IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 25 September 2025, accepted 14 October 2025, date of publication 20 October 2025, date of current version 31 October 2025.

*Digital Object Identifier 10.1109/ACCESS.2025.3623639*

RESEARCH ARTICLE

# RDF Fusion: An Extensible SPARQL Engine for Hybrid Data Models

**TOBIAS SCHWARZINGER[1], (Graduate Student Member, IEEE), MAX THOMA[1], THOMAS PREINDL[1], MARTIN KJÄER[1], VALENTIN PHILIPP JUST[1], AND GERNOT STEINDL[2], (Member, IEEE)**

[1]TU Wien, 1040 Vienna, Austria
[2]University of Applied Sciences Burgenland, 7000 Eisenstadt, Austria

Corresponding author: Tobias Schwarzinger (tobias.schwarzinger@tuwien.ac.at)

This work was supported in part by Austrian Research Promotion Agency Österreichische Forschungsförderungsgesellschaft (FFG) through Research Project "USEFLEDS–Unleashing Sector-Coupling Flexibility by Means of an Energy Data Space" under Contract 905128, and in part by the Technical University (TU) Wien Bibliothek for financial support through its Open Access Funding Program.

**ABSTRACT** The Internet of Things (IoT) generates vast streams of sensor data that often require enrichment with background knowledge about the system and domain. Although such data can be represented as graphs, purely graph-based models struggle with the temporal aspects of sensor observations, motivating hybrid approaches that integrate graphs with time series data. This creates a need for query engines that can handle both types of data within a single system. In the Semantic Web community, this drives demand for SPARQL engines that are flexible enough to support time series data and efficient for analytical workloads. Existing engines fall short as only some row-based systems focus on extensibility but perform poorly in time series analytics, while columnar systems could offer better analytical performance but lack the necessary extensibility. To address this gap, we present RDF Fusion, an extensible SPARQL engine built on Apache DataFusion, a modular columnar engine optimized for analytical workloads. RDF Fusion uses specialized encodings to represent the dynamic nature of RDF terms within the statically typed data model of DataFusion. These encodings enable efficient SPARQL query execution while preserving the extensibility to experiment with custom operators, optimizations, and hybrid time series support. Our evaluation shows that RDF Fusion complies with SPARQL 1.1 and provides competitive performance in analytical workloads. As an open-source system, it offers a solid foundation for research on hybrid data models in the IoT.

**INDEX TERMS** Semantic Web, SPARQL, hybrid data model, Internet of Things, analytics.

## I. INTRODUCTION

Historically, database systems were designed as tightly integrated and highly optimized solutions that followed a "one-size-fits-all" paradigm [1]. However, it became increasingly clear that such general-purpose systems could not deliver optimal performance and ergonomic queries for all workloads [1], leading to the rise of specialized databases optimized for domain-specific requirements [2]. Prominent examples include graph databases and time series databases, which are the focus of this work.

The Internet of Things (IoT) is a prime example of a domain where both data models are widely used. Graph data models, such as the Resource Description Framework (RDF) [3], capture metadata, system and domain knowledge, and relationships within a system [4]. In contrast, IoT devices continuously generate high-frequency sensor readings, which are naturally represented as time series [5]. Neither model alone is sufficient: Encoding time series as RDF triples causes storage and query overhead [6], while time series databases

The associate editor coordinating the review of this manuscript and approving it for publication was Francisco J. Garcia-Penalvo.

lack mechanisms to represent rich contextual information about devices [7].

To overcome these limitations, hybrid data models that integrate graphs and time series have recently been proposed [8], [9], [10]. However, currently there is no platform that facilitates rapid prototyping in this context. Researchers who want to explore custom query languages must either rely on inefficient prototypes or reimplement large parts of a query engine themselves. What is missing is an extensible query engine that natively supports queries that span both graph and time series data.

In parallel, modular query engines have emerged from the trend of composable data management systems [11], offering reusable building blocks for query processing and lowering the barrier to developing domain-specific engines. Prominent examples include Apache DataFusion [12] and Meta's Velox [13]. However, their primary focus remains on relational data. Consequently, researchers who are looking to combine graph and time series data must still implement a complete graph query language, such as SPARQL [14], on top of these systems. Nevertheless, because SPARQL can be translated into relational algebra [15], these systems enable platforms that combine SPARQL query operators with those of other data models.

To address this gap, we present *RDF Fusion*, an extensible SPARQL engine built on Apache DataFusion. RDF Fusion leverages DataFusion's extension points to implement SPARQL and introduces multiple RDF term encodings to represent the dynamic nature of RDF data within DataFusion's type system. By building on a modular query engine, RDF Fusion enables researchers to explore hybrid data models, where SPARQL query operators can be combined with time series and relational operators within a single system. The developed engine is open-source on GitHub [16] and our evaluation snapshot is available on Zenodo [17].

The remainder of the paper is structured as follows. Section II introduces the necessary background, while Section III elaborates on DataFusion in the context of hybrid data models. Then, Section IV outlines the requirements for a query engine for hybrid data models. Section V describes how dynamic RDF terms are mapped to DataFusion's internal data structures, which forms the basis for the discussion of RDF Fusion in Section VI. Section VII provides an evaluation, and Section VIII discusses the results and some design decisions. Finally, Section IX reviews related work, and Section X concludes the paper.

## II. BACKGROUND

This section provides a brief background on RDF, SPARQL, row-based and columnar query engines, and Apache DataFusion. Building on version 1.1 of RDF and SPARQL, this work does not introduce formal notation, since a detailed discussion of the semantics of SPARQL lies outside its scope. As this paper focuses on the graph processing aspect of hybrid data models, this work will not discuss those approaches in detail. We refer the reader to the original proposals for further information [8], [9], [10].

**TABLE 1. Used namespace prefixes.**

| Prefix | IRI |
|---|---|
| `rdfs:` | `<http://www.w3.org/2000/01/rdf-schema#>` |
| `xsd:` | `<http://www.w3.org/2001/XMLSchema#>` |
| `ex:` | `<http://example.org/>` |

### A. THE RESOURCE DESCRIPTION FRAMEWORK

The central data element in RDF is a *triple*, consisting of a subject, a predicate, and an object. Each element of a triple has a different domain: the subject must be an Internationalized Resource Identifier (IRI) or a blank node, the predicate must be an IRI, and the object may be an IRI, a blank node, or a literal. While IRIs are global identifiers, blank nodes can be thought of as local identifiers for resources without an IRI. Literals consist of a lexical value and, optionally, a datatype or a language tag.

RDF syntaxes allow serializing and desrializing of RDF triples. To improve readability and reduce verbosity, most RDF syntaxes allow defining namespace prefixes. These prefixes (e.g., `rdfs:`) allow for abbreviating IRIs. Table 1 lists the namespace prefixes used in this work.

An RDF graph is a set of triples that represents a collection of facts. Listing 1 shows an example RDF graph in Turtle syntax [18] that provides information on the Apache Arrow project, the in-memory format that underpins DataFusion's query execution. The relative IRI `<Apache>` is resolved using the `BASE` IRI. In addition to IRIs, the graph contains multiple literals with different data types.

```
BASE <http://example.org/>

<Apache> <hasTopLevelProject> <Arrow> .
<Arrow> rdfs:label "Apache Arrow"^^xsd:string ;
    <version> "20.0"^^xsd:decimal ;
    <firstRelease> "2016-10-10"^^xsd:date .
```

**LISTING 1. Example RDF graph.**

An important concept for this work is a *typed value* [14]. For example, consider two RDF literals: `"1"^^xsd::integer` and `"01"^^xsd::integer`. While they are distinct RDF terms, they share the same typed value, namely the integer 1. This is significant because some SPARQL operations, such as joins, must distinguish between these terms, while others, such as arithmetic operations, must not. To accommodate such scenarios, RDF Fusion allows for multiple encodings of RDF terms, which are presented in Section V.

### B. THE SPARQL QUERY LANGUAGE

SPARQL is the standard query language for RDF data. The central elements of SPARQL queries are *triple patterns*, which are similar to RDF triples, but can contain *variables*.

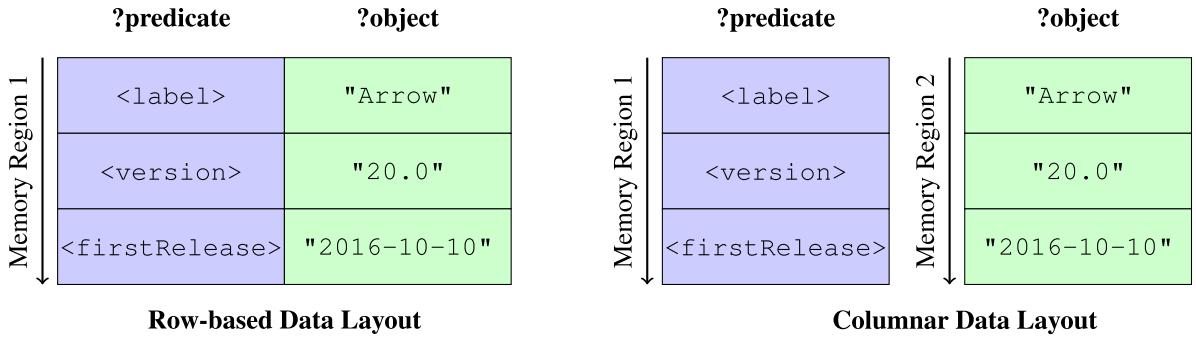


**FIGURE 1.** Difference between row-based and columnar data layout shown on simplified results for the query in Listing 2 and the RDF graph in Listing 1.

