Managing and Compiling Data Dependencies for Semantic Applications Using Databus Client

Johannes Frey(✉), Fabian Götz, Marvin Hofer, and Sebastian Hellmann

Knowledge Integration and Linked Data Technologies (KILT/AKSW), DBpedia Association/InfAI, Leipzig University, Leipzig, Germany
{frey,gotz,hofer,hellmann}@informatik.uni-leipzig.de

Abstract. Realizing a data-driven application or workflow, that consumes bulk data files from the Web, poses a multitude of challenges ranging from sustainable dependency management supporting automatic updates, to dealing with compression, serialization format, and data model variety. In this work, we present an approach using the novel Databus Client, which is backed by the DBpedia Databus - a data asset release management platform inspired by paradigms and techniques successfully applied in software release management. The approach shifts effort from the publisher to the client while making data consumption and dependency management easier and more unified as a whole. The client leverages 4 layers (download, compression, format, and mapping) that tackle individual challenges and offers a fully automated way for extracting and compiling data assets from the DBpedia Databus, given one command and a flexible dependency configuration using SPARQL or Databus Collections. The current vertical-sliced implementation supports format conversion within as well as mapping between RDF triples, RDF quads, and CSV/TSV files. We developed an evaluation strategy for the format conversion and mapping functionality using so-called round trip tests.

Keywords: Data dependency management · Data compilation · Data release management platform · Metadata repository · ETL

1 Introduction

With the growing importance of transparent, reproducible, and FAIR publishing of research results as well as the rise of knowledge graphs for digital twins in corporate and research environments in general, there is on the one hand an urging demand for (research) data infrastructure and management platforms to publish and organize produced data assets. On the other hand, there is a huge potential for plenty of research that depends on workflows using a variety of internal and external data dependencies or that creates applications which consume large amounts of data and try to make use of it (e.g. AI-based algorithms).

One major reason for the introduction of the Semantic Web was to make data on the Web more useful for machines such that they could automatically

E. Garoufallou et al. (Eds.): MTSR 2021, CCIS 1537, pp. 114–125, 2022.
https://doi.org/10.1007/978-3-030-98876-0_10

discover, access, read, understand and process it. While the Linked Data design principles provide a guideline to browse Linked (Open) Data in an automated and decentralized fashion, when it comes to workflows and applications that are driven by a variety of high volume data (bulk file dumps) and that aim to be automatically deployed and updated, several challenges arise with respect to managing and consuming these data dependencies.

Although data repositories or management platforms with rich homogeneous metadata catalogs like the DBpedia Databus [2] allow to manage, find, and access files in a unified way, difficulties arise if consumers want to use data from different publishers and domains. These files can be released in various serialization formats (e.g. RDF can be represented in more than 8 formats) and compression variants, that typically can not be read all by an application or workflow without any prior conversion. Moreover, in many research disciplines, data is stored in relational databases and exported into tabular-structured data formats (e.g. CSV) or specialized community-specific formats. Loading this data alongside knowledge graphs requires a mapping process to be performed on the consumer side. However, this mapping effort is usually lost on the local infrastructure or in a GitHub repository, where it is hard to find and reuse. Even if data dependencies are not fed manually into the system, plenty of custom scripted solutions per application becoming quickly chaotic tend to grow, making applications harder to maintain and reproduce, finally leaving users and consumers with the resulting decreased reusability and unclear provenance.

While some of the conversion to popular formats is already performed by publishers, we argue that this should not be the burden of the data provider in general. Instead, we envision a software client, that - given a dependency configuration - can dump any data asset registered on a data management platform and converts it to a format supported by the target infrastructure. A client that can execute different applications and ingest *compiled* data automatically, such that data is only one command away, like in traditional software dependency, built, and package management systems. Analogous to compiling of software, we define *compiling* of data as the process that converts, transforms or translates data geared to the needs of a specific target application.

This work introduces a conceptual approach implemented within the DBpedia Databus Client, that facilitates a more natural consumption and compiling of data from the DBpedia Databus and brings us one step closer towards our vision. Our main contributions are: a modular and extendable client that leads in combination with the Databus platform to less format conversion publishing effort (w.r.t. storage and time), enables easier and systematic data consumption with less conversion issues, allows for realizing data-driven apps using automatically updating data dependencies with clear provenance, and improves findability and reuse of mapping definitions.

