Janus: Querying RDF Streams Across Past and Present

Kushagra Singh Bisen[1,*], Femke Ongenae[1] and Pieter Bonte[2]

[1]IDLab, Ghent University-imec, Ghent, Belgium
[2]Department of Computer Science, KU Leuven Campus Kulak, Kortrijk, Belgium

Abstract

In its current form, RDF Stream Processing systems focus on query evaluation over live data, whereas historical RDF data is managed through a separate query and storage infrastructure. As a result, applications that need to compare live data with the historical data cannot express the complete task as a single query. Instead, they must issue separate queries to a live stream processor and a historical RDF store, manually coordinate the results and select the historical interval against which the live observations should be compared. For example, in healthcare monitoring a current heart rate observation may only become clinically relevant when compared with the same patient's previous observations, such as their average heart rate over the last week. In this paper, we introduce Janus-QL, a query specification that allows historical windows over historical data and live data over RDF streams to be expressed in a single query model. We present Janus as an implementation engine capable of executing Janus-QL queries. We evaluate Janus by combining queries over historical RDF event logs with live RDF stream processing. The results show that Janus produces results equivalent to those of a decomposed execution pipeline. The processing latency remains comparable. Janus provides substantially lower historical-access latency, uses less CPU and memory for selective access patterns and requires approximately 13× less persistent storage at one million historical events.

Keywords

RDF Stream Processing, RDF Event Logs, Hybrid Query Processing, Continuous Queries

1. Introduction

Intelligent applications in domains such as healthcare, smart city and Industry 4.0, increasingly depend on combining information from heterogeneous data sources to support real-time decision making [1, 2]. Semantic Web technologies, RDF Stream Processing (RSP) [3] in particular, provide mechanisms to integrate heterogeneous live data sources with domain knowledge, enabling a unified semantic view across distributed information sources [4]. In smart-city and healthcare applications ontology-based integration has been successfully applied to combine heterogeneous clinical, sensor, and contextual data for monitoring, decision support, and personalized services [5, 6]. While live event streams continuously capture evolving observations, domain knowledge is comparatively more stable and typically only changes through ontology updates or modifications to reasoning rules. However, many monitoring and analytics scenarios require more than the combination of live observations with static background knowledge. Often, **meaningful interpretation depends on understanding how current observations relate to the historical behaviour of the same monitored entities** to detect trends and anomalies over time [7]. For example, a current heart rate observation can only be accurately assessed when compared against the patient's personalized historical baseline, e.g. average heart rate over the current week, while a traffic congestion value is best interpreted relative to historical observations of the same location. Detecting trends, deviations, and anomalies therefore requires reasoning across both current and past observations. Such use cases require a hybrid evaluation model in which live RDF stream data can be seamlessly correlated with historical RDF stream events, which

DMKG'26: 2nd International Workshop on Data Management for Knowledge Graphs, October 2026, Bari, Italy
*Corresponding author.
kushagrasingh.bisen@ugent.be (K. S. Bisen); Femke.Ongenae@ugent.be (F. Ongenae); pieter.bonte@kuleuven.be (P. Bonte)
https://kushbisen.com/ (K. S. Bisen); https://sites.google.com/view/femkeongenae (F. Ongenae); https://pieterbonte.be/ (P. Bonte)
0000-0003-0950-6043 (K. S. Bisen); 0000-0003-2529-5477 (F. Ongenae); 0000-0002-8931-8343 (P. Bonte)
© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-11-09

CEUR-WS.org/Vol-4322/paper_10.pdf

are persisted in a log. Although **current RDF Stream Processing systems** provide declarative query languages for evaluating continuous queries over live RDF streams [8, 3, 9, 10], **access to historical data is typically delegated to a separate storage and query infrastructure**. Consequently, applications that need to compare current and past observations are forced to distribute the computation task across two systems with two different query paradigms. The application layer must then align the timestamps of both queries and combine the results afterwards. This makes the **hybrid computations difficult to specify and maintain** as the developer has to set up separate query executions, coordinate their outputs and implement the comparison logic outside the query language. Besides increasing development complexity, this approach **obscures the intended query semantics** as they are not expressed in one reusable declarative query but are hidden in application-specific coordination code. This paper addresses this limitation by introducing **Janus-QL**, a **query language specification for expressing windows over historical data and live data in a single declarative query**. We further present **Janus**, an RDF stream processing engine capable of executing the Janus-QL queries over both historical RDF event logs and live RDF streams. The name Janus, which refers to the Roman god [1] who is depicted as simultaneously looking to the past and the future, reflects this dual perspective.

The research is guided by the research question:

> **RQ.** How can a single declarative query model that enables correlations over historical RDF event logs and live RDF streams be represented and executed?

We address this through the following three sub-questions:

- **RQ 1.1** How can a query language express both historical windows over persisted RDF event logs and live windows over RDF streams?
- **RQ 1.2** Can a unified stream query correlating historical and live data preserve the same results as a decomposed execution in which the historical access and live stream processing are evaluated separately and correlated at the application layer?
- **RQ 1.3** How does unified historical-live RDF stream processing scale compared to a conventional decomposed execution as the size of the historical RDF event log increases?

To address these questions, we present the following three contributions:

- We introduce the Janus-QL specification[2] as a query language to express historical windows over RDF event logs and live windows over RDF streams within one declarative query.
- We present Janus[3], an RSP engine that executes Janus-QL queries by integrating historical RDF event log access with live stream processing to produce a unified result stream.
- We evaluate Janus against an equivalent conventional decomposed execution pipeline to study the result equivalence and scaling behaviour as the historical RDF event log grows in size.

The remainder of the paper is organized as follows. Section 2 discusses the related work. Section 3 details the Janus-QL specification and its evaluation model. Section 4 introduces the Janus engine and its RDF Event log storage model used for historical data. Section 5 presents the experimental evaluation, while Section 6 discusses the findings, limitations and future work. We conclude the paper in Section 7.