**TABLE 2.** Result of evaluating the query in Listing 2 against the RDF graph in Listing 1.

| ?predicate | ?object |
|---|---|
| rdfs:label | "Apache Arrow" |
| ex:version | "20.0"^^xsd:decimal |
| ex:firstRelease | "2016-10-10"^^xsd:date |

In principle, the query engine's task is to find all possible solutions by binding these variables to RDF terms in the graph, as defined in the SPARQL specification [14].

Listing 2 shows a basic SPARQL query that can be evaluated against the RDF graph from Listing 1. This query searches for all triples with the subject <Arrow> and returns their predicates and objects. The results of this query are shown in Table 2. Note that the triple <Apache> <hasTopLevelProject> <Arrow> does not match the query, as it has a different subject.

```
BASE <http://example.org/>

SELECT ?predicate ?object
{
    <Arrow> ?predicate ?object
}
```

**LISTING 2.** A SPARQL query that returns predicates and objects for triples with the subject <Arrow>.

Beyond these fundamentals, SPARQL also supports named graphs and more complex operations such as filtering, grouping, and aggregation. These features, while powerful, contribute to the complexity of building a SPARQL engine. However, elaborating on them in detail is beyond the scope of this work. For detailed discussions, we refer the reader to the SPARQL specification [14].

### C. ROW-BASED AND COLUMNAR QUERY ENGINES

The key distinction between row-based and columnar query engines lies in how intermediate query results are stored and processed in memory. As shown in Fig. 1, for SPARQL, row-based engines store each complete solution contiguously in memory, while columnar engines store all values of a single variable contiguously in memory.

Row-based layouts usually excel in Online Transaction Processing (OLTP) workloads, which typically involve a high volume of small, write-heavy operations. In contrast, columnar layouts usually excel in Online Analytical Processing (OLAP) workloads, where queries may read large amounts of data to perform complex analysis [1]. As a result, for analytical queries, a columnar query engine can provide significant performance benefits when realizing hybrid data models. Although most of the work on columnar engines has focused on less dynamic query languages, recent research suggests that SPARQL engines can also benefit from this execution model [19].

### D. APACHE DATAFUSION

Apache Arrow[1] defines an in-memory format designed for high-performance analytical data processing. The specification includes various data types and their corresponding memory layouts. In Arrow terminology, a column of data is called an *array*, which is backed by one or more contiguous memory regions called *buffers*. For example, an `Int64Array` consists of a validity buffer (for null values) and a value buffer (for integers).

Apache DataFusion [12] is an extensible query engine that uses Arrow as its internal data representation. Implemented in Rust,[2] a modern system programming language, DataFusion can be embedded directly into other applications, such as custom research prototypes. Fig. 2 presents a simplified architecture of its query processing pipeline, highlighting key extension points that allow for custom functionality. This work briefly introduces the depicted components. We refer the reader to [12] for an in-depth discussion.

Processing begins in a *front end* (usually an SQL [20] dialect), which parses and validates the query, generating a *logical plan*. This plan is a tree of relational operators and expressions that represents the query's semantics (e.g., a `Join` or a `Filter`). This logical plan undergoes a series of

[1]https://arrow.apache.org/
[2]https://www.rust-lang.org/

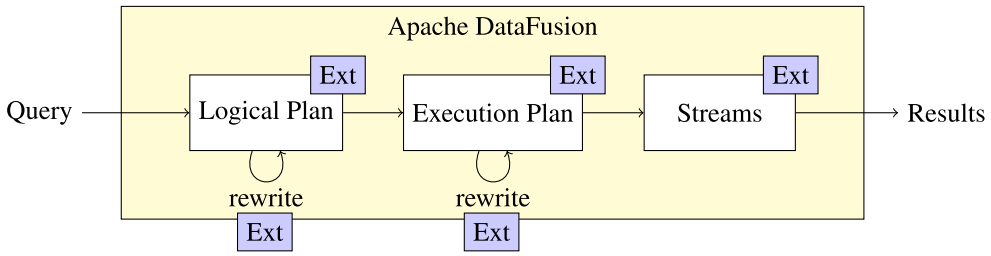


**FIGURE 2.** Simplified architecture of query processing in DataFusion. "Ext" marks extension points. These can be custom operators or rewriting rules.

*rewriting steps* for optimization and transformation. Both, the set of logical plan nodes and rewriting steps can be extended by the user.

The optimized logical plan is then translated into an *execution plan*. Contrary to the logical plan, the execution plan nodes have a concrete algorithm to execute a given operation. For example, a `Join` operator in the logical plan might be translated into a `HashJoin` or a `NestedLoopJoin`. Similarly to the logical plan, the execution plan undergoes a series of rewriting steps that optimize or transform it. Again, both can be extended by the user.

Finally, the execution plan is translated into a set of *streams*, which produces one or more *record batches*. Each record batch consists of one or more fields, backed by Arrow arrays. For example, the query in Listing 2 would result in a batch with two fields: one for the predicate and one for the object, as shown in the columnar data layout in Fig. 1. The record batches emitted from the root operator are the final result of the query.

A major strength of DataFusion is its extensibility, which supports user-defined functions, custom logical and physical nodes, and custom optimizer rules. This is where Arrow's standardized memory model is particularly beneficial: As long as a component reads and writes Arrow record batches, it can be integrated with DataFusion. RDF Fusion takes full advantage of this extensibility to implement a complete SPARQL engine within DataFusion.

## III. HYBRID DATA MODELS AND DATAFUSION

One of the core goals of RDF Fusion is to support hybrid data models that integrate graph and time series workloads. This requires a query engine capable of handling both efficiently. The key question is whether DataFusion can support both domains in a single query.

To elaborate on this, we start by discussing the support of graph and time series data independently. On the graph side, the existence of RDF Fusion demonstrates the feasibility of processing RDF graphs. Section V details our approach to this problem. On the time series side, several systems (e.g., InfluxDB[3]) have already built query engines on top of DataFusion. Arrow supports this, for example, by modeling a time series as two arrays: timestamps and values. While common time series database features like gap filling are currently not built into DataFusion, the existing systems mentioned before show that they can be supported.

The next challenge, therefore, is not merely handling graphs and time series data separately but integrating them within a single query. To illustrate this, consider an example that computes the average temperature for each Heating, Ventilation, and Air Conditioning (HVAC) zone in a building. Information about which sensor belongs to which HVAC zone is stored in the graph, while the corresponding time series data are maintained separately. Figure 3 presents a simplified query plan that demonstrates how a hybrid data model can support this type of use case.

1) A SPARQL query retrieves all zones and sensor pairs from the graph.
2) For each sensor, its time series are fetched directly in a format optimized for time series, avoiding conversion to RDF and the associated overhead.
3) Finally, the data are aggregated by zone, and the average temperature is computed.

DataFusion enables hybrid execution by leveraging Arrow's support for advanced data types and the ability for each column to have a distinct type. Consequently, parts of a result may originate from the graph domain, while others come from the time series domain. As long as the data can be mapped onto Arrow arrays, they can be processed together seamlessly. For instance, an operator can take an RDF-encoded column `?sensor` as input and produce both `?sensor` and `?temp`, where `?temp` corresponds to the temperature time series of the sensor. In the example, the column `?temp` could be materialized using Arrow's Variable-size List Layout.[4]

In summary, DataFusion can support both graph and time series data models in isolation and in combination. Other query engines that are purpose-built for SPARQL workloads may need to rely on workarounds to perform similar computations, as they typically restrict intermediate

[3]https://www.influxdata.com/

[4]https://arrow.apache.org/docs/format/Columnar.html#variable-size-list-layout

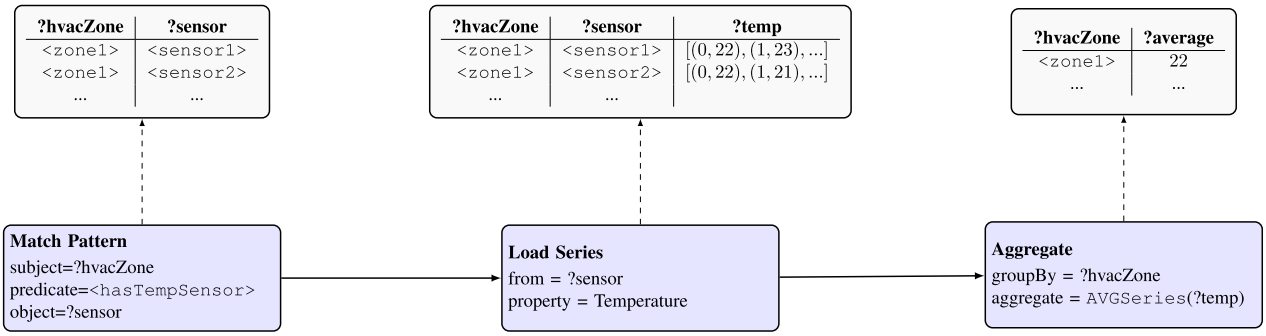


**FIGURE 3.** A simplified query plan that demonstrates the integration of graph and time series data. The query first accesses the graph data, then loads the corresponding time series, and finally computes an aggregate of the time series by grouping on the corresponding graph elements. The tables connected above show examples of intermediate query results.

results to data types prevalent in SPARQL. Consequently, extending these intermediate results to accommodate time series data may be challenging.

## IV. REQUIREMENTS

Based on the challenges identified in Section I, we define a set of key requirements for the graph processing part of a query engine that supports a hybrid data model. RDF Fusion is designed to meet these requirements, serving as a foundation for future research in this area. See Section VII for an evaluation of these requirements.

- **R1: SPARQL Conformance**. The engine must correctly interpret and execute the SPARQL 1.1 query language. This is essential to provide users with a formal specification that defines the expected results of queries. Furthermore, it allows performance comparisons with other SPARQL implementations and the use of RDF Fusion as a standalone SPARQL engine.
- **R2: Extensibility**. As the prototypes that motivate this work often require custom functionality, the engine must have a modular architecture that supports extensions. For example, in hybrid data models, it includes novel operators that bridge the gap between graphs and time series data (e.g., those proposed in [10]).
- **R3 SPARQL Performance**: The engine must perform reasonably well on graph workloads. However, given the significant engineering effort required, our initial mission is not to reimplement and fine-tune all SPARQL-specific optimizations found in other engines.

## V. RDF TERM ENCODINGS

Apache Arrow serves as DataFusion's central memory format for query processing. Consequently, any SPARQL implementation built on a framework like DataFusion must represent RDF terms using Arrow arrays. This introduces a significant challenge, as in SPARQL, each variable can be bound to arbitrary RDF terms within subsequent solutions. For example, the first solution could bind the variable *?object* to a string, while the second binds it to a decimal. This behavior contrasts with the conventional relational model, where each column is assigned a well-defined, fine-grained domain (e.g., integers). To bridge this gap, RDF Fusion must adopt a strategy capable of encoding the dynamic nature of SPARQL solutions within Arrow arrays.

### A. PLAIN TERM ENCODING

The *Plain Term Encoding* uses an Arrow "Struct Layout"[5] to represent RDF terms in their full lexical form. This encoding preserves all syntactic distinctions, which is crucial for operations that depend on term identity rather than just value equivalence.

As shown in Fig. 4, a term is composed of several fields. The *Term Type* field indicates whether the term is an IRI, a blank node, or a literal. A mandatory *Value* field stores the term's string value, while literals also include an optional *Data Type* and *Language Tag* field. The validity buffer is used to represent unbound variables, which are treated as null values within the array.

As the encoding retains the term identity, it can be used to join SPARQL solutions. Furthermore, evaluating SPARQL functions that do not operate on typed values can be very efficient. For example, consider the `STR` function that returns the lexical form of an RDF term. In the Plain Term Encoding, this is simply copying the reference counted pointer of the Value field, copying the null buffer, and creating the Data Type array. The actual values of the bound terms do not have to be inspected.

In addition, consider the `BOUND` function, which determines whether a given variable is bound in a solution. Its implementation only needs to examine the validity buffer of the column that represents the variable, checking whether an entry is valid to produce an output. All other buffers can be ignored for this operation. Furthermore, such operations can be efficiently vectorized using Single Instruction, Multiple Data (SIMD) instructions, which are typically found in modern hardware.

[5]https://arrow.apache.org/docs/format/Columnar.html#struct-layout

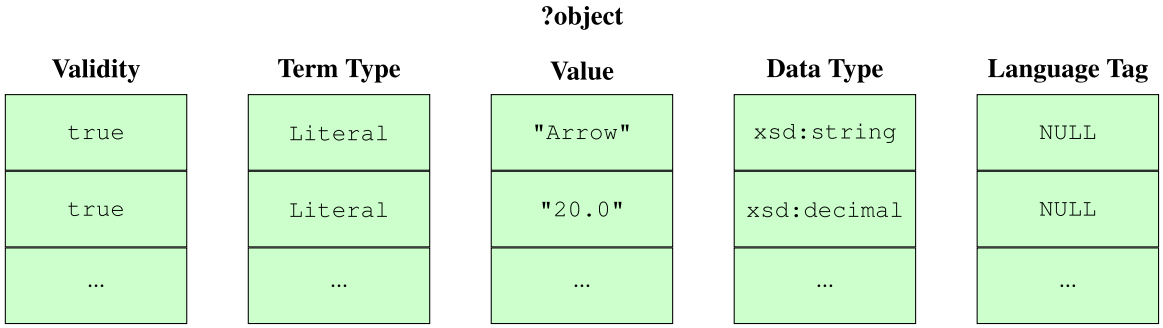


**FIGURE 4.** Simplified data layout for the `?object` variable of the example query (Listing 2) in the Plain Term Encoding.

### B. OBJECTID ENCODING

The *ObjectID Encoding* represents RDF terms using a mapping from a term to a fixed-size identifier called *object ID*. For example, a triple (`<Arrow>`, `rdfs:label`, `"Apache Arrow"`) could be encoded as (1, 2, 3), where each number is an object ID corresponding to the original term. RDF Fusion currently uses 32-bit unsigned integers for these identifiers. As a result, a single `UInt32` array is necessary to encode the RDF terms.

A key advantage of this encoding is its efficiency for operations that rely on term identity. RDF terms are identical if and only if they have the same object ID. As a result, operations like joins can be performed directly on these fixed-length identifiers, which is significantly faster than comparing variable-length strings. The goal is to materialize the original RDF terms as late as possible, sometimes even avoiding the decoding of entire variables (e.g., if a variable is used for grouping but is not part of the final result).

However, a drawback is that, for some queries, the overhead of looking up the original term outweighs the benefits of faster joins. This can be particularly problematic when a triple store contains many small literals [21]. While more sophisticated strategies exist to handle such cases by storing small values inline, these are currently not implemented in the RDF Fusion prototype.

### C. TYPED VALUE ENCODING

The encodings presented above materialize the RDF terms directly. However, for certain operations (e.g., arithmetic), the SPARQL engine requires access to the typed value of an RDF literal, as discussed in Section II-A. In the Plain Term Encoding, this value can only be obtained by parsing the string representation according to its data type and then using the parsed value in computations. To avoid repeated parsing of typed values, we introduce an encoding that represents typed values directly.

The *Typed Value Encoding* employs a Dense Union Layout[6] to represent RDF terms, using specialized child arrays for each supported typed value. Unlike the Plain Term Encoding, which stores all values as strings, this encoding can utilize native Arrow types (e.g., Int64 or Float32) for storing typed literals. This is critical for operations that build on the typed value of an RDF terms, such as numerical operations and comparisons.

Figure 5 shows a simplified schema for this encoding. A *Type Id* buffer indicates which child array holds the value. For example, a value with type id 1 would be stored in `Int64Array`, and a value with type id 2 would be in the `Float32Array`. By storing values of the same type contiguously, this layout makes them amenable to vectorized operations. As the union layout delegates null handling to the child arrays, there is also a `NullArray` that handles unbound variables. Literals with an invalid lexical form (e.g., `"str"^^xsd:integer`) are also encoded as null.

The set of child arrays is based on the data types defined in the SPARQL standard, natively supporting IRIs, blank nodes, and known literal types such as `xsd:integer` and `xsd:float`. For typed literals without a natively supported data type, an *Other Literal* array stores their lexical representation. In this encoding, a variable's data can thus be either homogeneous (i.e., a single large array) or heterogeneous (i.e., multiple smaller arrays for different data types). As shown in Figure 5, the *?object* variable from the query in Listing 2 would result in a heterogeneous array, while the *?predicate* variable would be an homogeneous array, as the variable is only bound to IRIs.

This encoding is crucial for evaluating any SPARQL query that relies on the typed value of a term. To demonstrate, consider evaluating the SPARQL expression `xsd:float(?a + ?b)` only using the Plain Term Encoding. The query engine would first need to parse all literals, then perform the addition, and lastly encode the result back into the Plain Term Encoding, as results of operators are also Arrow arrays in DataFusion. Then, for the `xsd:float` cast, these values would have to be parsed again, converted to a float, and re-encoded. This inefficient process of repeated parsing and encoding is eliminated in the Typed Value Encoding, as the data values are stored directly in their native formats.