The remainder of the paper is structured as follows: in the next section we sketch the process of data release and dependency management leveraging the DBpedia Databus. In Sect. 3 we present the conceptual design of the client, followed by the description of its implementation in Sect. 4. We evaluate the

approach in Sect. 5 and compare it to other related work in Sect. 6. We conclude with a discussion and future work.

2 DBpedia Databus Release Management Platform

Inspired by paradigms and techniques successfully applied in (Java) software release management and deployment, we started to think how we could transfer these to data engineering and management. Additionally driven by the need for a flexible, heavily automatable dataset management and publishing platform for a new and more agile DBpedia release cycle [5], we initiated the development of the DBpedia Databus Platform[1] over 3 years ago.

The Databus [2] uses the Apache Maven concept hierarchy *group*, *artifact*, *version* and ports it to a Linked Data based platform, in order to manage data pipelines and enable automated publishing and consumption of data. *Artifacts* form an abstract identity of a dataset with a stable dataset ID and can be used as entry point to discover all *versions*. A *version* usually contains the same set of *files* for each release. These concepts are embedded in the personal IRI (Internationalized Resource Identifier) space that is issued by the Databus for every user. The full IRI `https://databus.dbpedia.org/<publisher>/<group>/<artifact>/<version>/<file>` can be used as a stable ID for a particular dataset *file* in a particular *version*. *Groups* provide a coarse modularization or bundling of datasets forming (useful) units to improve overview and consumption. The overall structure is very flexible as software libraries, but once defined should be as fixed as software to prevent applications from breaking, if they update on a new version.

Additionally, every *file* can be described by key-value records (so-called *content-variants*) which allow another level of granularity as well as addressing and querying for particular files (e.g. split labels of entities based on their language into different files).

Databus metadata is represented with an extension of the DataID core vocabulary for *group*, *artifact*, *version*, and *file* entities that allows for flexible, fine-grained, as well as unified metadata access using SPARQL. Based on `dcat:downloadURL` links in this metadata, Databus file IDs form a stable (but redirectable) abstraction layer independent of file hosting (similar to w3id.org). Provenance can be added by specifying Databus IDs of the input data on *version* or *file* level.

Moreover, users can create automatically updating or stable catalogs of data assets via so-called Databus collections[2], which encode the information need or asset selection via SPARQL queries. Collections can be created and refined via a faceted browsing UI on the Databus Website similar to a shopping cart and used as easy way to specify data input dependencies while recording provenance.

An example collection which consists of 2 *artifacts* from 2 different publishers (a crawl of the German Energy Market Core Register (MaStR) filtered to files

[1] https://databus.dbpedia.org.
[2] https://www.dbpedia.org/blog/databus-collections-feature/.

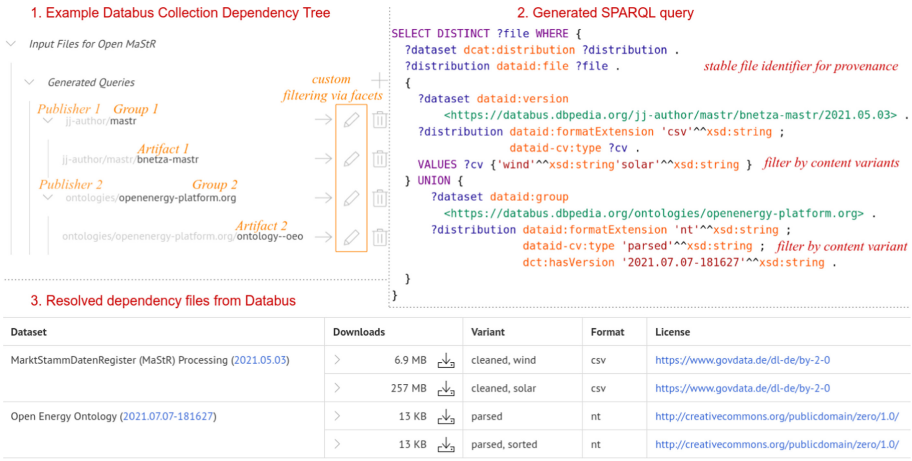

Fig. 1. Data dependency definition using Databus Collections

with wind and solar units, as well as parsed files of the Open Energy Ontology from DBpedia Archivo [4]) is shown as dependency tree in Fig. 1. Using the dedicated collection identifier, it is possible to retrieve the generated SPARQL query which encodes the dependency tree and optional filters based on the facet selection. When issuing this query against the Databus SPARQL endpoint, a list of Databus files will be returned, which is also displayed in the Collection view on the Databus website.