2. Related Work

RSP extends RDF with continuous query evaluation over RDF streams. Query languages, such as C-SPARQL [8], RSP-QL [9], SPARQLstream [11], Streaming SPARQL [12] and CQELS [10], introduce window operators to transform an unbounded RDF stream into finite chunks to enable continuous evaluation. Recent work has surveyed this area by comparing RSP languages, engines and benchmarks [3].

[1] https://en.wikipedia.org/wiki/Janus
[2] https://spec.knows.idlab.ugent.be/janus-ql-specification/latest/
[3] https://github.com/SolidLabResearch/janus

Other work has focused on unifying or implementing RSP semantics across systems with RSP4J [13]. It can be noted that the execution model of RSP engines is centred on live stream windows, while historical observations are typically assumed to reside outside the stream processor, as external datasets or static background knowledge rather than first-class historical windows in the same declarative query. A separate line of work studies the **extension of RDF or SPARQL with temporal annotations**, such as Temporal RDF [14], SPARQL-ST [15], and T-SPARQL [16]. These approaches are important because they make time explicit in RDF data and enable queries over validity time, transaction time, or temporal relations. However, they also do not combine the querying of both live and historical in the same data model. **RDF archive systems** address the management of RDF data whose contents evolve over time [17, 18, 19, 20]. These approaches support the representation, storage and the querying of historical RDF states, and versions. This enables applications to inspect how an RDF dataset changed across time. However, such systems were only designed for dataset access rather than continuous query execution over live RDF streams. Previous work has combined SQL-based stream processing with historical data stores to support real-time analyses over live and archived data [21]. However, the live and historical components are queried separately and coordinated externally. Related functionality also exists outside the RDF ecosystem. CQL supports continuous queries over both streams and stored relations [22]. However, Janus-QL differs by exposing bounded historical RDF event-log intervals as named time-varying graphs within the RSP-QL Streaming Dataset, allowing them to be queried alongside live RDF stream windows using the same RSP query model. Janus instead provides a unified query abstraction in which live RDF streams and historical RDF logs are addressed through explicit temporal windows within the same continuous query. Consequently, despite significant advances in RDF stream processing, temporal RDF, and RDF archiving, the problem of seamlessly correlating historical and live RDF observations within a single declarative query and execution model remains, to the best of our knowledge, largely unaddressed.

3. Janus-QL

Janus-QL is a declarative query language for expressing windows over live and historical data enabling correlation between them in a single query. It extends RSP-QL [9] with window declarations over historical data. In the following subsections, we describe the Query Language and the evaluation model. The specification has been published in our lab's catalogue, preceding further standardization efforts[4].

3.1. Query Language

The Stream Reasoning community is veering to RSP-QL, the reference model proposed by Dell'Aglio et al. [9], as a unifying semantics of existing continuous SPARQL extensions and the operational semantics. RSP-QL has been applied to explain the operational semantics of three popular RSP engines, e.g. C-SPARQL, CQELS, and Morphstream, and the semantics of the continuous extensions of SPARQL they implement, i.e. C-SPARQL, CQELS-QL, and SPARQLstream. It has been adopted by modern RSP Engines, e.g. RSP4J [13] & Kolibrie [23]. Therefore, a Janus-QL query follows a RSP-QL style syntax.

The distinction Janus-QL makes is by introducing the capability to define windows over historical logs along live RDF streams. Janus-QL extends the RSP-QL dataset clause syntax by allowing a named window to be declared over a persisted RDF event log using `ON LOG`. The remainder of the query syntax is inherited unchanged from RSP-QL, and thus also SPARQL, including projection, graph patterns, aggregation, filters, and live windows declared using `ON STREAM`. Consequently, Figure 1 specifies only the Janus-QL specific syntactic additions rather than reproducing the complete RSP-QL grammar. An example query can be seen in Listing 1. Janus-QL follows the RSP-QL query model, which extends the SPARQL dataset clause with named windows over RDF streams. Accordingly, standard SPARQL graph declarations and graph patterns can be used alongside RSP-QL window declarations. A static named graph is addressed through the usual SPARQL `FROM NAMED` and `GRAPH` constructs, whereas a

[4]https://spec.knows.idlab.ugent.be/janus-ql-specification/latest/

$$
\begin{aligned}
\textit{logWindowClause} &::= \texttt{ON LOG } \ell \texttt{ [} \textit{fixedLogWindow} \mid \textit{slidingLogWindow} \texttt{]} \\
\textit{fixedLogWindow} &::= \texttt{START } t_{start} \texttt{ END } t_{end} \\
\textit{slidingLogWindow} &::= \texttt{OFFSET } offset \texttt{ RANGE } r_h \texttt{ STEP } \delta_h
\end{aligned}
$$

Figure 1: Janus-QL extension to the RSP-QL window declaration syntax.

time-varying graph is declared using `FROM NAMED WINDOW` and accessed through `WINDOW`. Janus-QL preserves this distinction and extends the RSP-QL window declaration with `ON LOG`, allowing a named window to be constructed from a historical RDF event log in addition to a live RDF stream. Janus-QL supports the following window forms:

1. Sliding windows over live RDF streams with `ON STREAM` and defining the range and step of the query with `RANGE` and `STEP`, inherited from RSP-QL.
2. Tumbling windows are supported over the live RDF streams where the `RANGE` equals the `STEP`.
3. Fixed windows over historical RDF event logs with `ON LOG` and defining the bound of the window with a `START` and `END` for the query (Line 4 of Listing 1).
4. Sliding windows over historical RDF event logs with `ON LOG` and defining the offset `OFFSET` relative to the current evaluation time and the sliding behaviour of the window starting from the offset with `RANGE` and `STEP` for the query. At evaluation time T, the historical window advances by δ_h independently, live evaluations reuse the latest materialized historical context until the next historical evaluation.
5. Tumbling windows over historical RDF event logs are supported where the `RANGE` equals the `STEP`.