[6] https://arrow.apache.org/docs/format/Columnar.html#union-layout

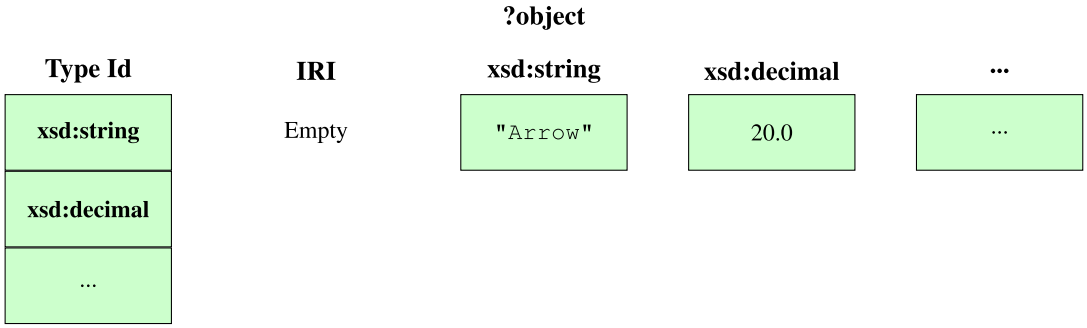


**FIGURE 5.** Simplified data layout for the `?object` variable from Listing 2 in the Typed Value Encoding. In this encoding, the distribution of the values between the arrays depends on their data type. Depending on the homogeneity of the bound terms, this can lead to many smaller arrays.

### D. CONVERTING BETWEEN ENCODINGS

RDF Fusion provides a set of User-Defined Functions (UDFs) for converting between its different encodings. These functions can then be used within query plans. Conversions involving the ObjectID Encoding have direct access to the internal mapping between object IDs and the corresponding RDF terms. RDF Fusion also maintains a separate mapping from object IDs to their typed values, enabling efficient conversion from the ObjectID Encoding to the Typed Value Encoding. Without this shortcut, the query engine would need to first convert to the Plain Term Encoding and then parse literals with known data types.

Currently, the query planner introduces encoding changes on demand. For example, when evaluating an expression such as `?a * -1`, the planner automatically inserts a conversion UDF if `?a` is not already in the Typed Value Encoding. This reactive strategy is local and simplifies query planning. However, it can result in repeated conversions of the same column if different nodes in the query plan require different encodings. These efforts can be costly, particularly when parsing or the object ID mapping is involved. We plan to explore more global strategies for managing encoding changes in future work.

## VI. RDF FUSION

RDF Fusion consists of several key components. This includes the encodings introduced in Section V. On top of these, we provide a set of functions, including encoding transformations, the SPARQL scalar and aggregate functions, and various utility functions required to interface with DataFusion (e.g., converting native booleans into RDF terms). RDF Fusion also introduces a number of custom logical plan nodes to represent parts of the SPARQL algebra, rewriting steps, and some custom physical operators and streams. Most custom logical plan nodes are lowered to existing DataFusion operators, allowing us to benefit from its built-in optimizations and implementations. For example, the `Extend` graph pattern of the SPARQL algebra is translated into a `Projection` in DataFusion. The most important physical operator is the quad pattern operator, described in Section VI-A.

RDF Fusion can be used in multiple ways. In this work, we focus on two usage scenarios: as a standalone SPARQL engine and as an extension to DataFusion. The former offers a ready-to-use system for working with RDF data, while the latter provides maximum extensibility for rapidly prototyping SPARQL extensions. In the following sections, we briefly summarize key aspects of RDF Fusion's implementation. This provides readers with the necessary context to assess the performance evaluation and to understand claims regarding potential future performance improvements.

### A. QUAD STORAGE

A key component of any SPARQL engine is how quads are stored and retrieved. RDF Fusion currently uses an in-memory implementation of the classical quad index approach.

The primary aim is to index quads in various permutations to speed up matching graph patterns. Here, we illustrate this concept using triple indexes, as the underlying principles remain the same. For example, the graph pattern `<Arrow> ?p ?o` benefits from an index that starts with the subject, since all triples not beginning with `<Arrow>` can be ignored. In contrast, a pattern such as `?s <label> ?o` does not benefit from a subject-first index, as every subject must be checked if the `<label>` predicate follows. Instead, a predicate-first index could improve performance. For more details, see [21].

RDF Fusion currently indexes `GSPO`, `GPOS`, and `GOPS`, where `G` denotes the graph name. Each index stores object IDs and is implemented as four sorted columns. To describe our in-memory structures, we adopt terminology from Apache Parquet,[7] an open file format based on similar principles. The quads are stored in a sorted list according to the permutation defined by the index. This list is partitioned into multiple *row groups*, whose size is aligned with DataFusion's `batch_size` configuration. Within a row group, values of

[7] https://parquet.apache.org/

the quad components (e.g., subjects) are stored contiguously in a columnar layout. Each such contiguous slice of a column is referred to as a *column chunk*.

The search on the sorted index supports only a few primitive operations, such as equality and range checks. Essentially, these are operations that select a contiguous slice of the index. The query evaluation then proceeds in three steps:

1) **Search space pruning**: Using the index, the system quickly identifies which row groups might contain quads that match the bound elements of the triple pattern. If a triple pattern element is unbound (i.e., a variable), pruning stops at that point. The stream operator remembers all pointers to the relevant row groups. Ideally, this step prunes the search space to just a few row groups.
2) **Boundary slicing**: The first and last matching row groups may include quads outside the desired range if the range does not align perfectly with row group boundaries. These "extra" quads can be trimmed, which may allow some filters to become fully applied at this stage. In other words, the engine can guarantee that every element in the remaining row group slices matches the filter. In the best case, this applies to all filters.
3) **Filtering and projection**: For each remaining row group, any filters that were not handled in previous steps are applied. The query engine projects the quad components to the variables in the pattern and collects the results. If all filters were handled earlier, this step can simply clone the pointer with the relevant column chunks without further computation. The resulting quads are then passed to the next stage of query evaluation.

We expect this strategy to extend naturally to on-disk storage using columnar formats, since similar pruning techniques are applied in DataFusion when scanning Parquet files. In future work, we also plan to explore compressing the first three columns of the index and operating directly on the compressed representation, as has been done in previous work on columnar query execution (e.g., [22]).

### B. OPTIMIZATIONS

DataFusion already provides a wide range of general-purpose optimizations, such as projection, filter, and limit pushdowns. Its philosophy is to provide broadly applicable optimizations while leaving domain-specific techniques to extensions built on top of it. In the case of RDF Fusion, these are SPARQL-specific optimizations. At present, RDF Fusion implements only a small set of algebraic transformations on expressions.

Even this limited set of algebraic transformations is essential in certain cases. For example, in SPARQL joins, unbound variables are considered compatible with any value. Since RDF Fusion encodes unbound variables as `NULL`, join semantics require that two terms match if at least one of them is `NULL`. In contrast, DataFusion only supports joins where `NULL` never matches or matches only other `NULL` values. To bridge this gap, we introduce an `IS_COMPATIBLE` UDF that enforces the semantics of SPARQL. However, this prevents DataFusion from using its efficient equi-join implementations (e.g., `HashJoin`), since the built-in engine cannot optimize joins involving `IS_COMPATIBLE`. To mitigate this, RDF Fusion rewrites the join to use plain equality whenever both variables are guaranteed to be bound (i.e., the columns are non-nullable). This allows DataFusion's equi-join to be applied in most cases, though it can cause issues in queries involving `OPTIONAL` graph patterns.

Note that RDF Fusion does not yet implement any sort of join ordering. Although DataFusion itself provides a general purpose algorithm, these are based on statistics that RDF Fusion's storage layer does not yet provide. Since join ordering is one of the most critical factors in SPARQL query performance [21], this remains an important area for future improvement.

### C. USING RDF FUSION AS A SPARQL ENGINE

RDF Fusion can be used as a standalone SPARQL engine. To enable this, we built on Oxigraph [23], a SPARQL implementation in Rust with a custom row-based engine. We forked Oxigraph and replaced its query engine with RDF Fusion. Although this required significant changes to the codebase, we continue to reuse key components of Oxigraph, including its test suite, RDF parsers, and serializers for SPARQL results. However, we do not aim for full compatibility, as some aspects of RDF Fusion fundamentally differ (e.g., its use of asynchronous Rust).

Fig. 6 illustrates this architecture. Programmers can interact with RDF Fusion through Oxigraph's `Store` Application Programming Interface (API) or through an HTTP endpoint. The `Store` API is a convenient wrapper around an RDF store that has methods to manipulate and query the database.

Query processing begins with the SPARQL parser, which converts the input query into a logical plan. Currently, this is a two-step process: RDF Fusion first invokes Oxigraph's SPARQL parser and then further processes the resulting SPARQL algebra. The resulting logical plan is then passed to DataFusion for execution, as described in Section II-D.

Each execution plan produces a stream of record batches. Depending on the use case, users can consume results as native Rust data structures or directly access the underlying Arrow arrays. The Stream Wrapper provides both options, allowing users to select the representation best suited to their needs.

### D. USING RDF FUSION AS AN EXTENSION

When used as an extension to DataFusion, RDF Fusion provides SPARQL support while preserving access to the full extensibility of DataFusion. To support SPARQL-specific functionality, RDF Fusion introduces new logical plan nodes

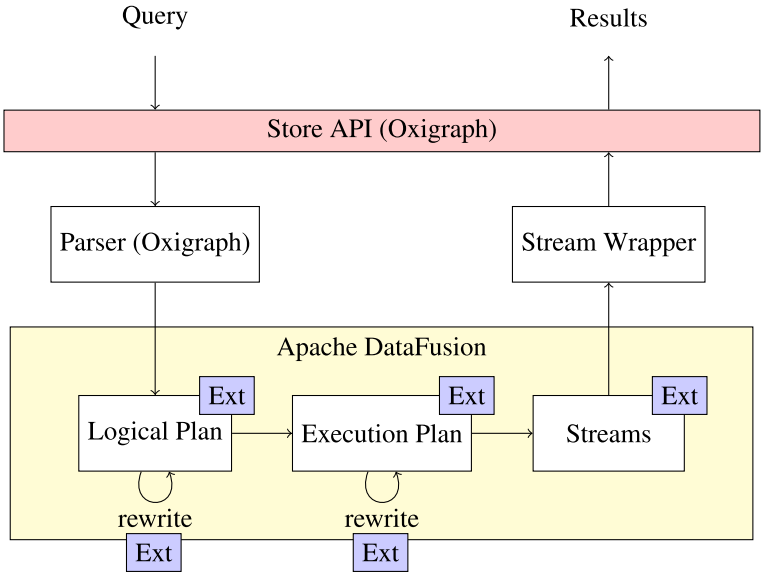


**FIGURE 6.** Using RDF Fusion via Oxigraph's `Store` API. "Ext" marks the extension points used by RDF Fusion.

which can be directly constructed by the user. These nodes can be inserted into a query's logical plan, and, once RDF Fusion's rewriting passes are registered, the query engine can execute them seamlessly.

This mode of use offers maximum extensibility, as users manage the DataFusion instance and control the construction of the initial logical plan. It enables advanced scenarios such as experimenting with alternative SPARQL dialects, mapping domain-specific query languages to SPARQL, and implementing new SPARQL operators.

RDF Fusion also exposes direct extension points for customizing SPARQL. For example, users can register their own UDFs or override built-in SPARQL functions (e.g., `STR(x)`). These features are also available when using the `Store` API, as the built-in SPARQL query processor uses this registry to look up function implementations. In addition, users could also integrate a custom storage layer into the query engine.

### E. DEVELOPMENT AND MAINTENANCE

RDF Fusion is implemented in Rust, built directly on top of DataFusion. The source code is publicly available on GitHub [16], along with source code documentation and example programs demonstrating how to use RDF Fusion. The GitHub repository serves as the main hub for issue tracking, feature requests, and contributions.

## VII. EVALUATION

The evaluation of RDF Fusion is based on the requirements presented in Section IV.

**TABLE 3.** Results of running the W3C "SPARQL Test Suite."

| Test Suite | # Tests | # Passing | # Failing | Passing (%) |
|---|---|---|---|---|
| SPARQL 1.0 | 246 | 235 | 9 | 95,53 |
| SPARQL 1.1 | 301 | 301 | 0 | 100 |
| **Total** | 547 | 536 | 9 | 97,99 |

### A. SPARQL COMPATABILITY (R1)

To evaluate the correctness of RDF Fusion's SPARQL implementation, we rely on the W3C "SPARQL Test Suite".[8] Table 3 provides an overview of the number of tests and their results. At the time of writing, approximately 98% of tests are passing.

The failing tests are due to a few outstanding issues that we plan to address in the future. For instance, RDF Fusion currently applies SPARQL 1.1 semantics by default, without allowing the user to select the SPARQL version. This causes some SPARQL 1.0 tests to fail, as some behavior differs between the two versions. Despite these known issues, we see no fundamental barriers to resolving them. For example, RDF Fusion can provide a different implementation for affected SPARQL 1.0 functions once a configuration option is available.

In addition to the W3C test suite, the test suite also covers queries from the benchmarks used in Section VII-C. All tests are executed as part of the Continuous Integration (CI) workflow. Taking into account these results, RDF Fusion already provides a solid foundation for columnar SPARQL

[8] https://github.com/w3c/rdf-tests/tree/main/sparql

query evaluation. Ongoing improvements aim to reach full compliance with the W3C recommendation.

### B. EXTENSIBILITY (R2)

A core design goal of RDF Fusion is to fully embrace the extensibility provided by DataFusion. Since RDF Fusion can itself be used as a DataFusion extension (see Section VI-D), users retain complete access to this extensibility. The architecture and its extension mechanisms have already been described in detail by Lamb et al. [12]. The following enumeration follows their listed extension points. The remainder of this section adds context from the perspective of the Semantic Web and hybrid data models.

1) **Scalar, Aggregate, and Window Functions** can be used to experiment with custom functions, as found in approaches such as GeoSPARQL [24], [25] and other enhancements provided by other popular SPARQL engines. As these functions have direct access to the Arrow arrays, they can perform on-par with built-in functions. Furthermore, window functions could support the implementation of streaming SPARQL extensions (e.g., [26]) and hybrid data models that support continuous querying (e.g., [10]) that use window operators to work with unbounded streams.
2) **Catalogs** provide an API to dynamically load and organize a list of tables into catalogs and schemas. In the context of Ontology-Based Data Access (OBDA) [27], this abstraction could be beneficial for loading virtual RDF graphs, allowing the integration of new data sources at runtime. In addition, this could also be used to expose "time series tables" that are part of hybrid data models.
3) **Data Sources** can be used to integrate new storage layers for RDF data. As long as the result conforms to one of RDF Fusion's encodings, one can use RDF Fusion's built-in SPARQL operators. As DataFusion allows for sophisticated optimizations that push the filter down to the storage layer, this presents optimization opportunities. For example, an RDF store that stores a sorted list of its literals could speed up queries by pushing down filters such as `?a > "2020-01-01"^^xsd:date` into the storage layer. For time series data, there are already production-ready data sources available.
4) The APIs in the **Execution Environment** can be used to restrict the use of resources, manage spills to disk, and access object stores. These features can help by supporting custom stream operators to work with larger-than-memory datasets (spilling) and provide easier integration with cloud storage (object stores).
5) **New Query / Language Frontends** make it possible to introduce alternative query languages. The SPARQL parser used in RDF Fusion is implemented using this mechanism and similar techniques could be used to integrate domain-specific query languages. Custom query languages for hybrid data models will make use of this extension point.
6) **Query Rewrites / Optimizer Passes** can enable SPARQL-specific optimizations. In addition, it is also possible to rewrite certain quad patterns to use custom indices. For example, using an R-tree [28] to efficiently return elements within a bounding box in GeoSPARQL [24]. Lastly, these capabilities could also be used for reasoning-aware query rewrites, such as PerfectRef [29].
7) **Relational Operators** allows users to implement custom operators that can run with the same performance as built-in operators. These are especially useful for experimenting with novel SPARQL extensions that cannot be built on top of existing operators. Furthermore, hybrid data models that require transformations between graphs and time series data could benefit from custom operators in the query plan.

**TABLE 4. Details on the experimental setup.**

| Component | Information |
|---|---|
| CPU | ADM Ryzen 9 9900X, 12 Cores, 24 Threads |
| Main Memory | 96 GiB |
| Operating System | Fedora 42, Linux Kernel 6.16.5-200 |
| Podman | 5.5.2 |
| JVM Heap Size | 80 GiB |
| DataFusion Version | 50.0 |
| DataFusion Memory Limit | 75 GiB |
| DataFusion Target Partitions | 1 |

### C. SPARQL PERFORMANCE (R3)

We evaluated the performance of RDF Fusion using the Berlin SPARQL Benchmark (BSBM) [30] benchmark suite. The evaluation compares RDF Fusion with popular open-source RDF stores that feature an in-memory storage option. All evaluated systems store the data in memory, as RDF Fusion does not yet feature an on-disk storage implementation.

Further details of the experimental setup can be found in Table 4. DataFusion is used in its default configuration, with the exception of fixing the number of target partitions and setting an upper bound on the memory usage. There is a 5 GiB difference between the JVM-based stores and DataFusion's memory limit because the latter does not account for the index size and is only on a "best-effort" basis. The SPARQL engines and benchmark drivers were running inside Podman pods that facilitate communication between the engine and the benchmark drivers. The code for running the benchmarks can be found on GitHub [31] and Zenodo [17].

The BSBM benchmark models an e-commerce platform and provides several "use cases" that target different aspects of the SPARQL query language. In this work, we use the "*Explore*" and "*Business Intelligence*" use cases to evaluate RDF Fusion. The Explore use case was executed with 24 parallel queries, while the Business Intelligence use case was executed with 12 parallel queries. We also attempted to execute the latter with 24 parallel queries. However, another

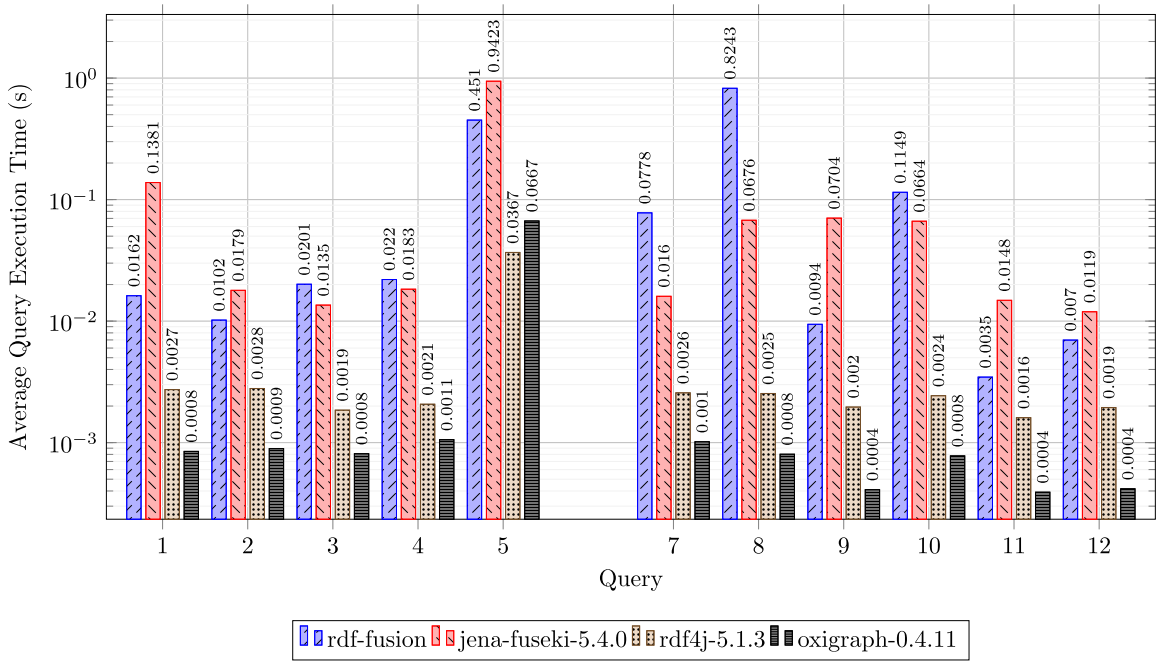


**FIGURE 7.** **BSBM Explore (25000 products, ≈ 8.9 million triples, 24 parallel queries). Lower is better.**

query engine (not RDF Fusion) crashed repeatedly during our experiments.

The Explore use case consists of data retrieval queries that simulate a user browsing the platform – for example, fetching reviews of a particular product. Query engines with low overhead for processing and data access typically perform well in this setting. The benchmark results for the Explore use case are shown in Fig. 7. Although RDF4J and Oxigraph outperform RDF Fusion in this scenario, RDF Fusion still achieves competitive execution times, comparable to Jena. Oxigraph yields the best performance on most Explore queries.

By contrast, the Business Intelligence use case contains analytical queries from different stakeholders. These queries often require processing larger intermediate results, such as aggregating the average price of a product type. Engines optimized for analytical workloads are expected to perform well here. The results for the Business Intelligence use case, depicted in Fig. 8, present a different picture. Here, Oxigraph performs worse than most other engines, while RDF Fusion and, in some cases, RDF4J deliver the best results.

To validate the correctness of the SPARQL results, we selected a representative query instance from each query type and manually compared its output against that of a battle-tested commercial SPARQL engine. Adapted versions of these queries (e.g., with additional orderings to ensure a stable result) are executed during the CI workflow.

Running the official BSBM qualification suite was not feasible because it requires loading 100 million triples into memory while still having enough memory to run the queries. Once an on-disk storage layer is implemented, we plan to incorporate the full qualification test into the CI workflow.

## VIII. DISCUSSION

Before proceeding with a detailed discussion, this section revisits the requirements outlined in Section IV. First, the evaluation provides evidence that RDF Fusion complies with the SPARQL 1.1 standard (**R1**). Next, the discussion, together with the example of hybrid data models in Section III, argues that RDF Fusion can support sophisticated extensions, including hybrid data models (**R2**). Providing a feature-complete prototype that implements a hybrid data model on top of RDF Fusion is beyond the scope of this paper. Finally, the performance evaluation shows that RDF Fusion achieves competitive performance for analytical SPARQL queries (**R3**).

*Performance:* The evaluation results highlight two key findings: RDF Fusion performs well enough on queries that involve relatively small intermediate results (BSBM Explore), and it performs well for queries with larger intermediate results (BSBM Business Intelligence). Further analysis reveals that, in the case of queries that quickly finish their execution, a significant portion of query execution time is spent in query planning. Since DataFusion is primarily optimized for large datasets, investing more time in query planning can often be beneficial. However, we believe that some parts of the planning machinery are less relevant for RDF Fusion's SPARQL queries (e.g., rewriting relational expressions). We are currently identifying a subset of optimizations most relevant to RDF Fusion to improve planning performance.

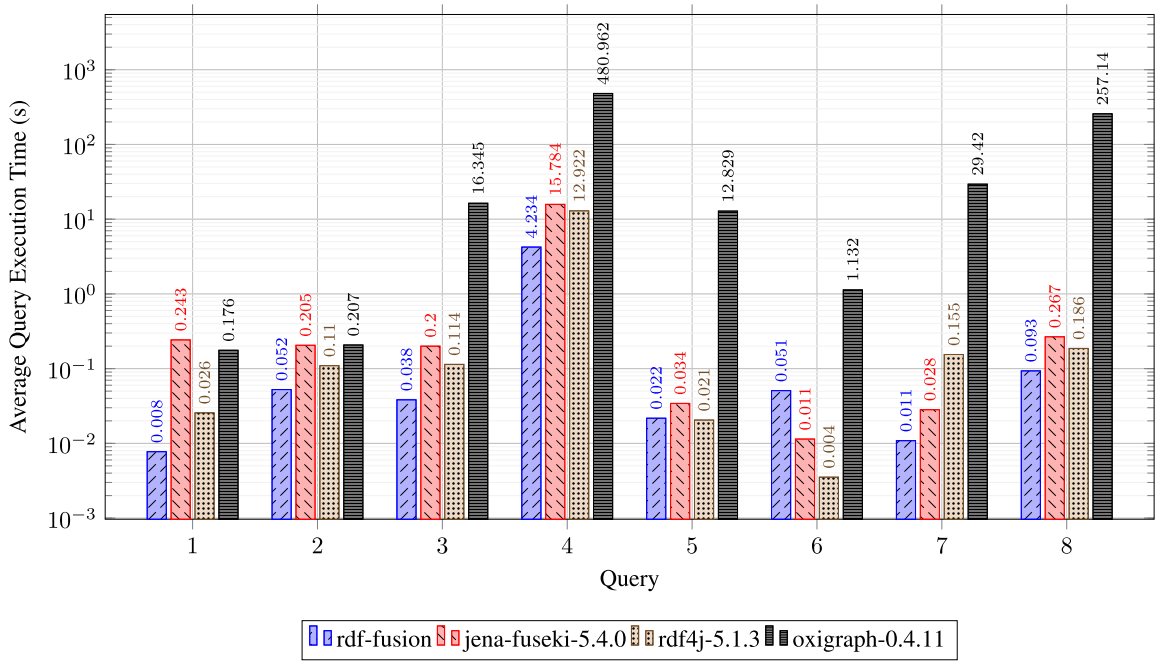


**FIGURE 8.** BSBM Business Intelligence (5000 products, ≈ 1.8 million triples, 12 parallel queries). Lower is better.

It is also important to note that RDF Fusion is still a relatively new system. Certain features that are crucial for specific query workloads are not yet implemented in our custom operators (e.g., statistics collection, sideways information passing). We are confident that continued improvements in DataFusion, as well as SPARQL-specific optimizations in RDF Fusion, will further strengthen its performance in graph processing. Even without further performance improvements, the results show that RDF Fusion already performs well enough to process graphs in hybrid data models. We plan on investigating RDF Fusion's performance and memory consumption characteristics further, once the aforementioned optimizations have been implemented.

*Vectorizing SPARQL Queries:* Solutions can bind variables to arbitrary RDF terms, which can lead to heterogeneous SPARQL results, as demonstrated in Section V-C. This heterogeneity can impede the vectorized execution of some SPARQL functions. Therefore, a detailed evaluation of the performance characteristics of vectorizing scalar and aggregate SPARQL functions in real-world workloads would be beneficial. Currently, most SPARQL functions in RDF Fusion are not yet optimized to take advantage of modern SIMD instructions. We are currently working on further investigating these characteristics.

*Extending RDF Fusion:* RDF Fusion is designed with a focus on extensibility. As noted in Section VI-D, it can be used as a DataFusion extension, giving users access to its full range of extension points. However, this requires more effort compared to the built-in RDF Fusion extension mechanisms (e.g., custom functions), since users must construct the logical plan themselves. More research and engineering efforts are required to further lower this barrier. Possible native extension points of RDF Fusion include user-defined types in the typed value encoding, custom object IDs that may store small values inline, and an extensible query parser.