3 Databus Client Concept

The Databus Client is designed in a modular way to achieve high reusability, which means that the components and functionalities such as the downloading component, and compression converter can be used separately and interchangeably. It leverages 4 functionality layers depicted in Fig. 2.

The fundamental **Download-Layer** is supposed to download exact copies of data assets via the DBpedia Databus in a flexible way. It can be understood as a simple extraction phase of the ETL (Extract-Transform-Load) process. Moreover, it is supposed to persist the input data provenance by recording stable file identifiers and additional metadata. The data assets to be downloaded can be selected in a fine-grained way via an interoperable data dependency specification. and optional compiling configurations tailored to the needs of a consuming app or workflow.

If any conversion process is required, the **Compression-Layer** takes action. It sniffs for the input compression format and decompresses the file. If the input file format differs from the output file format, the decompressed file is passed to the Format-Layer. The Compression-Layer takes the decompressed file, which

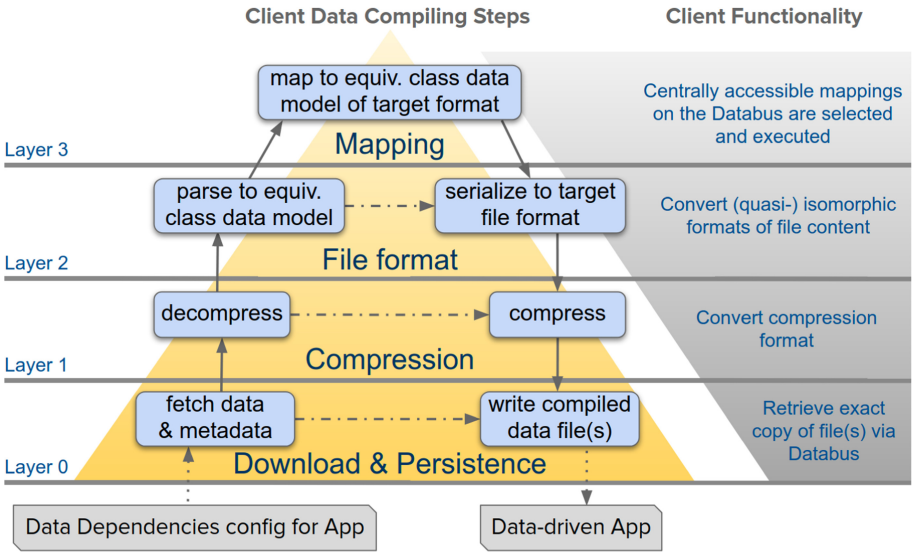

Fig. 2. Layers of the databus client data compiling process.

may be format converted by the Format-Layer or Mapping-Layer, and compresses it to the requested output compression format. This compressed file is passed back to the Download-Layer, after the conversion process has finished.

Within the data format conversion process, the Databus Client utilizes the Format-layer and the Mapping-Layer where required. The **Format-Layer** receives the uncompressed file and parses it to a unified internal data structure of the corresponding (format) equivalence class. Such an equivalence class contains all serialization formats that can be used interchangeably while representing the same amount of information, given a defined common data model for the class (e.g. a set of triples for RDF triple formats, a table of Strings for tabular-structured data formats). Subsequently, the Format-Layer serializes the internal data structure to the desired output file format. It passes the serialized data back to the Compression-Layer.

Whenever the input file format and the requested output file format are in different equivalence classes (e.g. Turtle/RDF triples and TSV/tabular-structured data), the **Mapping-Layer** is additionally used. However, it could also be used to manipulate the data of the same equivalence class (e.g. ontology mapping). With the aid of mapping configurations, the Mapping-Layer transforms the data represented using the internal data structure of the input equivalence class, to data of the internal data structure of the equivalence class of the target file format. After that process has finished, the data is passed back to the Format layer.

The Compression-Layer, File-Format-Layer, and Mapping-Layer represent the transformation-phase of the ETL process.

4 Implementation

We implemented a vertical slice of the four conceptional layers in the command-line tool *Databus Client*[3]. It is written in Scala and using Apache Maven. Subsequently, it is executable within a Java Virtual Machine (JVM) or a Docker[4] container, allowing it to be run on almost any machine and to be interoperable to a broad amount of applications. In addition, we provide a Maven package with interfaces to invoke the functions of the Databus Client from other JVM-based applications.

