COTTAS: Columnar Triple Table Storage for Efficient and Compressed RDF Management

Julián Arenas-Guerrero[1] and Sebastián Ferrada[2,3,4]

[1] Universidad Politécnica de Madrid, Madrid, Spain
[2] IDIA, Universidad de Chile, Santiago, Chile
[3] Millennium Institute for Foundational Research on Data (IMFD), Chile
[4] National Center for Artificial Intelligence Research (CENIA), Chile
julian.arenas.guerrero@upm.es, sebastian.ferrada@uchile.cl

Abstract. One of the main challenges in working with RDF data is its verbosity, as repeated IRIs and IRI prefixes lead to large files that are costly to store and process. HDT, a binary RDF format, addresses this by compressing data while supporting efficient triple pattern evaluation without decompression. However, its performance is highly dependent on index alignment with query patterns. In this paper, we introduce COTTAS, a storage model that encodes RDF graphs directly into the open-source Apache Parquet columnar format. COTTAS represents RDF as a triple table and leverages block range indexes (zone maps) to achieve high compression ratios and fast query execution over compressed data. We also provide pycottas, an open-source Python library that enables compression of RDF data into COTTAS format and supports efficient querying by translating triple patterns into SQL queries over COTTAS files. This implementation facilitates the adoption of COTTAS for managing RDF graphs. Experiments on the WDBench and DBpedia benchmarks show that COTTAS reduces storage requirements by around 50% with respect to HDT and exhibits competitive triple pattern evaluation, with less performance volatility across pattern types.

Keywords: Compression · Indexing · Efficiency · RDF · Parquet

1 Introduction

Efficient management and consumption of RDF datasets at scale is increasingly critical for Semantic Web applications. Knowledge graphs (KGs) such as SemOpenAlex [20], Wikidata [45], or the Microsoft Academic KG [19] comprise billions of triples. As RDF graphs grow larger, there is a pressing demand for storage techniques that minimize space consumption while preserving efficient query performance. The most popular and widely adopted solution for this problem is the HDT (Header-Dictionary-Triples) format [21,22], which compresses RDF triples using a dictionary-based encoding and creates a self-index with adjacency lists, enabling compact graph storage and querying without full decompression.

Even when HDT has proven highly effective for RDF exchange and querying, its dictionary-centric approach inherently limits compression rates, especially for

datasets with many distinct IRIs and literals. Although secondary indexes can be built for efficiently evaluating triple patterns that do not match the self-index, these are costly to compute and further increase file sizes. Since HDT's introduction 15 years ago, new compression approaches beyond the Semantic Web community have emerged that could potentially optimize RDF management.

The emergence of Big Data led to the development of column-oriented formats to facilitate data interchange in the Big Data ecosystem. Notable open-source columnar formats include Parquet, ORC, and CarbonData (find formal definitions and empirical evaluation of these formats in [48]). These formats incorporate lightweight compression schemes and can be combined with fast lossless compression algorithms. They also support vectorized query processing, projection skipping, and include indexing and filtering techniques for efficient querying without full decompression. Columnar formats have been used with RDF for KG processing in the Big Data ecosystem [40,41]; however, they have never been studied for RDF compression, interchange, and basic consumption.

In light of these considerations, this work investigates the use of column-oriented formats to compress and query RDF. Specifically, we use the Parquet format, a *de facto* standard [48], and empirically study its space and querying performance over RDF graphs. We also aim to provide an industry-ready implementation to enable the management of RDF with columnar formats and set the groundwork for future research in RDF compression in this direction.

With these goals in mind, this paper makes the following contributions:

- We introduce the **COTTAS file format** for RDF compression, based on the storage of a triple table in an open-source column-oriented file format, namely Apache Parquet. COTTAS employs dictionaries and run-length encoding as lightweight compression schemes, and uses the Zstandard algorithm [15] for further compression of data. Querying is optimized utilizing block range indexes with ordered data and employing Bloom filters [10].
- We present the **pycottas Python library** for generating and working with COTTAS files. pycottas can compress RDF data in plain text formats, merge and subtract COTTAS files, resolve triple patterns, and evaluate SPARQL queries by serving as an RDFLib backend. pycottas is open-source and available under the permissive Apache 2.0 License.
- We provide an **empirical evaluation**, examining the use of encodings, indexes, and filters in COTTAS, and comparing the format to HDT, demonstrating superior compression efficiency across diverse datasets and competitive triple pattern scan performance. HDT only performs better when the triple pattern aligns with the self-index, and it has high performance volatility across pattern types, which is not present in COTTAS.

The remainder of this paper is organized as follows: Section 2 introduces the COTTAS file format, and Section 3 describes the pycottas library; Section 4 presents the experimental evaluation of COTTAS, compares it with HDT, and discusses the results; Section 5 reviews related works; finally, Section 6 wraps up with some conclusions and outlines directions for future work.

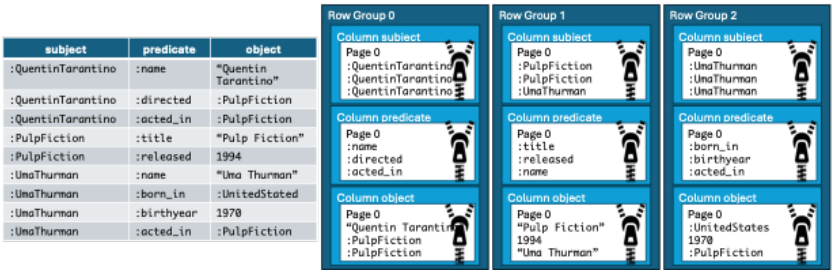

Fig. 1: RDF triple table (left) and its layout in the COTTAS format (right). Each column of the triple table is split into row groups of a fixed number of triples (three in this case). Each row group then lists the columns in column chunks, which are then divided into pages (in this case, a single page per column chunk) that are stored contiguously. The zippers indicate compression.

2 The COTTAS Format

The COTTAS file format utilizes Parquet to store RDF graphs as a triple table. In this section, we first outline the key features of Parquet and then describe how these features are leveraged for efficient RDF storage.

2.1 Column-Oriented Storage

The following features characterize Parquet's column-oriented storage model:

Data Layout: Column-oriented storage organizes data by storing values of each column contiguously, rather than storing entire rows together. The Partition Attributes Across (PAX) model [2] combines both approaches by dividing tables into fixed-size row groups, while still organizing data within each group in a column-oriented manner. In this layout, each row group contains a column chunk for every attribute, and each chunk is further divided into pages. This structure enables projection pushdown, allowing queries to read only the required columns while skipping the others. Figure 1 illustrates this layout in Parquet.

Encodings: Parquet supports various lightweight compression encodings, such as `PLAIN`, `PLAIN_DICTIONARY`, `DELTA_BINARY_PACKED`, and `RLE` (run-length encoding). In this work, we focus on the specific encodings most suited for RDF data, namely `RLE_DICTIONARY` and `DELTA_LENGTH_BYTE_ARRAY`.