Additional conditions are checked during Janus-QL validation to see whether the query is well-formed. Window identifiers must be unique, ranges and steps must be positive, and a fixed historical window must satisfy ($t_{start} < t_{end}$). For a sliding historical window with range (r_h) and offset *offset*, the condition ($r_h \leq offset$) ensures that the selected interval does not extend beyond the current evaluation instant. Additional constraints regarding the temporal relationship between historical and live windows are defined in Section 3.2.

Listing 1: Janus-QL hybrid query comparing a fixed historical congestion window for 12:00–12:01 UTC on 1 August 2026 (Unix epoch milliseconds) with a 60-second live window sliding every 30 seconds, returning the difference between historical and live average congestion per sensor.

```
1  PREFIX ex: <http://example.org/>
2  REGISTER RStream <output> AS
3  SELECT ?sensor ((AVG(?live) - ?histAvg) AS ?delta)
4  FROM NAMED WINDOW ex:hist ON LOG <LOG_IRI> [START 1785585600000 END 1785585660000]
5  FROM NAMED WINDOW ex:live ON STREAM <STREAM_IRI> [RANGE 60000 STEP 30000]
6  WHERE {
7    {
8      SELECT ?sensor (AVG(?hist) AS ?histAvg)
9      WHERE {
10       WINDOW ex:hist {
11         ?sensor ex:congestionLevel ?hist .
12       }
13     }
14     GROUP BY ?sensor
15   }
16   WINDOW ex:live {
17     ?sensor ex:congestionLevel ?live .
18   }
19 }
20 GROUP BY ?sensor ?histAvg
```

3.2. Evaluation Model

The evaluation model of Janus-QL defines how windows over live and historical data are interpreted within one query model. Janus-QL achieves this by extending the evaluation model of RSP-QL by introducing window operators over historical data stored in RDF event logs.

Definition 1 (RDF Event). *An* RDF event *is defined as a pair $e = (q, \tau)$, where q is an RDF quad and $\tau \in \mathbb{T}$ is the time instant associated with q.*

Definition 2 (Live RDF Stream). *A* live RDF stream *is defined as a potentially unbounded sequence of RDF events, $S_{\text{live}} = \langle e_1, e_2, e_3, \ldots \rangle$, in non-decreasing time order, such that for every $i \geq 1$, $\tau_i \leq \tau_{i+1}$.*

In RSP-QL, each declared window is evaluated at every evaluation instant to produce a time-varying graph (TVG). These graphs together with a default graph form the Streaming Dataset (SDS) defined as,

Definition 3 (RSP-QL Dataset SDS, adapted from Definition 14 of Dell'Aglio et al. [9]). *An RSP-QL dataset SDS is a set composed by an (optional) default graph, n ($n \geq 0$) named graphs and m ($m \geq 0$) named time-varying graphs obtained by application of time-based sliding windows over o ($o \leq m$) streams where:*

- *G_0 is the default time-varying graph;*
- *u_p and w_q are IRIs, with $u_p, w_q \in \mathcal{I}$, for each $p \in [1, n]$ and $q \in [1, m]$, where $\mathcal{I}$ denotes the set of IRIs;*
- *(u_p, G_p) identifies a time-varying named graph, for each $p \in [1, n]$;*
- *$\big(w_q, \mathscr{W}_q(S_{live})\big)$ identifies the named time-varying graph obtained by applying the time-based sliding-window operator $\mathscr{W}_q$ to RDF stream $S_{i}ive$, for each $q \in [1, m]$, where $r \in [1, o]$ identifies the stream associated with the window.*

Here, $[1, x]$ denotes the set of integers $\{1, \ldots, x\}$.

Definition 4 (Historical RDF Event Log). *A* historical RDF event log *is defined as a finite sequence of RDF events, $L = \langle e_1, e_2, \ldots, e_n \rangle$ Given the current query evaluation instant T, the visible portion of the historical RDF event log is,*

$$L(T) = \{(q, \tau) \in L \mid \tau \leq T\},$$

that is, the subset of events whose timestamps are less than or equal to T.

Throughout this section, T denotes the current query evaluation instant at which the Streaming Dataset is constructed. In contrast, t_{start} and t_{end} denote the absolute temporal bounds declared by a fixed historical window using START and END. Thus, T changes as the continuous query is evaluated over time, whereas t_{start} and t_{end} remain fixed for a given window declaration. Janus-QL introduces two window operators on the Historical RDF Event Log:

- A fixed window over the historical data stream selects all events between a declared interval creating a time-varying graph,

$$G(T) = \begin{cases} \{q \mid (q, \tau) \in L(T), t_{start} \leq \tau < t_{end}\} & \text{if } T \geq t_{end} \\ \varnothing & \text{if } T < t_{end} \end{cases}$$

  A fixed historical window represents the complete interval $[t_{start}, t_{end})$ rather than a progressively growing set for that interval. It therefore remains empty while $T < t_{end}$ and becomes available once the evaluation instant reaches or passes t_{end}. For example, consider the historical interval from 12:00 to 12:01 UTC on 1 August 2026 used in Listing 1. At an evaluation instant of 12:00:30, $L(T)$ already contains events observed up to 12:00:30, but the fixed historical window remains empty because its declared interval is not yet complete. At $T \geq$12:01, the complete interval is materialized as the historical time-varying graph. Thus, $L(T)$ determines which events are visible

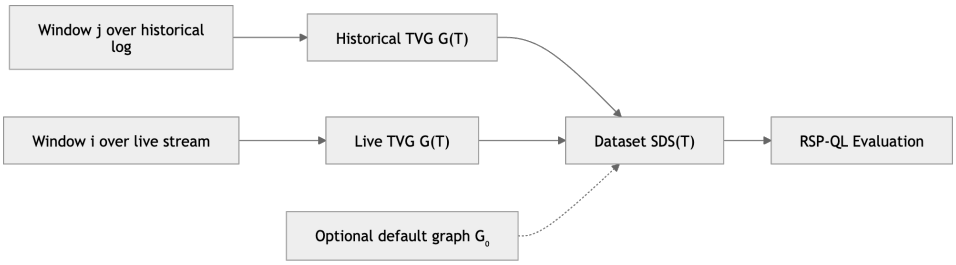


Figure 2: Construction of the Janus-QL Streaming Dataset at evaluation time T. Windows over live streams and historical logs produce TVGs that, together with the default graph, form SDS(T) for RSP-QL evaluation.

at evaluation instant T, whereas the fixed-window operator additionally requires the declared interval to be complete. This definition assumes that late-arriving and out-of-order events are not added to a completed interval, as these are not currently supported by Janus and are left for future-work.