Depending on the data compilation command parameters, the client applies different methods at each layer, either passing already processed data (as file, stream, or object) to the next higher layer or returning it to the one beneath.

Layer 0 (Download & Persistence) manages the data handling between the Databus, the Compression Layer, and the local file system. Its implementation consists of two modules: 1) the Download-Module that queries file metadata using the Databus, retrieves the files by accessing their download URLs, and finally verifies the download process; 2) the Persistence-Module which generates local provenance metadata and stores the target files in the correct file structure.

The Databus Client can download any file registered on the Databus as exact copy and verifies it according to its corresponding Databus metadata (using the SHA256 checksum). The files to be downloaded are specified via a SPARQL query or a Databus collection.

The Persistence-Module receives the target data as stream or file from either the Compression-Layer or directly from the Download-Module and stores the data on the local file system reproducing a directory structure similar to the Databus hierarchy `/<account>/<group>/<artifact>/<version>/<fileName>`. In addition, it creates a summary file tracking provenance of the Databus file identifiers and processing information, like applied mappings.

The **Compression-Layer** is implemented using Apache Commons Compress[5]. This library provides functions to detect and decompress several file compression formats, like *gzip*, *bzip2*, *snappy-framed*, *xz*, *deflate*, *lzma*, and *zstd*. The Compression-Module can either read/write from the local file system or a byte stream, getting data from the Download-Module and passing data to the Persistence-Module.

The **Format-Layer** that handles the format conversion within an equivalence class, currently supports three equivalence classes: 1) quad-based RDF formats, 2) triple-based RDF formats, and 3) Tabular structure formats.

For RDF formats, the implementation uses Apache Jena[6] either leveraging Jena's StreamRDF in combination with Apache SPARK[7] or the RDF Model/Dataset API, supporting various formats (see Table 1). Apache Spark

[3] https://github.com/dbpedia/databus-client.
[4] https://www.docker.com/.
[5] https://commons.apache.org/proper/commons-compress/.
[6] https://jena.apache.org/.
[7] https://spark.apache.org/.

utilizes Resilient Distributed Datasets (RDD) [10], which provide several relational algebra operations to transform and combine this kind of data structure in a salable way. A significant benefit of an RDD is that it can be partitioned and distributed over several computing nodes, including swapping (spill) partitions to disk to avoid out-of-memory exceptions that larger datasets could introduce. The inner type of an RDD can be any serializable JVM Object. In our case, the internal data structure of triple-based RDF formats is an instance of RDD[Triple], and the internal data structure of quad-based RDF formats is an instance of RDD[Quad].

For tabular-structured data, the conversion methods of Apache Spark's IO library are utilized, allowing to handle configurable CSV formats (specified by delimiter and escape characters). The internal representation of Tabular structured data is an instance of RDD[Row]. The Format-Layer is either passing the internal representation of an equivalence class to the Mapping-Layer, or a stream of the target format back to the Compression-Layer.

Mapping-Layer. To convert formats between different equivalence classes, the Format-Layer passes the internal data structure of an equivalence class to the Mapping-layer. With the aid of additional mapping information, the client can transform data between different equivalence classes. At the time of writing, the client supports conversion from tabular-structured data to RDF triples, or from RDF to tabular-structured data, or between RDF quads and triples.

Tabular to RDF. For mapping tabular-structured data to RDF triples, the client utilizes the Tarql[8] mappings language. Currently, the Tarql library only supports the mapping of tabular data to RDF triple formats. RDF quad formats can be supported in the future by using the RDF Mapping Language (RML)[9]. There are three strategies to apply a mapping from a table to RDF using the Databus Client: 1) a generic transformation from CSV to RDF, that generates a resource URI for each row and creates a triple for each column and its corresponding value (the column header is appended to a generic base URI to specify the property). The value is represented either as an IRI if it can be parsed as valid IRI or a literal value otherwise. 2) Databus managed - The Databus can be requested to find matching mapping files for the given Databus file identifiers. Users can associate mapping files (e.g., a published Tarql file) using metadata in a flexible way with Databus groups, versions, or file identifiers, allowing anyone to reuse and apply these with the client automatically. 3) manual mapping - The user can specify a mappings file for the Databus file selection (query, collection) with a command-line parameter.