`RLE_DICTIONARY` builds a dictionary of distinct values for each column chunk, storing it in a dedicated dictionary page. The remaining data pages store sequences of dictionary keys, compressed using a combination of bit packing, which finds the minimal number of bits needed for the key values, and run-length encoding (RLE), which replaces consecutive repetitions of a key with the key value and its count. Figure 2 illustrates this organization.

`DELTA_LENGTH_BYTE_ARRAY` stores all data values contiguously and compresses their lengths using delta encoding. By encoding only the differences between consecutive lengths and separating length information from the actual data, this

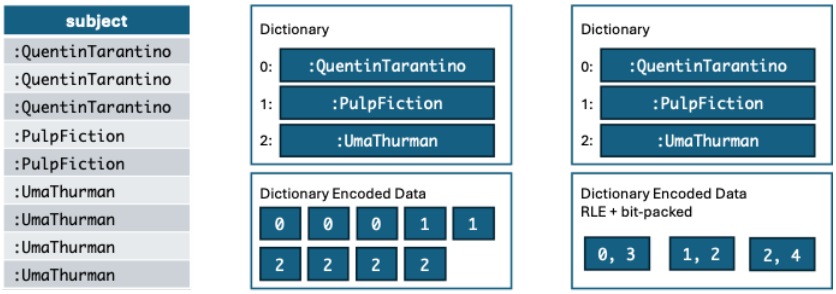

Fig. 2: Illustrative example of `RLE_DICTIONARY`. First, dictionary encoding is applied to the RDF terms in the column chunk (left). As a result, each RDF term is assigned an integer key (middle). Then, bit packing and run-length encoding are applied to the dictionary keys. As a result, runs of dictionary keys are replaced with pairs consisting of the value and the number of times it repeats.

approach enables compression algorithms (discussed next) to more effectively identify patterns in both sequences. As an example, the strings `:Q1`, `:Q2`, and `:Q3` have equal lengths, allowing their length sequence `[3, 3, 3]` to be efficiently represented as an initial value `3` followed by deltas `[0, 0]`.

Compression: Dictionary and data pages can be further compressed with algorithms such as Zstandard [15], Brotli [3], Snappy,[5] or Lempel–Ziv–Oberhumer (LZO).[6] By tuning a compression factor, these algorithms offer a configurable balance between performance and storage efficiency.

Block Range Indexes: Parquet uses block range indexes (BRIs, also referred to as zone maps or min-max indexes) to efficiently access data. BRIs record statistics about the minimum and maximum values within the column chunks and data pages. These indexes can significantly optimize scans by pruning irrelevant data blocks; if a predicate is not within the min-max range, the row group or data page can be skipped. BRIs are more effective over well-clustered columns, and their full potential is achieved with an ordered column.

Bloom Filters: Parquet supports Bloom filters [10] to accelerate query processing. For each column chunk, a Bloom filter summarizes the set of distinct values present in it, allowing efficient probabilistic membership tests. During query evaluation, Parquet reads Bloom filters to skip entire row groups that cannot satisfy a given predicate. If the Bloom filter for a column chunk indicates that a queried value is not present, the entire row group is skipped, reducing unnecessary I/O and improving query performance. Although Bloom filters can produce false positives, they never produce false negatives, ensuring correctness while effectively pruning irrelevant data blocks. The false positive rate can be controlled by adjusting the size of the filter, with larger filters reducing the likelihood of false positives.

[5] https://google.github.io/snappy/
[6] https://www.oberhumer.com/opensource/lzo/

2.2 Compressed RDF Triple Table

A *triple table* is a ternary relation (s, p, o) used to store an RDF graph. Each row in the triple table represents a triple in the graph. This tabular representation is widely used for RDF data management [1,18,36,8]. COTTAS adopts a column-oriented representation of this triple table, leveraging the Parquet format for efficient compression. Although other relational representations of RDF can be used with columnar storage (e.g., property tables [4]), they are not schema-oblivious and would require an arbitrary number of tables. Consequently, multiple COTTAS files would be needed, thus hindering RDF publication and exchange.

A *triples group* corresponds to a Parquet row group containing the column chunks s, p, and o. Each column chunk is of type `BYTE_ARRAY` (i.e., strings) representing RDF terms serialized in Notation3 (N3) format.

In addition to storing RDF as a triple table, COTTAS also supports *quad tables* to represent datasets with named graphs. In this case, each row includes a *graph* column, resulting in a quaternary relation (s, p, o, g). The *graph* column stores the name of the graph as a string or `NULL` if the triple belongs to the default graph. Thus, a valid COTTAS file can represent either a triple or a quad table, depending on the dataset requirements.

To improve query efficiency over compressed data, COTTAS leverages indexing strategies supported by Parquet. In particular, it makes use of BRIs. These indexes are especially effective when data is clustered, allowing large portions of data irrelevant to the query to be skipped during evaluation. COTTAS can then be indexed by sorting the triple table columns. There are six possible indexes: `SPO`, `SOP`, `PSO`, `POS`, `OSP`, and `OPS`. For instance, the `SPO` index first sorts the table by subject, then by predicate, and finally by object. The granularity of indexes has a direct impact on their effectiveness. Smaller triples groups improve pruning efficiency by allowing finer-grained filtering but increase metadata overhead due to the larger number of triples groups and associated dictionary pages. Similarly, without page-level indexes, pruning cannot be performed at the page level, limiting the ability to skip irrelevant data.

COTTAS can be hive-partitioned, i.e., the triple table can be split into multiple files based on a partitioning key. This is particularly useful for vertical partitioning [1] or for partitioning RDF datasets by named graph. Partitioning can be combined with BRIs and Bloom filters for enhanced query performance.

In addition to Parquet's standard file-level metadata, COTTAS files may include custom metadata, such as the index type, whether the data is stored as a quad table, and the number of distinct properties. By default, COTTAS uses the Zstandard compression algorithm [15] due to its widespread adoption and effective balance between compression ratio and speed. Nevertheless, the design remains flexible, allowing the use of alternative columnar formats and compression algorithms for storing RDF graphs.

3 The pycottas Library

The *pycottas* Python library provides a programmatic interface for managing and querying COTTAS files, following an interaction model similar to that of HDT. It supports efficient loading, compression, and querying of RDF data stored in COTTAS format. Built on top of DuckDB [39], pycottas leverages Oxigraph's RDF parsers [37] for data ingestion and integrates seamlessly with RDFLib [30] to ensure compatibility with existing RDF processing workflows. In the following, we describe its design and core functionalities.

pycottas provides tools for compressing and decompressing RDF data using the COTTAS format. Compression is performed through chunking, where subsets of triples are parsed and incrementally loaded into the triple table. The library supports a variety of input RDF serialization formats, including N-Triples, N-Quads, Turtle, TriG, N3, and RDF/XML. By default, Zstandard with maximum compression level is applied to optimize storage efficiency. Decompression follows a similar chunk-based process, reading triples from the triple table and converting them back into plain text formats. To handle large datasets, pycottas combines in-memory processing with out-of-core execution, automatically spilling data to disk when needed. Additionally, it supports merging (cat) and subtracting (diff) COTTAS files, facilitating dataset management operations.