- A sliding window over the historical data stream with an offset ($offset$) and a range (r_{hist}) defining the historical duration to go back (from evaluation time T) from where the sliding behaviour should start, $G(T) = \{q \mid (q, \tau) \in L(T), T - offset \leq \tau < T - of\,fset + r_{hist}\}$

The essence of the evaluation model of Janus-QL, as illustrated in Figure 2, lies in how it extends the standard RSP-QL Streaming Dataset (SDS) without altering the underlying semantics regarding the query execution. In RSP-QL, the SDS is traditionally populated with a default graph, static named graphs and named time-varying graphs (TVGs) derived exclusively from windows over the live stream. Janus-QL extends this architecture by injecting the historical data through a mapping process:

1. **Log to TVG:** The window operators over the historical log data stream are evaluated against the finite event log $L(T)$. As defined above, the output of these operators, $G(T)$, is structurally identical to a standard TVG.
2. **TVG to SDS:** As the RSP-QL dataset definition accommodates named TVGs regardless of their state of origin, the TVGs derived from the historical log are seamlessly appended to the dataset as additional named TVGs alongside the live stream graphs.
3. **Semantic Reuse:** With the historical data now formalized as TVGs within the SDS, Janus-QL inherits the RSP-QL evaluation function unmodified. For any evaluation instant T, the algebraic expression SE is evaluated over the unified dataset as eval(SDS(G_0), SE, T)

For a sliding historical window, Janus-QL requires $r_{hist} \leq offset$, ensuring that the selected historical interval does not extend beyond the current evaluation instant T. When the query additionally contains a live window with range $r_{live} > 0$, the current Janus-QL specification applies the stronger constraint

$$r_{hist} + r_{live} \leq offset,$$

which also implies the past-only condition and ensures that the historical and live windows do not overlap. This non-overlap condition is a design restriction of the current Janus-QL specification rather than a requirement of the underlying RSP-QL evaluation model and it ensures a clear temporal separation between the historical context and the current live observations. Allowing intentionally overlapping historical and live windows is left for future work in this direction.

4. Janus

Janus implements the Janus-QL query model described in Section 3 and provides the execution components needed to evaluate Janus-QL queries. It is worth noting that Janus-QL does not prescribe a particular storage implementation for historical data and the event log described below is a minimal implementation choice for the Janus engine. The implementation separates live stream processing from the historical stream access, and combines the context during live evaluation.

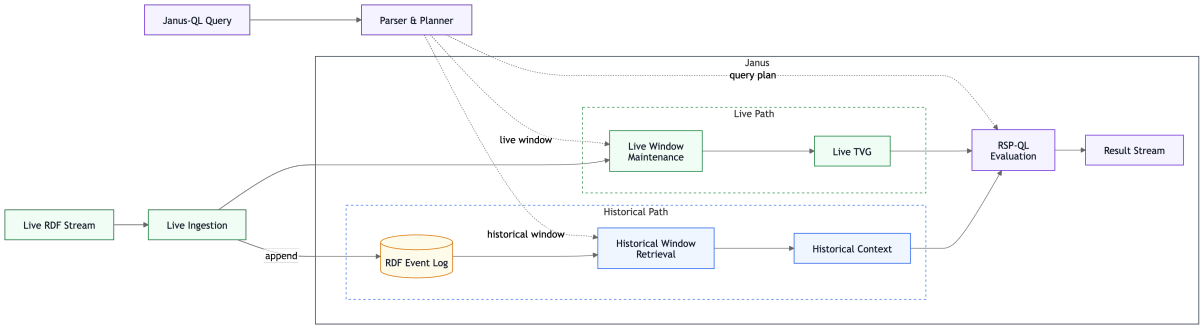


Figure 3: Architecture of Janus. A Janus-QL query is parsed into live and historical execution components. Incoming RDF events are processed by the live path and concurrently appended to the RDF Event Log for future historical access. Historical window queries retrieve and evaluate previously persisted events to materialize RDF context. This historical context is supplied to the live RSP-QL evaluation, where it is combined with the live TVGs to produce a single result stream.

4.1. System Architecture

Figure 3 shows the architecture of Janus. A Janus-QL query is parsed for different execution components. The window declarations on live data are passed to the live RSP execution path, while historical window declarations are passed into historical access and query evaluation over the RDF event log. Incoming RDF stream events are handled by the live ingestion path. As the present will be the past in the future, each event is simultaneously added to the live stream processor and persisted in the RDF Event log to make it available for future historical queries. The live stream processing is delegated to an underlying RSP engine. We have chosen to integrate the RSP-RS engine [5] in Janus. RSP-RS is a lightweight engine which provides the specific functionality needed for Janus. Moreover, both Janus and RSP-RS are implemented in Rust allowing the engine to be embedded directly into the Janus runtime and adapted where necessary. Janus retrieves the historical context when required based on windows on the live stream. At each evaluation instant T, Janus materializes the result of each historical window as context with its declared window identifier. The context is static during that specific evaluation, but it is time-varying across evaluation instants because its contents are recomputed according to the historical window operator. The resulting context is then supplied to the RSP-RS engine together with the live TVGs.