RDF to Tabular. The client implements a generic approach for mapping RDF into a wide table. Each RDF triple <subject, predicate, object> is mapped

[8] https://tarql.github.io/.
[9] https://rml.io/.

to one or more table cells, whereas each row contains information about one subject/entity. The first column of the table contains the subject's IRI. Then, for each occurring property of the source dataset, either one or two columns are created depending on the stored value. In case of an IRI, one column is created. Otherwise, two columns are created, one with the lexical form and a separate one to encode the original value's datatype or language tag information.

In addition to the resulting tabular file, the process generates a Tarql file that contains information for mapping the resulting table back to the original RDF structure.

RDF to RDF. The Databus Client can also convert between RDF triples and RDF quads formats. The mapping of RDF triples to RDF quads assigns a configurable graph name to the triples. The graph name setting can be given via a command-line option.

RDF quads to RDF triples are converted by splitting the input (quads) file into multiple triple files, one for each named graph.

Table 1. Equivalence class implementation overview: Reported are the equivalence classes with their supported serialization formats and mapping strategies between each other (inter equivalence class mapping)

Equivalence class		Supported mapping strategy		
Name	Serial. formats	to Quads	to Triples	to Tabular
RDF Quad	trig, trix, nquads, json-ld	–	File split	Wide table
RDF Triple	turtle, ntriples, rdf-xml	Conf. graph	–	Wide table
Tabular	tsv, csv	N/A	Tarql	–

5 Evaluation

We created a test suite and performed so-called round trip tests to verify the correctness of the file compiling for the reading and writing functionalities for every supported input/output format combination. We distinguish between round trip format conversion tests (Layer 2) and round trip mapping tests (Layer 3). Layer 0 and 1 are tested with regular unit tests.

A **round trip format conversion test** runs as follows. We take a file i and read it into the internal data structure of its equivalence class. Subsequently, that internal data structure is serialized to a file o, which is of the same format as file i. If the information in both files is equal, the round trip test succeeds. Within a **round trip mapping test**, we take a file i and convert it to file c of the format of another equivalence class before we convert c back to file o of the same format as i. Therefore, i first has to be read into the internal data structure of the equivalence class of i (see (1) in Fig. 3). Then this data is mapped to the internal data structure of the equivalence class of c (2) before it is written out

to c (3). Next, c is read into the internal data structure of its equivalence class (4). That resulting data is mapped back to the internal data structure of the equivalence class i (5). In the last step (6), this internal data structure data is written out to o. If the information of the input file i is equal to the information of output file o, the round trip test succeeds.

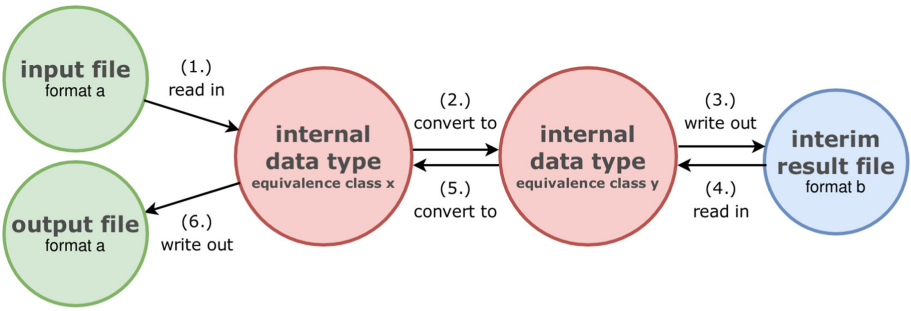

Fig. 3. Walk-through of a round trip mapping test

A round trip test is considered successful if we detect equality in the amount of information between the input and output file. There may be differences in syntax especially if no canonical version of the format is available. As a consequence, the files are parsed (ideally using a different software library) into some kind of internal/abstract data model again to be compared. In case of a non-bijective mapping between two equivalence classes (mapping is not reversible without information loss), these predictable losses have to be taken into account when evaluating the amount of information. We call this "quasi"-equal.

