Incremental Knowledge Graph Construction from Heterogeneous Data Sources

Dylan Van Assche [a,*], Julián Andrés Rojas [a], Ben De Meester [a] and Pieter Colpaert [a]
[a] *IDLab, Department of Electronics and Information Systems, Ghent University – imec, Belgium*
E-mail: dylan.vanassche@ugent.be

Abstract. Sharing real-world datasets that are subject to continuous change (creates, updates, and deletes) poses challenges to data consumers, e.g., reconciling historical versioning, handling change frequency. This is evident for Knowledge Graphs (KG) that are materialized from such datasets, where keeping the graph synchronized with the original datasets is only achieved by frequently and fully regenerating the KG. However, this approach is time-consuming, loses history, and wastes computing resources due to reprocessing of unnecessary data. In this paper, we present a KG generation approach that is capable of efficiently handling evolving data sources with different data change signaling strategies. We investigate different change signaling strategies observed in real-world data sources, propose the corresponding algorithms to detect data changes, and introduce a declarative approach that relies on RML and FnO to materialize and characterize data changes for evolving KGs. Our approach allows to optionally and automatically publish detected data changes in the form of a Linked Data Event Stream (LDES), relying on the W3C Activity Streams 2.0 vocabulary to describe changes semantically. This way, changes can be communicated to consumers over the Web. We implement our approach in the RMLMapper as IncRML (Incremental RML). We functionally evaluate our approach on a set of test cases, and quantitatively using a modified version of the GTFS Madrid Benchmark (taking change into account), and various real-world data sources (bike-sharing, transport timetables, weather, and geographical). On average, our approach reduces the necessary storage and used computing resources for generating and storing multiple versions of a KG (up to 315.83x less storage, 4.59x less CPU time, and 1.51x less memory), reducing KG construction time up to 4.41x. The performance gains are more evident for larger datasets, while for smaller datasets, our approach's overhead partially nullifies such performance gains. Our approach reduces the overall cost of publishing and maintaining KGs, which may contribute to the uptake of semantic technologies. Moreover, the use of LDES as a Web-native publishing protocol enables that not only the KG publisher benefits from the concise and timely communication of changes, but also third-parties if the publisher chooses to make it public. In the future, we plan to explore end-to-end performance, possible optimizations on our change detection algorithms and the use of windows to expand our approach to streaming data sources.

Keywords: RML, LDES, Incremental Knowledge Graph Construction

1. Introduction

Most real-world datasets are not static. Whether it be the inclusion of new data points, the rectification of errors, or the evolution of data collection methodologies, datasets are, by their very nature, subject to continuous change. The change frequency may vary, with some datasets undergoing rapid transformations while others experiencing a more gradual evolution. Regardless of the particular change frequency, data consumers find themselves impacted by these alterations. Consumers must take this continuous change flow into account when using the data [1] and incorporate these changes to stay synchronized with the current state of the original dataset. This leads them to

*Corresponding author. E-mail: dylan.vanassche@ugent.be.

face challenges such as storing multiple versions to keep historic records, having enough capacity to handle any dataset's size when processing changes, and keeping up with dataset's change frequency. Moreover, data source changes impact data integration processes, such as Knowledge Graph (KG) generation, which are forced to fully repeat their computations *every time one of the original data sources changes* [2, 3]. This approach could become unsustainable when dealing with large volumes of data. It could also affect applications depending on (near) real-time information: a generated KG supporting the application might be already outdated if the source data sources change faster than the (re-)generation process. In general, this may result into: (i) wasted computing resources and time due to unnecessary reprocessing of all data, especially when large parts remain unchanged, (ii) loss of historic records given that commonly only the latest version remains available, and (iii) outdated KGs when the data integration process is not able to keep up with the change frequency of the original data sources.

Change Data Capture (CDC) techniques [4, 5] try to address this problem by capturing and characterizing data changes using, e.g., database transaction logs, database triggers, comparing snapshots of data dumps, etc. [2]. However, current approaches for integrating heterogeneous data sources into KGs [6–10] cannot use existing CDC techniques, as they mainly focus on a specific type of data source (e.g., relational database logs). In this paper, we introduce a *declarative and incremental KG construction* approach, capable of handling heterogeneous and changing data sources based upon a CDC-based technique. Our approach can cope with different data change signaling strategies, allowing to detect and incorporate these changes in an incremental and continuous fashion into an existing KG. We enumerate and discuss different *data change signaling strategies* (both explicit and implicit) observed in real-world datasets, that are used to communicate changes to data consumers. We also propose a set of algorithms to handle these change signaling strategies for and during KG construction. Our approach only regenerates the parts of a KG that were changed in the original data source(s) (created and updated), while deleted data entities are detected, labeled as such and made explicit.

We opt for a declarative approach to incrementally generate KGs given its engine-agnostic and extensible nature to handle different types of data sources [11]. We implement our approach using RML as declarative mapping language to generate RDF, and FnO for describing data transformations and state management functions, which together constitute our approach called *IncRML* (Incremental RDF Mapping Language). However, our approach is not dependent on these particular technologies and could be implemented with any other declarative mapping language and data transformation vocabulary. Furthermore, we allow to optionally and automatically publish these changes on the Web in the form of a Linked Data Event Stream (LDES) [12]. LDES allows publishing data changes as an append-only event log that consumers can read to synchronize their KG with the latest data available. We show how it is possible to produce an LDES from the detected data changes, materialized with our IncRML approach, which we choose to describe semantically with the W3C Activity Streams 2.0 vocabulary [13].

We extend our previous work [14] on aligning LDES and RML, where we introduced a preliminary approach for continuously generating a KG by detecting and materializing changes in its data sources. Our previous work focused only on handling data creations and updates. In this paper, we extend that work to also detect and communicate deletions. We also include extensive functional and performance evaluations using (i) different types of real-world datasets, (ii) an extended version of the GTFS Madrid Benchmark [15] that allows to control the type and amount of changes in a data source, and (iii) a set of test cases that guarantee the support for different change signaling strategies. To keep this paper self-contained, we include the discussions of our previous work along with its extensions. Concretely, our contributions are the following:

1. An overview of different history and change signaling strategies observed in real-world data sources.
2. A set of CDC-based algorithms to detect changes in any of these change signaling categories.
3. The integration of these algorithms into a KG construction pipeline using RML, FnO, and (optionally) LDES.
4. An extensive evaluation and benchmark on the impact and performance of incrementally generating and updating KGs.

Overall, our approach is more efficient in terms of execution time and computing resources for generating the RDF quads of a KG, in exchange for a small overhead when the KG is constructed the first time. We observe that by materializing only detected changes when generating KGs, our approach uses 3.24–315.83x less storage to store multiple versions of a KG while also consuming less computing resources (0.85–4.59x less CPU time, 0.72–1.51x less memory consumption) depending on the dataset. The RDF generation time is reduced by 0.97–4.41x depending

on the data size. To guarantee a fair comparison with a traditional full KG re-materialization approach, we measured the total time that it takes to go from existing data to an updated triplestore using SPARQL `UPDATE` queries, so that a consistent and up-to-date KG is achieved for both cases. For our approach, an additional step is required to produce the corresponding SPARQL query that either creates, updates or deletes data entities. Despite this additional step, we observe that our approach is 11.07–57.66x faster to ingest all versions of a dataset, and on average 4.5–28.5x faster to ingest individual dataset updates. However it was only possible to measure this improvement on datasets whose updates where not larger than 20K quads, since the triplestore would fail due to reaching internal query length limits, to execute the SPARQL `UPDATE` queries for both the traditional and our approach.

IncRML is usable for any kind of data source, as supported by the underlying declarative mapping language. While we use RML as mapping language in our implementation, the logical definitions and algorithms of IncRML may be implemented with other mapping languages since they rely on widely supported features such as IRI templates and data transformations [11]. IncRML also allows for semantically describing data changes using any ontology of choice (e.g., W3C Activity Streams 2.0) and then publishing such changes with a structured approach such as LDES or any other data publishing strategy. Thanks to our work, data source changes can be integrated faster and with less resources into live and replicated KGs while allowing to keep access to the historical records in the form of an LDES. Although we observed that further optimizations are needed for triplestores to effectively support larger data updates through standard SPARQL UPDATE queries.

The remainder of this paper is structured as follows: Section 2 discusses related works, Section 3 introduces the main technological concepts used in this work, namely RML, FnO, and LDES. Section 4 presents different identified change signaling strategies and describes the rationale of our approach. Section 5 shows how we implement our approach. In Section 6, we present our evaluation design. Section 7 discusses our results, and Section 8 concludes.

2. Related Work

In this section, we discuss related work on (i) mapping rules for declarative Knowledge Graph (KG) generation describing how a KG can be generated, (ii) Change Data Capture approaches for detecting changes in data sources, (iii) versioning strategies for KGs to store the history of data sources and its impact on storage for producers and consumers, (iv) versioned generation of KGs, and (v) incremental mapping rules execution for optimizing execution time and resource usage.

2.1. Mapping rules for declarative Knowledge Graph generation

Declarative mapping rules for KG generation is an active research domain since the introduction of the R2RML W3C Recommendation [6] for transforming relational databases into an RDF KG [16]. R2RML was extended as the RDF Mapping Language (RML) [7, 17] to support heterogeneous data sources (e.g., JSON, XML, CSV) while keeping backwards compatibility with R2RML. Recently, a survey [11] was performed of existing approaches and systems for declarative KG generation from heterogeneous data. RML is widely used for declarative KG generation and was extended to support exporting RDF to various targets such as files and SPARQL endpoints [18], RDF Collections and Containers [19, 20], and access to Web APIs [18, 21].

Besides RML, other declarative mapping languages were proposed to transform heterogeneous data sources into RDF such as xR2RML [20, 22], SPARQL-Generate [9], SPARQL-Anything [8], ShExML [23], D-REPR [10], and OTTR [24]. xR2RML also extends R2RML with support for heterogeneous data sources and RDFS Collections and Containers [25]. SPARQL-Generate and SPARQL-Anything do not extend R2RML, but SPARQL instead to transform heterogeneous data sources into a KG. Therefore, they can reuse existing SPARQL engines and syntax. Similar to SPARQL-based approaches, ShExML uses Shape Expressions (ShEx) [26] as syntax, while D-REPR defines its own syntax.

Currently, the W3C Community Group on KG Construction[1] is working on standardizing RML as a W3C Recommendation [17] and is supported by multiple implementations such as the RMLMapper [7], Morph-KGC [27],

[1] https://www.w3.org/community/kg-construct/

or SDM-RDFizer [28]. Therefore, we decided to implement our approach with RML as the declarative mapping language in this work, but any mapping language may be used.

Data transformation support is an important requirement when generating KGs from heterogeneous data sources [11], using, e.g., the Function Ontology (FnO) [29], SPARQL Functions [30], or FunUL [31]. FnO is a popular vocabulary for describing functions to perform data transformations, and it is integrated with RML through FNML[2]. This way, RML+FnO mapping rules can perform both the generation of a KG and data transformations without requiring ad hoc or use case specific scripts. We use FnO in this paper to integrate the implementation of our change detection algorithms with RML to incrementally generate a KG from heterogeneous data sources.

2.2. Change Data Capture

Change Data Capture (CDC) [4, 5] refers to a technique, primarily used in databases, to identify and capture data changes so that those changes can be tracked, recorded, and propagated to other systems or applications. CDC's primary purpose is identifying and capturing creations, updates, and deletions of data in a dataset to enable (near-)real-time synchronization of data across different systems. Most approaches focus on data sources which provide some sort of change signaling mechanism such as transaction logs, snapshots, triggers, or timestamps. Moreover, data sources may offer different granularity regarding change signaling, e.g., change signaling on parts of the data source or the whole data source [32].

Log-based approaches [4, 33] use database transaction logs to determine which changes were performed to the underlying data. As log systems are implementation-specific, these approaches are specific to each database. *Snapshot-based approaches* [2] compare the current version of a data source with previous versions to extract changes, requiring sufficient resources to store and compare these versions. *Trigger-based approaches* [1, 34, 35] hook into a data source to execute a trigger on each change, requiring support for triggers from the data source (e.g., stored procedures in relational databases). *Timestamp-based approaches* [36] analyze a last-modified timestamp of a data source to detect and extract changes, requiring data sources to provide timestamp-annotated data.

Although these approaches clearly have their merit, many data sources do not signal their changes nor support triggers when a data record is changed (e.g., data streams, files, or Web APIs). Therefore, existing CDC approaches are insufficient to cover heterogeneous data sources which do not signal their changes to consumers. In this work, we combine and extend *Timestamp-based* and *Snapshot-based* approaches for detecting implicit and explicit changes.

2.3. Versioning of Knowledge Graphs

Several approaches for versioning of (RDF) KGs have been proposed [37]. Three main RDF archive storage strategies can be identified [38]: (i) Change-Based, (ii) Timestamp-Based, and (iii) Independent Copies. *Change-Based* only stores the changes; *Timestamp-Based* uses timestamps to define when a specific version is valid; and *Independent Copies* stores a copy of the data source each time it is updated.

Change-Based approaches include: R&Wbase [39], based on Git[3]; a Version Control Based RDF storage approach using patches, similar to Frommhold M. et al. [40]; Cassidy et al. [41], SemVersion [42], R43ples [43], and Im et al [44]. *Timestamp-Based* approaches for accessing different data source versions include: Memento (HTTP) [45] and x-RDF-3X [46] (SPARQL). OSTRICH [47] and TailR [48] are both *hybrid* approaches, combining all three strategies for efficient query operations. All approaches implementing the 3 strategies put the burden of resolving versions on the data producer.

LDES [12] uses an *Independent Copies* approach on a entity level (aka. member in LDES terminology) for versioning. An LDES entity/member may be defined as a named node and its properties, as defined by its Concise-Bounded Description (CBD)[4], or as named graph and its contained triples. However, Independent Copies suffers from scalability problems as storage is not infinite [38], LDES addresses this by dropping the oldest versions of

[2]We use the most implemented version of FnO with RML (https://fno.io/rml), but our approach can be used with the latest version as well (http://w3id.org/rml/fnml).

[3]https://git-scm.com/

[4]https://www.w3.org/submissions/CBD/

entities in the event stream, according to a specific and configurable *retention policy*. Since LDES consumers are aware of the retention policy, they can decide to store the LDES members themselves if they need to for their particular use case. For example, if an LDES producer's retention policy is 7 days and a consumer requires at least the last 30 days of data, the consumer must store a copy. If a consumer does not require history information longer than 7 days, it can solely rely on the LDES producer's data. LDES allows consumers to synchronize their local copy of the member collection, similar to a Copy and Log approach [49]. This way, versioning is resolved on the consumer side and several versioning strategies can be applied independent of the publisher. In this work, we allow our approach to use LDES as a publishing strategy for communicating KG changes on the Web.

2.4. Versioned generation of Knowledge Graphs

Ontologies (Change Detection Ontology [50, 51]) and benchmarks (EvoGen Benchmark Suite [52], BEAR benchmark [38]) were proposed for versioned KGs. However, they are tied to a specific ontology or focus on querying the versions while we focus on the generation in this work. The EvoGen Benchmark Suite [52] allows generating synthetic versioned RDF data for benchmarking purposes. BEAR [38] proposed a benchmark for Semantic Web archiving systems to evaluate full materialization of different versioned KGs, only materializing the changes, or annotating triples when they were created, updated, and deleted. However, both focus on materialized RDF for benchmarking query systems, while in this work, we focus on the generation of different KG versions from non-RDF heterogeneous data. The Change Detection Ontology [50, 51] allows describing changes inside the original data sources as a changelog and is used in the MQ framework [53] to generate new KGs if changes are detected, using R2RML mappings from CSV, XML, and relational databases. However, this solution is tightly coupled with a specific ontology, while we aimed for a more generalized approach, that allowed to use any ontology to describe the detected changes.

2.5. Incremental mapping rules execution

Execution planning of mapping rules has seen uptake in the KG community [11] as seen in tools such as Morph-KGC [27] and SDM-RDFizer [28]. Both systems plan their execution and remove duplicates before they execute RML mapping rules to reduce the number of data records and improve performance. However, they consider that executing these mapping rules only happens once: if the datasets change, all the mapping rules must be fully executed again. Thus, incremental KG generation is not considered. Besides execution planning with Morph-KGC and SDM-RDFizer, existing work on incremental KG generation does not consider heterogeneous data as it focus on relational databases such as using log files to determine changes [54], triggers to update virtualized views [55], or indexing triples [56]. There is no approach which supports heterogeneous data – besides relational databases – e.g., JSON, XML, CSV. In this work, we present a novel approach to also detect changes in heterogeneous data and making them available to consumers, even if the data sources do not signal their changes. While our experiments focus on incremental KG generation from single data sources, multiple and heterogenous data source cases are also supported through the use of RML.

3. Background

In this section, we provide an introduction to the (i) RDF Mapping Language (RML), (ii) Function Ontology (FnO), and (iii) Linked Data Event Streams (LDES). These technologies came forward from Section 2 as prevalent methods to describe how a Knowledge Graph (KG) could be constructed from input datasets (RML), performing data transformations and generic functions during KG construction (FnO), and semantically describing and continuously publishing the KG changes (LDES) to allow for KG replication and synchronization.