*DataFusion as a Foundation:* We built RDF Fusion on DataFusion instead of creating a specialized query engine. Although other works [19] mention the challenges of extending modular execution engines to support SPARQL queries, we believe that building on DataFusion is a good decision. It provides a state-of-the-art query engine architecture and many features that would otherwise have been reimplemented from scratch. However, some SPARQL features currently require workarounds. For example, at the time of writing, users cannot provide a custom sort order to DataFusion's sort operator. As a result, RDF Fusion cannot sort directly on the typed value encoding as the sort implementation is not implemented for `Union` layouts, thus forcing us to add another "Sortable" encoding for this purpose. As DataFusion matures, these restrictions may be lifted. Furthermore, RDF Fusion may provide its own physical sort operator to circumvent this problem in the future.

*Scalability:* DataFusion's core project focuses on single-node execution. Although it has shown excellent single-node performance, it cannot scale to multiple machines by itself. However, there are multiple projects that extend or use DataFusion in a distributed manner (e.g., DataFusion Ballista[9]). Building on these approaches, RDF Fusion may

[9]https://datafusion.apache.org/ballista/

also be able to scale across multiple nodes in the future. Furthermore, in modern shared-disk database architectures (e.g., [32]) that rely on a scalable storage layer, a simple scaling strategy would be to run multiple stateless compute nodes that can handle load-balanced queries. Although this model would not allow users to leverage multiple compute nodes for a single query, they could leverage the compute nodes across multiple queries.

## IX. RELATED WORK

We identified two areas that are closely related to this work. Firstly, there are other open source SPARQL engines that could be used to implement hybrid data models. Secondly, there are other applications of the Arrow ecosystem within the Semantic Web.

### A. OPEN SOURCE SPARQL ENGINES

Apache Jena ARQ is the SPARQL engine that powers the Apache Jena system [33]. It is an extensible and embeddable SPARQL engine and well suited for writing extensions, as demonstrated by many different research prototypes (e.g., [24]). In contrast to Jena ARQ, RDF Fusion provides a different query execution model that can outperform Jena, as demonstrated in Section VII-C. Furthermore, RDF Fusion can leverage a large ecosystem by integrating with Arrow and DataFusion. For example, if someone writes high-performance geo-spatial functions for Arrow, RDF Fusion can use these implementations by using the same data type in the typed value encoding. Similarly, the larger Arrow community can also benefit from improvements made by RDF Fusion to their ecosystem.

RDF4J[10] is a modular framework for working with RDF data that also includes a SPARQL implementation. It's SAIL (Storage and Inference Layer) API allows users to plug-in, among other things, custom storage layers, custom reasoners, and extensions such as GeoSPARQL [24]. However, similar to Apache Jena, RDF4J's query engine employs a row-based approach, and therefore exhibits the same performance characteristics discussed previously.

Oxigraph [23] is a SPARQL engine implemented in Rust. It employs a custom row-based query engine designed for SPARQL workloads. The evaluation results show that Oxigraph performs particularly well on queries that benefit from engines with low overhead. However, its performance is less competitive on analytical workloads. In addition, since extensibility is not a primary design focus, the same limitations as above also apply here.

QLever [34] also adopts a columnar layout for its query engine. In contrast to approaches that emphasize the integration of time-series data, QLever focuses on enabling full-text search within SPARQL queries. It employs a custom data layout that is not based on a standardized memory representation. Although this design allows for a query engine tailored specifically to SPARQL evaluation, it prevents QLever from leveraging the broader ecosystem built around Arrow and DataFusion. In addition, RDF Fusion distinguishes itself by its emphasis on extensibility.

In addition to the systems mentioned above, there are commercial SPARQL engines, some of which employ a columnar data processing strategy (e.g., Stardog's BARQ [19], Virtuoso [35]), as well as discontinued open-source systems (e.g., Blazegraph [36]). While some of these platforms provide limited extensibility (e.g., custom functions), none offer extension points as comprehensive or flexible as those available in DataFusion.

In summary, there are compelling open-source projects for extending SPARQL, either by using native extension points or by forking the open-source code. However, none of them combine the focus on extensibility and analytical query processing in a single system.

### B. ARROW IN THE SEMANTIC WEB

Arrow has already been used for high-performance Semantic Web applications. Chrontext [37] allows integrating data from industrial data sources in an OBDA SPARQL engine. Processing is performed using Pola.rs,[11] a data frame library that uses Apache Arrow as an internal format. While Chrontext allows one to issue SPARQL queries against IoT data sources, it does so by mediating between an existing SPARQL and time series databases.

Arrow has also been used in maplib [38] (also through Pola.rs), a library for creating RDF graphs with template expansions from data frames. The library allows users to query the graph directly with SPARQL, construct further triples with `CONSTRUCT` clauses, and validate the resulting graph using SHACL [39]. SPARQL is implemented by mapping it directly to dataframe operations. However, maplib does not support SPARQL queries where a single variable has multiple data types.

While these existing approaches bring parts of RDF Fusion's premises to the Semantic Web world, they are either specialized for a particular application or are limited in the SPARQL compatibility they provide. Furthermore, they do not share RDF Fusion's focus on extensibility.

## X. CONCLUSION & FUTURE WORK

This work presented RDF Fusion, a SPARQL engine built on Apache DataFusion. Leveraging DataFusion's extensible query engine architecture, RDF Fusion provides a foundation for exploring columnar SPARQL query processing and hybrid data models, benefiting from the surrounding ecosystem. To bridge the gap between the RDF data model and Apache Arrow, we introduced several RDF term encodings and demonstrated that this mapping can be used to efficiently implement SPARQL on a modular analytical query engine. We also provided evidence that RDF Fusion complies with the SPARQL 1.1 standard and discussed how researchers

[10] https://rdf4j.org/

[11] https://pola.rs/

working on the Semantic Web and hybrid data model can leverage the discussed extension points.

RDF Fusion is a first step toward embracing the trend of composable data management systems in the Semantic Web and hybrid data models. Other building blocks, such as index definition and update mechanisms, could further reduce the complexity of SPARQL implementations, allowing engineering efforts to focus on specialized solutions like time series data or full-text search. The goal is to enable exploration of these specialized data models without the effort of building a SPARQL engine from scratch, making these technologies more accessible for practical use cases.

Future work, partially discussed in Section VII-B, includes building an on-disk storage layer to allow RDF Fusion to function as a standalone SPARQL engine, implementing various performance improvements, providing additional built-in extensibility options, and implementing a distributed version of RDF Fusion. Furthermore, we are aiming to build a full-fledged hybrid data model on top of RDF Fusion.

## REFERENCES

[1] M. Stonebraker and U. Cetintemel, "'One size fits all': An idea whose time has come and gone," in *Proc. 21st Int. Conf. Data Eng. (ICDE)*, 2005, pp. 2–11.

[2] A. Davoudian, L. Chen, and M. Liu, "A survey on NoSQL stores," *ACM Comput. Surveys*, vol. 51, no. 2, pp. 1–43, Apr. 2018.

[3] World Wide Web Consortium. *RDF 1.1 Concepts and Abstract Syntax*. Accessed: Sep. 12, 2025. [Online]. Available: https://www.w3.org/TR/rdf11-concepts/

[4] M. Sabou, S. Biffl, A. Einfalt, L. Krammer, W. Kastner, and F. J. Ekaputra, "Semantics for cyber-physical systems: A cross-domain perspective," *Semantic Web*, vol. 11, no. 1, pp. 115–124, Jan. 2020.

[5] S. K. Jensen, T. B. Pedersen, and C. Thomsen, "Time series management systems: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 11, pp. 2581–2600, Nov. 2017.

[6] G. Steindl and W. Kastner, "Query performance evaluation of sensor data integration methods for knowledge graphs," in *Proc. Big Data, Knowl. Control Syst. Eng. (BdKCSE)*, Nov. 2019, pp. 1–8.

[7] S. Zhang, W. Zeng, I.-L. Yen, and F. B. Bastani, "Semantically enhanced time series databases in IoT-edge-cloud infrastructure," in *Proc. IEEE 19th Int. Symp. High Assurance Syst. Eng. (HASE)*, Jan. 2019, pp. 25–32.

[8] E. Bollen, R. Hendrix, and B. Kuijpers, "Managing data of sensor-equipped transportation networks using graph databases," *Geoscientific Instrum., Methods Data Syst.*, vol. 13, no. 2, pp. 353–371, Nov. 2024.

[9] M. Ammar, C. Rost, R. Tommasini, S. Agarwal, A. Bonifati, P. Selmer, E. Kharlamov, and E. Rahm, "Towards Hybrid graphs: Unifying property graphs and time series," in *Proc. 28th Int. Conf. Extending Database Technol.*, 2025, pp. 970–977.