4.2. RDF Event Log Storage

As shown in Figure 4, Janus persists incoming RDF stream events in an append-only RDF event log, enabling future queries to access historical stream data. Since historical query evaluation is performed frequently and new events continue to arrive, optimizing the storage layer is essential. It must support high-throughput writes and low-latency historical access. As the size of the log grows, a naive storage layout would increase both the storage consumption and latency for historical access. To address this challenge, the RDF event log employed by Janus is minimal by design. The purpose is not to provide a general-purpose RDF store, but support the specific access pattern required by Janus by appending incoming RDF events and retrieving events over a timestamp interval. By restricting the storage layer only to this timestamp-based access pattern, we avoid additional complexity and overhead of supporting arbitrary SPARQL queries, updates or complex indexing structures. It keeps the historical event access lightweight. To reduce storage overhead, each timestamped RDF quad, consisting of a subject, predicate, object and graph, is encoded as compact numeric identifier before being written to the log. This avoids redundant storage of the same terms across event records, while preserving the ability to reconstruct the original events during query evaluation. The term dictionary is maintained separately and persisted in the file `dictionary.bin`. The event log is organized into immutable segments. The encoded events

[5] https://github.com/SolidLabResearch/rsp-rs

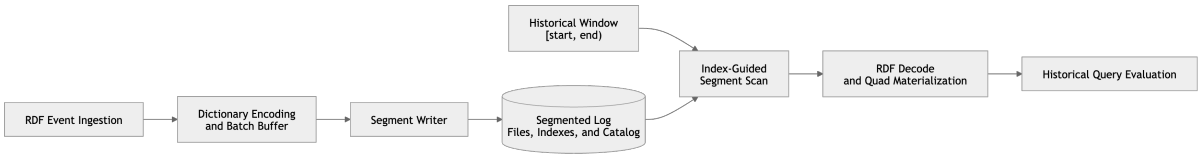


Figure 4: Segmented RDF event log storage in Janus. RDF events are encoded, buffered, and flushed into append-only segments with sparse indexes. Historical-window queries use segment metadata and checkpoint indexes to scan relevant records before materializing them for historical SPARQL evaluation.

are first accumulated into an in-memory batch buffer and are flushed to the disk when configurable buffer-size thresholds are reached. Each flush creates a new segment consisting of the 1) log file `segment-N.log`, which stores the fixed-size encoded event records, and 2) index file `segment-N.idx`, which stores the offset checkpoints in relation to the timestamp to allow efficient location of relevant log regions without requiring a full scan of all stored events. Historical windows in Janus-QL are resolved to timestamp ranges over this event log. To retrieve the range, Janus first checks the in-memory batch buffer, enabling low-latency access to recently ingested events. It then identifies storage segments whose timestamp intervals overlap with the requested range. For each overlapping segment, Janus utilizes the sparse index checkpoint to seek near the requested start time to start and scans forward until the end of the range. The storage layer returns the encoded events after reading, which are decoded into RDF quads with timestamp and passed into the historical query evaluator for SPARQL execution.

5. Evaluation

This section evaluates Janus using a unified query over a historical window from a persisted RDF event log and a live window from an incoming RDF stream. We compare Janus against a baseline decomposed pipeline and measure how execution costs evolve as the historical RDF event log grows.

5.1. Dataset & Query

The evaluation uses a synthetic traffic stream inspired by the traffic scenario of Citybench and replayed to an MQTT topic. Each event represents a congestion-level observation for a traffic sensor. Citybench models traffic monitoring through sensor observations such as congestion level, vehicle count, average speed and the estimated travel time. Since the original Citybench benchmark targets continuous queries over live streams only, it does not provide the historical event logs or scalable historical datasets required for Janus. Therefore, we use a configurable traffic generator that produces the congestion observations and adapting the Q3 scenario for traffic monitoring allowing historical RDF event log to be queried as well. Moreover, we can generate historical RDF event log at different sizes for the experiments related to scalability. The workload adapts the congestion-level aspect of the Q3 query from Citybench to the hybrid setting. The query computes an average live traffic-value for the sensor over the live data and compares it with the corresponding historical average over historical data.

> Q3: What is the average congestion level and estimated travel time to my destination?
>
> Q3-Adapted: How does the current average congestion level for each traffic sensor compare with its historical average congestion level?

The adapted query keeps the aggregate aspect of Q3, but replaces estimated travel time with a per sensor historical congestion. Listing 1 presents the Janus-QL hybrid query used for the evaluation. The query computes the average congestion level over a historical window selected from the RDF event log and a live window selected from the RDF stream, then reports the difference between both averages.

5.2. Baseline

The experiment compares Janus against a decomposed execution pipeline. It is composed of the historical logs stored as RDF with Oxigraph and a live RSP pipeline with an RSP-RS Engine [6]. In the decomposed pipeline, separate SPARQL and RSP-QL queries are evaluated and their results are correlated externally.

5.3. Experiment Set-up

For each configuration, we measure the execution cost and whether the results are equivalent. We measure the latency of the historical RDF event log access, and processing latency from window closure to result as well as the resource usage. The historical RDF event log is scaled with the number of events, i.e. 10k, 50k, 100k, 500k and 1M. Moreover, we vary the range of the considered events in the queries to calculate the historical context as well with the following query types, 1) Point lookup, 2) Fixed Window of 60 seconds, 3) 10% Range Query, 4) 50% Range Query and 5) 100% Full Range Query, while keeping the RDF stream behaviour and query stable to emphasize the impact of scaling the historical events in the RDF event log. For a historical RDF event log containing N events ordered by timestamp, a k% range query retrieves the contiguous suffix containing exactly $kN/100$ events. The temporal bounds of the query are given by the timestamps of the first and last events in the selected suffix. For the streaming data, the experiment employs a live window of 60 seconds with a slide of 30 seconds. The evaluation was executed on a server running Ubuntu 22.04.5 LTS, equipped with an Intel(R) Core(TM) i5-9500 CPU @ 3.00 GHz processor and 62 GiB of RAM. Each configuration was executed 35 times. We report results over iterations 4–33, discarding the others as warm-up and cool-down.