3.1. RDF Mapping Language (RML)

RDF Mapping Language (RML) [7, 17][5] is an extension of W3C Recommendation R2RML [6] to support heterogeneous data sources besides relational databases. RML mapping rules consist of Triples Maps (Listing 1: lines 1-17) which define how the terms (subject, predicate, and object) of an RDF triple are generated. A named graph can also be specified using a Graph Map for generating RDF quads. Each Triples Map has one Logical Source (Listing 1: lines 2-4), one Subject Map, and zero or more Predicate Object Maps. The Subject Map (Listing 1: lines 6-9) defines how the subject IRIs are generated from the data source as defined by the Logical Source. This Subject Map also includes a Graph Map to specify the named graph of the RDF quad (Listing 1: line 8). Predicate Object Maps (Listing 1: lines 11-16) consist of Predicate Maps (Listing 1: line 12) to specify the quad's predicate and (Referencing) Object Maps (Listing 1: lines 13-15) for the quad's object. The Subject Map, Predicate Map, Object Map, and Graph Map are all Term Maps, generating an RDF term (an IRI, blank node, or literal). A Term Map may always generate the same RDF term with `rr:constant`, a referenced value from the data source with `rml:reference`, or construct a value based on a template with `rr:template`. If a Subject Map, Predicate Map, or Object Map uses an `rr:constant` for its RDF term generation, a shortcut can be used, e.g., `rr:predicate`.

Listing 1: RML uses Triples Maps with a Logical Source, a Subject Map, Graph Map, and zero or more Predicate Object Maps to specify how RDF quads must be generated from the referenced data source.

```
1  <#RMLMapping> a rr:TriplesMap;
2    rml:logicalSource [ a rml:LogicalSource;
3      rml:source "/path/to/data.csv";
4    ];
5
6    rr:subjectMap [ a rr:SubjectMap;
7      rr:template "http://example.org/{ID}";
8      rr:graphMap [ rr:constant ex:MyGraph ];
9    ];
10
11   rr:predicateObjectMap [ a rr:PredicateObjectMap;
12     rr:predicate foaf:name;
13     rr:objectMap [ a rr:ObjectMap;
14       rml:reference "name";
15     ];
16   ];
17 .
```

3.2. Function Ontology (FnO)

The Function Ontology (FnO) [29] semantically describes and declares implementation-independent functions and their relations to related concepts such as input parameters, outputs, mappings to concrete implementations, and executions. The alignment between RML and FnO (via FNML [29]) specifies how a data transformation must be performed by specifying the function to execute (Listing 2: line 5) and its values (Listing 2: lines 8-11). FnO is standalone which allows describing data transformations in a declarative way with or without RML. FnO is integrated in RML through an `fnml:FunctionMap` (Listing 2: lines 1-14) which is an RML Term Map. Therefore, an FnO function can be used as Subject Map, Predicate Map, Object Map, or other RML Term Maps.

[5] We use the most implemented version of RML (https://rml.io/spec), but our approach can be used with the latest version as well (http://w3id.org/rml/core).

Listing 2: FnO defines a data transformation by specifying the function and the function's values. FnO is aligned with RML through a `fnml:FunctionMap` which is an RML Term Map. The function `toUppercase` is executed on all referenced `name` data values of the data source.

```
<#FunctionMap> a fnml:FunctionMap
  fnml:functionValue [
    rr:predicateObjectMap [ a rr:PredicateObjectMap;
      rr:predicate fno:executes;
      rr:object grel:toUppercase;
    ];
    rr:predicateObjectMap [ a rr:PredicateObjectMap;
      rr:predicate grel:inputString;
      rr:objectMap [ a rr:ObjectMap;
        rml:reference "name";
      ];
    ];
  ];
.
```

3.3. Linked Data Event Streams (LDES)

Linked Data Event Stream (LDES) is an RDF data publishing approach fostered by the EU Semantic Interoperability Community (SEMIC)[6] and officially adopted as a standard specification by the Flemish government through its Flemish Smart Data Space project[7]. LDES defines datasets in terms of a collection of immutable objects (a.k.a. members) such as versioned entities or observations (Listing 3), where every member must have its own unique IRI [12]. The main goal of a LDES is to enable efficient replication and synchronization of datasets over the Web. LDES allows data consumers to traverse the collection by relying on the TREE specification [57][8] to semantically describe hypermedia relations among subsets or fragments of the data (Figure 1). These hypermedia relations can be defined in multiple ways, e.g., by publishing fragments organized by time or by version, configuring tree-like data structures that can be efficiently traversed by clients. TREE also allows further describing the content of each member in an LDES collection through a SHACL shape (`tree:shape`, Listing 3: line 4). Consumers may use the related SHACL shape to validate but also to understand the type and properties of the LDES members (`tree:member`, Listing 3: line 5), e.g., for discoverability and source selection purposes. Additional metadata on how a collection of LDES members is structured, is available via properties such as `ldes:timestampPath` (Listing 3: line 2) or `ldes:versionOfPath` (Listing 3: line 3), which describe the property paths that provide timestamp and versioning information in every member, similarly to the SHACL `sh:path` property. This way, LDES remains domain model agnostic and members are not limited to a specific ontology for describing timestamp-based versions or referring to other members' versions. Consumers may poll or subscribe to an LDES to obtain the latest (versions of) members published, which then can be further processed and materialized into a consumer's local environment to remain synchronized with the original data source.

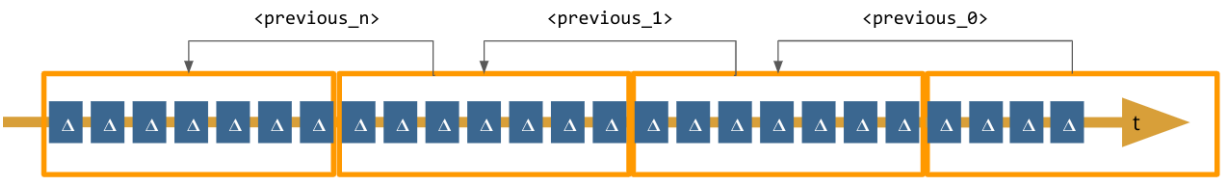

Fig. 1. LDES's structure allows to traverse the collection by clients with semantic descriptions of the collection's relations.

[6] https://joinup.ec.europa.eu/collection/semic-support-centre/linked-data-event-streams-ldes
[7] https://www.vlaanderen.be/vlaamse-smart-data-space-portaal
[8] https://w3id.org/tree/specification/

Listing 3: A data collection as an LDES with one member. The LDES member provides a sensor value at a given time and version.

```
<#LDESDataCollection> a ldes:EventStream;
  ldes:timestampPath sosa:resultTime;
  ldes:versionOfPath dcterms:isVersionOf;
  tree:shape <http://example.org/shacl/shape/>;
  tree:member <LDESMember>;
.

<#LDESMember> a sosa:Observation;
  sosa:resultTime "2023-01-01T00:00:00Z"^^xsd:dateTime;
  sosa:versionOfPath <http://example.org/sensor/result/>;
  sosa:hasSimpleResult "5"^^xsd:integer;
.
```

4. Approach

In this section, we describe our incremental Knowledge Graph (KG) construction approach, which consists of the following high-level steps:

1. detect changes in any heterogeneous dataset;
2. construct RDF data from these changes;
3. explicitly publish the changes with explicit semantics.

Section 4.1 introduces the overall approach. How to detect changes in any heterogeneous dataset requires us to first understand how a particular dataset may signal changes (step 1a), hence, Section 4.2 discusses the various change signaling strategies along 3 dimensions: history, change communication, and change types. Then, Section 4.3 describes the algorithms to detect both implicit and explicit changes in data collections (step 1b). Where Sections 4.1–4.3 focus on detecting and describing changes in existing datasets, Section 4.4 contextualizes this work as part of an entire processing pipeline, from original data to a published event stream of RDF data, integrated in a mature KG construction and publication pipeline (steps 2 and 3). The overall approach is shown in Figure 3, top (green) row.

4.1. IncRML

IncRML (Incremental RML) refers to the implementation of our proposed approach for incrementally generating a KG from heterogeneous data sources. It consists mainly of 2 high-level steps: (i) detect changes in data sources, independent of whether they are signaled explicitly or implicitly, and (ii) enriching the original target ontology mapping to include additional metadata that makes explicit which quads are created, updated, and deleted. In general, our approach maximally relies on existing standards and specifications, both for the generation and the publishing of KGs. We aim at reducing execution time and resource consumption during the RDF generation process, as only the changes between consecutive data source versions are materialized. Through a CDC-based approach, we detect and extract changes during the KG generation process and (re-)generate the RDF triples/quads of all the entities (or members in LDES terminology) affected by such change. Each materialized member may include additional metadata specifying, for example, the type of change, the time of change, etc, as specified by LDES. Our approach does not limit the description of changes in data members to a specific ontology, thus it may be applied to any data collection modeled by an ontology with the semantics and expressivity to guarantee unique identification and describe member changes, or extensions thereof. This allows consumers to keep their local version of a KG in sync with the original producer across the Web, by interpreting the change semantics present in the materialized members and performing the corresponding create, update, and delete operations instead of fully re-fetching and re-ingesting a complete version of the KG every time there is an update.

Currently, RDF triplestores lack support for integrating LDES data directly, which poses the need for an additional intermediate step to interpret and execute the corresponding SPARQL `UPDATE` operation for each LDES member

(e.g., `INSERT DATA`, `DELETE WHERE`, etc.), in order to keep the triplestore up to date. In this work we focus on studying the impact of our approach on the generation aspect of a CDC-based KG publishing system that adheres to the LDES specification, and provide a proof-of-concept implementation of an interpreter library that translates the change semantics of LDES members into the corresponding SPARQL `UPDATE` queries that ingest data updates into a target triplestore.

4.2. Change signaling strategies

We identified a set of change signaling strategies differing along three 3 dimensions: (i) availability of historical records, (ii) how changes are communicated to consumers, and (iii) type of change; by analyzing real-world datasets from various domains e.g., bike-sharing data, public transport timetables, geographical data, traffic data, and meteorological data (Section 6.1.3).

History We identified 3 different types of history availability:

- *latest state*: Latest state refers to datasets that publish only the latest version of all its members on every change.
- *latest changes*: Latest changes refers to datasets that publish only changed members (aka delta updates). Thus, consumers must have access to an initial complete version of the data upfront and reconcile updates over it.
- *full history*: Full history refers to datasets that are published including both historical and current versions of its members. The number of available versions is defined by the data publisher.

Change communication We identified 2 change communication strategies:

- *explicit*: Dataset changes are *explicitly* communicated if metadata is also provided to point that a change has occurred (e.g., via uniquely identified members using timestamps, hashes, or logs).
- *implicit*: Dataset changes are *implicitly* communicated if members are changed without providing any kind of metadata indicating that a change happened, e.g. changes such as property updates, member deletions, or member creations, all happening silently across new versions of the data collection.

Change types We identified 3 change types:

- *create*: A member is added to the dataset and it didn't exist before.
- *update*: An existing member of the dataset is modified. New properties are added to the member or existing properties are modified.
- *delete*: An existing member of the dataset is deleted.

Moreover, change communication may differ depending on the type of change. For example: created and updated members may have a unique identifier (explicit change), while deleted members are simply removed in newer versions of a data collection (implicit change). Table 1 shows a summary of the different change signaling strategy combinations with respect to history availability, change communication, and change types identified on the set of real-world datasets analyzed in this work.

4.3. Implicit and explicit change detection

Datasets communicate their changes mainly in 2 ways, regardless of historical records availability: (i) explicitly through uniquely identified members, logs, etc., and (ii) implicitly by *silently* changing dataset members. The latter imposes the need for consumers to detect these changes themselves. Our approach (Figure 2) combines Timestamp-based and Snapshot-based Change Data Capture (CDC) approaches [2] to handle both explicit and implicit changes. In our approach, explicit changes are detected by relying on uniquely identified dataset members (e.g., via subject IRIs that depend on last modified timestamps). We detect implicit changes by comparing consecutive snapshots of dataset members, checking their subject IRIs, and (a subset of) their corresponding properties for changes.

It is important to note that implicit deletion communication is not possible when combined with full history or latest changes. *Full history* datasets must communicate deletions explicitly otherwise a data consumer cannot determine that a member was deleted implicitly, since it will encounter it as part of the historical records also

Table 1

Change signaling strategies according to their history availability and change communication. Change communication can differ per type of change, even in the same dataset.

History availability	Change communication	Change types
Latest state	Explicit	Create
		Update
	Implicit	Delete
Latest changes	Explicit	Create
		Update
	Implicit	Delete
Full history	Explicit	Create
		Update
	Implicit	Delete

present in the dataset. Even when a data consumer can tell apart historical records from new data, if a member stops occurring in a full history dataset, it could be ambiguously interpreted either as being deleted or as a non-updated member. Assuming that the member is to be deleted could result in both false-positive and false-negative deletion detection. *Latest changes* datasets are similar to *full history* regarding deletions. If such dataset do not explicitly communicate that a member is deleted, consumers cannot determine with full certainty if a member was deleted or simply remains unchanged.

Explicit changes can be directly detected and do not require additional processing effort during the KG generation process. Implicit changes, however, require a stateful processing approach, to keep track of members' state across KG generation executions. Our approach introduces 3 algorithms to handle implicit changes, which scale in direct proportion to the number of members in the dataset. Each detection algorithm corresponds with a particular type of change: (i) *create*, detects when a new member is added to the dataset, (ii) *update*, detects when an existing member is modified in the dataset, and (iii) *delete*, detects when an existing member is deleted from the dataset. Next, we describe the logical flow of each algorithm in the case of implicit change communication.

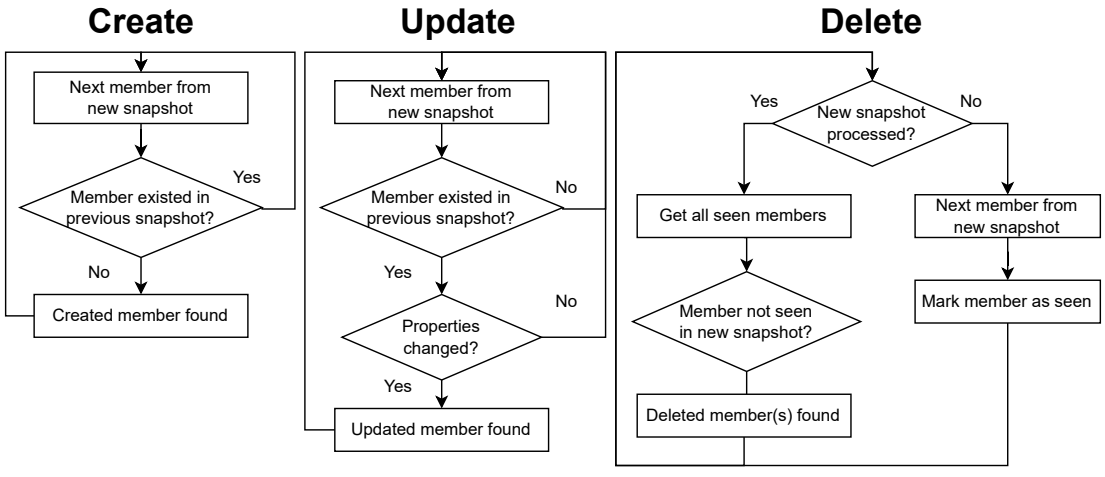

Fig. 2. Implicit updates must be detected by a CDC algorithm. Creates and updates are detected by checking if the member already exists in a previous snapshot. Members in a dataset are identified through their generated IRI. If the IRI does not exist in a previous snapshot, we conclude that the member was created. If the IRI does exist and the properties of the member have changed, we conclude that the member was updated. Deletions are detected by checking if the member is no longer present compared to the previous snapshot of the dataset.

Implicit Create (Figure 2, left) For every dataset member being processed in the current version, the algorithm checks if the member was already present in a previous version of the dataset, based on its subject IRI. If not, a *created* member is found.

Implicit Update (Figure 2, middle) Similar to create, the algorithm checks if the member was already present through its subject IRI and its properties. The algorithm does not necessarily check all properties of a member. It can simply check a predetermined subset which were labeled as *watched properties*. These properties can be used to, for example, compute a hash that allows to determine if a change has occurred. If the member's subject IRI was present and its watched properties' values were changed, a *updated* member is found.

Implicit Delete (Figure 2, right) At KG generation time, the algorithm marks every generated member as seen. Once all members have been processed in the current version of the dataset, it identifies the members which were not seen in the current execution, with respect to the previous KG generation execution (if any). These are marked as the *deleted* members in the current version of the dataset.

4.4. Incremental KG construction and publishing pipeline

Our approach effectively brings together (i) RML for declaratively defining generation rules for a KG; (ii) FnO for defining change detection functions across data source versions (Section 4.3); and (iii) LDES as an optional publishing strategy to publish semantically described changes as an event stream for consumers on the Web. Figure 3 presents a schematic view of a data processing pipeline using our approach to incrementally construct and publish a KG. If changes are detected, the corresponding RML mapping(s) is/are executed to generate and publish only the changed members to be later integrated into the KG. In practice, a member is generated by a RML Triples Map[9] (`rr:TriplesMap`) with one Subject Map (`rr:SubjectMap`) and zero or more Predicate Object Maps (`rr:PredicateObjectMap`).