Triple patterns are evaluated by unfolding to SQL and delegating execution to DuckDB. The `COTTASDocument` and `COTTASStore` classes provide access to COTTAS files. The former allows native evaluation of triple patterns, while the latter serves as an RDFLib backend for SPARQL query evaluation. In both cases, the metadata of COTTAS files is cached in memory to accelerate data access.

The package can be used as a library and also via command line. It supports compressing and querying RDF datasets (quads). It also allows querying multiple files at once using glob patterns and partitioned COTTAS.

Impact: The increasing volume of RDF data in large KGs makes efficient compression essential for ensuring their practical storage, exchange, and query performance, reinforcing its importance to the Semantic Web community. COTTAS significantly outperforms the state-of-the-art compression format HDT in terms of space and is competitive in performance, as demonstrated in Section 4. Additionally, pycottas is the first Python library that fully supports compressed RDF operations. This enhances its potential adoption, as other Python libraries such as rdflib-hdt [34] can only query HDT but cannot create HDT files.

Availability: The source code of pycottas is maintained on GitHub.[7] It is distributed through the Python Package Index[8] (PyPI), and the releases are archived at Zenodo [6]. The package is available under the Apache License 2.0.

Reusability: The documentation of pycottas is hosted on Read the Docs.[9] The Apache License 2.0 permits future scientific work and its use in industry.

[7] `https://github.com/arenas-guerrero-julian/pycottas`
[8] `https://pypi.org/project/pycottas/`
[9] `https://pycottas.readthedocs.io`

Design & technical quality: pycottas stands on the shoulders of giants: it builds on an open-source columnar format (Apache Parquet) and open-source software (DuckDB, Oxigraph, and RDFLib). It has been tested on massive datasets with over 1 billion triples (see Section 4), demonstrating the suitability of the design of the COTTAS file format and the pycottas implementation.

4 Experimental Evaluation

This section presents an empirical evaluation of the COTTAS format and the features introduced in Section 2. Our goal is to assess its storage efficiency and query performance, and to determine whether it constitutes a viable alternative to existing RDF compression formats. We also aim to provide experimental insights using a baseline configuration based on default Parquet settings to guide future research on RDF compression with columnar storage formats.

4.1 Experiment Setup

We run the experiments on an Ubuntu 24.04.2 LTS server with 64 vCPUs Intel Xeon Gold 5218R CPU @ 2.10GHz, 125GB RAM, and 1TB SSD. The COTTAS files are generated with pycottas utilizing DuckDB v1.2.1 with `PARQUET_VERSION v2` and Zstandard with compression level `22`. Writing page indexes is not supported in DuckDB (although the system can read them); hence, page indexes are not considered. For the rest, we use the following default configuration of DuckDB. The row group size is 122,880 rows. The number of distinct values (NDV) ratio to apply dictionary encoding is 10%. Bloom filters are generally created when DuckDB opts for dictionary encoding of a chunk, and its size is determined by a false positive probability of 1%. Triples in HDT are bitmap-encoded. For running triple patterns in HDT, we use the Python bindings of the HDT C++ library in rdflib-hdt. Similarly, pycottas uses the Python bindings of DuckDB, which is also implemented in C++. Reported times are the average of three executions. We use a timeout of 1,000 seconds for join queries.

In terms of dataset and queries, we use WDBench [5] and the DBpedia testbed used in [26,12]. WDbench is ∼156GB with ∼1.25 billion triples. DBpedia is ∼34GB with ∼222 million triples. Both benchmarks provide a set of triple patterns for evaluating query performance. We also use the join patterns from the DBpedia testbed to analyze join evaluation in COTTAS and HDT.

4.2 Encoding Analysis

We examine the Parquet metadata and summarize the encoding usage in Table 1. For each dataset and index, we report the frequency of each encoding method. Our observations show that the applied sorting strategy influences the selected encoding. When column values are clustered, `RLE_DICTIONARY` is used more frequently, likely because the NDV falls below the 10% threshold required for this encoding. This clustering effect is less pronounced for the subject column,

Table 1: Encodings used by COTTAS for the WDBench and DBpedia datasets with all possible indexes. Values correspond to the number of column chunks that have been encoded with `RLE_DICT` or `DELTA_LENGTH_BYTE_ARRAY` (`DLBA`).

Dataset	Column	Encoding	SPO	SOP	PSO	POS	OSP	OPS
WDBench	s	RLE_DICT	21	21	36	0	0	0
		DLBA	10,200	10,200	10,187	10,221	10,224	10,222
	p	RLE_DICT	8,146	8,153	10,223	10,221	10,079	10,111
		DLBA	2,075	2,068	0	0	145	111
	o	RLE_DICT	0	0	36	2,942	2,053	2,055
		DLBA	10,221	10,221	10,187	7,279	8,171	8,167
DBpedia	s	RLE_DICT	98	101	35	0	0	0
		DLBA	1,910	1,905	1,972	2,008	2,007	2,007
	p	RLE_DICT	747	740	2,007	2,007	1,681	1,689
		DLBA	1,261	1,266	0	1	326	318
	o	RLE_DICT	0	0	128	162	158	163
		DLBA	2,008	2,006	1,879	1,846	1,849	1,844

which typically presents a higher NDV ratio, as subject terms act as identifiers and show high variability. As a result, subject column chunks are mostly encoded using `DELTA_LENGTH_BYTE_ARRAY`. In contrast, predicate column chunks are predominantly encoded with `RLE_DICTIONARY`, given their usually low NDV ratio, an observation that is also the rationale behind vertical partitioning [1].

Predicate column chunks, and object column chunks to a lesser extent, are more suitable for `RLE_DICTIONARY`. To prioritize the use of this encoding by maintaining the 10% threshold for the NDV ratio, COTTAS files should be indexed by predicate or object. Future research should investigate how aggressively `RLE_DICTIONARY` should be applied for each index and subject, predicate, and object column chunks. Given the heavy use of IRIs (especially in subject and predicate positions), the use of specialized encodings in COTTAS for this type of string data could be further studied. Potential candidates for this are dictionary-based order-preserving delta compression [9] and Fast Static Symbol Table (FSST) [11], which exploits frequently-occurring substrings in data.

