Jelly: a fast and convenient RDF serialization format

Piotr Sowiński[1,2,*], Karolina Bogacka[1,2], Anastasiya Danilenka[1,2] and Nikita Kozlov[1,2]

[1]NeverBlink, ul. Wspólna 56, 00-684 Warsaw, Poland
[2]Warsaw University of Technology, Pl. Politechniki 1, 00-661 Warsaw, Poland

Abstract

Existing RDF serialization formats such as Turtle, N-Triples, and JSON-LD are widely used for communication and storage in knowledge graph and Semantic Web applications. However, they suffer from limitations in performance, compression ratio, and lack of native support for RDF streams. To address these shortcomings, we introduce Jelly, a fast and convenient binary serialization format for RDF data that supports both batch and streaming use cases. Jelly is designed to maximize serialization throughput, reduce file size with lightweight streaming compression, and minimize compute resource usage. Built on Protocol Buffers, Jelly is easy to integrate with modern programming languages and RDF libraries. To maximize reusability, Jelly has an open protocol specification, open-source implementations in Java and Python integrated with popular RDF libraries, and a versatile command-line tool. To illustrate its usefulness, we outline concrete use cases where Jelly can provide tangible benefits. By combining practical usability with state-of-the-art efficiency, Jelly is an important contribution to the Semantic Web tool stack.

Keywords

RDF, Serialization format, RDF stream processing, Knowledge graphs, Semantic Web data formats

1. Introduction

Knowledge graph and Semantic Web systems use serialization formats such as Turtle, N-Triples, and JSON-LD to exchange and store RDF data in scenarios ranging from database dumps to client-server communication. However, common RDF formats are limited in terms of their serialization/deserialization speeds, compression ratios, and processing efficiency, which can become a performance bottleneck. Additionally, no W3C format can natively represent streams of RDF data, which could be beneficial in some applications (e.g., industrial IoT, clickstreams, real-time user interaction). Although several new RDF formats were proposed in the past to solve some of these issues (see Section 5), none have reached wider adoption, either due to missing tooling, incomplete use case coverage, or other practical issues.

In this contribution we present **Jelly**, a high-performance binary RDF serialization format designed specifically to be easy to use, utilize as little compute resources as possible, and cover practical use case requirements. It natively supports RDF stream processing, but can also be applied in batch settings, in place of W3C-standard formats. In this work, we describe Jelly's contribution to the Semantic Web community tool stack that includes: (1) a robust and open Jelly protocol specification; (2) implementations for Java and Python compatible with popular RDF libraries; (3) an easy to use command-line tool; (4) several concrete use cases where Jelly can provide tangible benefits.

2. Jelly protocol

Jelly is a binary serialization format for streams of RDF triples, quads, graphs, or datasets [1]. It can be used in place of W3C-standard formats like N-Triples or Turtle, to simply represent a sequence of statements (a flat RDF stream in the RDF Stream Taxonomy [2]). It can also be used to represent a

SEMANTiCS 2025 Developers Workshop, September 03, 2025, Vienna, Austria
*Corresponding author.
piotr@neverblink.eu (P. Sowiński); karolina@neverblink.eu (K. Bogacka); anastasiya@neverblink.eu (A. Danilenka); nikita@neverblink.eu (N. Kozlov)
https://ostrzyciel.eu/ (P. Sowiński)
0000-0002-2543-9461 (P. Sowiński); 0000-0002-7109-891X (K. Bogacka); 0000-0002-3080-0303 (A. Danilenka); 0009-0003-1634-9194 (N. Kozlov)

sequence of graphs or datasets (grouped RDF stream) – this would be a series of files in W3C-standard formats. A Jelly file is split into *frames*, where each frame corresponds to either a batch of RDF statements, or a complete RDF graph/dataset. In streaming settings (e.g., Kafka, gRPC), each frame corresponds to a separate message, allowing for efficient compression over the entire stream.