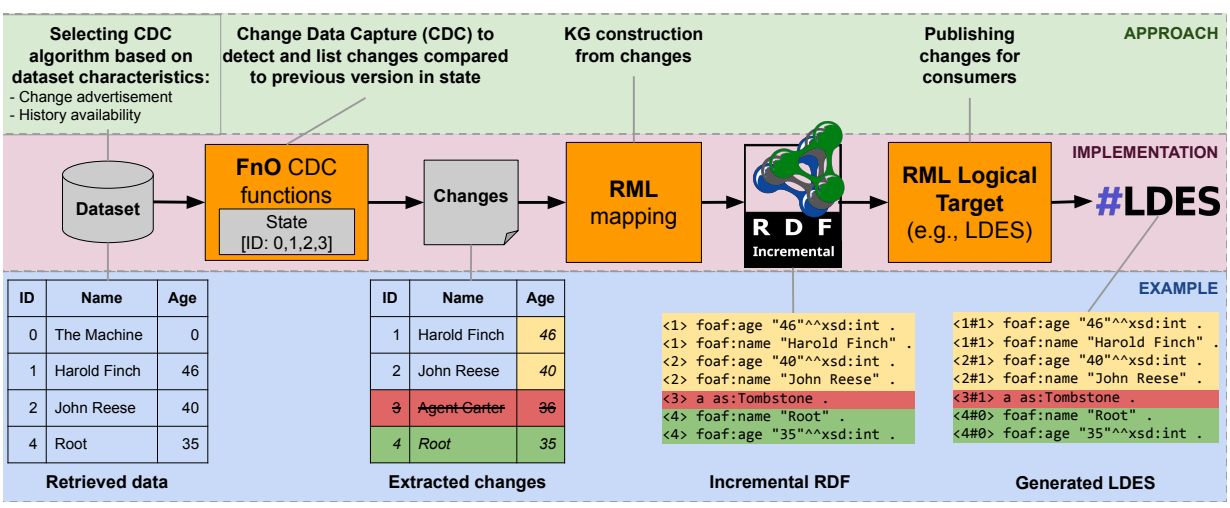

Fig. 3. Our approach IncRML (top green row) for which we use RML+FnO (middle pink row). We use FnO described functions to perform CDC based on the characteristics of the dataset and RML for constructing RDF from the detected changes. Changed RDF quads may be published as an LDES via an LDES Event Stream Logical Target. The pipeline is continuously executed to extract changes from new versions of the datasets. Our approach can be used by any RML engine with support for FnO (orange squares). Example data (bottom blue row) shows how data creations (green), data updates (yellow), and data deletions (red) are detected through CDC FnO functions. It is assumed that the previous state contains info on data rows with IDs 0, 1, 2, and 3. The extracted changes are then incrementally transformed into RDF and published as LDES members.

Our CDC FnO functions monitor the Subject Maps and their correspondent Predicate Object Maps at execution time, to determine if there were any changes with respect to the previous execution. If so, the correspondent member

[9]The `rr` prefix expands to http://www.w3.org/ns/r2rml#

is materialized and (optionally) published as a typed event in an LDES. As an example, we show how we can semantically annotate changed members using the W3C Activity Streams 2.0 vocabulary: `as:Create`[10] for created members, `as:Update` for updated members, `as:Delete` for deleted members. Created and updated members are fully re-materialized while for deleted members only a tombstone (a simplified version of the member) which indicates that the member was deleted without any other properties is generated. Annotating these changes with the type of change, allows consumers to interpret and incorporate these changes over their local copies of the KG. If published as an LDES, data consumers can subscribe to it to receive the changes and remain in sync over the Web. Our approach generates an unique IRI for version of a member as defined by the LDES specification[11]. Note that this requirement is solely for publishing a valid LDES and it is not required to perform the change detection process discussed in Section 4.3. Ultimately, both LDES and non-LDES data consumers can interpret the change semantics present in the materialized members produced by our incremental generation approach and use, e.g., SPARQL `UPDATE` queries to incorporate the changed members into their KG.

5. Implementation

In this section, we describe how we implement our proposed approach (Section 4) in the RMLMapper[12] based on a set of CDC FnO functions, RML mapping rules to construct a Knowledge Graph (KG), and LDES for publishing an incrementally constructed KG as an event stream. Section 5.1 explains how we implement our CDC FnO functions, Section 5.2 discusses how we integrated RML with the CDC FnO functions, Section 5.3 explains how we implemented an LDES Logical Target for directly generating LDES annotated data, and Section 5.4 demonstrates how our implementation is applied on an example dataset.

5.1. Change Data Capture FnO functions

We implement our CDC algorithms from Section 4.3[13] as FnO described functions. A dedicated FnO function was implemented for each change type and change communication strategy (Table 2). This way, we semantically describe which type of detection is applied on the dataset, even if the underlying detection algorithms operate similarly. Listing 4 shows the FnO description of a CDC FnO function for detecting implicit updates in a dataset. It detects dataset updates based on the subject IRI of a member (`idlab-fn:iri`) and a set of watched properties (`idlab-fn:watchedProperty`), coalesced into as a structured string. By using a structured string, our FnO functions remain independent of the source data structure and allows us to use RML constructs such as `rml:template` for capturing the values of potentially complex watched properties through the used reference formulation (e.g., XPath or JSONPath) in the RML rules.

The function persists the state of the properties per member in the location specified by `idlab-fn:state`. Storing of state is implementation dependent, thus it can be stored as a plain file but also in a database for example. The state is a lookup table of the member's properties with the member subject IRI as key. Through the state, we can keep track of any changes to members of a data source and detect added and deleted members by monitoring the presence of the member's IRI.

Each function requires an `rr:template` to construct the IRI of the member being checked for changes. For explicitly communicated changes, each function only needs to check if a member IRI was already generated in the past: since IRIs are guaranteed to be unique when changes are explicit, there is no need to materialize previously seen members. The implicit creation function `idlab-fn:implicitCreate` is identical to its explicit counterpart, (`idlab-fn:explicitCreate`), given that implicit member creation also occurs when new IRIs are detected. However, the `idlab-fn:implicitUpdate` and `idlab-fn:implicitDelete` functions are different. `idlab-fn:implicitUpdate` requires an additional list of properties (watched properties)

[10] The `as` prefix expands to https://www.w3.org/ns/activitystreams#
[11] LDES specification: https://w3id.org/ldes/specification
[12] RMLMapper repository: https://github.com/RMLio/rmlmapper-java
[13] FnO functions for Change Data Capture: https://github.com/FnOio/idlab-functions-java/blob/main/src/main/java/be/ugent/knows/idlabFunctions/IDLabFunctions.java

Table 2

FnO functions for explicitly and implicitly communicated changes in members. Each type of change (creation, update, and deletion) has its own dedicated function.

FnO function	Purpose
`idlab-fn:explicitCreate`	Detect explicitly created members by checking if the member IRI existed already.
`idlab-fn:explicitUpdate`	Detect explicitly updated members by checking if the member IRI existed already.
`idlab-fn:explicitDelete`	Detect explicitly deleted members by checking if the member IRI existed already.
`idlab-fn:implicitCreate`	Detect implicitly created members by checking if the member IRI existed already.
`idlab-fn:implicitUpdate`	Detect implicitly updated members by checking if the member IRI already existed and its watched properties have changed.
`idlab-fn:implicitDelete`	Detect implicitly deleted members by marking each member IRI as seen and after processing all the members, returning the set of member IRIs which were not seen compared to the previous version.

Listing 4: RML usage of a CDC FnO function for detecting implicit updates in a dataset.

```
<#FunctionMap> a fnml:FunctionMap
  fnml:functionValue [
    # CDC FnO function to use.
    rr:predicateObjectMap [
      rr:predicate fno:executes ;
      rr:objectMap [ rr:constant idlab-fn:implicitUpdate; ];
    ];
    # IRI template of a member
    rr:predicateObjectMap [
      rr:predicate idlab-fn:iri ;
      rr:objectMap [ rr:template "https://example.org/{id}"; ]
    ];
    # Properties to watch, can be one or multiple, might differ from IRI template.
    rr:predicateObjectMap [
      rr:predicate idlab-fn:watchedProperty ;
      rr:objectMap [ rr:template "prop1={prop1}&prop2={prop2}"; ];
    ];
    # Directory path to store the function's state
    rr:predicateObjectMap [
      rr:predicate idlab-fn:state ;
      rr:objectMap [ rr:constant "/path/to/state"; rr:dataType xsd:string; ];
    ];
  ];
.
```

for each member, to be compared with the previous version of the dataset and determine if anything changed. `idlab-fn:implicitDelete` keeps a state (e.g., a hashmap) of all seen member IRIs. This state is used at the end of every processing round to detect missing – thus, deleted – members from the dataset. It returns a list with all the deleted member IRIs.

5.2. KG generation with RML and CDC FnO functions

We integrate our FnO functions with RML through FNML in a RML Triples Map (Listing 4). For each type of change we have an independent RML Triples Map with a conditional Subject Map referencing one of these FnO functions. When a function returns the IRI that it received as input, the Triples Map is executed completely, thus materializing the member and its properties. If no IRI is returned by the FnO function, the Triples Map is not executed, which means that the member did not change in the dataset. A separate Triples Map can be used (Listing 5) to generate additional metadata in the form of an event log to describe which type of changes took place in the dataset. A possible ontology to describe these change types is the W3C Recommendation Activity Streams 2.0 [13], but other ontologies can be used in the mappings as well, i.e., Change Detection Ontology [51, 53], or PROV-O [14].

[14] Activities of the PROV-O ontology: https://www.w3.org/TR/prov-o/

5.3. LDES Logical Target

We use LDES to publish a stream of dataset member changes events. LDES allows publishing only changed members as a stream which can be consumed by third-parties to replicate and synchronize with a dataset. We incrementally generate the detected changes (Section 5.2) and publish them as an LDES through an *Event Stream Target* in the RML mapping rules. An Event Stream Target is a subclass of an RML Logical Target with extra properties such as `rmlt:ldesBaseIRI`, `rmlt:ldes`, and `rmlt:generateImmutableIRI`. The Event Stream Target allows to indicate if unique and immutable IRIs are to be generated for each materialized member, as required by the LDES specification[15]. This way, consumers can uniquely identify individual changes in a dataset over time. We use an Event Stream Target for all member change types and one for generating an event log.

Listing 5 provides an example of our alignment between LDES and RML with CDC FnO functions. In this example, we use `as:Create`, `as:Update`, and `as:Delete` of W3C Activity Streams 2.0 [13] to semantically annotate the type of change for LDES consumers in the event log. Each change type has a dedicated RML Triples Map with an FnO function (Listing 5: lines 65-82, 84-106, 108-121) which outputs the generated members to a LDES Event Stream Target (Listing 5: lines 13-23). The W3C Activity Streams 2.0 event log is outputted to another LDES Event Stream Target (Listing 5: lines 1-11).

Listing 5: RML mapping for generating an LDES from a CSV file as data source (`data.csv`). Each change type has a separate Triples Map with an FnO function as Subject Map. These functions return an IRI when changes are detected, thus triggering the full execution of the Triples Map. The generated RDF triples are written to the LDES Event Stream Target with the necessary LDES properties specified in the LDES Logical Target. W3C ActivityStreams 2.0 is used to indicate the type of change through an LDES-based event log.

```
# Logical Target for outputting W3C ActivityStreams 2.0 event log as an LDES
<LDESLogicalTargetAS> a rmlt:EventStreamTarget ;
    rmlt:target [ a void:Dataset ; void:dataDump <file:///eventlog.nq> ; ] ;
    rmlt:serialization formats:N-Quads ;
    rmlt:ldes [ a ldes:EventStream ;
      ldes:timestampPath dct:created ; ldes:versionOfPath dct:isVersionOf ;
      tree:shape <https://example.org/shape/> ;
    ] ;
    rmlt:ldesBaseIRI <https://example.org/ldes/eventlog/> ;
    rmlt:ldesGenerateImmutableIRI "true"^^xsd:boolean .

# Logical Target for outputting data collection member changes as an LDES
<LDESLogicalTargetMember> a rmlt:EventStreamTarget ;
    rmlt:target [ a void:Dataset ; void:dataDump <file:///members.nq> ; ] ;
    rmlt:serialization formats:N-Quads ;
    rmlt:ldes [ a ldes:EventStream ;
      ldes:timestampPath dct:created ; ldes:versionOfPath dct:isVersionOf ;
      tree:shape <https://example.org/shape/> ;
    ] ;
    rmlt:ldesBaseIRI <https://example.org/ldes/members/> ;
    rmlt:ldesGenerateImmutableIRI "true"^^xsd:boolean .

# Input CSV file as datasource
<DataSource> a rml:LogicalSource ;
  rml:source "data.csv" ;
  rml:referenceFormulation ql:CSV .

# Dedicated named graph for each change type
# W3C ActivityStreams 2.0 eventlog generation of created members
<TriplesMapASCreate> a rr:TriplesMap ;
  rml:logicalSource <DataSource> ;
  rr:subjectMap [
    rr:constant "http://blue-bike.be/event/create" ; rr:class as:Create ;
    rml:logicalTarget <LDESLogicalTargetAS> ;
  ] .

# W3C ActivityStreams 2.0 eventlog generation of updated members
<TriplesMapASUpdate> a rr:TriplesMap ;
```

[15] LDES specification: https://w3id.org/ldes/specification#introduction

```
  rml:logicalSource <DataSource> ;
  rr:subjectMap [
    rr:constant "http://blue-bike.be/event/update" ; rr:class as:Update ;
    rml:logicalTarget <LDESLogicalTargetAS> ;
  ] .

# W3C ActivityStreams 2.0 eventlog generation of deleted members
<TriplesMapASDelete> a rr:TriplesMap ;
  rml:logicalSource <DataSource> ;
  rr:subjectMap [
    rr:constant "http://blue-bike.be/event/delete" ; rr:class as:Delete ;
    rml:logicalTarget <LDESLogicalTargetAS> ;
  ] .

# Data collection member
<PersonName> a rr:PredicateObjectMap ;
  rr:predicate schema:name ;
  rr:objectMap [ rml:reference "name" ; rr:datatype xsd:string ; ] .

# Dedicated Triples Map per change type
# Detection of explicit member creations with FnO function,
# if the member IRI is not found in the state, a new created member is generated.
<TriplesMapObjectCreate> a rr:TriplesMap ;
  rml:logicalSource <DataSource> ;
  rr:subjectMap [
    fnml:functionValue [
      rr:predicateObjectMap [ rr:predicate fno:executes ; rr:object idlab-fn:explicitCreate ; ] ;
      rr:predicateObjectMap [
        rr:predicate idlab-fn:iri ;
        rr:objectMap [ rr:template "https://example.org/member/{id}" ]
      ] ;
    ] ;
    rr:graph <http://example.org/event/create> ; rr:class foaf:Person ;
    rml:logicalTarget <LDESLogicalTargetMember> ;
  ] ;
  rr:predicateObjectMap <PersonName> .

# Detection of implicit member updates with FnO function
# Looks up the property 'name' of a member with the IRI of the member,
# if changed, an updated member is generated.
<TriplesMapObjectUpdate> a rr:TriplesMap ;
  rml:logicalSource <DataSource> ;
  rr:subjectMap [
    fnml:functionValue [
      rr:predicateObjectMap [ rr:predicate fno:executes ; rr:object idlab-fn:implicitUpdate ; ] ;
      rr:predicateObjectMap [
        rr:predicate idlab-fn:iri ;
        rr:objectMap [ rr:template "https://example.org/member/{id}" ]
      ] ;
      # Watch property 'name' of member for changes
      rr:predicateObjectMap [
        rr:predicate idlab-fn:watchedProperty ;
        rr:objectMap [ rr:template "name={name}" ]
      ] ;
    ] ;
    rr:graph <http://blue-bike.be/event/update> ; rr:class foaf:Person ;
    rml:logicalTarget <LDESLogicalTargetMember> ;
  ] ;
  rr:predicateObjectMap <PersonName> .

# Detection of implicit member deletions with FnO function by IRI
# If member IRI is removed in the new version, a member as tombstone is generated.
<TriplesMapObjectDelete> a rr:TriplesMap ;
  rml:logicalSource <DataSource> ;
  rr:subjectMap [
    fnml:functionValue [
      rr:predicateObjectMap [ rr:predicate fno:executes ; rr:object idlab-fn:implicitDelete ; ] ;
      rr:predicateObjectMap [
        rr:predicate idlab-fn:iri ; rr:objectMap [ rr:template "https://example.org/member/{id}" ] ] ;
    ] ;
    rr:graph <http://blue-bike.be/event/delete> ; rr:class foaf:Person ;
    rml:logicalTarget <LDESLogicalTargetMember> ;
  ] .
```

5.4. Example demonstrating our approach

We demonstrate our approach through an example dataset (Figure 3) which consists of an initial version (Table 3a) and an implicitly updated version (Table 3b). Figure 3 also visualizes this example in the pipeline. The following changes (Table 3c) are extracted between the initial data (Table 3a) and the newer version (Table 3b) through Change Data Capture:

- **ID 0**: Excluded from the changes as it is unchanged.
- **IDs 1,2**: Members are updated because the 'age' property changed, marked as 'update'.
- **ID 3**: A tombstone is generated since it is removed, marked as 'delete'.
- **ID 4**: New member, marked as 'create'

The resulting RDF quads of the initial (Listing 6) and updated (Listing 7) datasets consist of 3 named graphs[16]: `Create`, `Update`, and `Delete`, indicating the change type of each member. These named graphs are described in our examples (Listing 8) using W3C Activity Streams 2.0 [13][17]. On each execution of our approach, the resulting RDF quads (Listing 7) are generated depending on the type of change compared to the previous execution.