4.3 Space Efficiency

We measured the COTTAS file sizes across several RDF graphs, with results summarized in Table 2. For each graph and index, we compare COTTAS against HDT and the uncompressed data in N-Triples format. It must be noted that the OPS index could not be generated for the larger graphs using HDT due to a software error, which we reported to the developers of the indexing library.[10] As shown in the table, COTTAS consistently achieves better compression ratios than HDT across all index configurations. We report the Average Compression

[10] `https://github.com/rdfhdt/hdt-java/issues/213`

Table 2: File sizes of RDF graphs in COTTAS, HDT, and (uncompressed) N-Triples. For COTTAS and HDT, sizes in MB for all possible indexes are listed. ACR is the average compression ratio of the indexes w.r.t. the N-Triples file size. Dashes (–) indicate that a certain index could not be created.

Dataset	# of triples	N-Triples	Format	SPO	SOP	PSO	POS	OSP	OPS	ACR
WDBench [5]	1,253,567,798	156,233	HDT	13,143	13,887	14,484	12,668	13,577	–	8.6%
			COTTAS	7,838	8,030	5,940	5,259	5,464	5,440	4.1%
DBpedia [35]	221,982,313	34,128	HDT	4,590	4,718	4,652	4,561	4,693	–	13.6%
			COTTAS	2,281	2,301	2,658	2,802	2,547	2,550	7.3%
ORKG [43]	3,331,305	469	HDT	34.9	38.6	36.7	33.1	38.6	32.2	7.6%
			COTTAS	14.0	14.3	11.9	12.4	13.5	13.4	2.8%
CimpleKG [13]	136,837	29.9	HDT	6.3	6.4	6.5	6.3	6.4	6.3	21.4%
			COTTAS	2.5	2.5	2.5	2.5	2.7	2.7	8.7%
LauNutsKG [46]	1,181,549	148.7	HDT	10.6	12.8	12.8	10.8	12.8	10.1	7.8%
			COTTAS	3.6	3.7	3.2	3.4	3.8	3.8	2.4%
DSKG [23]	850,288	101.1	HDT	8.2	10.3	8.2	9.8	9.6	8.6	9.0%
			COTTAS	3.0	3.0	2.7	3.0	3.0	3.0	2.9%
ESCO [16]	8,756,008	1,595	HDT	216.2	236.3	225.4	224.0	235.3	215.6	14.1%
			COTTAS	130.7	131.2	158.9	168.1	164.3	164.4	9.5%
LPWC [24]	10,648,236	2,583	HDT	720.9	744.9	735.6	727.6	742.3	718.0	9.5%
			COTTAS	220.2	220.0	258.2	262.6	261.9	263.3	3.8%
ArCoKG [14]	128,571,853	34,570	HDT	1,323	1,309	1,498	1,399	1,282	1,258	3.8%
			COTTAS	265.3	268.5	361.9	372.8	325.0	324.9	0.9%

Ratio (ACR), defined as the percentage of the uncompressed file size occupied by the compressed file. Lower ACR values indicate better compression. COTTAS achieves an ACR below 10% for all datasets, reaching as low as 0.9% in the best case. For several datasets, COTTAS indexes are more than three times smaller than their HDT counterparts. Additionally, no single COTTAS index configuration consistently outperforms the others in terms of space efficiency, suggesting that compression effectiveness is dataset-dependent.

For applications constrained by storage space where querying performance is not prioritary—such as archiving—COTTAS is the preferred alternative. Note that compression can be further tuned in Parquet, and space-performance trade-offs can be made in COTTAS. For instance, lowering the compression level of Zstandard can speed up decompression during file scans, at the cost of a lower compression ratio. In the extreme case, block compression could be disabled and rely entirely on lightweight compression encodings for maximizing performance.

4.4 Filters and Indexes

In COTTAS, BRIs are explicitly selected by users, while Bloom filters are created automatically based on the characteristics of the data. Table 3 reports the number of filters generated for WDBench and DBpedia under different index configurations. We observe that the presence of filters correlates with the choice of encoding strategies (as shown earlier in Table 1). In particular, Bloom filters

Table 3: The use of Bloom filters by COTTAS for the WDBench and DBpedia datasets with all possible indexes. Values correspond to the number of column chunks with a Bloom filter. The total number of column chunks for each dataset is also reported to illustrate the filter availability proportion.

Dataset	# col. chunks	Column	SPO	SOP	PSO	POS	OSP	OPS
WDBench	10,222	s	21	21	36	0	0	0
		p	8,146	8,153	10,222	10,221	10,079	10,111
		o	0	0	1,173	2,942	2,053	2,055
DBpedia	2,007	s	98	101	35	0	0	0
		p	747	740	2,007	2,007	1,681	1,689
		o	0	0	128	162	158	163

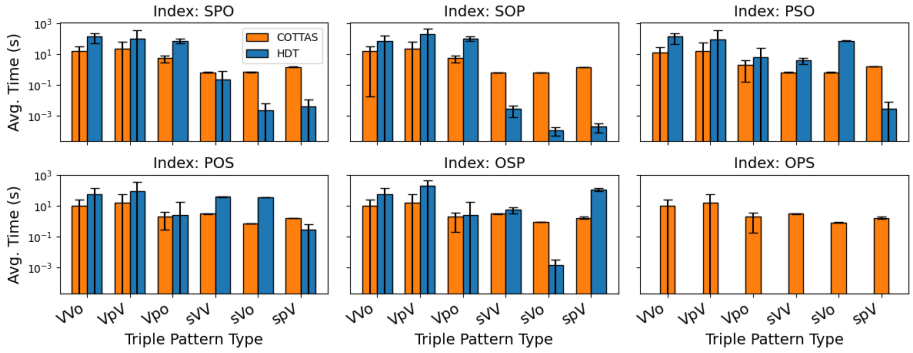

Fig. 3: Average execution time for different **triple pattern** types and indexes over **WDBench** in the COTTAS and HDT formats. Time is reported in seconds using log scale. Bars are missing for indexes that failed in their creation.

are typically created when the `RLE_DICT` encoding is used, since this encoding results in low cardinality within column chunks, making filters both space-efficient and effective for pruning data during query evaluation.

More aggressive use of filters in COTTAS could be enabled by increasing the NDV ratio required to apply dictionary encoding. As COTTAS files are much smaller than HDT, this can be done while using less space than HDT. Also, the space difference between COTTAS and HDT allows for the incorporation of other advanced indexing and filtering structures to accelerate data access.