Serialization. Jelly uses the industry-standard Protocol Buffers (Protobuf) serialization framework as its basis. The binary layout of Jelly is specified in a Protobuf interface definition file, which can then be used by compilers to generate serialization/deserialization code in one of the many supported programming languages (e.g., C++, Python, Java, Rust, C#). This greatly simplifies implementing the format, as all binary-level manipulation code is generated automatically, and the developer only needs to implement a translation layer between Protobuf messages and RDF types of the given library. The mechanism for this translation is defined in the open Jelly specification[1].

Compression. The design goals of Jelly are to: (1) maximize serialization/deserialization throughput, (2) provide a relatively good compression ratio, and (3) operate in a fully streaming manner. We understand the third goal as meeting the common criteria for stream processing systems: process only one triple at time, use a limited amount of time per triple, use a limited amount of memory (overall), output the results on demand, and adapt to temporal changes [3]. This essentially would allow Jelly to process arbitrarily large (potentially indefinite) RDF files in constant memory.

To meet these criteria, Jelly employs a streaming compression algorithm, defined in the aforementioned specification. The full description of the algorithm is beyond the scope of this contribution – here we provide only a brief summary. Three fixed-sized string lookup tables are used, for IRI prefixes, suffixes, and datatypes. The serializer populates these tables until they are full, after which old lookup entries can be replaced by new ones using any policy (e.g., least recently used). Entries in these tables are referenced using variable-length integer identifiers, also utilizing simple delta compression to minimize their size. For consecutive RDF statements with repeating terms (e.g., same triple subject), the repeated term is not serialized, further reducing the file size. These compression methods are relatively easy to implement and can work in a fully streaming manner, processing one triple at a time.

Performance. We regularly publish performance benchmarks for the Jelly-JVM implementation on the Jelly website[2], using a diverse mix of datasets from RiverBench [4]. In summary, Jelly-JVM 2.7.0 achieves an average compression ratio of 16.2% (compared to a baseline of 100% for N-Triples), serialization speed of 7.28 MT/s (millions of triples per second), and deserialization speed of 15.16 MT/s with Eclipse RDF4J. As far as we are aware, this currently makes Jelly the most compressed and the fastest RDF format implemented in either Apache Jena or RDF4J.

3. Implementations and tooling

Jelly currently has implementations for Python and the Java Virtual Machine. Both implementations are designed around a generic core that can be used in conjunction with multiple different RDF libraries – at the moment supported are: Apache Jena, Eclipse RDF4J, Titanium RDF API, and rdflib.

3.1. Java Virtual Machine: Jelly-JVM

Jelly-JVM[3] is the more mature and heavily optimized implementation. It is organized into a set of modular components. The `jelly-core` module encapsulates the base functionality for encoding, decoding and transcoding Jelly-formatted data, independent of any particular RDF library, along with necessary utilities and abstractions to facilitate the development of integrations with various RDF libraries. The

[1] https://w3id.org/jelly/dev/specification/serialization
[2] https://w3id.org/jelly/dev/performance
[3] https://w3id.org/jelly/jelly-jvm

`jelly-jena` module provides full interoperability with Apache Jena by converting between Jelly Protobuf messages and Jena's data structures. It includes an integration with Jena's RIOT subsystem, making it possible to use Jelly in Jena just like any other RDF format (e.g., Turtle), including proper support for content negotiation. Module `jelly-rdf4j` offers analogous adapters for Eclipse RDF4J, in addition to an integration with the Rio serialization subsystem. Module `jelly-titanium-rdf-api` supplies an adapter layer for the minimalistic Titanium RDF API. Finally, the `jelly-pekko-stream` module written in Scala integrates with the powerful Apache Pekko Streams framework, allowing for efficient and scalable processing of RDF streams in more advanced use cases (e.g., IoT data aggregation).