ID	Name	Age
0	The Machine	0
1	Harold Finch	44
2	John Reese	38
3	Agent Carter	36

(a) *Initial* dataset produces 4 members with change type 'create'.

ID	Name	Age
0	The Machine	0
1	Harold Finch	46
2	John Reese	40
4	Root	35

(b) *Changed* dataset has 2 updated, 1 unchanged, 1 created, and 1 deleted member(s).

ID	Name	Age
1	Harold Finch	46
2	John Reese	40
~~3~~	~~Agent Carter~~	~~36~~
4	Root	35

(c) Extracted changes by the Change Data Capture functions.

Table 3

Example dataset with the initial dataset (left), a newer version of the dataset (middle), and extracted changes by the Change Data Capture functions between the initial dataset and the newer version.

[16] We split up in 3 named graphs by change type for showcasing purposes. However, some use cases are only interested in a specific change type. For example within public transport: history of canceled routes requires only the Delete named graph while providing traveling information requires all named graphs to indicate added, updated, and deleted routes.

[17] https://www.w3.org/ns/activitystreams

Listing 6: The materialized KG in TriG of the *initial* dataset. All dataset members are materialized because this is the first time the dataset is processed. Thus, they are part of the Create named graph. The subject IRIs are versioned as required by LDES to publish versioned members.

```
:Created {
 <http://ex.org/Mbr0#0> a foaf:Person .
 <http://ex.org/Mbr0#0> foaf:name "The Machine" .
 <http://ex.org/Mbr0#0> foaf:age "0"^^xsd:int .

 <http://ex.org/Mbr1#0> a foaf:Person .
 <http://ex.org/Mbr1#0> foaf:name "Harold Finch" .
 <http://ex.org/Mbr1#0> foaf:age "44"^^xsd:int .

 <http://ex.org/Mbr2#0> a foaf:Person .
 <http://ex.org/Mbr2#0> foaf:name "John Reese" .
 <http://ex.org/Mbr2#0> foaf:age "38"^^xsd:int .

 <http://ex.org/Mbr3#0> a foaf:Person .
 <http://ex.org/Mbr3#0> foaf:name "Agent Carter" .
 <http://ex.org/Mbr3#0> foaf:age "36"^^xsd:int .
}
```

Listing 7: The materialized KG in TriG of the *changed* dataset. A member is deleted (ID 3), a member's age property is updated (ID 1 and 2), and a new member is created (ID 4). Unchanged members (ID 0) are not materialized. The subject IRIs are versioned as required by LDES to publish versioned members.

```
:Created {
 <http://ex.org/Mbr4#0> a foaf:Person .
 <http://ex.org/Mbr4#0> foaf:name "Root" .
 <http://ex.org/Mbr4#0> foaf:age "35"^^xsd:int .
}

:Updated {
 <http://ex.org/Mbr1#1> a foaf:Person .
 <http://ex.org/Mbr1#1> foaf:name "Harold Finch" .
 <http://ex.org/Mbr1#1> foaf:age "46"^^xsd:int .

 <http://ex.org/Mbr2#1> a foaf:Person .
 <http://ex.org/Mbr2#1> foaf:name "John Reese".
 <http://ex.org/Mbr2#1> foaf:age "40"^^xsd:int .
}

:Deleted {
 <http://ex.org/Mbr3#1> a foaf:Person .
}
```

Listing 8: The different named graphs are described using the W3C Activity Streams 2.0 ontology as `as:Create`, `as:Update`, and `as:Delete`.

```
# Named graph for created members of data collection
:Created a as:Create ;
  as:actor <http://ex.org/data-collection> .
# Named graph for updated members of data collection
:Updated a as:Update ;
  as:actor <http://ex.org/data-collection> .
# Named graph for deleted members of data collection
:Deleted a as:Delete ;
  as:actor <http://ex.org/data-collection> .
```

6. Evaluation

In this section, we describe our methodology to evaluate our approach on several datasets using both synthetic and real-world data. First, we discuss our datasets and how they can be classified according to the different change signaling strategies identified in this work (Section 6.1). Then, we introduce our evaluation setup (Section 6.2).

6.1. Methodology

We apply our approach on (i) a set of artificial test cases (Section 6.1.1) to verify if our approach is able to handle all change signaling strategies listed in Section 4.2; (ii) an extended version of the GTFS Madrid Benchmark (Section 6.1.2) to measure the scalability, performance and resource consumption of our approach for incrementally generating KGs; and (iii) 5 different real-world datasets (Section 6.1.3) to investigate its performance and resource impact on different types of datasets (Section 4.2),

6.1.1. Functionality

Through a set of test cases (Table 4), we evaluate 3 change signaling strategy dimensions (history availability, change communication, and change type (Section 4.2)) to determine if our approach: (i) can handle all dimensions, and (ii) is feasible to implement. We combined the 3 types of history availability (latest state, latest changes, and full history) with the 2 types of change signaling (explicit and implicit), and 3 change types (create, update, delete) resulting into 18 test-cases. We also included a test case where no change is applied to verify if an unchanged dataset is handled correctly by our approach, which yields additionally 6 more test-cases (24 in total). Since 2 combinations are not possible (Section 4: implicit deletion signaling for full history and latest changes), we removed these from the test cases, bringing the number of test cases to 22.

Table 4

22 test cases for detecting changes in datasets. Our approach is feasible to implement and covers detecting all possible change types and change signaling strategies.

# test cases	History availability	Signaling	Change type	Purpose
8	Latest state	Explicit	No change	Empty output
			Create	Added new entity
			Update	Existing entity changed
			Delete	Tombstone deleted entity
		Implicit	No change	Empty output
			Create	Added new entity
			Update	Existing entity changed
			Delete	Tombstone deleted entity
7	Latest Changes	Explicit	No change	Empty output
			Create	Added new entity
			Update	Existing entity changed
			Delete	Tombstone deleted entity
		Implicit	No change	Empty output
			Create	Added new entity
			Update	Existing entity changed
7	Full History	Explicit	No change	Empty output
			Create	Added new entity
			Update	Existing entity changed
			Delete	Tombstone deleted entity
		Implicit	No change	Empty output
			Create	Added new entity
			Update	Existing entity changed

We validate that all combinations of these dimensions are possible and use these test cases to verify that our implementation covers all possible scenarios. The test cases are publicly available on GitHub[18].

6.1.2. GTFS Madrid Benchmark extension

We extended the GTFS Madrid Benchmark [15] data generator[19] to allow generating different versions of the benchmark data by supporting creations, updates, and deletions based on the GTFS data model. We implemented creations, updates, and deletions in the GTFS Madrid Benchmark similar to how real-world GTFS datasets are changed at the Belgian public transport agencies De Lijn and NMBS. This way, we keep the characteristics of GTFS Madrid Benchmark which aims at using real-world data from the metro in Madrid. Creations are applied by adding GTFS `routes` and their associated GTFS `trips`, `stops`, `stoptimes`, and `service` entities. For example: 25% creations will provide 25 % additional new routes with respect to the original amount of routes.

[18]Repository: https://github.com/RMLio/rml-ldes-testcases, DOI: https://doi.org/10.5281/zenodo.10171394

[19]Repository: https://github.com/oeg-upm/gtfs-bench, DOI: https://doi.org/10.5281/zenodo.10256865

Updates are performed by modifying the GTFS services. For example: 50% updates will modify 50% of the GTFS service entries in the GTFS calendar. Deletes are applied by removing GTFS routes and their associated trips and services. Example: 10% deletes will remove 10% of the routes in the original data, together with the associated data. We use a random number generator to randomly select GTFS routes and services.

We extend the GTFS Madrid Benchmark with 4 additional configuration parameters:

- *seed*: The random seed value used for configuring the random number generator.
- *additions*: The percentage defining how many creations must be added to the generated data.
- *modifications*: The percentage defining how many updates must be performed on the generated data.
- *deletions*: The percentage defining how many deletions must be applied on the generated data.

We aim on analyzing the impact of our approach on datasets of varying size and change characteristics, by measuring execution time and resource usage (storage, CPU time, and RAM usage) to detect and materialize changes into a KG (CHANGE) or materialize the complete KG (ALL). In particular, we analyze the overhead of our approach for detecting changes and the reduction in execution time and resource usage achieved by only materializing the changes instead of the complete KG. The reduction of our approach might be affected by the amount of changes, type of changes, and the dataset size. Therefore, we use data size scales (1, 10, and 100) with a fixed change percentage of 50%, equally divided among the different change types (16.67% creations, 16.67% updates, and 16.67% deletions), to obtain reference measurements of increasing dataset size scales. This way, we avoid that other dimensions e.g., change percentage and change types impacts our analysis of results from the data size dimension. We divide the changes equally among change types and use scale 100 to analyze the change percentage dimension (0%, 25%, 50%, 75%, and 100%). We also experiment with the type of change by using a fixed change percentage of 50% and scale 100 for either creations, updates, or deletions to analyze the impact of each change type on execution time and resources. For each experiment we use 10 new versions of the GTFS dataset which we apply as updates over an initial base version. We select 50% as fixed change percentage to avoid outliers from 0% or 100% and scale 100 for analyzing the change type dimension.

6.1.3. Real-world datasets

We apply our approach on 5 types of real-world datasets: (i) BlueBike & JCDecaux bike-sharing data; (ii) timetables in GTFS format from the Belgian public transport agencies NMBS and De Lijn; (iii) OpenStreetMap geographical data; (iv) dynamic message boards from the Flemish traffic controller center Vlaams Verkeerscentrum; and (v) meteorological sensor data from the Belgian meteorological institute KMI. These datasets stand as representative examples for all the identified change signaling strategies (Table 5) regarding change communication, change types, and history availability. For each dataset, we collected released versions during 24 hours, with the exception of De Lijn and NMBS, since they only provide a new version per day. In this case, we collect new versions during a week. We use a timeframe instead of the number of versions to have different frequencies when new versions of the datasets are published, e.g., VVC is changed more frequently (every 2-3 secs) compared to KMI (every 10 mins). Next we describe in detail each of the analyzed datasets, which are also summarized in Table 5.

BlueBike & JCDecaux These datasets provide information related to bike-sharing services including data about stations and currently available bikes. They are publicly accessible via HTTP APIs[20]. Both datasets communicate creations explicitly through unique identifiers, and updates implicitly by modifying the number of available bikes and a last updated timestamp property. However, for some stations, they do not provide this timestamp. Thus, we use implicit change detection by *watching* the available bikes at each station. Deletes are implicitly communicated by removing stations from the dataset. Regarding history availability, these datasets follow the *latest state* strategy, overwriting the whole dataset with the current state of each member on every new version. We use a GBFS-based vocabulary[21] to transform this bike-sharing data into RDF and retrieve the data every minute to check for changes.

NMBS & De Lijn These datasets contain public transport timetables in GTFS format, which are updated daily. They are publicly accessible as data dumps[22]. In GTFS timetables, creations are explicitly communicated by adding

[20] BlueBike: https://api.blue-bike.be/pub/location; JCDecaux: https://developer.jcdecaux.com/#/home

[21] https://github.com/jiaoxlong/gbfs-json-schema/tree/gbfs-ld/GBFS-LD/v2.3

[22] NMBS: https://gtfs.irail.be/nmbs/gtfs/latest.zip, De Lijn: https://gtfs.irail.be/de-lijn/de_lijn-gtfs.zip

new entries with a new ID. Updates are implicit, silently changing properties of members. They do not provide unique identifiers nor timestamps for identifying changes. Deletes are also implicitly communicated by silently removing members. GTFS also follows the *latest state* approach for history availability. We use the Linked GTFS ontology[23] to transform the GTFS timetables into RDF, based on the RML mappings of the GTFS-Madrid-Bench [15].

OpenStreetMap (OSM) This is a crowdsourced geographical dataset which provides a feed [24] of changed data on a minutely basis. These updates publish OpenStreetMap's latest changes with explicit change communication for all types of change: creations, updates, and deletions are all identified with a unique ID. OpenStreetMap data is mapped using the Routable Tiles specification [58], which provides an ontology for OpenStreetMap's Nodes and Ways. OSM follows the *latest changes* approach regarding history availability in their replication feeds (although historical full latest-state data dumps are also available). Since OpenStreetMap already explicitly provides unique IDs for each change, no additional processing is needed by our approach. Thus, semantically publishing the dataset changes can be achieved with regular RML mappings without applying the CDC functions on the dataset. We include this dataset in this work to show the wide range of dataset types regarding change signaling, history availability, and change types, supported by our approach.

Vlaams Verkeerscentrum (VVC) This dataset contains live information from traffic boards that include data about traffic jams, road works, accidents, and other incidents on the road across Flanders. These dynamic message boards are changed every 2–3 seconds and published as Open Data. We include this dataset due to its high update frequency, compared to the other datasets: the RDF generation process must keep up with this high frequency to avoid high latency. We use the DATEX II v3 VMS ontology[25] for the content of the dynamic message board messages and collect the data every 3 seconds. This dataset follows the *latest state* approach for history availability and contains a modified timestamp for each published message, which allows for explicit change communication. Messages are not modified in this dataset, but republished as new message instances instead. Therefore, there are no updates, only creations and deletions. Deletions are marked in the data as an out-of-service board. If a board is marked as out-of-service, we consider it deleted from the dataset. If the board is put in service again, it is considered as a creation.

Koninklijk Meteorologisch Instituut van België (KMI) This dataset is published by the Belgian meteorological institute which allows access to the measurement history of their weather sensors. The dataset contains the *full history* of sensor measurements labeled with unique IDs. However, it does not explicitly communicate deletions. Since detecting implicit deletions in a full history is not directly possible (Section 4), and each measurement has a unique ID, only creations are possible in this dataset. We use the W3C Semantic Sensor Network ontology (SSN)[26] to transform the sensors' measurements into RDF and collect the data every 10 minutes.

Table 5

Real-world datasets addressed in our evaluation. Each dimension of dataset types is covered: change signaling, change type and history availability. Some datasets only provide certain change types, inapplicable change types are marked as Not Applicable (NA).

Dataset	Number of collected versions	Collection frequency	Change communication			History availability
			Create	Update	Delete	
BlueBike	1440	1 min	Explicit	Implicit	Implicit	Latest state
JCDecaux	1440	1 min	Explicit	Implicit	Implicit	Latest state
NMBS	7	1 day	Explicit	Implicit	Implicit	Latest state
De Lijn	7	1 day	Explicit	Implicit	Implicit	Latest state
OSM	1440	1 min	Explicit	Explicit	Explicit	Latest changes
VVC	28760	3 secs	Explicit	NA	Explicit	Latest state
KMI	144	10 mins	Explicit	NA	NA	Full history

[23] http://vocab.gtfs.org/terms#
[24] https://planet.openstreetmap.org/replication/
[25] https://datex2.eu/vocab/3/Vms/
[26] https://www.w3.org/TR/vocab-ssn/

6.1.4. Ingestion

In practice, the outcome of a KG generation process is typically a set of RDF triples/quads that are subsequently ingested into an RDF triplestore, so that they can be queried and used in applications. In a traditional full re-materialization KG generation scenario (ALL), the ingestion process usually entails a complete deletion of the KG in the triplestore (if any), followed by an insertion of all the newly generated RDF quads. However, the materialized members generated by our approach after the CDC process (CHANGE), do not constitute on their own a complete and standalone version of the KG (except for the initial generation), and require an additional interpretation step to determine the proper operations to be performed over the triplestore that will bring the KG to its latest state. We implemented a proof-of-concept library[27] that performs this interpretation step (Figure 4) and produces the corresponding SPARQL UPDATE queries for each materialized member, based on their change semantics (i.e., type of change). By using a triplestore as the same system to make the standalone KG accessible for both the ALL and CHANGE strategy, we make sure we are not monitoring side-effects when using reconciliation systems that are different for the ALL and CHANGE strategy.

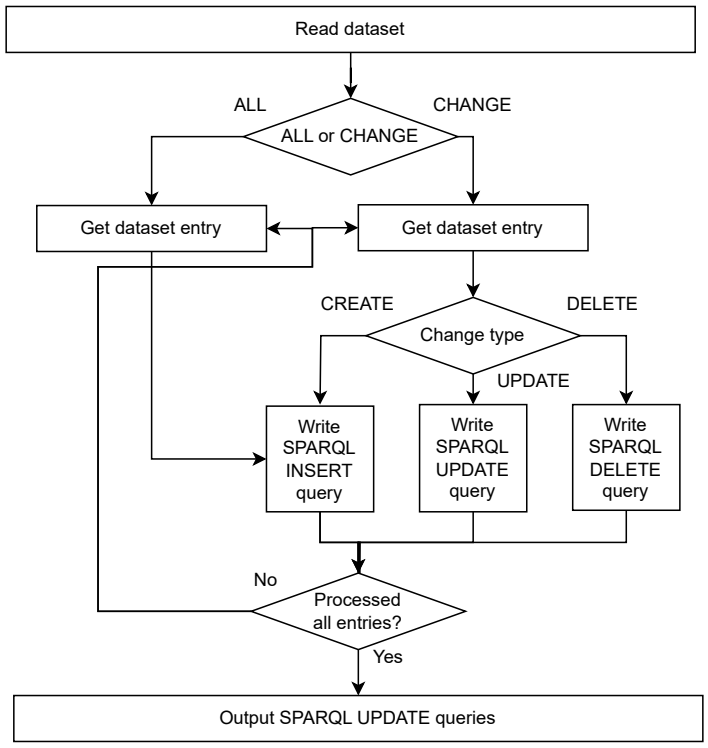

Fig. 4. `incrml2sparql` transforms the generated RDF into SPARQL UPDATE queries for ingestion by a triplestore. ALL datasets always generate a SPARQL INSERT query for all data entries. CHANGE datasets generate a SPARQL INSERT/UPDATE/DELETE based on the type of change and only for the changed dataset entries.

To guarantee a fair comparison of our approach (CHANGE) with a traditional approach (ALL), we initially observe the number of RDF quads generated by each approach. Our premise is that less RDF quads translate into faster ingestion and thus more efficiently updated KGs. To verify this, we also measured the time that takes to ingest via SPARQL `UPDATE` (i) the fully materialized KGs (ALL); and (ii) to produce and execute the SPARQL `UPDATE`

[27] https://github.com/julianrojas87/incrml2sparql.git

queries corresponding to the materialized members (CHANGE), for every real world dataset considered in this work (Section 6.1.3). We perform this measurement over a Virtuoso triplestore v7.2.14[28].

6.2. Evaluation setup

We evaluate our approach on the described datasets and a modified version of the GTFS Madrid Benchmark by measuring the following metrics: (i) execution time to detect changes and materialize the complete or changed parts of a dataset into RDF; (ii) CPU usage to determine how CPU intensive is our generation approach; (iii) peak RAM memory usage to investigate the memory overhead of our approach; and (iv) storage usage of the generated KG on each new version to verify the impact of our approach for reducing storage of historical data. We execute this evaluation with and without our approach using the RMLMapper v6.3.0[29] to investigate how our approach impacts the measured metrics during RDF KG generation.

For each experiment, we materialize a KG from a given dataset and its corresponding updated versions as RDF quads. We manually verify if the output is complete and correct for each experiment. We compare 2 KG generation-and-version strategies:

- *ALL* is the traditional strategy to generate versioned RDF KGs, where a set of RDF quads are initially produced and completely re-materialized when a new version of the data source(s) is/are available.
- *CHANGE* corresponds to our approach to incrementally generated RDF KGs, where we initially materialize a complete version of the KG, but only materialize into RDF the actual changed members upon updates of the data source(s), not the complete KG. We also opt to use the LDES Logical Target, to add additional metadata that semantically describes how the KG changes to help consumers reconcile the changes into their own KG, e.g., via SPARQL `UPDATE` queries over a triplestore.

In this work, we focus on measuring the generation step of both strategies to investigate the feasibility and impact of our approach over materializing the complete KG on each dataset version. Since our focus is the generation, we only perform a comparative experiment to have an indication of how our approach affects ingestion into triplestores using SPARQL `UPDATE` queries, given it is the standardized way to update KGs in any triplestore. Moreover, most triplestores offer custom approaches to update quads besides SPARQL `UPDATE`. Each of these ingestion approaches have their own benefits and drawbacks depending on the triplestore and vendor. Thus, we focus on comparing the amount of generated quads for each dataset version with and without our approach since we want to investigate how our approach affects KG generation itself.

The experiments are executed on an Ubuntu 22.04.1 LTS machine (Linux 5.15.0-83-generic, x86_64) with an Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60Ghz, 48GB RAM memory and 2GB swap memory. Java JVM heap space is set to 90% of the available RAM memory. All experiments are executed 5 times, from which we report the median measurements of the execution. All resources and instructions to reproduce the experiments are available on Zenodo[30].

7. Results

In this section, we present the results obtained during the execution of our evaluation with and without our approach. We first perform a functional evaluation through a set of test cases (Section 7.1). Then, we evaluate the performance in terms of execution time and computing resources of our approach on an extended version of the GTFS Madrid Benchmark (Section 7.2), and real-world datasets (Section 7.3). We then reconcile the results of both the ALL and CHANGE scenario (i.e., make the latest version of a knowledge graph accessible after a set of changes) and report the results of our comparative experiment after triplestore ingestion (Section 7.4). In Section 7.5, we discuss the impact of our approach on the measured metrics.

[28] OpenLink's Virtuoso triplestore: https://virtuoso.openlinksw.com/

[29] Our approach is engine-agnostic, thus can be implemented in any KG construction engine that support a declarative mapping language and data transformations.

[30] DOI: https://doi.org/10.5281/zenodo.10171156

7.1. Functionality

We integrated our set of 22 test cases (Table 4 of Section 6.1.1) as unit tests in the RMLMapper v6.3.0[31] to validate if all possible change types and change signaling strategies are covered by our approach and are feasible to implement. Through these test cases, we confirm the feasibility of our approach and its coverage of all possible dimensions regarding change signaling and history availability.

7.2. GTFS Madrid Benchmark

In this subsection, we discuss the results obtained for the 2 generation-and-version strategies (Section 6.2) `ALL` and `CHANGE` per scenario: (i) scaling data size, (ii) scaling amount of changes, and (iii) different change types. We compare the results of the initial version of the KG (Table 6), and the updates applied after the initial version (Table 7). We execute each experiment by starting with the base dataset to generate the initial KG and apply the corresponding dataset updates on top of the base dataset. Each experiment is executed 5 times from which the median value is taken for each metric of each dataset version. We report the average of these median metrics, e.g., execution time, CPU time, and memory usage across multiple dataset updates in Tables 6 and 7. Storage usage is reported as the sum of KG sizes generated from the base dataset and all its updates (Table 8).

The results show that our approach **significantly reduces the storage requirements** for storing multiple versions of a KG and **reduces execution time and CPU time, without impacting memory usage** (Table 7). For the **initial KG generation** (Table 6), **the overhead of our approach causes an increase in execution time (2.22x in average), storage (1.57x in average), and CPU time (1.87x in average), while memory usage is mostly unaffected (1.15x in average)**. This is the result of constructing the initial KG but also initializing the state to keep track of dataset members, as the state is completely empty during the initial KG construction. Only the tracking of dataset members by our approach has a slight increase (+2.30% compared to average) in memory for `CHANGE`. Our approach remains unaffected by the amount of changes or type of change, being solely impacted by the dataset size.

Table 6

Initial execution results for all GTFS Madrid Benchmark for strategies `ALL` (no change detection) and `CHANGE` (with change detection). We scale one dimension for each experiment to analyze its impact. $GTFS_{SCALE}$ and $GTFS_{TYPE}$ use a fixed change percentage (50%). $GTFS_{CHANGE}$ and $GTFS_{TYPE}$ use a fixed data size (scale 100) which results into the same storage usage for the initial generation. $GTFS_{SCALE}$ and $GTFS_{CHANGE}$ spread the changes equally among the change types. Lower is better.

Scenario	Execution time (s)			CPU time (s)			Peak memory (GB)		
	ALL	CHANGE	Ratio A/C	ALL	CHANGE	Ratio A/C	ALL	CHANGE	Ratio A/C
$GTFS_{SCALE}$ 1	12.29	16.18	0.76	26.80	36.31	0.74	4.25	4.44	0.96
$GTFS_{SCALE}$ 10	82.15	124.84	0.66	129.45	206.05	0.63	7.57	14.00	0.54
$GTFS_{SCALE}$ 100	922.43	1943.38	0.47	2 260.62	4 626.85	0.49	46.30	51.78	0.89
$GTFS_{CHANGE}$ 0%	919.25	2 736.11	0.34	2 259.27	5 002.98	0.45	45.29	52.30	0.87
$GTFS_{CHANGE}$ 25%	914.98	1 575.10	0.58	2 303.53	4 521.50	0.51	46.38	51.64	0.90
$GTFS_{CHANGE}$ 50%	922.43	1 943.38	0.47	2 260.62	4 626.85	0.49	46.30	51.78	0.89
$GTFS_{CHANGE}$ 75%	924.12	2 357.18	0.39	2 224.46	4 603.18	0.48	46.77	52.09	0.90
$GTFS_{CHANGE}$ 100%	912.37	1 420.53	0.64	2 336.65	4 431.36	0.53	46.32	51.70	0.90
$GTFS_{TYPE}$ CREATE	990.30	1 397.49	0.71	2 448.92	4 414.65	0.55	48.07	51.73	0.93
$GTFS_{TYPE}$ UPDATE	924.16	1 910.59	0.48	2 239.66	4 507.61	0.50	47.13	51.64	0.91
$GTFS_{TYPE}$ DELETE	929.74	1 671.17	0.56	2 302.00	4 702.38	0.49	46.58	52.30	0.89

[31] Repository: https://github.com/RMLio/rmlmapper-java, DOI: https://doi.org/10.5281/zenodo.10142511

Table 7

Execution results for all GTFS Madrid Benchmark for strategies `ALL` (no change detection) and `CHANGE` (with change detection). Only results of dataset updates are included, initial execution is not included. We scale one dimension for each experiment to analyze its impact. $GTFS_{SCALE}$ and $GTFS_{TYPE}$ use a fixed change percentage (50%). $GTFS_{CHANGE}$ and $GTFS_{TYPE}$ use a fixed data size (scale 100) which results into the same storage usage for the initial generation. $GTFS_{SCALE}$ and $GTFS_{CHANGE}$ spread the changes equally among the change types. Lower is better.

Scenario	Execution time (s)			CPU time (s)			Peak memory (GB)		
	ALL	CHANGE	Ratio A/C	ALL	CHANGE	Ratio A/C	ALL	CHANGE	Ratio A/C
$GTFS_{SCALE}$ 1	12.20	8.74	1.40	27.40	26.58	1.03	4.25	4.25	1.00
$GTFS_{SCALE}$ 10	83.41	44.93	1.87	132.63	88.73	1.49	7.53	12.58	0.60
$GTFS_{SCALE}$ 100	928.93	473.55	1.96	2 306.50	1 083.96	2.13	47.41	47.01	1.00
$GTFS_{CHANGE}$ 0%	927.82	470.73	1.97	2 292.39	1 072.47	2.14	47.19	47.33	1.00
$GTFS_{CHANGE}$ 25%	912.50	481.62	1.89	2 303.02	1 090.74	2.11	46.86	47.29	0.99
$GTFS_{CHANGE}$ 50%	928.93	473.55	1.96	2 306.50	1 083.96	2.13	47.41	47.01	1.00
$GTFS_{CHANGE}$ 75%	929.53	466.87	1.99	2 292.81	1 056.88	2.17	47.21	47.19	1.00
$GTFS_{CHANGE}$ 100%	914.68	460.30	1.98	2 301.77	1 063.46	2.16	47.05	47.35	0.99
$GTFS_{TYPE}$ CREATE	1 021.04	454.06	2.25	2 386.46	1 063.90	2.24	48.55	46.61	1.04
$GTFS_{TYPE}$ UPDATE	953.32	479.87	1.99	2 298.29	967.63	2.38	46.77	47.32	0.99
$GTFS_{TYPE}$ DELETE	922.04	464.78	1.98	2 266.11	1 056.56	2.14	46.69	47.33	0.99

Table 8

Storage usage for initial and all updates per strategy of the GTFS Madrid Benchmark. We scale one dimension for each experiment to analyze its impact. $GTFS_{SCALE}$ and $GTFS_{TYPE}$ use a fixed change percentage (50%). $GTFS_{CHANGE}$ and $GTFS_{TYPE}$ use a fixed data size (scale 100) which results into the same storage usage for the initial generation. $GTFS_{SCALE}$ and $GTFS_{CHANGE}$ spread the changes equally among the change types. Total storage usage is the sum of the base dataset and all applied updates upon it. Lower is better.

Scenario	Initial storage usage (kb)			Total storage usage (kb)		
	ALL	CHANGE	Ratio A/C	ALL	CHANGE	Ratio A/C
$GTFS_{SCALE}$ 1	92 537.13	116 662.92	0.79	1 023 192.74	125 962.43	8.12
$GTFS_{SCALE}$ 10	732 959.66	1 168 374.14	0.63	8 100 043.47	1 234 254.86	6.56
$GTFS_{SCALE}$ 100	7 347 846.47	11 701 971.43	0.62	81 178 874.08	12 323 348.17	6.59
$GTFS_{CHANGE}$ 0%	7 347 846.47	11 701 971.43	0.62	80 826 311.20	11 703 851.89	6.91
$GTFS_{CHANGE}$ 25%	7 347 846.47	11 701 971.43	0.62	81 005 737.97	12 019 442.49	6.74
$GTFS_{CHANGE}$ 50%	7 347 846.47	11 701 971.43	0.62	81 178 874.08	12 323 348.17	6.59
$GTFS_{CHANGE}$ 75%	7 347 846.47	11 701 971.43	0.62	81 355 130.62	12 631 882.64	6.44
$GTFS_{CHANGE}$ 100%	7 347 846.47	11 701 971.43	0.62	81 530 274.97	12 934 632.82	6.30
$GTFS_{TYPE}$ CREATE	7 347 846.47	11 701 971.43	0.62	82 031 098.47	13 530 545.51	6.06
$GTFS_{TYPE}$ UPDATE	7 347 846.47	11 701 971.43	0.62	80 826 311.20	11 712 313.42	6.90
$GTFS_{TYPE}$ DELETE	7 347 846.47	11 701 971.43	0.62	80 689 654.29	11 727 476.53	6.88

Scaling data size Our approach **reduces the execution time and resource usage (storage, CPU, and memory) of the different GTFS Madrid Benchmark scales** (1, 10, 100) by only materializing the actual changed members of the KG (Figure 5). However, **the initial generation of the KG from the base dataset has a longer execution time and higher resource consumption since all dataset members must be checked to initialize the internal state for tracking the members.** Moreover, the storage usage increases because of the metadata we generate to indicate that all members were created. When solely considering the dataset changes, the overhead of detecting changes and LDES event stream metadata is mostly noticeable with $GTFS_{SCALE}$ 1 (execution time reduced by 30%, no CPU time reduction). For larger GTFS scales the impact of the overhead is less noticeable, since they achieve an overall reduction in execution time (50% faster) and resource usage (CPU time is reduced up to 53%, and up to 8.10x less storage usage). Only for $GTFS_{SCALE}$ 10, the memory usage increases since the Java VM does not perform garbage collection, given that the Java heap space still has enough free space. This is not the case for $GTFS_{SCALE}$ 100 where memory usage is similar again compared to no change detection.

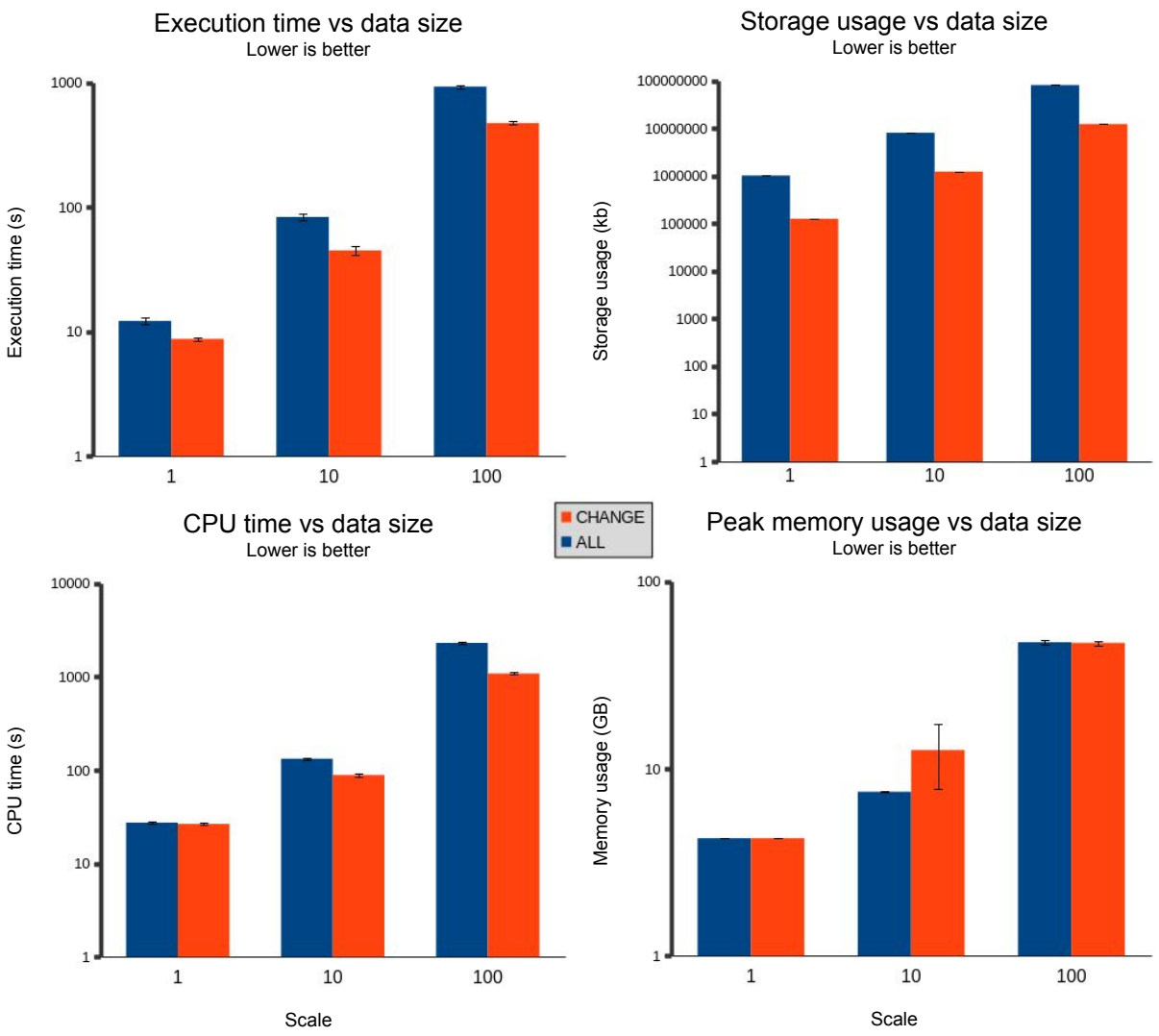

Fig. 5. GTFS Madrid Benchmark results for different data scales with a fixed amount of changes (50%). Only results of dataset updates are included, initial execution is not included.