4.5 Triple Pattern Scan Performance

We evaluate the performance of triple pattern queries using COTTAS across different index configurations and compare the results with HDT. Figures 3 and 4 show the average query evaluation times for each type of pattern for WDBench

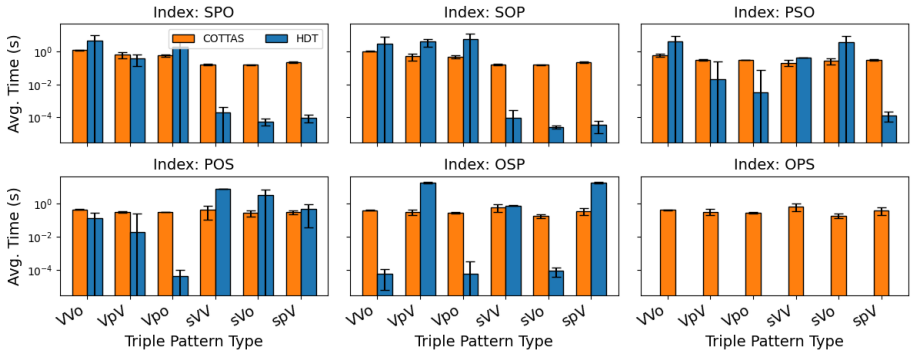

Fig. 4: Average execution time for different **triple pattern** types and indexes over **DBpedia** in the COTTAS and HDT formats. Time is reported in seconds using a log scale. Bars are missing for indexes that failed in their creation.

and DBpedia, respectively. For WDBench, COTTAS is more rapid on average for 21 out of 30 combinations of triple pattern type and index (excluding `OPS` as it is not available for HDT). The `SPO` and `SOP` indexes favor HDT, where it significantly outperforms COTTAS in cases of exact index-pattern alignment. In contrast, for the `PSO` and `OSP` orderings, HDT only outperformed COTTAS when two bound terms aligned with the index. For DBpedia, COTTAS is fastest in 14 out of the 30 combinations. Overall, the results indicate that index selection has a much stronger impact on HDT's performance, often resulting in dramatic variations (orders of magnitude) depending on index-pattern alignment. In contrast, COTTAS exhibits more consistent performance across different indexes, with improvements introduced by indexing, but without the same level of performance volatility observed in HDT.

Although there is no clear winner regarding performance, the distinct behaviour of both solutions may determine their suitability for specific applications. HDT is notably more efficient than COTTAS when the index matches the triple pattern, but it becomes significantly slower than COTTAS when it does not. This makes COTTAS preferable for time-constrained applications, regardless of the triple pattern type. Conversely, HDT might be preferable in applications that require optimizing specific patterns. As previously mentioned, COTTAS files use far less space than HDT, and thus, additional indexes and filters could be stored to catch up to HDT.

4.6 Join Performance

Finally, we evaluate and compare the join performance in COTTAS and HDT formats using various join patterns provided by the DBpedia benchmark [12]. The testbed defines two query sets—small and large—based on the size of their result sets, and categorizes joins between two triple patterns as follows:

Fig. 5: Average query execution time for the different types of **small joins** and indexes over **DBpedia** in the COTTAS and HDT formats. Time is reported in seconds using log scale. Bars are missing for indexes that failed in their creation.

- No unbound predicates:
  - No unbound subject or object: **A** ($s\ p_1\ ?x\ .\ ?x\ p_2\ o$).
  - One unbound subject or object: **B** ($?s\ p_1\ ?x\ .\ ?x\ p_2\ o$).
  - Two unbound subject or object: **C** ($?s\ p_1\ ?x\ .\ ?x\ p_2\ ?o$).
- One unbound predicate:
  - No unbound subject or object: **D** ($s\ p_1\ ?x\ .\ ?x\ ?p_2\ o$).
  - One unbound subject or object:
    - $\mathbf{E_1}$ ($?s\ p_1\ ?x\ .\ ?x\ ?p_2\ o$).
    - $\mathbf{E_2}$ ($?s\ ?p_1\ ?x\ .\ ?x\ p_2\ o$).
  - Two unbound subject or object: **F** ($?s\ p_1\ ?x\ .\ ?x\ ?p_2\ ?o$).
- Two unbound predicates:
  - No unbound subject or object: **G** ($s\ ?p_1\ ?x\ .\ ?x\ ?p_2\ o$).
  - One unbound subject or object: **H** ($?s\ ?p_1\ ?x\ .\ ?x\ ?p_2\ o$).

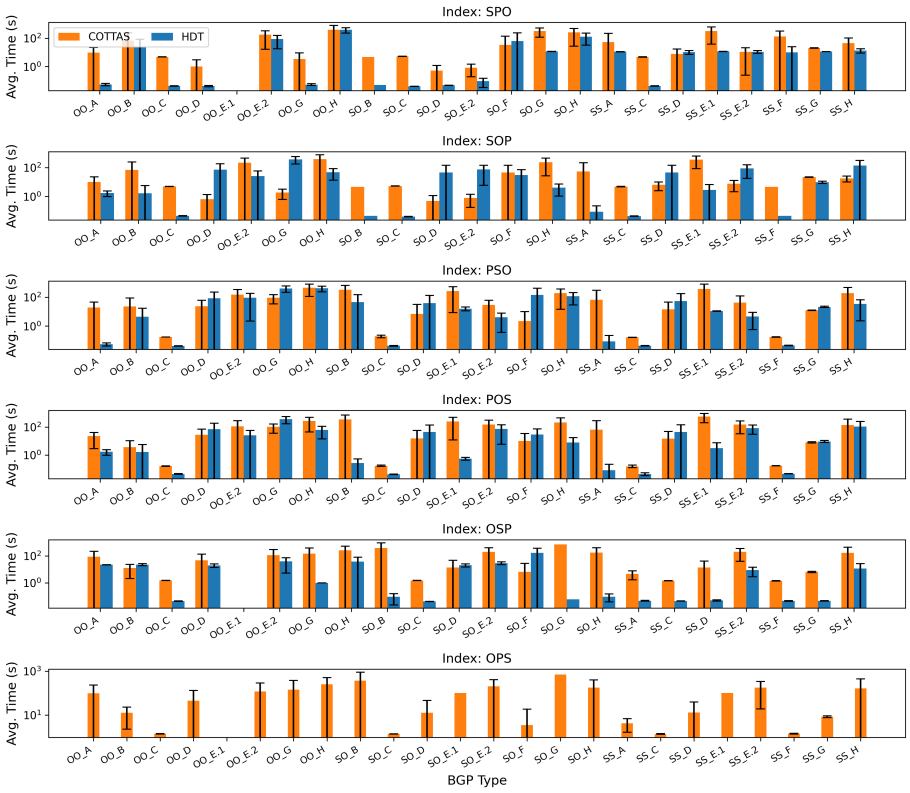

Fig. 6: Average execution time for the different type of **big joins** and indexes over the **DBpedia** dataset in the COTTAS and HDT formats. Time is reported in seconds using log scale. Bars are missing for indexes that failed in their creation.

Additionally, each join type has variants based on join positions:

- Subject-subject: **SS** ($?x\ p_1\ o_1\ .\ ?x\ p_2\ o_2$).
- Subject-object: **SO** ($s_1\ p_1\ ?x\ .\ ?x\ p_2\ o_2$).
- Object-object: **OO** ($s_1\ p_1\ ?x\ .\ s_2\ p_2\ ?x$).