Jelly-JVM can be used as a plugin with Jena or RDF4J[4]. For example, for Apache Jena Fuseki, full Jelly support can be installed by placing the plugin JAR in the `extra/` directory of the Fuseki installation. This will enable full support in SPARQL UPDATE queries, the graph store protocol, and in the graphical user interface. Alternatively, for programmatic use, one can include either the `jelly-jena` (for Jena) or `jelly-rdf4j` (for RDF4J) Maven artifact by adding the following to pom.xml:

```
<dependency>
  <groupId>eu.neverblink.jelly</groupId>
  <artifactId>jelly-jena</artifactId>
  <version>3.1.0</version>
</dependency>
```

This pulls in `jelly-core` and all necessary converters. With this dependency in place, Jelly support is registered with Jena or RDF4J automatically, enabling the use of Jelly as an RDF format in code. For example, one can load a Jelly-serialized graph via Jena RIOT using:

```
Model m = RDFDataMgr.loadModel("https://w3id.org/riverbench/v/2.1.0.jelly");
```

To write this graph back to a Jelly file, call:

```
RDFDataMgr.write(new FileOutputStream("m.jelly"), m, JellyLanguage.JELLY);
```

Streaming serialization/deserialization with Jena, RDF4J, and Titanium is also possible, hooking into the iterator-like interfaces of these libraries, which allows for processing RDF data one statement at a time[5]. Using the Pekko Streams integration, it is also possible to manipulate the RDF data in more complex ways, e.g., batching a stream of quads into a stream of datasets[6].

3.2. Python: pyjelly

`pyjelly`[7] is a Python implementation of Jelly, designed to make the benefits of the format accessible to the broad Python audience. Distributed through PyPI[8], it can be easily installed on all major operating systems (Linux, Windows, macOS) using standard Python package managers such as pip. To support users in getting started with pyjelly, a documentation suite is available[9] including usage examples, up-to-date functionality information, and API reference.

Like Jelly-JVM, pyjelly also has a generic serialization core, on top of which integrations with other libraries are built. Currently, only the popular `rdflib` library is supported in this manner, however, more integrations are planned. Serializing an rdflib graph to the Jelly format is as simple as installing pyjelly with rdflib support, e.g., `pip install pyjelly[rdflib]`, and specifying the Jelly format during serialization. As shown below, this is as easy as using built-in rdflib formats:

[4] https://w3id.org/jelly/jelly-jvm/dev/getting-started-plugins/
[5] https://w3id.org/jelly/jelly-jvm/dev/user/rdf4j
[6] https://w3id.org/jelly/jelly-jvm/dev/user/reactive
[7] https://github.com/Jelly-RDF/pyjelly
[8] https://pypi.org/project/pyjelly/
[9] https://w3id.org/jelly/pyjelly

```python
from rdflib import Graph
g = Graph()
g.parse("https://www.w3.org/2013/N-TriplesTests/nt-syntax-subm-01.nt")
g.serialize(destination="triples.jelly", format="jelly")
```

Similarly, a Jelly file can be parsed back to an rdflib `Graph`:

```python
g = Graph()
g.parse("triples.jelly", format="jelly")
```

3.3. Command-line tool: jelly-cli

To make it easier to use Jelly in production, development, and CI pipelines, we have developed a user-friendly command-line tool, `jelly-cli`[10]. This utility supports manipulating Jelly files without the need to write any code beyond a single terminal command. `jelly-cli` supports the adoption and growth of the Jelly ecosystem by lowering the barrier to entry and helps with common development tasks. Below we briefly showcase several commands supported by `jelly-cli` (version 0.4.7), highlighting its utility for both common and advanced scenarios:

- **Convert RDF to Jelly:**
  `jelly-cli rdf to-jelly input.nq --to=out.jelly`
  This command converts RDF data written in any W3C-standard format into Jelly. Additional options allow for configuring the compression settings.
- **Convert Jelly to RDF:**
  `jelly-cli rdf from-jelly input.jelly --to=out.trig`
  This command converts a Jelly file to a chosen RDF syntax. Supported formats include W3C-standard formats and a human-readable version of Jelly, useful for debugging (`jelly-text`).
- **Inspect a specific Jelly frame in human-readable binary form:**
  `jelly-cli rdf from-jelly input.jelly --out-format=jelly-text --take-frames=3..5`
  Combined with `jelly-text`, the `--take-frames` flag allows extracting individual frames for direct inspection and debugging of parts of the Jelly file.
- **Merge and transcode Jelly files:**
  `cat in1.jelly in2.jelly in3.jelly | jelly-cli rdf transcode --to=merged.jelly --opt.max-name-table-size=8192`
  The transcode command allows for merging multiple Jelly files into one, as well as recompressing the contents with new settings, using a very efficient transcoding algorithm.
- **Collect Jelly file statistics:**
  `jelly-cli rdf inspect ./in.jelly --per-frame=true`
  Summarizes information about the Jelly file's contents and used compression options. When run with `--per-frame=true`, returns detailed statistics for each frame.
- **Validate a Jelly file:**
  `jelly-cli rdf validate file.jelly --compare-to-rdf-file=file.nq --compare-ordered=true`
  Validates the Jelly file and optionally compares its contents against a reference RDF file. This command can also verify whether specific compression options were used during serialization.