Scaling amount of changes We observe an **increase in storage usage when the amount of changes increases**, while it has no impact on the resource consumption (Figure 6). This is due to each dataset member being evaluated regardless of the amount of changes. Similar to scaling the data size, the initial KG generation from the base dataset introduces overhead to initialize the state for tracking changes. When only considering the dataset updates, if no changes are found between 2 versions (0%), only the overhead of generating LDES event stream metadata is affecting storage usage, resulting in the lowest storage usage (11.7GB in total). Scaling up the amount of changes, increases the storage usage (12.93GB in total).

Type of changes **The type of change in GTFS Madrid Benchmark mostly affects storage usage** because our algorithms evaluate each dataset member, for each change type (Figure 7). Therefore, CPU and memory usage is unaffected by the type of change. The same overhead of initially generating the KG applies as mentioned in the previous sections. Creations cause a higher storage usage because they result in a higher number of materialized RDF quads. In the case of GTFS-Madrid-Benchmark, properties of new GTFS routes and associated data such

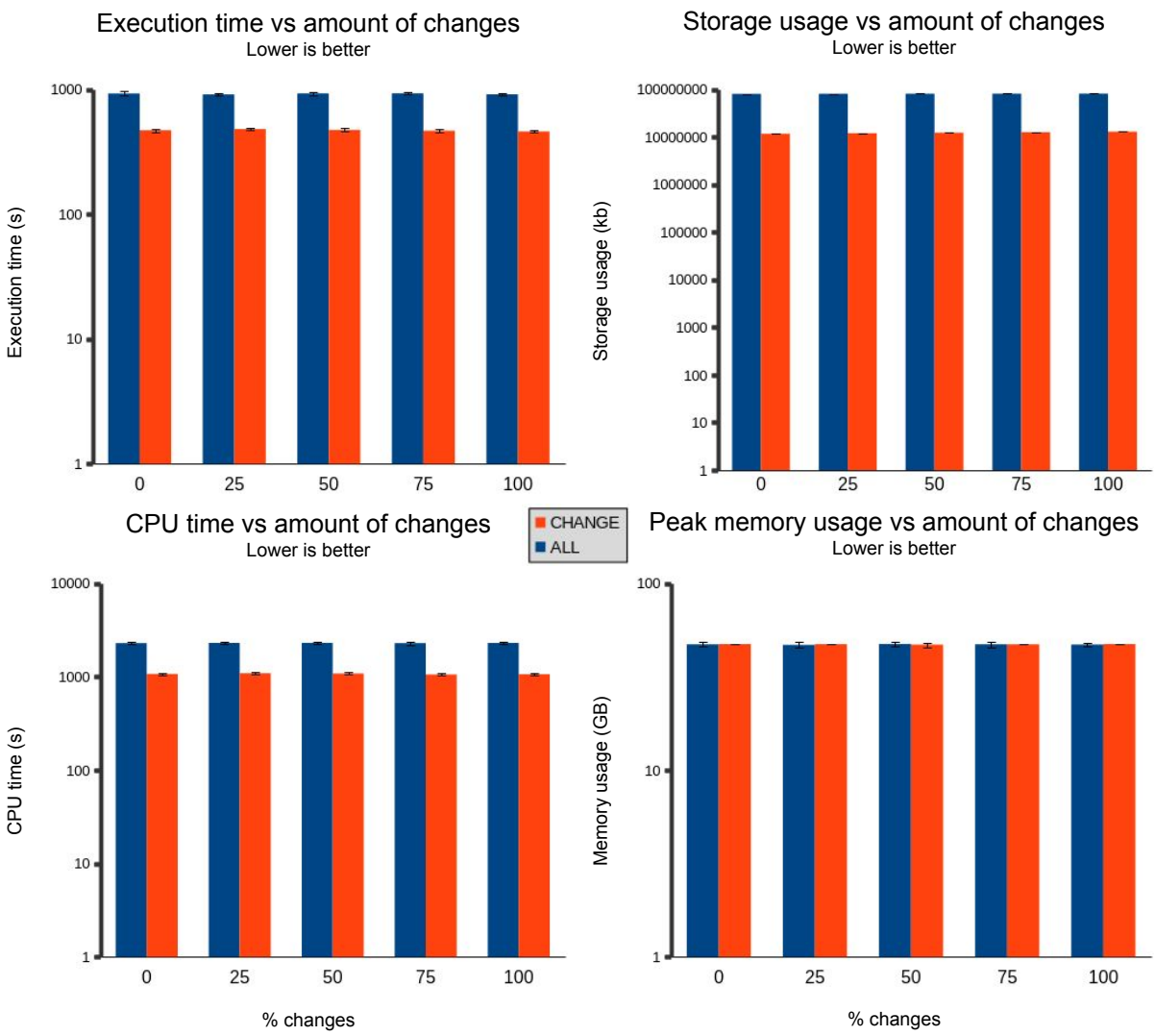

Fig. 6. GTFS Madrid Benchmark results for different amount of changes with a fixed data size (scale 100). Only results of dataset updates are included, initial execution is not included.

as GTFS trips, shapes, etc., must be generated as new triples. Deletions only keep a tombstone of the member, e.g., a deleted GTFS route or trip. Updates also trigger the generation of new RDF quads, e.g., the GTFS trip's service dates are modified in the GTFS Madrid Benchmark, which causes fewer changes among the GTFS datasets compared to creates or deletes. GTFS trip's service dates do not affect other information about a GTFS trip such as its route, shapes, or stops. Thus, updates have a lower storage usage for GTFS Madrid Benchmark since fewer changes happened in the GTFS dataset.

7.3. Real-World datasets

In this subsection, we discuss the results obtained from applying our approach over the set of real-world datasets described in Section 6.1.3, with respect to storage usage, execution time, CPU time, and memory usage. For each dataset, we measured these metrics following strategies `ALL` (no change detection) and `CHANGE` (using our change detection approach). Table 9 reports the initial execution results, while Table 10 shows the impact of our approach on

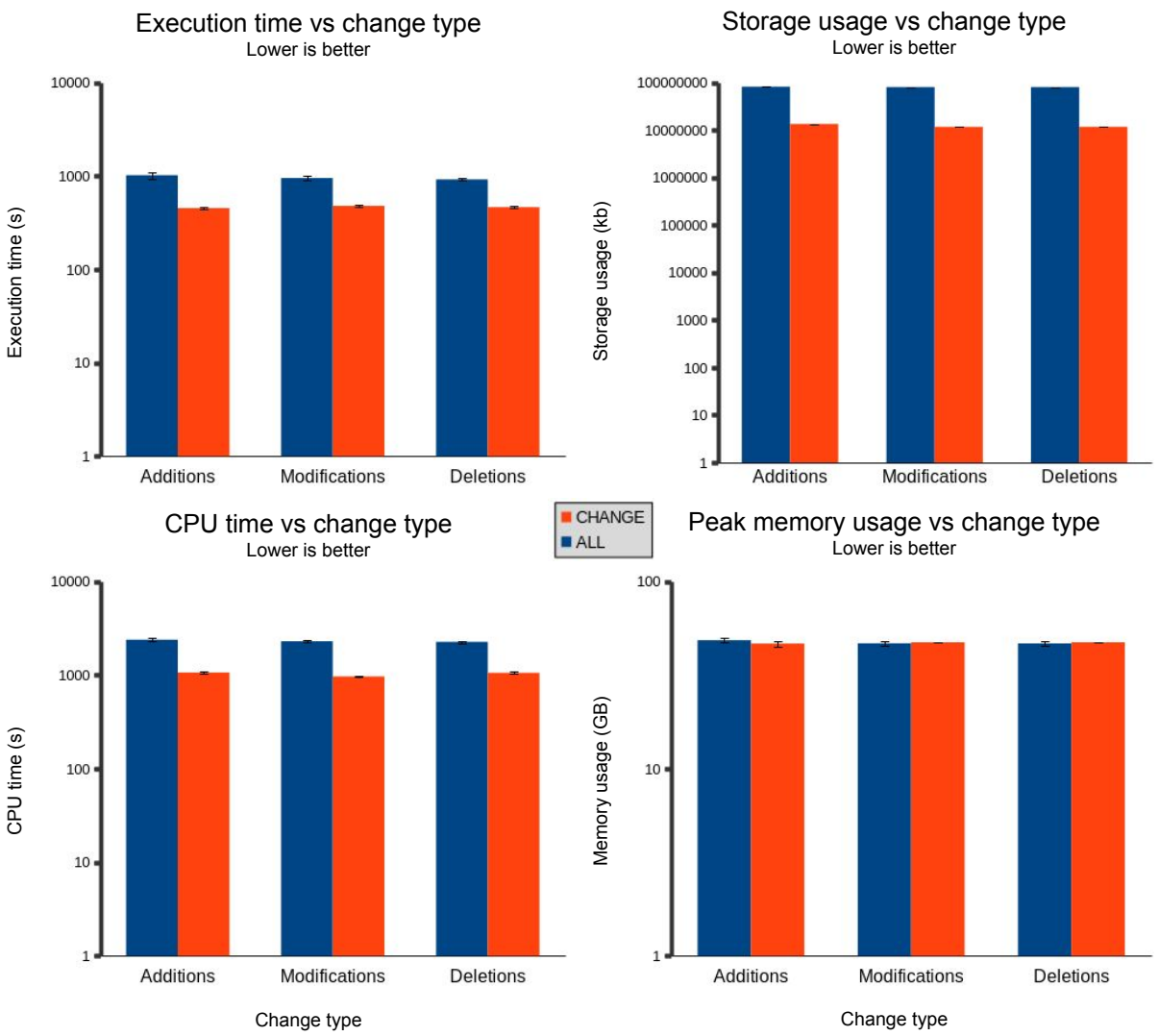

Fig. 7. GTFS Madrid Benchmark results for different change types with a fixed data size (scale 100) and amount of changes (50%). Only results of dataset updates are included, initial execution is not included.

multiple dataset updates on the measured metrics. We execute each experiment by starting with the base dataset to generate the initial KG and apply the corresponding dataset updates on top of it. Each experiment is executed 5 times from which the median value is taken for each metric of each dataset update. We report the average of these median metrics e.g., execution time, CPU time, and memory usage across multiple dataset updates in Tables 9 and 10. Storage usage is reported as the sum of KG sizes generated from the base dataset and all its updates (Table 11). OSM signals explicitly all changes, thus there is no OSM_{ALL}.

We observe that our approach **reduces execution time and CPU time without impacting memory usage** (Table 10), while also **reducing the storage needed to store multiple versions of a KG**. For the initial KG generation (Table 9), the overhead of change detection causes a higher execution time, storage, and CPU time, while memory usage is unaffected, similar to the results observed and discussed in Section 7.2.

Table 9

Initial execution results for all real-world use-cases for strategies `ALL` (no change detection) and `CHANGE` (with change detection). Lower is better.

Scenario	Execution time (s)			CPU time (s)			Peak memory (GB)		
	ALL	CHANGE	Ratio A/C	ALL	CHANGE	Ratio A/C	ALL	CHANGE	Ratio A/C
BlueBike	2.59	2.80	0.93	7.87	9.12	0.86	2.39	2.42	0.99
JCDecaux	3.73	4.65	0.80	16.82	21.63	0.77	2.13	2.33	0.91
NMBS	151.02	177.11	0.85	232.11	292.61	0.79	17.41	23.58	0.74
De Lijn	1 156.67	1 542.23	0.75	3 101.31	6 142.43	0.50	49.98	50.09	1.00
KMI	534.93	637.28	0.84	850.56	1 147.12	0.74	36.11	38.86	0.93
VVC	19.94	17.94	1.11	44.67	43.09	1.04	7.61	6.46	1.18
OSM	NA	3.10	NA	NA	11.22	NA	NA	2.67	NA

Table 10

Execution results for all real-life datasets for strategies `ALL` (no change detection) and `CHANGE` (with change detection). Only results of dataset updates are included, initial execution is not included. Lower is better.

Scenario	Execution time (s)			CPU time (s)			Peak memory (GB)		
	ALL	CHANGE	Ratio A/C	ALL	CHANGE	Ratio A/C	ALL	CHANGE	Ratio A/C
BlueBike	2.56	2.63	0.97	7.68	8.21	0.94	2.39	2.40	1.00
JCDecaux	3.79	3.78	1.00	16.74	19.77	0.85	2.13	2.16	0.99
NMBS	146.98	86.17	1.71	227.29	144.21	1.58	20.08	27.88	0.72
De Lijn	1 072.17	385.53	2.78	2 523.12	1 268.39	1.99	48.88	48.61	1.00
KMI	527.95	119.63	4.41	845.98	184.25	4.59	34.03	31.14	1.09
VVC	17.47	6.36	2.75	41.10	22.73	1.94	7.35	4.87	1.51
OSM	NA	3.56	NA	NA	16.64	NA	NA	2.92	NA

Table 11

Storage usage for initial and all updates per strategy of the real-world datasets. Total storage usage is sum of the base dataset and the applied updates upon it. Lower is better.

Scenario	Initial storage usage (kb)			Total storage usage (kb)		
	ALL	CHANGE	Ratio A/C	ALL	CHANGE	Ratio A/C
BlueBike	108.60	164.00	0.66	156 381.58	5 241.14	29.84
JCDecaux	3120.69	3665.46	0.85	4 493 804.23	173 649.86	25.88
NMBS	1 242 672.71	1 319 257.81	0.94	8 564 018.79	2 646 384.42	3.24
De Lijn	8 705 837.26	10 050 766.41	0.87	57 132 528.26	11 638 171.53	4.91
KMI	5 130 955.63	5 925 769.79	0.87	739 036 278.96	5 928 882.90	124.66
VVC	10 581.32	10 957.26	0.97	304 588 651.83	964 395.21	315.83
OSM	NA	541.25	NA	NA	12 745 567.48	NA

Storage usage **Our approach reduces storage usage by a factor ranging from 3.24 to 315.83 depending on the dataset** (Figure 8). We observe a large reduction in storage for *full history* datasets such as KMI: the history itself does not change across new version releases and is considerably larger than the size of the members created/updated in new versions of the dataset. The same applies to datasets that have few changed members per version (e.g., VVC). In contrast, datasets with a large number of changes between versions, e.g., NMBS, have a lower reduction in storage usage. Initial KG generation results in a higher storage usage for all datasets, as all members are generated along with the additional metadata indicating their creation.

CPU time Similar to execution time, **our approach reduces CPU time depending on the dataset size** (Figure 8), with a factor up to 4.59. On larger datasets (e.g., KMI, De Lijn, NMBS, or VVC) more members avoid unnecessary materialization, thus obtaining a higher reduction in CPU time. However, smaller datasets (e.g., BlueBike, JCDecaux) have a higher CPU time due to the overhead of continuously applying our change detection approach (7–18% CPU time increase). The initial KG generation causes a higher CPU consumption for all datasets, as the state for each dataset is initialized to track each dataset member.

Memory usage Results show that **our approach does not have a significant impact on the memory usage** (Figure 8) during the execution of KG generation processes. We attribute this to the compensation effect in our approach: avoiding materializing unchanged members compensates for the memory overhead of detecting changes. NMBS is an exception in this case because the Java VM does not execute garbage collection while processing this dataset, since the heap space does not reach its limits yet. Bigger datasets, e.g., KMI and De Lijn reach the heap space limit, triggering garbage collection. Therefore, memory usage grows and varies more for NMBS compared to the other datasets. The VVC dataset has a higher reduction in memory because processing XML data is costly regarding memory: when only the changes are processed, only a fraction of the XML is processed and kept in memory, causing a lower memory consumption. The initial KG generation has no impact on memory consumption, similar to processing dataset updates because the same amount of memory is needed to evaluate each data member of the base dataset and its corresponding updates.