[10] T. Schwarzinger, G. Steindl, T. Frühwirth, T. Preindl, K. Diwold, K. Ehrenmüller, and F. J. Ekaputra, "SigSPARQL: Signals as a first-class citizen when querying knowledge graphs1," in *Studies on the Semantic Web*, 2025.

[11] P. Pedreira, O. Erling, K. Karanasos, S. Schneider, W. McKinney, S. R. Valluri, M. Zait, and J. Nadeau, "The composable data management system manifesto," *Proc. VLDB Endowment*, vol. 16, no. 10, pp. 2679–2685, Jun. 2023.

[12] A. Lamb, Y. Shen, D. Heres, J. Chakraborty, M. O. Kabak, L.-C. Hsieh, and C. Sun, "Apache arrow DataFusion: A fast, embeddable, modular analytic query engine," in *Proc. Companion Int. Conf. Manage. Data*, Jun. 2024, pp. 5–17.

[13] P. Pedreira, O. Erling, M. Basmanova, K. Wilfong, L. Sakka, K. Pai, W. He, and B. Chattopadhyay, "Velox: Meta's unified execution engine," *Proc. VLDB Endowment*, vol. 15, no. 12, pp. 3372–3384, Aug. 2022.

[14] World Wide Web Consortium. *SPARQL 1.1 Query Language*. Accessed: Sep. 12, 2025. [Online]. Available: https://www.w3.org/TR/sparql11-query/

[15] R. Cyganiak, "A relational algebra for SPARQL," Digital Media Systems Laboratory HP Laboratories, Bristol, U.K., Tech. Rep. HPL-2005-170, 2005.

[16] *Github: RDF Fusion*. Accessed: Sep. 12, 2025. [Online]. Available: https://github.com/tobixdev/rdf-fusion

[17] T. Schwarzinger and M. Thoma, "Supplementary data for 'RDF fusion: An extensible SPARQL engine for hybrid data models,"' Zenode, CERN Eur. Org. Nucl. Res., Genève, Switzerland, Tech. Rep., Sep. 2025, doi: 10.5281/zenodo.17200132.

[18] World Wide Web Consortium. *RDF 1.1 Turtle*. Accessed: Sep. 12, 2025. [Online]. Available: https://www.w3.org/TR/turtle/

[19] S. Grätzer, L. Heling, and P. Klinov, "BARQ: A vectorized SPARQL query execution engine," in *Proc. 8th Joint Workshop Graph Data Manage. Exper. Syst. (GRADES) Netw. Data Anal. (NDA)*, Jun. 2025, pp. 1–9.

[20] *Information Technology–Database Languages–SQL*, Standard 9075-1:2023, 2023.

[21] W. Ali, M. Saleem, B. Yao, A. Hogan, and A.-C.-N. Ngomo, "A survey of RDF stores & SPARQL engines for querying knowledge graphs," *VLDB J.*, vol. 31, no. 3, pp. 1–26, May 2022.

[22] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik, "C-store: A column-oriented DBMS," in *Proc. 31st Int. Conf. Very Large Data Bases*, Aug. 2005, pp. 553–564.

[23] T. P. Tanon, "Oxigraph," Zenode, CERN Eur. Org. Nucl. Res., Genève, Switzerland, Tech. Rep., 2023, doi: 10.5281/zenodo.7408022.

[24] R. Battle and D. Kolas, "GeoSPARQL: Enabling a geospatial semantic web," *Semantic Web J.*, vol. 3, no. 4, pp. 355–370, 2011.

[25] N. J. Car and T. Homburg, "GeoSPARQL 1.1: Motivations, details and applications of the decadal update to the most important geospatial LOD standard," *ISPRS Int. J. Geo-Inf.*, vol. 11, no. 2, p. 117, Feb. 2022.

[26] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus, "C-SPARQL: SPARQL for continuous querying," in *Proc. 18th Int. Conf. World Wide Web*, France, Apr. 2009, pp. 1061–1062.

[27] A. Poggi, D. Lembo, D. Calvanese, G. De Giacomo, M. Lenzerini, and R. Rosati, "Linking data to ontologies," *Journal on Data Semantics X* (Lecture Notes in Computer Science), vol 4900, S. Spaccapietra, Ed., Berlin, Germany: Springer, 2008. [Online]. Available: https://doi.org/10.1007/978-3-540-77688-8_5

[28] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proc. 1984 ACM SIGMOD Int. Conf. Manage. Data*, Jun. 1984, pp. 47–57.

[29] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati, "Tractable reasoning and efficient query answering in description logics: The DL-lite family," *J. Automated Reasoning*, vol. 39, no. 3, pp. 385–429, Oct. 2007.

[30] C. Bizer and A. Schultz, "The Berlin SPARQL benchmark," in *Semantic Services, Interoperability and Web Applications*, 2011, pp. 81–103.

[31] *Github: SPARQL Bencher*. Accessed: Sep. 12, 2025. [Online]. Available: https://github.com/tobixdev/sparql-bencher

[32] S. Loesing, M. Pilman, T. Etter, and D. Kossmann, "On the design and scalability of distributed shared-data databases," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, May 2015, pp. 663–676.

[33] B. McBride, "Jena: A semantic web toolkit," *IEEE Internet Comput.*, vol. 6, no. 6, pp. 55–59, Nov. 2002.

[34] H. Bast and B. Buchhold, "QLever: A query engine for efficient SPARQL+text search," in *Proc. ACM Conf. Inf. Knowl. Manage.*, Nov. 2017, pp. 647–656.

[35] O. Erling and I. Mikhailov, "Virtuoso: RDF support in a native RDBMS," in *Semantic Web Information Management*, 2010, pp. 501–519.

[36] B. Thompson, M. Personick, and M. Cutcher, "The bigdata RDF graph database," in *Linked Data Management*. Boca Raton, FL, USA: CRC Press, 2014.

[37] M. Bakken and A. Soylu, "Chrontext: Portable SPARQL queries over contextualised time series data in industrial settings," *Expert Syst. Appl.*, vol. 226, Sep. 2023, Art. no. 120149.

[38] M. Bakken, "maplib: Interactive, literal RDF model mapping for industry," *IEEE Access*, vol. 11, pp. 39990–40005, 2023.

[39] World Wide Web Consortium. (Jul. 2017). *Shapes Constraint Language (SHACL)*. Accessed: Sep. 12, 2025. [Online]. Available: https://www.w3.org/TR/shacl/

**TOBIAS SCHWARZINGER** (Graduate Student Member, IEEE) received the B.Sc. and M.Sc. degrees in computer science, from TU Wien, Vienna, Austria, in 2020 and 2023, respectively. He is currently pursuing the Ph.D. degree in informatics with the Institute of Computer Engineering, TU Wien.

Since 2023, he has been an University Assistant. His current research interests include knowledge graphs and time series data in the context of cyber-physical systems.

**MAX THOMA** received the B.Sc. degree in computer engineering from TU Wien, in 2023. He is currently pursuing the M.Sc. degree. He is the Member of the student staff. His research interests include semantic technologies in automation and model-based systems engineering.

**THOMAS PREINDL** received the B.Sc. and M.Sc. degrees in computer science from TU Wien, Vienna, Austria, in 2012 and 2019, respectively. He is currently pursuing the Ph.D. degree in informatics with the Institute of Computer Engineering, TU Wien.

Since 2019, he has been an Research Assistant. His research focuses on architectures for trustworthy and compliant data exchange in decentralized multi-stakeholder ecosystems.

**MARTIN KJÄER** received the B.Sc. degree in business informatics from the University of Vienna, in 2016, and the M.Sc. degree in business informatics from TU Wien, in 2021. Since 2018, he has been with the Institute of Computer Engineering, TU Wien, where he is a Researcher and the Ph.D. Candidate. His research interests include blockchain and other trust-enabling technologies, with a focus on software architecture.

**VALENTIN PHILIPP JUST** received the B.Sc. and M.Sc. degrees in mechatronics and information technology from KIT, Karlsruhe, Germany, in 2017 and 2024, respectively. He is currently pursuing the Ph.D. degree in informatics with the Institute of Computer Engineering, TU Wien. Since 2021, he has been an University Assistant. His research interests include process mining techniques, the Industrial Internet of Things, and semantics in automation systems.

**GERNOT STEINDL** (Member, IEEE) received the B.S. degree in electrical engineering from Technical University Wien, Vienna, Austria, in 2010, the first M.S. degree in electrical engineering from Technical University Wien, in 2013, the second M.S. degree in building technology from the University of Applied Science Burgenland, Eisenstadt, Austria, in 2016, and the Ph.D. degree in computer science from the Technical University Wien, in 2021. From 2015 to 2018, he was a Researcher with the Research Burgenland GmbH. In 2018, he joined the Institute of Computer Engineering, Technical University Wien as a Predoctoral Researcher. From 2022 to 2025, he was a Postdoctoral Researcher with the Research Unit Automation Systems, Technical University Wien. Since 2025, he has been the Head of the Study Program AI Solution Engineering with the University of Applied Sciences Burgenland. His research interests include hybrid AI for cyber-physical Systems (CPS), semantic modeling, and causal reasoning.