4. Use cases

Jelly was designed to flexibly fit the requirements of many practical use cases, improving processing speed and reducing compute resource usage. Below we list the intended technical use cases, along with two examples of how Jelly is used in other projects.

[10] https://github.com/Jelly-RDF/cli

- **Client-server communication** – for example, a user interface frontend component can be efficiently linked to the backend. Jelly can reduce the latency between user input and the result, improving the user experience.
- **Inter-service communication** – complex backend applications often consist of a network of microservices that must exchange RDF data (e.g., databases, cron job workers, analytics pipelines). Jelly can be integrated into existing APIs or with a complete gRPC stack, improving efficiency.
- **Database dumps and bulk loads** – large RDF datasets can be quickly written and read with Jelly, while taking up less storage space. This can reduce the time needed for routine database maintenance tasks and help minimize infrastructure costs.
- **Streaming ingest** – Jelly can eliminate ingestion bottlenecks in systems processing large amounts of incoming streaming data, improving responsiveness to new events, and allowing the system to scale to larger problems.
- **Database replication and change capture** – changes to append-only databases can be recorded with the Jelly-RDF protocol described in Section 2. Additionally, add/delete operations with transaction support can be recorded using the Jelly-Patch format[11], based on RDF Patch.

Example use case 1: Nanopublication Network. The Nanopublication Network is a decentralized infrastructure for publishing and querying scientific knowledge using Semantic Web technologies [5]. It consists of various services (e.g., storage, querying) exchanging small RDF datasets – nanopublications. This inter-service communication has proven to be a bottleneck, mainly due to the need for requesting each nanopublication individually which adds overhead. We switched over the communication to Jelly streams over HTTP, which reduced the time to retrieve 60,000 nanopubs from the server from over an hour to less than 4 seconds. This solution is now used in the live Nanopublication Network[12].

Example use case 2: RiverBench. RiverBench[13] is a community-driven collaborative benchmark suite for RDF systems [4]. Each dataset in RiverBench is distributed as both a single N-Triples file (flat RDF stream) and an archive containing a sequence of Turtle files (grouped RDF stream). We have added Jelly as a third option that can replace both of these distribution formats. Jelly natively supports streams of graphs/datasets, which simplifies parsing the file, removing the need to manipulate TAR archive entries. On top of that, Jelly is much faster to parse, reducing the time needed to set up a benchmark.

5. Related work

Several efficient binary formats for streaming RDF data were proposed in the past, such as ERI [6] or S-HDT [7]. However, to the best of our knowledge, none have public implementations compatible with any modern RDF library, and therefore they are not usable in practice. The Jelly protocol uses the findings from these studies, but also focuses on broader tooling support and usability in general.

Both Apache Jena and Eclipse RDF4J have their own binary formats. In Jena, there are two nearly identical formats (one based on Apache Thrift, the other on Protobuf) that have no built-in compression and are largely analogous to N-Triples [8]. RDF4J has a compressed binary format that uses streamed dictionary compression (similar to Jelly) and a custom serialization framework [9]. In our benchmarks, RDF4J Binary is by far the closest contender to Jelly, with similar compression ratios, 1.09–1.31x slower serialization, and 1.96–2.14x slower deserialization than Jelly-JVM 2.7.0[14]. However, none of these formats has a complete, up-to-date specification, and they only work within one ecosystem (either Jena or RDF4J), limiting their impact on the community. They also do not support streams of graphs/datasets (only streams of triples/quads).