The layer design and round trip test approach reduce the quadratic amount of transformation/compiling combinations to be probed, and therefore help to realize a reliable and sustainable way to extend the client with other formats while maintaining/ensuring quality and correctness with a manageable effort. The number of tests performed on format layer is $T_f = n_{e_1} + n_{e_2} + ... + n_{e_{x-1}} + n_{e_x}$, whereas n_{e_i} is the number of formats in equivalence class e_i and x the total number of classes. Currently, the Databus Client has three implemented equivalence classes (see Table 1), which add up to $4 + 3 + 2 = 9$ format round trip tests that need to be performed.

To test the correctness of the mapping layer, we pick one format for each class and do a round trip mapping test for every (ordered) pair of equivalence classes. If we have two classes e_1 and e_2 we perform one mapping round trip test starting with a format from e_1 and a second test starting with a format from e_2. Picking one format is sufficient since we already tested the format conversion process within the equivalence classes for Layer 2. In summary, for x equivalence classes, the number of round trip mapping tests can be calculated by $T_m = \frac{x!}{(x-2)!}$. However, this formula assumes that there is exactly one mapping implementation in

every direction from/to every class. There could be cases where two equivalence classes can not be mapped or only in one direction (because the underlying data models differ too much), or multiple implementations for one mapping transition exist (that would need to be tested additionally).

When having 3 equivalence classes using exactly one mapping implementation between every class $\frac{3!}{(3-2)!} = 3! = 6$ round trip mapping tests need to performed. Since there is currently no mapping from tabular to RDF quads, 5 tests were performed.

Round trip tests allow to automate the conversion test, but they also have a limitation in spotting two interfering, systematic implementation errors, (e.g. one in the parser and one in the serializer), that counteract themselves. However, we consider them as sufficient in the scope of this work, especially when using frameworks that are broadly used and already tested in itself.

6 Related Work

The following section reports related work that also aims to improve the consumption of Linked Data into applications.

With the aid of **HTTP content negotiation**, HTTP clients can request files in formats that suite best for their demands, sending a list of weighted MIME types in the `Accept` header. Content negotiation is considered best practice for consuming Linked Data [8]. By using an `Accept-Encoding` header, the compression can be additionally specified. However, all conversion and implementation overhead as well as the complexity is on the server/publisher side, while leaving the consumer with the resulting technical heterogeneity, failures, and varying availability for common formats and compressions.

HDT [1] addresses a similar problem as the Databus Client, making RDF accessible better for consumers while being more efficient for the party hosting the data. It decomposes the original data into three components: Header, Dictionary, and Triples. With the help of the dictionary it realizes a compression, and makes additional compression of RDF files obsolete. An optional index can speed up simple lookup queries on the file. Unfortunately, there is no widespread native reading support for semantic applications and tools (SPARQL stores, reasoners, etc.). However, HDT parsing support could be integrated into the Databus Client, to allow transparent consumption of HDT files for applications.

OntoMaven [7] uses Maven repositories to release ontologies and optionally its dependencies (i.e. imported ontologies). As a consequence, transitive imports can be resolved and downloaded locally (using the Maven client) and then rewritten to use the locally mirrored (transitive) imports via a Maven plugin.

Although we were not able to find any announced public repository, the ontology organization structure is very similar to the one that is realized on the Databus using DBpedia Archivo [4] and which can be leveraged in combination with the Databus Client to manage and consume over 1300 ontologies as dependencies alongside instance data.

While plenty of ETL frameworks exist, we mention **UnifiedViews** [6] as an open-source ETL framework that supports RDF and ontologies. A data processing pipeline in UnifiedViews, consists of one or more data processing units (DPUs) and data flows between them. The DPUs offer basic functions that obtain data from external sources (e.g. CSV, XLS, RDF), convert data between various formats (e.g., CSV files to RDF, relational tables to RDF), perform data transformations (such as executing SPARQL Construct queries, XSLT, linking/fusing RDF data), and load the data to various (RDF) databases and file sinks.

While UnifiedViews has more powerful options in the individual steps, it has a weakness when it comes to provenance and repeatability (e.g. when the sources have changed). In contrast, the Databus Client harnesses the clear versioning and provenance model of the DBpedia Databus.

DataHub.io[10] is a data management platform, based on CKAN . A command line tool is provided that can download a single dataset alongside its datapackage JSON metadata file. A rudimentary versioning strategy allows to download the latest or an older version of a dataset. Furthermore, DataHub converts tabular data into normalized CSV and JSON files. However, the rich DataID metadata Model of the Databus in combination with collections or SPARQL queries provide a much more flexible and fine-grained download configuration method for the Databus Client.