7.4. Ingestion

Comparing the size of each update between ALL and CHANGE indicates that processing CHANGE updates requires less resources and time because they are smaller. To fully compare our approach with the ALL strategy (i.e., having access to the latest standalone version of a knowledge graph), we measure the total processing time of the ingestion step by ingesting the real-world datasets, via SPARQL `UPDATE` queries, into a Virtuoso triplestore.

Table 12 presents the total time of processing all collected versions for each real-world dataset (as presented in Table 5), for each step of the pipeline: i) KG generation time (as presented in Section 7.3), ii) interpretation of the generated changes as SPARQL updates (using `incrml2sparql`, Section 6.1.4), and iii) ingestion time in a triplestore, for both the ALL and CHANGE strategy. We present these metrics for the successfully completed real-world cases and the total execution time of the pipeline.

Unfortunately, most of these SPARQL queries are immediately rejected if the number of the changes increases, due to reaching Virtuoso's internal query length limits (10Mb). We overcame this problem by splitting up queries into multiple smaller ones, using the following pragmantic method: we started with 100 triples as upper limit and divided the queries in 2 parts each time a query failed, thus minimizing the number of queries needed. Depending on how the data is structured in RDF, Virtuoso was able to handle bigger queries or not, for example: large string literals have a higher memory impact on Virtuoso. Therefore, only BlueBike, JCDecaux, and VVC were successfully ingested into Virtuoso. We further tried to optimize the queries by splitting them into smaller queries. This helped already for the JCDecaux dataset to ingest it, but not for bigger datasets with larger change sets such as NMBS, De Lijn, KMI, or GTFS Madrid Benchmark.

The results show a reduction on the overall ingestion time for the datasets that were successfully ingested, which includes also the time needed to interpret the change semantics and generate the corresponding SPARQL `UPDATE` queries for the CHANGE strategy. **We observe that the overall execution time (i.e., the time required to generate, interpret, and ingest all versions of a dataset) is reduced 1.05-2.23 times, depending on the dataset**. This results provide already a very promising indication of the usefulness of our approach and highlight a broader need for more effective implementations of SPARQL `UPDATE` query execution in triplestores.

7.5. Lessons learned

In this subsection, we discuss the lessons learned (Table 13) from analyzing different real and synthetic datasets with varying change signaling strategies, change types, and history availability, for incremental KG generation.

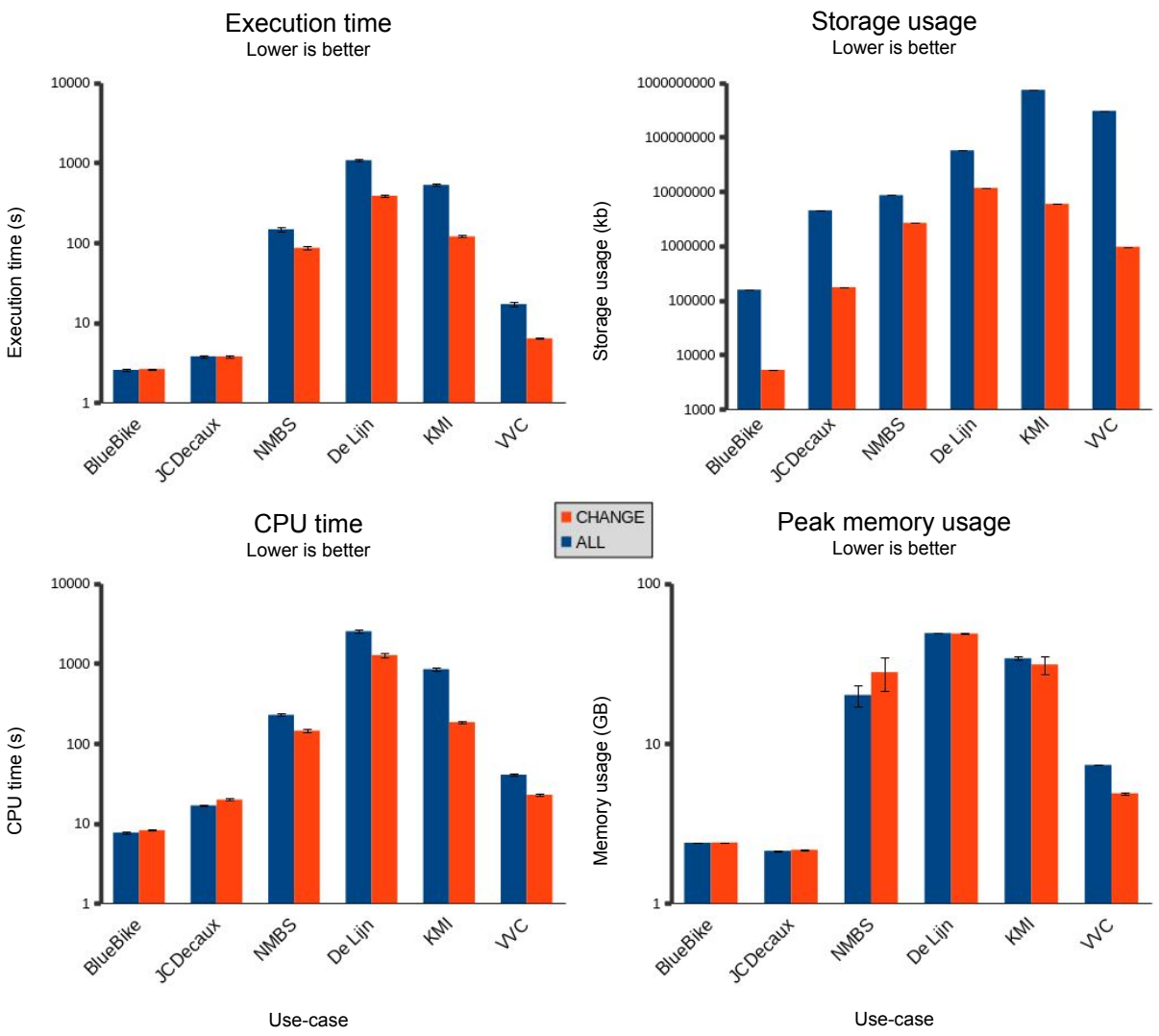

Fig. 8. Our approach reduces the necessary resources to generate different versions of a KG. Storage is reduced 3.24–315.83 times depending on dataset type. Execution time is reduced between 0.97–4.41 times depending on the dataset type. CPU time usage is reduced 0.85–4.59 times. Memory usage is mostly unaffected.

Functionality Our approach handles all possible dimensions and is feasible to implement. We implemented our approach by extending the RMLMapper v6.3.0 with **100% coverage for all the test cases** (Table 4). Therefore, we confirmed that our approach can detect all change types, both explicit and implicit, with all identified history signaling types and for all change types.

GTFS Madrid Benchmark Our **approach reduces the storage (~6x), execution time (~2x), CPU time (~2x) and memory consumption is mostly unaffected** for any of the benchmark scales (data size, amount of changes, and change types). However, the initial KG generation has a higher execution time and consumes more resources because all members are created and tracked by our approach, which causes this initial overhead.

Scaling the data size (1, 10, 100) has the most impact for $\text{GTFS}_{\text{SCALE}}$ 1 because the dataset is rather small resulting in a more visible impact of the overhead from tracking dataset members. CPU time usage remains the same for all scales, while execution time has a larger increase for $\text{GTFS}_{\text{SCALE}}$ 1 (-30%) compared to $\text{GTFS}_{\text{SCALE}}$

Table 12

Results for generating, interpreting, and ingesting of the different real-world datasets into a triplestore (Virtuoso). Failed datasets are caused by rejections of the SPARQL `UPDATE` queries by the triplestore when the initial graph or updates are too large. Larger datasets such as KMI fails to ingest, same with GTFS-based datasets, i.e., NMBS and De Lijn. GTFS Madrid Benchmark is already larger than the failing GTFS-based datasets, thus it fails as well. Note that the number of versions is the same for ALL and CHANGE, the reported number of versions is lower for CHANGE since unchanged versions are skipped. Lower is better.

Dataset	Generation time (s)			Interpretation time (s)			Ingestion time (s)			Total execution time (s)		
	ALL	CHANGE	Ratio A/C	ALL	CHANGE	Ratio A/C	ALL	CHANGE	Ratio A/C	ALL	CHANGE	Ratio A/C
BlueBike	3794.13	3691.84	1.02	601.74	607.65	0.99	123.24	11.13	11.07	4519.11	4310.72	1.05
JCDecaux	5450.90	5461.16	1.00	994.05	653.36	1.52	5746.05	189.84	30.27	12191.00	6304.46	1.93
VVC	503328.91	242419.45	2.08	22967.47	11721.36	1.96	43322.92	751.29	57.66	569619.30	254892.10	2.23
OSM	NA	5907.71	NA	NA	1852.16	NA	NA	6235.21	NA	NA	13995.08	NA

Table 13

Analysis goals on change signaling strategies, change types, and history availability from datasets using incremental KG generation.

Experiment	Goal
Functionality	Feasibility and coverage of our approach
GTFS Madrid Benchmark	Execution time and resource consumption analysis in multiple dimensions e.g., data size, change percentage per version, and change types.
Real-world datasets	Real-world execution time and resource consumption analysis, including end-to-end comparison.

10 and $\text{GTFS}_{\text{SCALE}}$ 100. Only for $\text{GTFS}_{\text{SCALE}}$ 10 the peak memory usage increases by 50%, due to the Java VM not performing garbage collection, as the Java heap space still has enough free memory. $\text{GTFS}_{\text{SCALE}}$ 100 is larger which causes the Java VM to trigger garbage collection, thus lowering memory usage. Storage usage is lower for any data size scale (6.56–8.12 times).

Scaling the amount of changes (0%, 25%, 50%, 75%, 100%) increases storage usage when more changes are present (11.70GB–12.93GB) since more new versions of dataset members are generated. CPU time usage and memory usage are unaffected because the number of dataset members remains unchanged, each dataset member is assessed to determine if it was changed or not.

Change types (create, update, delete) mostly affects the storage usage, for created members the storage usage (13.5GB) is higher compared to deleted (11.72GB) or updated members (11.71GB) because the new members that are added to the dataset need to be materialized. Deletions reduce the storage usage, but not significantly because deleting a GTFS Route only affects a subset of the dataset. Moreover, a tombstone comprising a few triples is still materialized to indicate thedeletion of the member for historical purposes.

Real-world datasets Depending on the data size, **our approach's RDF generation process runs faster (up to a factor of 4.41x)**. Besides faster RDF generation, the amount of resources is also **reduced in terms of storage** (factor 3.24–315.83) and **CPU time** (up to a factor of 4.59) depending on the dataset size. **No significant impact is observed fo memory usage** because the additional memory required by our approach for tracking changes is compensated by avoiding materializing the unchanged members. Detecting changes has an overhead, but it is mitigated when only a certain part of the KG must be regenerated, which is usually the case when processing new versions of datasets in practice. The type of change does not affect our approach in execution time, CPU and memory usage because every dataset member is checked for changes in any case. However, storage usage is affected since additional created members need to be stored, deleted members release occupied storage, and updated members could increase or decrease storage usage based on the change. However, in a complete pipeline, **interpreting and ingesting the changes into a triplestore reduces the efficiency of our approach, but we are still faster by a factor of 1.05–2.23** compared to ingested complete datasets. Smaller datasets are more affected since they have more overhead.

For smaller datasets (e.g., BlueBike or JCDecaux), **the overhead of our approach increases CPU time (+6.96% – +18.12%) and execution time (-0.20% – +2.76%) while storage usage is significantly reduced (-96.14% – -96.65%)**. For the initial KG generation from the base dataset, the execution time is longer and more resources are consumed due to the initialization of the internal state used for tracking all dataset members. Once the dataset members are processed, the execution time is almost the same, but the CPU time is still higher. This is the

overhead of tracking each member which impacts smaller datasets more visibly compared to larger ones. Storage usage is still heavily reduced. Despite the slight overhead on execution and CPU time introduced by our approach, the important reduction on storage usage may be decisive motivating factor for data publishers, which now could also store and offer historical records efficiently. Moreover, the semantic annotation of the different type of changes happening in a dataset, could also enable different and independent application behaviors both for the publishers and for remote data consumers if published via LDES.

For larger datasets (e.g., VVC, NMBS, KMI, or De Lijn), **our approach has a much larger impact on reducing CPU time usage (-33.57% – -78.22%), storage usage (-69.10% – -99.68%), and execution time (-41.37% – -77.34%)** with respect to its overhead, and compared to (re-)materializing the complete dataset. Note that memory consumption has an increase for NMBS (+38.8%) because the Java VM did not perform garbage collection as still enough heap memory is free for this dataset. Other datasets achieve a reduction of -0.55% – -31.31%.

In general, this results show a clear advantage of our approach, in terms of required computational resources, with respect to the traditional way of generating RDF KGs. Even considering the additional step required to fully ingest and integrate the materialized member changes into a KG (Table 12), we observe a clear reduction on the total time execution (SPARQL generation + ingestion). The beneficial impact of our approach increases with the size of the original datasets and provided that the number of changes remain relatively low compared to the total size, which is usually the case in most practical scenarios. This positions our approach as a scalable solution with promising potential to be used in production.

Use cases benefiting from incremental KG generation Our approach is the most beneficial for use cases where data changes frequently or requires historical data. For example: real-time information for public transport, IoT sensor data for smart cities, or weather history for analyzing the impact of climate change (Section 7). Real-time data needs to be integrated quickly to provide up-to-date information and the time to regenerate a KG for each data change is longer and consumes more computing resources when dealing with large datasets, compared to incrementally generating KGs. If historical information is important, it is beneficial to use our approach as only changes throughout history are incorporated into the KG which saves storage. This way, consumers can access and store the complete history for analytical purposes. However, if the data is rather small, e.g., BlueBike or JCDecaux bicycle data, our approach's overhead to detect changes does not reduce the execution time or computing resources, but still provides an important reduction on storage requirements while providing access to historical information.

8. Conclusion

In this paper, we investigated how to detect and materialize only dataset updates towards establishing an incremental publishing approach for KGs. To achieve this, we designed an approach that combines established KG generation technologies and a novel KG publishing approach (IncRML), which we implemented by extending the RMLMapper, and evaluated on 5 types of heterogeneous datasets. We observed that in general, our IncRML achieves a reduction in execution time for RDF generation (up to 4.41x), CPU time (up to 4.59x), memory usage (up to 1.51x), and storage (up to 315.83x). In terms of ingesting and fully updating a KG hosted in a triplestore, we also observed faster ingestion in IncRML both overall for all updates (up to 57.66x) and on average for individual updates (up to 28.5x), although we were only able to measure this for smaller datasets in one triplestore (Virtuoso) due to internal query size limitations.

Through this work, we establish a trade-off for generating and publishing KGs, where following our approach can lead to significant time and computing resource savings to generate the raw RDF quads of a KG, at the cost of introducing an additional step for change reconciliation. On the other hand, a traditional KG generation is capable of producing an already updated and integrated KG, at the cost of additional computing resources and processing time. Nevertheless, our experiments indicate that despite the additional processing step required by IncRML, the overall processing time to update KGs is still lower (by a factor of 1.05–2.23) than with a fully re-materialization approach.

We evaluated our approach on a heterogeneous set of real-world datasets, including weather sensor data, public transport timetables, bike sharing data, live road traffic information and crowdsourced geospatial data. We show that

our *IncRML* implementation is able to cover the different change communication strategies used by these set of real-world datasets (explicitly by the data source, or implicitly by silently changing the data), change types (creations, updates, and deletions), and history availability in the dataset (latest state, latest changes, or full history datasets).

Thanks to our approach, we provide the means to generate semantically annotated data and metadata from both implicitly and explicitly changed datasets. Therefore, we help to bring more transparency/provenance, in particular to implicitly signaled datasets. We integrate LDES as a Web native alternative to publish a semantically and structurally described stream of events that could enable data consumers to replicate and continuously synchronize with a KG, whether their changes are explicitly or implicitly signaled in the original data sources. Furthermore, our approach facilitates and reduces the cost of storing and publishing historic data, which is commonly an important requirement for e.g., data analysis and machine learning applications. Existing LDES clients[32] can already take advantage of our incrementally generated KGs given that only changed members need to be processed, which is usually lower compared to the total size of a dataset. In general, IncRML show great potential to be further developed into production ready solutions that could lower the costs of creating and consuming KGs, with the goal of increasing adoption of Semantic Web technologies.

Further research includes expanding the pipeline to investigate the impact on *end-to-end performance* with different triplestores and increasing amount of consumers. On this direction, triplestores could be extended to allow direct native ingestion of incremental KGs (e.g., as an LDES) instead of performing full re-ingestion, thus benefiting from the usual lower amount of data that needs to be processed. On the other hand, we also highlight the need for more effective SPARQL `UPDATE` query processing in triplestores, in order to remain standard compliant and not having to rely only on custom and vendor-dependent data update techniques.