Figures 5 and 6 show the results for the small and big query sets, respectively. The average query execution time for each join pattern is reported, excluding queries that resulted in timeouts or errors for COTTAS or HDT. In this experiment, join execution is delegated to RDFLib for both COTTAS and HDT. However, Parquet is optimized for vectorized query execution, which RDFLib does not exploit, resulting in more cases where HDT achieved lower execution times than COTTAS. The performance of join execution in COTTAS could be improved by pushing down its execution to DuckDB via SQL unfolding and leveraging vectorized execution.

5 Related Work

In this section, we review related works on RDF compression and discuss their relevance to our approach. Section 5.1 focuses on RDF compression formats, with special emphasis on HDT and its successors, while Section 5.2 covers approaches based on columnar storage formats.

5.1 RDF Compression

HDT [21] is the state-of-the-art format for compressing and querying RDF graphs. It comprises three components: (i) the *header* stores file metadata, including dataset statistics, publication, and format information; (ii) the *dictionary* maps RDF terms to IDs; and (iii) the *triples* consists of tuples of term IDs. There are notable similarities between HDT and COTTAS. Both formats record file-level metadata, and COTTAS additionally records row group- and page-level metadata. COTTAS also maintains a dictionary of RDF terms when a column chunk is encoded with `RLE_DICTIONARY`. However, while the dictionary in HDT is global to the entire file, COTTAS uses a separate dictionary for each column chunk. The HDT dictionary is divided into four sections: subjects, predicates, objects, and terms occurring as both subject and object. Given that COTTAS dictionaries apply to a single column chunk, they also encode terms in a specific position. HDT offers three encodings for the triples: *plain*, *compact*, and *bitmap*. Bitmap is the most efficient, and implies sorting by subject, predicate, and object, and creating nested predicate and object adjacency lists, similar to the Turtle RDF serialization. Sequences of triple IDs for predicates and objects are stored alongside bitmaps that determine the shape of the graph. This allows for efficient resolution of triple patterns with a bound subject or with a bound subject and predicate. Other triple patterns can also be efficiently evaluated using different orderings for the adjacency lists, similar to BRIs indexes in COTTAS. HDT also supports external indexes [33,29] to efficiently evaluate all possible triple patterns. However, creating these indexes is costly in terms of space and time, and HDT files are already larger than their counterparts in COTTAS.

HDTQ [22] extends HDT to support RDF datasets by adding a section in the dictionary for storing named graphs and incorporating the *quad information* component to mark the presence of triples within each graph of the RDF dataset. RDF datasets are straightforwardly supported in COTTAS using a quad table.

HDTCat [17] efficiently merges two HDT files by exploiting their initial ordering (assuming they use the same triple ordering). HDTCat has been further extended to merge an arbitrary number of files and to remove triples, enabling updates [47]. These operations are supported in COTTAS by pushing down computation to SQL. Notably, multiple COTTAS files can be simultaneously queried by using glob patterns without the need to merge them.

Other RDF compression techniques, such as K2Triples [25], Permuted Trie Index [38], and RDFCSA [12], focus on the structural part of the RDF graph (indeed, they serve as a replacement of the HDT triples component). However, their implementations are research prototypes and not industry-ready. A major

advantage of COTTAS is that it is built on top of a popular columnar format and can be implemented in other programming languages beyond Python with minimal effort by reusing existing Parquet implementations.

5.2 Columnar Compression

Column-oriented compressed formats emerged with the rise of Big Data nearly two decades ago. The most popular formats today are Parquet, ORC, and CarbonData, all of which have a PAX-based [2] layout. Recent studies [48,32] have empirically evaluated Parquet and ORC, laying the groundwork for future, more efficient columnar formats. COTTAS can directly benefit from new columnar formats (e.g., BtrBlocks [31] and Vortex [44]), as well as new lightweight compression schemes (e.g., FSST [11]), and indexing and filtering techniques (e.g., column imprints [42] and SuRF [49]).

Parquet was previously used for the scalable and distributed processing of RDF with Big Data frameworks. Examples include Sempala [40], which uses Impala as a processing engine, and S2RDF [41], which uses Spark. However, these works focus on distributed RDF processing, whereas our work emphasizes the use of Parquet for RDF compression, exchange, and querying.

6 Conclusion and Future Directions

In this paper, we explored RDF compression using column-oriented storage and introduced COTTAS, a file format that encodes RDF as a triple table in Parquet. COTTAS leverages Parquet's lightweight compression schemes along with the Zstandard algorithm to achieve high space efficiency. It also exploits block range indexes and Bloom filters to enable fast triple pattern evaluation over compressed RDF graphs. Additionally, we presented pycottas, an open-source Python library for creating and managing COTTAS files. It supports the compression of RDF in plain-text serializations and enables efficient querying by translating triple patterns into SQL queries over COTTAS files. This implementation facilitates seamless integration of COTTAS in existing RDF data processing ecosystems and lowers the barrier to adoption.

Experiments using the WDBench and DBpedia benchmarks show that COTTAS reduces storage requirements by around 50% with respect to HDT, and generally outperforms HDT in triple pattern evaluation, except when these align with the index. COTTAS presents a more stable behavior than HDT, whose performance differs by orders of magnitude for different types of triple patterns.

Future research should analyze COTTAS using other column-oriented formats, and more lightweight compression encodings, indexes, and filters for enhanced compression and querying of RDF graphs. Although COTTAS does not have a large memory footprint, ways to optimize its memory consumption can be further researched. Investigating the pushdown of joins into SQL and leveraging vectorization capabilities for efficient join execution by unfolding them to SQL are also important areas for future work. In addition, it is crucial to study

how to extend COTTAS to support the upcoming RDF 1.2 standard [28], for which recent advances have been made for HDT [27]. We also plan to integrate pycottas with the KG construction system Morph-KGC [7] to directly generate compressed, ready-for-consumption RDF graphs from heterogeneous data.

Acknowledgements

Arenas-Guerrero received partial financial support from the INESData project (Infrastructure to Investigate Data Spaces in Distributed Environments at UPM - TSI-063100-2022-0001), a project funded under the UNICO I+D CLOUD call by the Ministry for Digital Transformation and the Civil Service, within the framework of the recovery plan PRTR financed by the European Union (NextGenerationEU). Ferrada was funded by ANID - Millennium Science Initiative Program - Code ICN17_002; by CENIA FB210017, Basal ANID; by FONDECYT Grant 11251322; and by the Vicerrectoría de Investigación y Desarrollo (VID) of Universidad de Chile, project code UI-016/24.