[11] https://w3id.org/jelly/dev/specification/patch
[12] https://nanopub.net
[13] https://w3id.org/riverbench
[14] https://w3id.org/jelly/dev/performance/rdf4j

There are also binary RDF formats with different design goals than Jelly, resulting in a very different set of trade-offs. CBOR-LD is a tightly compressed binary version of JSON-LD [10]. It is primarily designed to be as compressed as possible (for use in, e.g., QR codes), while sacrificing ease of use (need to design compression contexts by hand) and serialization speed. CBOR-LD has an active community, with a tool stack similar to Jelly, e.g., the `ld-cli` tool. Finally, HDT [11] is an indexed binary format that allows for direct querying of RDF files. It does not support streaming compression, as the entire contents of the graph must be known before writing the file. It is typically integrated with RDF libraries as a storage backend (not a serialization format), making it applicable for different use cases than Jelly.

6. Conclusion and future work

In this contribution, we present Jelly, a high-performance RDF serialization format designed specifically to be fast, easy to use, and meet the practical requirements of many real-life use cases. It is currently integrated with several popular RDF libraries and has a CLI tool available to aid with production workloads, testing, and development.

We are constantly improving Jelly and its tooling. In the near future, we plan to: finalize protocol test cases for conformance testing of implementations, expand the feature set of the Python implementation, and integrate Jelly with commonly-used Semantic Web tools. Further plans include implementations for more programming languages (e.g., JavaScript, Rust, Kotlin), building new integrations (e.g., Neo4j, NetworkX, Pandas), and adding support for serializing SPARQL result sets.

We invite the community to contribute to the Jelly protocol and its tooling, with feature requests, bug reports, new integrations, documentation and more. More information on contributing can be found here: **https://w3id.org/jelly/dev/contributing**

References

[1] P. Sowiński, K. Wasielewska-Michniewska, M. Ganzha, M. Paprzycki, et al., Efficient RDF streaming for the edge-cloud continuum, in: 2022 IEEE 8th World Forum on Internet of Things (WF-IoT), IEEE, 2022, pp. 1–8. doi:10.1109/WF-IoT54382.2022.10152225.

[2] P. Sowiński, P. Szmeja, M. Ganzha, M. Paprzycki, RDF Stream Taxonomy: Systematizing RDF stream types in research and practice, Electronics 13 (2024) 2558.

[3] A. Bifet, R. Gavalda, G. Holmes, B. Pfahringer, Machine learning for data streams: with practical examples in MOA, MIT press, 2023.

[4] P. Sowiński, M. Ganzha, Realizing a collaborative RDF benchmark suite in practice, arXiv preprint arXiv:2410.12965, 24th International Conference on Knowledge Engineering and Knowledge Management (EKAW 2024), 26–28 November 2024, Amsterdam, Netherlands (2024).

[5] T. Kuhn, R. Taelman, V. Emonet, H. Antonatos, et al., Semantic micro-contributions with decentralized nanopublication services, PeerJ Computer Science 7 (2021) e387.

[6] J. D. Fernández, A. Llaves, O. Corcho, Efficient RDF interchange (ERI) format for RDF data streams, in: International Semantic Web Conference, Springer, 2014, pp. 244–259.

[7] H. Hasemann, A. Kröller, M. Pagel, RDF provisioning for the Internet of Things, in: 2012 3rd IEEE International Conference on the Internet of Things, IEEE, 2012, pp. 143–150.

[8] Apache Software Foundation, RDF binary using Apache Thrift, 2025. URL: https://jena.apache.org/documentation/io/rdf-binary.html, accessed on 12 June 2025.

[9] Eclipse Foundation, Inc., RDF4J binary RDF format, 2025. https://rdf4j.org/documentation/reference/rdf4j-binary/, accessed on 12 June 2025.

[10] M. Sporny, D. Longley, CBOR-LD 1.0 Draft Community Group Report, 2025. https://json-ld.github.io/cbor-ld-spec/, accessed on 11 June 2025.

[11] J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, M. Arias, Binary RDF representation for publication and exchange (HDT), Journal of Web Semantics 19 (2013) 22–41.