5.4. Results

Figure 5 shows the historical access latency observed inside the hybrid benchmark. The Janus hybrid engine keeps the point and the fixed-window access close to constant across the evaluated history sizes, whereas decomposed baseline increases approximately linearly with the number of historical quads present in the RDF event log. For queries with a selective range, Janus scales with the amount of data needed, the 10% range query increases from 1.313 ms at 10k quads to 111 ms at 1M quads while the 100% range query increases from 12.31 ms to 1454.357 ms. The result is expected as the larger historical windows require more events to be constructed and materialized. We see in these results that Janus remains faster than decomposed baseline for all of the historical access patterns except the smallest full-range case with 10k quads where both systems are close.

Before comparing latency, we verified that Janus and the baseline pipeline produced equivalent results across all query patterns and across all runs. Table 1 reports the latency for the first hybrid result and should be interpreted together with the historical access latencies reported in Figure 5. Since both the decomposed baseline and Janus's event log are fast enough to be done fetching and computing before the window is supposed to be triggered, no additional delay is observed in the RSP execution of a window. In our experiment, all configurations produce the first hybrid result after approximately 251 ms the corresponding window closes, with negligible differences between Janus and the decomposed baseline.

Figure 6 summarizes both runtime resource usage and the persistence costs. Panels (a) and (b) report peak memory and CPU utilization at one million historical RDF events evaluated across various queries for the access patterns. Janus utilizes less memory than the decomposed baseline for point lookup, fixed window access and the 10% range query. For the 50% and 100% range queries, memory usage of both systems becomes similar to Janus when querying a larger fraction of the event log. Regarding CPU, Janus remains around 12–13.5% across the access patterns, whereas the decomposed baseline remains around 20–21%. Panels (c) and (d) report the persistence cost of storing historical RDF events. Janus maintains a near-constant storage footprint of around 24-25 bytes per event, growing from 0.24MB

[6] https://github.com/SolidLabResearch/rsp-rs

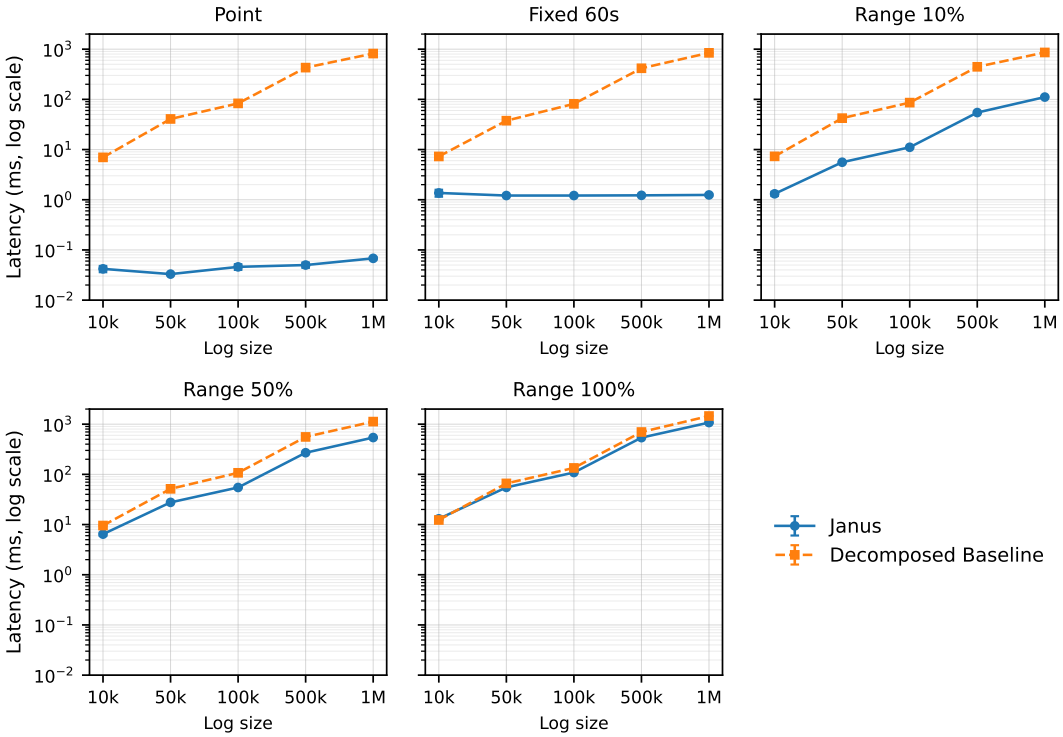


Figure 5: Historical access latency as the RDF event log grows from 10k to 1M RDF events. Points show mean latency and error bars show standard deviation. All panels use the same logarithmic y-axis to make the access patterns comparable. Janus provides the largest latency reduction for point and fixed-window access, while the gap narrows for larger range queries because more historical data must be materialized.

Table 1
Processing latency for the first hybrid result (calculated as $T_{\text{emission}} - T_{\text{window_close}}$ for a window with a 30-second step). Values are mean ± standard deviation in ms.

System	Fixed 60s	Range 10%	Range 50%	Range 100%
Janus	251.4 ± 0.1	251.4 ± 0.1	251.4 ± 0.1	251.3 ± 0.1
Decomposed baseline	251.1 ± 0.1	251.1 ± 0.1	251.1 ± 0.1	251.1 ± 0.1

at 10k events to 231.28 MB at 10 M events. In contrast, the Oxigraph-based persistent store of the decomposed baseline grows from 3.31 MB to 6.22 GB over the same range. At one million historical events, Janus requires 23.14 MB of persistent storage compared with 302.83 MB for the Oxigraph-based baseline, meaning that the baseline requires approximately 13.1× more storage than Janus. The median ingestion time follows the same trend at 10M events with Janus writing the event log in 9.32 s, whereas the decomposed baseline in 63.6 s. The results support using a minimal append-oriented RDF log for historical stream access rather than relying on a general-purpose RDF store. Overall, the results show that Janus preserves the observable behaviour of the decomposed execution while reducing the cost of the historical part of the query. Janus has a comparable processing latency compared to the decomposed baseline. However, Janus retrieves and materializes the historical context faster, uses less CPU, and stores the historical events with a substantially smaller persistent footprint. It provides the Janus implementation more headroom for queries with shorter steps and larger histories.

6. Discussion

We revisit the research questions introduced in Section 1. With respect to the overall **RQ** and **RQ 1.1**, Janus-QL addresses the language aspect of the question by enabling windows over historical and

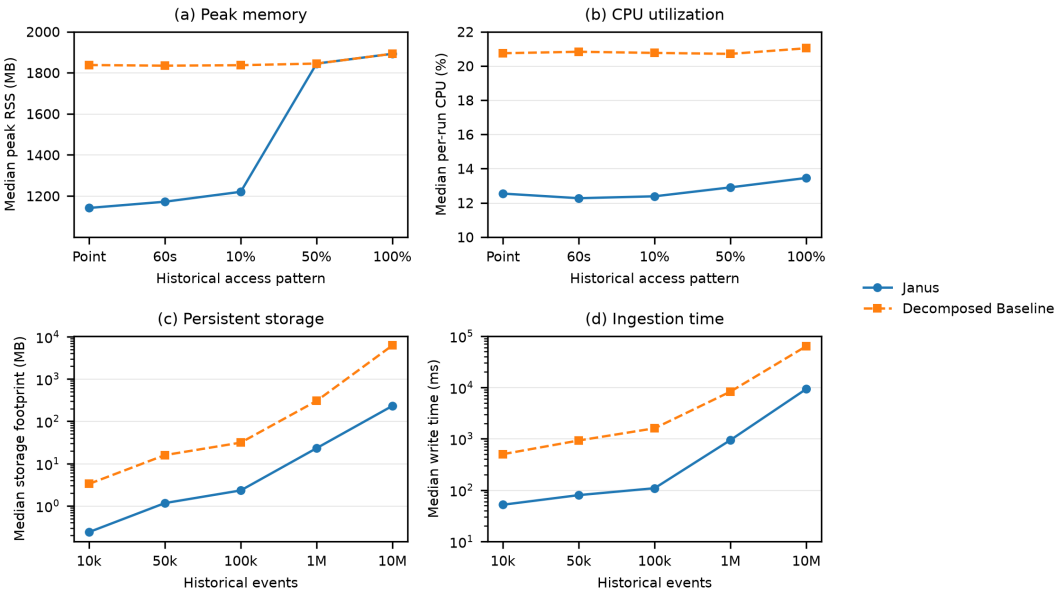


Figure 6: Resource usage and storage costs. Panels (a) and (b) show median peak RSS memory and median per-run CPU utilization at one million historical quads across historical access patterns. Panels (c) and (d) show median storage footprint and median ingestion time for writing historical RDF events to disk when increasing event counts. Janus uses less CPU across all access patterns and less memory for selective access patterns, while memory converges for larger historical ranges. Janus also maintains a substantially smaller persistent footprint and lower ingestion time than the Oxigraph persistent store. Panels (c) and (d) use logarithmic y-axes.

live RDF streams in one query. Janus addresses the execution aspect by evaluating windows over historical data over an event log and injecting the historical context into live stream processing. With respect to the result correctness of **RQ 1.2**, the evaluation shows that the unified Janus execution produces the same answers as the decomposed execution. It supports the claim that historical-live correlations can be expressed and executed within a single declarative query language. With respect to **RQ 1.3**, the scaling results show that Janus is most beneficial for specific historical retrieval patterns. Queries related to point lookups, fixed historical windows and smaller range queries benefit from the segmented RDF event log, however, for larger range scans, such as for the 50% and 100% range queries, the advantage decreases as Janus becomes similar to the baseline. The behaviour is consistent with the design goal of the implementation by providing efficient partial access to events from the entire history. Optimizing the historical access reduces the internal cost of the execution plan of the hybrid query. Faster historical retrieval, however, gives the Janus implementation more headroom for workloads with shorter window steps, larger histories and frequent queries. The resource and persistence results support the use of a minimal storage layer for historical events as Janus leads to a lower CPU usage, lower memory usage for selective access patterns and a substantially smaller persistent footprint. The trade-off is important for systems where full SPARQL functionality might not be needed and a simple time-bounded query is enough for retrieval. The performance results characterize the Janus reference implementation rather than Janus-QL alone. Janus-QL specifies historical-window semantics but not the underlying storage or indexing. Since Janus uses a segmented event log while the baseline uses Oxigraph, the observed differences reflect both storage design and execution model. Thus, the comparison demonstrates one implementation of Janus-QL rather than isolating the language's contribution to a specific implementation. The evaluation focuses on a single hybrid query shape combining one historical and one live window and evaluating more complex Janus-QL queries involving multiple historical windows or joins across several live and historical sources is left for future work.

7. Conclusion

The paper presented Janus-QL as an extension to RSP-QL to declare the windows over historical and live data in a single declarative query and introduces Janus as an implementation of Janus-QL with a segmented historical RDF event log that stores the historical events. The evaluation shows that Janus preserves result equivalence with a decomposed baseline while reducing the cost of historical access. Janus reduces historical-access latency, CPU and memory usage for selective access patterns, and the persistent storage footprint for most patterns. The processing latency remains similar for the first hybrid result. These results demonstrate that Janus can execute Janus-QL queries correlating historical and live RDF data without requiring applications to coordinate separate query pipelines. Future work will extend Janus with result stream output, such as `IStream` and `DStream`, and evaluate the system under concurrent workloads, irregular event rates, out-of-order events, and different RDF event-log segment configurations.

Acknowledgments

The work is partly funded by the following projects: FWO FRACTION (Nr.G086822N), BOF FRACTION (BOF/24J/2021/388), and the imec.icon PACSOI, which was co-financed by imec and VLAIO and brings together the following partners: FAQIR Foundation, FAQIR Institute, MoveUP, Byteflies, AContrario, and Ghent University – IDLab.