References

1. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: Scalable semantic web data management using vertical partitioning. In: Proceedings of the 33rd International Conference on Very Large Data Bases. p. 411–422. VLDB Endowment (2007), https://www.vldb.org/conf/2007/papers/research/p411-abadi.pdf
2. Ailamaki, A., DeWitt, D.J., Hill, M.D., Skounakis, M.: Weaving Relations for Cache Performance. In: Proceedings of the 27th International Conference on Very Large Data Bases. p. 169–180 (2001), https://www.vldb.org/conf/2001/P169.pdf
3. Alakuijala, J., Farruggia, A., Ferragina, P., Kliuchnikov, E., Obryk, R., Szabadka, Z., Vandevenne, L.: Brotli: A General-Purpose Data Compressor. ACM Trans. Inf. Syst. **37**(1) (2018). https://doi.org/10.1145/3231935
4. Ali, W., Saleem, M., Yao, B., Hogan, A., Ngomo, A.C.N.: A survey of RDF stores & SPARQL engines for querying knowledge graphs. The VLDB Journal **31**(3) (2022). https://doi.org/10.1007/s00778-021-00711-3
5. Angles, R., Aranda, C.B., Hogan, A., Rojas, C., Vrgoč, D.: WDBench: A Wikidata Graph Query Benchmark. In: Proceedings of the 21st International Semantic Web Conference. pp. 714–731. Springer International Publishing (2022). https://doi.org/10.1007/978-3-031-19433-7_41
6. Arenas-Guerrero, J.: arenas-guerrero-julian/pycottas (2025). https://doi.org/10.5281/zenodo.15350990
7. Arenas-Guerrero, J., Chaves-Fraga, D., Toledo, J., Pérez, M.S., Corcho, O.: Morph-KGC: Scalable knowledge graph materialization with mapping partitions. Semantic Web **15**(1), 1–20 (2024). https://doi.org/10.3233/SW-223135
8. Arenas-Guerrero, J., Corcho, O., Pérez, M.S.: Intermediate triple table: A general architecture for virtual knowledge graphs. Knowledge-Based Systems **314**, 113179 (2025). https://doi.org/10.1016/j.knosys.2025.113179

9. Binnig, C., Hildenbrand, S., Färber, F.: Dictionary-based order-preserving string compression for main memory column stores. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data. p. 283–296. ACM (2009). https://doi.org/10.1145/1559845.1559877
10. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM **13**(7), 422–426 (1970). https://doi.org/10.1145/362686.362692
11. Boncz, P., Neumann, T., Leis, V.: FSST: Fast Random Access String Compression. Proc. VLDB Endow. **13**(12), 2649–2661 (2020). https://doi.org/10.14778/3407790.3407851
12. Brisaboa, N.R., Cerdeira-Pena, A., de Bernardo, G., Fariña, A., Navarro, G.: Space/time-efficient RDF stores based on circular suffix sorting. The Journal of Supercomputing **79**(5), 5643–5683 (2023). https://doi.org/10.1007/s11227-022-04890-w
13. Burel, G., Mensio, M., Peskine, Y., Troncy, R., Papotti, P., Alani, H.: CimpleKG: A Continuously Updated Knowledge Graph on Misinformation, Factors and Fact-Checks. In: Proceedings of the 23rd International Semantic Web Conference. pp. 97–114. Springer Nature Switzerland (2025). https://doi.org/10.1007/978-3-031-77847-6_6
14. Carriero, V.A., Gangemi, A., Mancinelli, M.L., Marinucci, L., Nuzzolese, A.G., Presutti, V., Veninata, C.: ArCo: The Italian Cultural Heritage Knowledge Graph. In: Proceedings of the 18th International Semantic Web Conference. pp. 36–52. Springer International Publishing (2019). https://doi.org/10.1007/978-3-030-30796-7_3
15. Collet, Y., Kucherawy, M.: RFC 8878: Zstandard Compression and the 'application/zstd' Media Type (2021). https://doi.org/10.17487/RFC8878
16. De Smedt, J., Le Vrang, M., Papantoniou, A.: ESCO: Towards a Semantic Web for the European Labor Market. In: Proceedings of the Workshop on Linked Data on the Web 2015. vol. 1409. CEUR Workshop Proceedings (2015), https://ceur-ws.org/Vol-1409/paper-10.pdf
17. Diefenbach, D., Giménez-García, J.M.: HDTCat: Let's Make HDT Generation Scale. In: Proceedings of the 19th International Semantic Web Conference. pp. 18–33. Springer International Publishing (2020). https://doi.org/10.1007/978-3-030-62466-8_2
18. Erling, O., Mikhailov, I.: RDF Support in the Virtuoso DBMS. In: Networked Knowledge - Networked Media: Integrating Knowledge Management, New Media Technologies and Semantic Systems, pp. 7–24. Springer Berlin Heidelberg (2009). https://doi.org/10.1007/978-3-642-02184-8_2
19. Färber, M.: The Microsoft Academic Knowledge Graph: A Linked Data Source with 8 Billion Triples of Scholarly Data. In: Proceedings of the 18th International Semantic Web Conference. pp. 113–129. Springer International Publishing (2019). https://doi.org/10.1007/978-3-030-30796-7_8
20. Färber, M., Lamprecht, D., Krause, J., Aung, L., Haase, P.: SemOpenAlex: The Scientific Landscape in 26 Billion RDF Triples. In: Proceedings of the 22nd International Semantic Web Conference. pp. 94–112. Springer Nature Switzerland (2023). https://doi.org/10.1007/978-3-030-30796-7_8
21. Fernández, J.D., Martínez-Prieto, M.A., Gutiérrez, C., Polleres, A., Arias, M.: Binary RDF representation for publication and exchange (HDT). Journal of Web Semantics **19**, 22–41 (2013). https://doi.org/10.1016/j.websem.2013.01.002

22. Fernández, J.D., Martínez-Prieto, M.A., Polleres, A., Reindorf, J.: HDTQ: Managing RDF Datasets in Compressed Space. In: Proceedings of the 15th Extended Semantic Web Conference. pp. 191–208. Springer International Publishing (2018). https://doi.org/10.1007/978-3-319-93417-4_13
23. Färber, M., Lamprecht, D.: The data set knowledge graph: Creating a linked open data source for data sets. Quantitative Science Studies **2**(4), 1324–1355 (2021). https://doi.org/10.1162/qss_a_00161
24. Färber, M., Lamprecht, D.: Linked Papers With Code: The Latest in Machine Learning as an RDF Knowledge Graph. In: The 22nd International Semantic Web Conference, P&D. vol. 3632. CEUR Workshop Proceedings (2024), https://ceur-ws.org/Vol-3632/ISWC2023_paper_467.pdf
25. Álvarez García, S., Brisaboa, N., Fernández, J.D., Martínez-Prieto, M.A., Navarro, G.: Compressed vertical partitioning for efficient RDF management. Knowledge and Information Systems **44**(2), 439–474 (2015). https://doi.org/10.1007/s10115-014-0770-y
26. Álvarez García, S., Brisaboa, N.R., Fernández, J.D., Martínez-Prieto, M.A.: Compressed k2-Triples for Full-In-Memory RDF Engines. In: Proceedings of the 17th Americas Conference on Information Systems (2011), https://aisel.aisnet.org/amcis2011_submissions/350
27. Gimenez-Garcia, J.M., Gautrais, T., Fernández, J.D., Martínez-Prieto, M.A.: Compact Encoding of Reified Triples Using HDTr. In: Proceedings of the 22nd International Semantic Web Conference. pp. 309–327. Springer Nature Switzerland (2023). https://doi.org/10.1007/978-3-031-47240-4_17
28. Hartig, O., Champin, P.A., Kellogg, G., Seaborne, A.: RDF 1.2 Concepts and Abstract Syntax. W3C Working Draft, World Wide Web Consortium (2025), https://www.w3.org/TR/rdf12-concepts/
29. Hernández-Illera, A., Martínez-Prieto, M.A., Fernández, J.D., Fariña, A.: iHDT++: improving HDT for SPARQL triple pattern resolution. Journal of Intelligent & Fuzzy Systems **39**(2), 2249–2261 (2020). https://doi.org/10.3233/JIFS-179888
30. Krech, D., Grimnes, G.A., Higgins, G., Car, N., Hees, J., Aucamp, I., Lindström, N., Arndt, N., Sommer, A., Chuc, E., Herman, I., Nelson, A., McCusker, J., Gillespie, T., Kluyver, T., Ludwig, F., Champin, P.A., Watts, M., Holzer, U., Summers, E., Morriss, W., Winston, D., Perttula, D., Kovacevic, F., Chateauneu, R., Solbrig, H., Cogrel, B., Stuart, V.: RDFLib (2025). https://doi.org/10.5281/zenodo.6845245, https://github.com/RDFLib/rdflib
31. Kuschewski, M., Sauerwein, D., Alhomssi, A., Leis, V.: BtrBlocks: Efficient Columnar Compression for Data Lakes. Proc. ACM Manag. Data **1**(2) (2023). https://doi.org/10.1145/3589263
32. Liu, C., Pavlenko, A., Interlandi, M., Haynes, B.: A Deep Dive into Common Open Formats for Analytical DBMSs. Proc. VLDB Endow. **16**(11), 3044–3056 (2023). https://doi.org/10.14778/3611479.3611507
33. Martínez-Prieto, M.A., Arias Gallego, M., Fernández, J.D.: Exchange and Consumption of Huge RDF Data. In: Proceedings of the 9th Extended Semantic Web Conference. pp. 437–452. Springer Berlin Heidelberg (2012). https://doi.org/10.1007/978-3-642-30284-8_36
34. Minier, T.: rdflib-hdt (2025), https://github.com/RDFLib/rdflib-hdt
35. Morsey, M., Lehmann, J., Auer, S., Ngomo, A.C.N.: DBpedia SPARQL Benchmark – Performance Assessment with Real Queries on Real Data. In: Proceedings of the 10th International Semantic Web Conference. pp. 454–469. Springer Berlin Heidelberg (2011). https://doi.org/10.1007/978-3-642-25073-6_29

36. Neumann, T., Weikum, G.: The RDF-3X engine for scalable management of RDF data. The VLDB Journal **19**(1), 91–113 (2010). https://doi.org/10.1007/s00778-009-0165-y
37. Pellissier Tanon, T.: Oxigraph (2025). https://doi.org/10.5281/zenodo.7408022, https://github.com/oxigraph/oxigraph
38. Perego, R., Pibiri, G.E., Venturini, R.: Compressed Indexes for Fast Search of Semantic Data. IEEE Transactions on Knowledge and Data Engineering **33**(9), 3187–3198 (2021). https://doi.org/10.1109/TKDE.2020.2966609
39. Raasveldt, M., Mühleisen, H.: DuckDB: An Embeddable Analytical Database. In: Proceedings of the 2019 International Conference on Management of Data. p. 1981–1984. Association for Computing Machinery (2019). https://doi.org/10.1145/3299869.3320212
40. Schätzle, A., Przyjaciel-Zablocki, M., Neu, A., Lausen, G.: Sempala: Interactive SPARQL Query Processing on Hadoop. In: Proceedings of the 13th International Semantic Web Conference. pp. 164–179. Springer International Publishing (2014). https://doi.org/10.1007/978-3-319-11964-9_11
41. Schätzle, A., Przyjaciel-Zablocki, M., Skilevic, S., Lausen, G.: S2RDF: RDF Querying with SPARQL on Spark. Proc. VLDB Endow. **9**(10), 804–815 (2016). https://doi.org/10.14778/2977797.2977806
42. Sidirourgos, L., Kersten, M.: Column imprints: a secondary index structure. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. p. 893–904. ACM (2013). https://doi.org/10.1145/2463676.2465306
43. Stocker, M., Oelen, A., Jaradeh, M.Y., Haris, M., Oghli, O.A., Heidari, G., Hussein, H., Lorenz, A.L., Kabenamualu, S., Farfar, K.E., Prinz, M., Karras, O., D'Souza, J., Vogt, L., Auer, S.: FAIR scientific information with the Open Research Knowledge Graph. FAIR Connect **1**(1), 19–21 (2023). https://doi.org/10.3233/FC-221513
44. Vortex: Vortex, https://github.com/vortex-data/vortex
45. Vrandečić, D., Krötzsch, M.: Wikidata: A Free Collaborative Knowledgebase. Communications of the ACM **57**(10), 78–85 (2014). https://doi.org/10.1145/2629489, 10.1145/2629489
46. Wilke, A., Ngonga Ngomo, A.C.: LauNuts: A Knowledge Graph to Identify and Compare Geographic Regions in the European Union. In: Proceedings of the 20th Extended Semantic Web Conference. pp. 408–418. Springer Nature Switzerland (2023). https://doi.org/10.1007/978-3-031-33455-9_24
47. Willerval, A., Diefenbach, D., Bonifati, A.: Generate and Update Large HDT RDF Knowledge Graphs on Commodity Hardware. In: Proceedings of the 21st Extended Semantic Web Conference. pp. 128–144. Springer Nature Switzerland (2024). https://doi.org/10.1007/978-3-031-60635-9_8
48. Zeng, X., Hui, Y., Shen, J., Pavlo, A., McKinney, W., Zhang, H.: An Empirical Evaluation of Columnar Storage Formats. Proc. VLDB Endow. **17**(2), 148–161 (2023). https://doi.org/10.14778/3626292.3626298
49. Zhang, H., Lim, H., Leis, V., Andersen, D.G., Kaminsky, M., Keeton, K., Pavlo, A.: SuRF: Practical Range Query Filtering with Fast Succinct Tries. In: Proceedings of the 2018 International Conference on Management of Data. p. 323–336. ACM (2018). https://doi.org/10.1145/3183713.3196931