7 Discussion and Future Work

In this work, we presented a concept and vertical-focused implementation of an interoperable and modular Databus Client, that shifts effort from the servers to the client while making data consumption and dependency management easier and more unified as a whole. The Databus Client offers a fully automated way for extracting and compiling data assets from the DBpedia Databus. Data that is only available in one RDF or tabular format can be used for many different semantic applications that support only a subset of these formats. Publishers can save storage and processing power of servers as well as human effort for publishing data in multiple formats, and instead invest resources in organizing the release and registering it with appropriate metadata.

The client‘s modular layer structure allows to implement, test, and use different functionalities individually and extend the client easily in the future. We can imagine to add one or multiple integration layers which normalize and merge schema and entity identifiers [3]. Moreover, we can support more formats (e.g. HDT, Manchester Syntax) and more mapping frameworks (like RML) by expanding existing layers.

While the Databus Client allows a flexible and via DataID metadata fine-grained access to files, this granularity is still dependent on the file partitioning strategy of the dump publisher. Although a monthly DBpedia release is separated into over 3,000 files, if information for only a small set of entities is

[10] https://datahub.io.

consumed by an application, a SPARQL or Linked Data fragments [9] endpoint is more convenient. We plan to extend the current file-based focus of the client to an even more flexible extraction phase that can use e.g. SPARQL to filter the compiled data.

At the current stage, the Databus Client is considered passive in the loading phase of the ETL process. The interface to consume data is on file/folder level, which is simple and powerful, but for better flexibility and complex workflows we see potential in advancing the client to orchestrate the loading phase as well.

Acknowledgments. This work was partially supported by grants from the Federal Ministry for Economic Affairs and Energy of Germany (BMWi) to the projects LOD-GEOSS (03EI1005E) and PLASS (01MD19003D).

References

1. Fernández, J.D., MartÍnez-Prieto, M.A., Gutiérrez, C., Polleres, A., Arias, M.: Binary RDF representation for publication and exchange (HDT). J. Web Semant. **19**, 22–41 (2013). https://doi.org/10.1016/j.websem.2013.01.002
2. Frey, J., Hellmann, S.: Fair linked data - towards a linked data backbone for users and machines. In: WWW Companion (2021). https://doi.org/10.1145/3442442.3451364
3. Frey, J., Hofer, M., Obraczka, D., Lehmann, J., Hellmann, S.: DBpedia FlexiFusion the best of wikipedia > wikidata > your data. In: Ghidini, C., et al. (eds.) ISWC 2019. LNCS, vol. 11779, pp. 96–112. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30796-7_7
4. Frey, J., Streitmatter, D., Götz, F., Hellmann, S., Arndt, N.: DBpedia archivo: a web-scale interface for ontology archiving under consumer-oriented aspects. In: Blomqvist, E., et al. (eds.) SEMANTICS 2020. LNCS, vol. 12378, pp. 19–35. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59833-4_2
5. Hofer, M., Hellmann, S., Dojchinovski, M., Frey, J.: The new dbpedia release cycle: increasing agility and efficiency in knowledge extraction workflows. In: Semantic Systems (2020). https://doi.org/10.1007/978-3-030-59833-4_1
6. Knap, T., et al.: Unifiedviews: an ETL tool for RDF data management. Semant. Web **9**(5), 661–676 (2018). https://doi.org/10.3233/SW-180291
7. Paschke, A., Schäfermeier, R.: OntoMaven - maven-based ontology development and management of distributed ontology repositories. In: Nalepa, G.J., Baumeister, J. (eds.) Synergies Between Knowledge Engineering and Software Engineering. AISC, vol. 626, pp. 251–273. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-64161-4_12
8. Sauermann, L., Cyganiak, R.: Cool uris for the semantic web. W3c interest group note, W3C (2008). https://www.w3.org/TR/cooluris/
9. Verborgh, R., Sande, M.V., Colpaert, P., Coppens, S., Mannens, E., de Walle, R.V.: Web-scale querying through linked data fragments. In: Proceedings of the 7th Workshop on Linked Data on the Web, vol. 1184. CEUR (2014). http://ceur-ws.org/Vol-1184/ldow2014_paper_04.pdf
10. Zaharia, M., et al.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: USENIX Symposium on Networked Systems Design and Implementation (NSDI 2012), pp. 15–28. USENIX Association (2012). https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia