It's About Time: Time Functions for Comparing Partial and Floating Time Literals in SPARQL

Ieben Smessaert, Julián Rojas Meléndez, Pieter Colpaert

IDLab, Department of Electronics and Information Systems, Ghent University - imec, Ghent, Belgium

Abstract

Working with temporal data on the Semantic Web remains challenging due to SPARQL's limited support for comparing time literals of different data types and handling floating times without explicit time zones. These issues are especially problematic when dealing with partial time literals (such as xsd:date, xsd:gYearMonth, or xsd:gYear) and floating times, both of which are common in real-world knowledge graphs like Wikidata. To showcase the relevance and urgency of the problem, we gathered and reviewed existing discussions, specifications, draft proposals, and examples from deployed knowledge graphs, providing a consolidated starting point for further community dialogue. We then proposed a solution in the form of a set of SPARQL extension functions—**Time Functions**—designed to reinterpret time literals as time intervals, enabling consistent and type-agnostic temporal comparisons. These functions are formally described using the Function Ontology (FnO), and implemented in the Comunica query engine, with a publicly available demo application that allows users to interactively explore and test the functions. The demo includes curated example queries that highlight both the limitations of existing SPARQL behavior and how the Time Functions enable more accurate filtering and sorting of temporal data. In addition to providing a technical proposal, we advocate for improved temporal data publishing practices, urging data providers to use accurate data types and explicit time zones to support reliable temporal reasoning in the open-world context of RDF.

Keywords

SPARQL, Time Literals, Extension Functions, Partial Times, Floating Times

Demo: https://smessie.github.io/TimeFunctions-SPARQL-Editor/
Canonical version: https://smessie.github.io/Article-ISWC2025-TimeFunctions/

1. Introduction

Temporal literals are common in many datasets on the Semantic Web. SPARQL, the standard query language for RDF, is frequently used to query, filter, and compare such temporal information. However, current support for comparing and reasoning over time-related literals in SPARQL is limited, especially when it comes to *partial time literals* (e.g., xsd:gYear, xsd:gYearMonth, xsd:date) and *floating times* (i.e., time literals without explicit time zones).

Although the RDF 1.1 standard [1] recommends the use of various built-in XML Schema temporal data types (e.g., xsd:dateTime, xsd:date, and xsd:gYearMonth), the operator mappings from SPARQL 1.1 [2] define comparison semantics only for literals of the same data type. Cross-datatype comparisons (e.g., comparing an xsd:date with an xsd:dateTime) are not defined, and existing SPARQL engines

ISWC 2025 Companion Volume, November 2–6, 2025, Nara, Japan

ieben.smessaert@ugent.be (I. Smessaert); julianandres.rojasmelendez@ugent.be (J. Rojas); pieter.colpaert@ugent.be (P. Colpaert)

0009-0004-5281-0723 (I. Smessaert); 0000-0002-6645-1264 (J. Rojas); 0000-0001-6917-2167 (P. Colpaert)

© 2025 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

and querying frameworks such as Virtuoso, BlazeGraph, and Comunica return false or empty results in these cases, even if the date parts of the literals are comparable.

This issue becomes problematic in large-scale, real-world knowledge graphs like Wikidata, where users can specify date precision (e.g., year, month, day) when entering temporal data. Its SPARQL endpoint, however, returns fully qualified xsd:dateTime values without indicating the original precision. For example, a historical event entered as 27th century BCE may appear as such in the user interface, but is internally represented and queryable only as "–2650-01-01T00:00:00Z"^^xsd:dateTime. This loss of precision in the queryable data undermines temporal reasoning and can produce misleading query results, such as asserting that this historical event happened in the month of January.

Another critical challenge arises with *floating times*, which are time literals that lack time zone information. A literal like "2025-08-01T12:00:00"^^xsd:dateTime can be interpreted differently depending on the context, or may be wrongly adjusted to the user's local time zone. This ambiguity makes consistent comparison and sorting unreliable. While the W3C Group Draft Note on Working with Time and Timezones [3] recommends treating such floating times as UTC by default, this approach is insufficient in the open and distributed context of the Semantic Web, where data originates from diverse sources with potentially different implicit time zone assumptions. As discussed in the GitHub issue regarding implicit time zones in SPARQL[1], it is problematic to consider floating times from different sources as equal because they can represent different time instants and thus have different implicit time zones.

A more robust approach is to treat all temporal literals—particularly floating and partial ones—as time intervals, bounded by their earliest and latest possible interpretations. For example, a floating time can be represented by the time interval it could occupy across the full range of legal time zone offsets (–14:00 to +14:00, as per XML Schema Part 2: Datatypes Second Edition [4]). This perspective enables meaningful comparison between floating, partial, and fully-qualified time literals by aligning them to their temporal bounds rather than requiring exact matches.

Despite the growing interest in temporal reasoning on knowledge graphs, e.g., in works such as Soulard et al. 2025 [5], issues around time zones and floating times are rarely addressed explicitly. Moreover, while enhancement proposals like SEP-0002[2] improve SPARQL's handling of date-time arithmetic, they do not cover cross-type comparison or floating time semantics.

This paper aims to highlight these overlooked issues and present a concrete, extensible solution. We propose a set of SPARQL extension functions—called *Time Functions*—that enable comparison across different temporal data types, interpretation of floating and partial times as intervals, and consistent and explainable time-based filtering and sorting.

The remainder of this paper is structured as follows. Section 2 introduces the proposed SPARQL extension functions for time handling, along with their formal semantics. Subsequently, Section 3 presents a demo application showcasing how these functions enable richer and more accurate temporal queries over RDF data. Finally, Section 4 concludes with a discussion of how Time Functions address the limitations of SPARQL for temporal reasoning, highlights their relevance for future standardization, and encourages improved temporal data publishing practices.

2. Time Functions

To address the limitations of SPARQL when dealing with temporal data, particularly the inability to compare partial or floating time literals, we propose a set of SPARQL extension functions called **Time Functions**. These functions allow for consistent, meaningful comparison and reasoning over heterogeneous temporal literals by interpreting them as **time intervals**.

The formal specification and accompanying ontology for these functions are available at https://w3id.org/time-fn. The ontology uses the namespace https://w3id.org/time-fn# with the recommended prefix tfn:. This specification defines the semantics of each function and provides an ontology for integrating time-aware logic into SPARQL queries. The ontology relies on the Function Ontology (FnO) [6] to formally describe the semantics, inputs, and outputs of each function. FnO provides a reusable and machine-readable vocabulary for specifying function metadata, well-suited for describing SPARQL extension functions. This approach aligns with Semantic Web best practices and mirrors GeoSPARQL's extension function set defined using FnO[3]. By adopting this method, the Time Functions can be consistently documented, discovered, and potentially reused by other tools and specifications.

2.1. Motivation And Design

Time Functions treats all temporal literals as time intervals, interpreted as the range of time they could represent based on their earliest and latest possible interpretations. For instance, a literal "2025-08"^^xsd:gYearMonth spans from August 1st ("2025-08-01T00:00:00-14:00"^^xsd:dateTime) to August 31st ("2025-08-31T23:59:59+14:00"^^xsd:dateTime). As discussed in Section 1, the W3C XML Schema Recommendation [4] is followed to interpret floating date-time values as intervals that encompass all possible time zone offsets, using the full ±14:00 hour range, rather than defaulting to a specific time zone like UTC.

By shifting from point-based to interval-based reasoning, these functions enable meaningful comparisons across data types, handle ambiguities introduced by missing time zones or precision, and make temporal filtering in SPARQL more reliable and consistent.

2.2. Overview Of Time Functions

The current Time Functions include five core functions:

- **tfn:periodMinInclusive(?timeLiteral)**: Returns the inclusive lower bound of the time period represented by the given temporal literal, as an xsd:dateTime. For example, for "2025-08"^^xsd:gYearMonth, it returns "2025-08-01T00:00:00.000+14:00"^^xsd:dateTime.
- **tfn:periodMaxInclusive(?timeLiteral)**: Returns the inclusive upper bound of the time period represented by the given temporal literal. For the same example, it returns "2025-08-31T23:59:59.999-14:00"^^xsd:dateTime.
- **tfn:periodMinExclusive(?timeLiteral)**: Returns the exclusive lower bound of the time period. This is particularly useful for defining open-ended or non-overlapping intervals in filtering logic. For the same example, it returns "2025-07-31T23:59:59.999+14:00"^^xsd:dateTime.
- **tfn:periodMaxExclusive(?timeLiteral)**: Returns the exclusive upper bound of the time period. For the same example, it returns "2025-09-01T00:00:00.000-14:00"^^xsd:dateTime.
- **tfn:bindDefaultTimezone(?timeLiteral, ?timeZone)**: For a given floating time literal, this function returns a new literal of the same type with the specified time zone bound. If the literal already includes a time zone, no default time zone needs to be bound to it, and it is returned unchanged. For

the same example, it returns "2025-08+02:00"^^xsd:gYearMonth when bound to the +02:00 time zone. The function aligns with the approach proposed in Working with Time and Timezones [3], which recommends interpreting floating times as UTC by default. However, it also supports more flexible, context-specific interpretations by allowing users to explicitly specify an alternative time zone. Caution is advised when applying this function across data from heterogeneous sources, as there is no universally correct default time zone. Nevertheless, the function enables binding a default time zone retrieved dynamically from the dataset itself. For example, a SERVICE clause may be used to ensure that the time zone is sourced from the same dataset or endpoint as the time literal, preserving consistency within federated queries. An example of such a query is included in the demo application, showcasing how default time zones can be retrieved and applied dynamically based on the queried data source.

2.3. Use Cases

The Time Functions are applicable in a wide range of practical scenarios. When comparing dates of different data types, such as matching an xsd:dateTime with an xsd:date, the functions allow both values to be interpreted as intervals and compared accordingly. This is particularly useful in knowledge graphs where schema constraints are loose and data often lacks uniform temporal granularity.

Floating time literals pose challenges in distributed and heterogeneous datasets. Without a defined time zone, their interpretation is ambiguous, leading to incorrect comparisons or missed matches. Time Functions make it possible to consistently bind a default time zone where appropriate or interpret the literal as an interval that spans all possible time zones, depending on the application's needs.

These functions also support more advanced temporal logic, such as detecting overlaps between time periods, validating temporal boundaries, and improving sorting behavior. In knowledge graphs like Wikidata, where date precision is user-defined but not preserved in SPARQL query results, Time Functions allow users to reconstruct and reason about the intended temporal scope of such data. More generally, they provide a principled foundation for integrating diverse temporal representations in SPARQL queries, enabling more accurate and expressive querying of temporal knowledge.

3. Demo

To illustrate the practical utility of the Time Functions, we developed an online demo application, available at https://smessie.github.io/TimeFunctions-SPARQL-Editor/. The application is a lightweight SPARQL query editor that allows users to experiment interactively with the Time Functions described in Section 2.

Users can write and execute SPARQL queries, demonstrating how the Time Functions can address limitations in standard SPARQL when comparing partial or floating time literals. Under the hood, the demo runs on the Comunica query engine [7], which has been extended[4] to support these custom SPARQL extension functions. The query interface itself is powered by the open-source YASQE editor[5], which offers features such as syntax highlighting and autocompletion.

The application includes example queries that highlight common pitfalls when working with time literals in SPARQL, as well as how the Time Functions can be used to resolve them. Fig. 1 shows a screenshot of the demo with a query that compares an xsd:date to an xsd:dateTime. Since SPARQL does not support comparisons between these types, the query yields no results—even though the date components are logically comparable. In Fig. 2, the same query is rewritten using the Time Functions to map both literals to their corresponding time intervals. This allows for a meaningful comparison, and

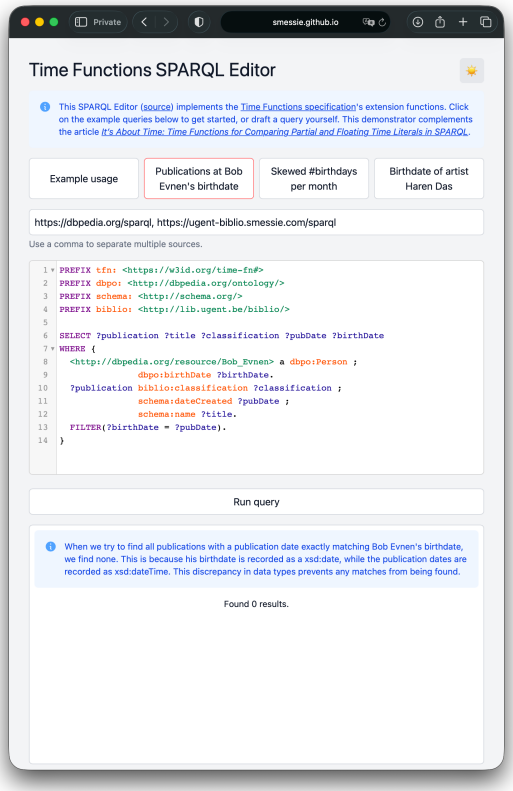

Fig. 1: Demo app with default query comparing an xsd:date to an xsd:dateTime yielding no results.

Fig. 2: Demo app with query using Time Functions to compare an xsd:date to an xsd:dateTime, yielding expected results.

the query returns the expected result.

4. Conclusion

In this paper, we have highlighted an important gap in the current processing of SPARQL queries dealing with temporal data. This gap includes the inability to directly compare logically comparable data types such as xsd:date and xsd:dateTime, as well as potential ambiguity when dealing with floating time literals, especially in federated or heterogeneous data settings. Moreover, current data modeling practices further complicate temporal reasoning: for instance, Wikidata represents imprecise time periods such as “27th century BCE” using a single xsd:dateTime literal (e.g., “-2650-01-01T00:00:00Z”), thereby flattening a broad time range into a misleading instant. These challenges highlight a broader need for improved temporal data publishing practices: data should be expressed using accurate data types that reflect temporal granularity, and time zones should be made explicit wherever possible to avoid implicit and potentially conflicting assumptions.

To address these limitations, we proposed a set of SPARQL extension functions, **Time Functions**,

which reinterpret time literals as time intervals, enabling consistent and type-agnostic comparisons. These functions support partial dates, floating times, and even cross-type comparisons by interpreting each time literal as a bounded interval, which can then be compared meaningfully using SPARQL. Additionally, a function for binding a default time zone enables explicit handling of floating times in a configurable way. These functions are formally specified using FnO, following established practices such as those in GeoSPARQL, and are implemented in a demo application using the Comunica query engine. Our demonstrator provides an interactive environment to explore the functions in action and illustrates how they resolve common pitfalls in temporal querying.

While the functions offer an immediate and practical solution, they are also intended to contribute to the broader discussion around improving temporal reasoning in the upcoming SPARQL 1.2 and beyond. By surfacing the challenges and demonstrating a viable path forward, we hope this work sparks further standardization efforts and community feedback.

Endnotes

1. https://github.com/w3c/sparql-query/issues/116 ↩
2. https://github.com/w3c/sparql-dev/blob/main/SEP/SEP-0002/sep-0002.md ↩
3. https://github.com/opengeospatial/ogc-geosparql/blob/master/vocabularies/functions.ttl ↩
4. https://github.com/smessie/TimeFunctions-SPARQL-Editor/blob/main/src/assets/queryWorker.ts ↩
5. https://docs.triply.cc/yasgui-api/#yasqe ↩

Declaration On Generative AI

During the preparation of this work, the author(s) used ChatGPT, DeepL in order to: Grammar and spelling check, Paraphrase and reword. After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] Cyganiak, R., Wood, D., Lanthaler, M.: RDF 1.1 Concepts and Abstract Syntax. https://www.w3.org/TR/2025/WD-rdf12-concepts-20250704/ (2014).

[2] Harris, S., Seaborne, A.: SPARQL 1.1 Query Language. https://www.w3.org/TR/2013/REC-sparql11-query-20130321/#OperatorMapping (2013).

[3] Phillips, A.: Working with Time and Timezones. https://www.w3.org/TR/timezone/ (2024).

[4] V. Biron, P., Malhotra, A.: XML Schema Part 2: Datatypes Second Edition. https://www.w3.org/TR/2004/REC-xmlschema-2-20041028/ (2004).

[5] Soulard, T., Saïs Fatiha, Raad, J.: Explainable Temporal Fact Validation Through Constraints Discovery in Knowledge Graphs. In: European Semantic Web Conference. pp. 227–244. Springer (2025).

[6] De Meester, B., Dimou, A., Verborgh, R., Mannens, E.: An ontology to semantically declare and describe functions. In: European Semantic Web Conference. pp. 46–49. Springer (2016).

[7] Taelman, R., Van Herwegen, J., Vander Sande, M., Verborgh, R.: Comunica: a Modular SPARQL Query Engine for the Web. In: Proceedings of the 17th International Semantic Web Conference (2018).