Declaration on Generative AI

During the preparation of this work, the authors used OpenAI's ChatGPT (GPT-5.5) for grammar and spelling checks, paraphrasing and rewording, improving writing style, and peer-review simulation. OpenAI Codex was used to assist with debugging and implementation-related code revisions. All AI-assisted outputs were critically reviewed, edited, tested, and validated by the authors as appropriate. The authors take full responsibility for the content of the publication.

References

[1] B. P. L. Lau, S. H. Marakkalage, Y. Zhou, N. U. Hassan, C. Yuen, M. Zhang, U.-X. Tan, A survey of data fusion in smart city applications, Information Fusion 52 (2019) 357–374. URL: http://dx.doi.org/10.1016/j.inffus.2019.05.004. doi:10.1016/j.inffus.2019.05.004.

[2] C. Peng, P. Goswami, Meaningful integration of data from heterogeneous health services and home environment based on ontology, Sensors 19 (2019). URL: https://www.mdpi.com/1424-8220/19/8/1747. doi:10.3390/s19081747.

[3] P. Bonte, C. Callé, O. Curé, H. Kondylakis, R. Tommasini, Languages and systems for rdf stream processing, a survey: P. bonte et al., The VLDB Journal 34 (2025) 50.

[4] S. Hoseini, J. Theissen-Lipp, C. Quix, A survey on semantic data management as intersection of ontology-based data access, semantic modeling and data lakes, J. Web Semant. 81 (2024) 100819. URL: https://api.semanticscholar.org/CorpusID:269437421.

[5] F. Ongenae, J. Famaey, S. Verstichel, S. D. Zutter, S. Latré, A. Ackaert, P. Verhoeve, F. D. Turck, Ambient-aware continuous care through semantic context dissemination, BMC Medical Informatics and Decision Making 14 (2014). URL: https://api.semanticscholar.org/CorpusID:15094013.

[6] F. Gao, M. I. Ali, A. Mileo, Semantic discovery and integration of urban data streams, in: Proceedings of the Fifth International Conference on Semantics for Smarter Cities - Volume 1280, S4SC'14, CEUR-WS.org, Aachen, DEU, 2014, p. 15–30.

[7] F. Ongenae, M. Claeys, T. Dupont, W. Kerckhove, P. Verhoeve, T. Dhaene, F. De Turck, A probabilistic ontology-based platform for self-learning context-aware healthcare applications, Expert

Systems with Applications 40 (2013) 7629–7646. URL: https://www.sciencedirect.com/science/article/pii/S0957417413005174. doi:https://doi.org/10.1016/j.eswa.2013.07.038.

[8] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, M. Grossniklaus, C-sparql: Sparql for continuous querying, in: Proceedings of the 18th international conference on World wide web, 2009, pp. 1061–1062.

[9] D. Dell'Aglio, E. Della Valle, J.-P. Calbimonte, O. Corcho, Rsp-ql semantics: A unifying query model to explain heterogeneity of rdf stream processing systems, Int. J. Semant. Web Inf. Syst. 10 (2014) 17–44.

[10] D. Le-Phuoc, M. Dao-Tran, J. Xavier Parreira, M. Hauswirth, A native and adaptive approach for unified processing of linked streams and linked data, in: International Semantic Web Conference, Springer, 2011, pp. 370–388.

[11] J.-P. Calbimonte, H. Jeung, O. Corcho, K. Aberer, Enabling query technologies for the semantic sensor web, International Journal On Semantic Web and Information Systems (IJSWIS) 8 (2012) 43–63.

[12] A. Bolles, M. Grawunder, J. Jacobi, Streaming sparql-extending sparql to process data streams, in: European Semantic Web Conference, Springer, 2008, pp. 448–462.

[13] R. Tommasini, P. Bonte, F. Ongenae, E. Della Valle, Rsp4j: an api for rdf stream processing, in: European Semantic Web Conference, Springer, 2021, pp. 565–581.

[14] C. Gutierrez, C. Hurtado, A. Vaisman, Temporal rdf, in: European Semantic Web Conference, Springer, 2005, pp. 93–107.

[15] M. Perry, P. Jain, A. P. Sheth, Sparql-st: Extending sparql to support spatiotemporal queries, in: Geospatial semantics and the semantic web: foundations, algorithms, and applications, Springer, 2011, pp. 61–86.

[16] F. Grandi, T-sparql: a tsql2-like temporal query language for rdf (2010).

[17] O. Pelgrin, L. Galárraga, K. Hose, Towards fully-fledged archiving for rdf datasets, Semantic Web 12 (2021) 903–925.

[18] R. Taelman, M. Vander Sande, R. Verborgh, Ostrich: versioned random-access triple store, in: Companion Proceedings of the The Web Conference 2018, 2018, pp. 127–130.

[19] T. Windels, W. Slabbinck, P. Bonte, S. Verstichel, P. Colpaert, S. Van Hoecke, F. Ongenae, Ldests: enabling efficient storage and querying of large volumes of time series data on solid pods, in: ISWC2023: the International Semantic Web Conference, volume 3632, CEUR, 2023.

[20] N. Arndt, N. Radtke, M. Martin, Distributed collaboration on rdf datasets using git: Towards the quit store, in: Proceedings of the 12th International Conference on Semantic Systems, 2016.

[21] F. Reiss, K. Stockinger, K. Wu, A. Shoshani, J. M. Hellerstein, Enabling real-time querying of live and historical stream data, in: Proceedings of the 19th International Conference on Scientific and Statistical Database Management, IEEE Computer Society, 2007, p. 28.

[22] A. Arasu, S. Babu, J. Widom, The cql continuous query language: Semantic foundations and query execution, The VLDB Journal 15 (2006) 121–142. doi:10.1007/s00778-004-0147-z.

[23] V. Kadzhaia, P. Bonte, Neuro-symbolic stream reasoning with kolibrie, in: European Semantic Web Conference, Springer, 2026, pp. 39–57.