Investigating *optimizations for the proposed change detection algorithms* is also possible given that currently the implementation of our CDC algorithms as FnO functions uses in-memory lookup tables to detect changes. Avoid keeping the lookup tables in memory (e.g., using key-value stores) could reduce the memory footprint of IncRML even further. Also, exploring *windowing techniques for streaming data*, could allow to handle deletions on unbounded data streams. Our approach can perform change detection on streaming data, but requires a window definition for detecting deletions. Specifying a window is not supported yet in any declarative mapping language, but is considered by the W3C Community Group on Knowledge Graph Construction[33]. Another potential path for future work is *querying of incremental KGs*. Performing SPARQL queries on incrementally generated KGs requires further investigation to optimize query execution as we only tackled incremental generation. The study of approaches to *handle schema-level (ontology and mappings) changes* efficiently, is an important aspect to be consider [59] since also the ontology and mappings can evolve besides the data itself. Lastly, performing a survey on dataset change signaling could be performed to validate our change signaling and communication strategies.

References

[1] C.R. Valencio, M.H. Marioto, G.F. Donega Zafalon, J.M. Machado and J.C. Momente, Real Time Delta Extraction Based on Triggers to Support Data Warehousing, in: *2013 International Conference on Parallel and Distributed Computing, Applications and Technologies*, 2013, pp. 293–297. doi:10.1109/PDCAT.2013.52.

[2] Denny, I.P.M. Atmaja, A. Saptawijaya and S. Aminah, Implementation of Change Data Capture in ETL Process for Data Warehouse using HDFS and Apache Spark, in: *2017 International Workshop on Big Data and Information Security (IWBIS)*, 2017, pp. 49–55. doi:10.1109/IWBIS.2017.8275102.

[3] J.A. Rojas, M. Aguado, P. Vasilopoulou, I. Velitchkov, D.V. Assche, P. Colpaert and R. Verborgh, Leveraging semantic technologies for digital interoperability in the European Railway domain (2021). https://julianrojas.org/papers/iswc2021-in-use/.

[4] L. Hao, T. Jiang, Y. Lin and Y. Lu, Methods for Solving the Change Data Capture Problem, in: *Advances in Natural Computation, Fuzzy Systems and Knowledge Discovery*, 2023, pp. 781–788.

[5] S. Gupta and V. Giri, *Capture Streaming Data with Change-Data-Capture*, in: *Practical Enterprise Data Lake Insights: Handle Data-Driven Challenges in an Enterprise Big Data Lake*, Apress, 2018, pp. 87–123. doi:10.1007/978-1-4842-3522-5_3.

[6] S. Das, S. Sundara and R. Cyganiak, R2RML: RDB to RDF Mapping Language, Working Group Recommendation, World Wide Web Consortium (W3C), 2012. http://www.w3.org/TR/r2rml/.

[32] https://github.com/rdf-connect/ldes-client

[33] Windowing operation for streaming data sources: https://github.com/kg-construct/rml-core/issues/85

[7] A. Dimou, M. Vander Sande, P. Colpaert, R. Verborgh, E. Mannens and R. Van de Walle, RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data, in: *Proceedings of the 7th Workshop on Linked Data on the Web*, 2014.

[8] E. Daga, L. Asprino, P. Mulholland and A. Gangemi, Facade-X: An Opinionated Approach to SPARQL Anything, in: *Further with Knowledge Graphs – Proceedings of the 17th International Conference on Semantic Systems, 6–9 September 2021, Amsterdam, The Netherlands*, 2021, pp. 58–73. doi:10.3233/SSW210035.

[9] M. Lefrançois, A. Zimmermann and N. Bakerally, A SPARQL Extension for Generating RDF from Heterogeneous Formats, in: *The Semantic Web 14th International Conference, ESWC 2017, Portorož, Slovenia, May 28 – June 1, 2017, Proceedings*, 2017, pp. 35–50. doi:10.1007/978-3-319-58068-5_3.

[10] B. Vu, J. Pujara and C.A. Knoblock, D-REPR: A Language for Describing and Mapping Diversely-Structured Data Sources to RDF, in: *Proceedings of the 10th International Conference on Knowledge Capture*, 2019, pp. 189–196. doi:10.1145/3360901.3364449.

[11] D. Van Assche, T. Delva, G. Haesendonck, P. Heyvaert, B. De Meester and A. Dimou, Declarative RDF graph generation from heterogeneous (semi-)structured data: A systematic literature review, *Journal of Web Semantics* (2022). doi:10.1016/j.websem.2022.100753.

[12] D. Van Lancker, P. Colpaert, H. Delva, B. Van de Vyvere, J. Rojas Meléndez, R. Dedecker, P. Michiels, R. Buyle, A. De Craene and R. Verborgh, Publishing Base Registries as Linked Data Event Streams, in: *Proceedings of the 21th International Conference on Web Engineering*, M. Brambilla, R. Chbeir, F. Frasincar and I. Manolescu, eds, Lecture Notes in Computer Science, Vol. 12706, Springer, Cham, 2021, pp. 28–36. ISBN 9783030742966. doi:10.1007/978-3-030-74296-6_3.

[13] J.M. Snell and Prodromou, Activity Streams 2.0, Recommendation, World Wide Web Consortium (W3C), 2017. http://www.w3.org/TR/activitystreams-core/.

[14] D. Van Assche, S.M. Oo, J.A. Rojas and P. Colpaert, Continuous generation of versioned collections' members with RML and LDES, in: *Proceedings of the 3rd International Workshop on Knowledge Graph Construction (KGCW 2022) co-located with 19th Extended Semantic Web Conference (ESWC 2022)*, 2022.

[15] D. Chaves-Fraga, F. Priyatna, A. Cimmino, J. Toledo, E. Ruckhaus and O. Corcho, GTFS-Madrid-Bench: A benchmark for virtual knowledge graph access in the transport domain, *Journal of Web Semantics* **65** (2020), 100596. doi:https://doi.org/10.1016/j.websem.2020.100596. https://www.sciencedirect.com/science/article/pii/S1570826820300354.

[16] R. Cyganiak, D. Wood and M. Lanthaler, RDF 1.1 Concepts and Abstract Syntax, Recommendation, World Wide Web Consortium (W3C), 2014. http://www.w3.org/TR/rdf11-concepts/.

[17] A. Iglesias-Molina, D. Van Assche, J. Arenas-Guerrero, B. De Meester, C. Debruyne, S. Jozashoori, P. Maria, F. Michel, D. Chaves-Fraga and A. Dimou, The RML Ontology: A Community-Driven Modular Redesign After a Decade of Experience in Mapping Heterogeneous Data to RDF, in: *The Semantic Web – ISWC 2023*, T.R. Payne, V. Presutti, G. Qi, M. Poveda-Villalón, G. Stoilos, L. Hollink, Z. Kaoudi, G. Cheng and J. Li, eds, Springer Nature Switzerland, Cham, 2023, pp. 152–175. ISBN 978-3-031-47243-5.

[18] D. Van Assche, G. Haesendonck, G. De Mulder, T. Delva, P. Heyvaert, B. De Meester and A. Dimou, Leveraging Web of Things W3C Recommendations for Knowledge Graphs Generation, in: *Web Engineering, 21st International Conference, ICWE 2021, Proceedings*, M. Brambilla, R. Chbeir, F. Frasincar and I. Manolescu, eds, Lecture Notes in Computer Science, Vol. 12706, Springer, Cham, 2021, pp. 337–352. ISBN 9783030742966. doi:10.1007/978-3-030-74296-6_26.

[19] C. Debruyne, L. McKenna and D. O'Sullivan, Extending R2RML with Support for RDF Collections and Containers to Generate MADS-RDF Datasets, in: *Research and Advanced Technology for Digital Libraries: 21st International Conference on Theory and Practice of Digital Libraries, TPDL 2017, Thessaloniki, Greece, September 18-21, 2017, Proceedings*, 2017, pp. 531–536. doi:10.1007/978-3-319-67008-9_42.

[20] F. Michel, L. Djimenou, C. Faron-Zucker and J. Montagnat, xR2RML: Relational and Non-Relational Databases to RDF Mapping Language, Rapport de Recherche, Laboratoire d'Informatique, Signaux et Systèmes de Sophia-Antipolis (I3S), 2017. https://hal.archives-ouvertes.fr/hal-01066663/document/.

[21] A. Chortaras and G. Stamou, Mapping Diverse Data to RDF in Practice, in: *The Semantic Web – ISWC 2018*, D. Vrandečić, K. Bontcheva, M.C. Suárez-Figueroa, V. Presutti, I. Celino, M. Sabou, L.-A. Kaffee and E. Simperl, eds, Lecture Notes in Computer Science, Vol. 11136, Springer, Cham, 2018, pp. 441–457. ISBN 978-3-030-00671-6. doi:10.1007/978-3-030-00671-6.

[22] F. Michel, L. Djimenou, C. Faron-Zucker and J. Montagnat, Translation of Heterogeneous Databases into RDF, and Application to the Construction of a SKOS Taxonomical Reference, in: *International Conference on Web Information Systems and Technologies*, 2015, pp. 275–296. doi:10.1007/978-3-319-30996-5_14.

[23] H. García-González, I. Boneva, S. Staworko, J.E. Labra-Gayo and J.M.C. Lovelle, ShExML: improving the usability of heterogeneous data mapping languages for first-time users, *PeerJ Computer Science* (2020), e318.

[24] M.G. Skjæveland, D.P. Lupp, L.H. Karlsen and H. Forssell, Practical Ontology Pattern Instantiation, Discovery, and Maintenance with Reasonable Ontology Templates, in: *The Semantic Web – ISWC 2018*, D. Vrandečić, K. Bontcheva, M.C. Suárez-Figueroa, V. Presutti, I. Celino, M. Sabou, L.-A. Kaffee and E. Simperl, eds, Springer International Publishing, Cham, 2018, pp. 477–494. ISBN 978-3-030-00671-6.

[25] D. Brickley and R.V. Guha, RDF Schema 1.1, Recommendation, World Wide Web Consortium (W3C), 2014. http://www.w3.org/TR/rdf-schema/.

[26] E. Prud'hommeaux, J.E. Labra Gayo and H. Solbrig, Shape expressions: an RDF validation and transformation language, in: *Proceedings of the 10th International Conference on Semantic Systems*, H. Sack, A. Filipowska, J. Lehmann and S. Hellmann, eds, Association for Computing Machinery, New York, NY, United States, 2014, pp. 32–40, ACM. doi:10.1145/2660517.2660523. http://dl.acm.org/citation.cfm?id=2660523.

[27] J. Arenas-Guerrero, D. Chaves-Fraga, J. Toledo, M.S. Pérez and O. Corcho, Morph-KGC: Scalable knowledge graph materialization with mapping partitions, *Semantic Web* (2022), 1–20. doi:10.3233/sw-223135.

[28] E. Iglesias, S. Jozashoori, D. Chaves-Fraga, D. Collarana and M.-E. Vidal, SDM-RDFizer: An RML Interpreter for the Efficient Creation of RDF Knowledge Graphs, in: *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, 2020. doi:10.1145/3340531.3412881.

[29] B. De Meester, T. Seymoens, A. Dimou and R. Verborgh, Implementation-independent Function Reuse, *Future Generation Computer Systems* (2020), 946–959. doi:10.1016/j.future.2019.10.006.

[30] S. Harris and A. Seaborne, SPARQL 1.1 Query Language, Recommendation, World Wide Web Consortium (W3C), 2013. https://www.w3.org/TR/sparql11-query/.

[31] A.C. Junior, C. Debruyne, R. Brennan and D. O'Sullivan, An evaluation of uplift mapping languages, *International Journal of Web Information Systems* (2017), 405–424. doi:10.1108/IJWIS-04-2017-0036.

[32] J. Umbrich, B. Villazón-Terrazas and M. Hausenblas, Dataset Dynamics Compendium: A Comparative Study, in: *COLD*, 2010. https://api.semanticscholar.org/CorpusID:15551988.

[33] K. Ma and B. Yang, Log-based Change Data Capture from Schema-free Document Stores using MapReduce, in: *2015 International Conference on Cloud Technologies and Applications (CloudTech)*, 2015, pp. 1–6. doi:10.1109/CloudTech.2015.7336969.

[34] Q. Hu, Z. Gan and B. Zhang, Design and Implementation of Oracle Database Incremental Data Capture Based on Trigger and Identification Table, *Journal of Physics: Conference Series* (2019), 022161. doi:10.1088/1742-6596/1237/2/022161.

[35] I. MadeSukarsa, N. Wisswani, I. Putra and L. Linawati, Change Data Capture on OLTP Staging Area for Nearly Real Time Data Warehouse Base on Database Trigger, *International Journal of Computer Applications* (2012), 32–37. doi:10.5120/8248-1762.

[36] A. Goyal and C. Dyreson, Temporal JSON, in: *2019 IEEE 5th International Conference on Collaboration and Internet Computing (CIC)*, 2019, pp. 135–144. doi:10.1109/CIC48465.2019.00025.

[37] V. Papakonstantinou, G. Flouris, I. Fundulaki, K. Stefanidis and G. Roussakis, Versioning for Linked Data: Archiving Systems and Benchmarks., *BLINK@ ISWC* **1700** (2016).

[38] J.D. Fernández, A. Polleres and J. Umbrich, Towards Efficient Archiving of Dynamic Linked Open Data, 2015.

[39] M. Vander Sande, P. Colpaert, R. Verborgh, S. Coppens, E. Mannens and R. Van de Walle, R&Wbase: git for triples, in: *Proceedings of the 6th Workshop on Linked Data on the Web*, 2013.

[40] M. Frommhold, R.N. Piris, N. Arndt, S. Tramp, N. Petersen and M. Martin, Towards Versioning of Arbitrary RDF Data, *Proceedings of the 12th International Conference on Semantic Systems* (2016). https://api.semanticscholar.org/CorpusID:14113981.

[41] S. Cassidy and J. Ballantine, Version Control for RDF Triple Stores, in: *International Conference on Software and Data Technologies*, 2007. https://api.semanticscholar.org/CorpusID:12177206.

[42] M. Völkel, W. Winkler, Y. Sure, S.R. Kruk and M. Synak, SemVersion: A Versioning System for RDF and Ontologies, 2005. https://api.semanticscholar.org/CorpusID:14892100.

[43] M. Graube, S. Hensel and L. Urbas, R43ples: Revisions for Triples - An Approach for Version Control in the Semantic Web, in: *LDQ@SEMANTiCS*, 2014. https://api.semanticscholar.org/CorpusID:14184753.

[44] D.-H. IM, S.-W. LEE and H.-J. KIM, A VERSION MANAGEMENT FRAMEWORK FOR RDF TRIPLE STORES, *International Journal of Software Engineering and Knowledge Engineering* **22**(01) (2012), 85–106. doi:10.1142/S0218194012500040.

[45] H.V. de Sompel, M. Nelson and R. Sanderson, HTTP Framework for Time-Based Access to Resource States – Memento, *Request for Comments*, RFC Editor, 2013. doi:10.17487/RFC7089.

[46] T. Neumann and G. Weikum, X-RDF-3X: Fast Querying, High Update Rates, and Consistency for RDF Databases, *Proc. VLDB Endow.* **3**(1–2) (2010), 256–263–. doi:10.14778/1920841.1920877.

[47] R. Taelman, M. Vander Sande, J. Van Herwegen, E. Mannens and R. Verborgh, Triple Storage for Random-Access Versioned Querying of RDF Archives, *Journal of Web Semantics* (2018). https://rdfostrich.github.io/article-jws2018-ostrich/.

[48] P. Meinhardt, M. Knuth and H. Sack, TailR: A Platform for Preserving History on the Web of Data, in: *Proceedings of the 11th International Conference on Semantic Systems*, 2015, pp. 57–64. doi:10.1145/2814864.2814875.

[49] B. Salzberg and V.J. Tsotras, Comparison of Access Methods for Time-Evolving Data, *ACM Comput. Surv.* (1999), 158–221. doi:10.1145/319806.319816.

[50] A. Randles and D. O'Sullivan, Modelling & Analyzing Changes within LD Source Data, in: *MEPDaW@ISWC*, 2022. https://api.semanticscholar.org/CorpusID:257081241.

[51] A. Randles and D. O'Sullivan, Preserving the Alignment of LD with Source Data, in: *KGCW@ESWC*, 2023. https://api.semanticscholar.org/CorpusID:259266370.

[52] M. Meimaris and G. Papastefanatos, The EvoGen Benchmark Suite for Evolving RDF Data, in: *MEPDaW/LDQ@ESWC*, 2016. https://api.semanticscholar.org/CorpusID:12789745.

[53] A. Randles, D. O'Sullivan, J. Keeney and L. Fallon, Applying a Mapping Quality Framework in Cloud Native Monitoring, in: *International Conference on Semantic Systems*, 2022. https://api.semanticscholar.org/CorpusID:252919576.

[54] N. Konstantinou, D. Kouis and N. Mitrou, Incremental Export of Relational Database Contents into RDF Graphs, in: *Proceedings of the 4th International Conference on Web Intelligence, Mining and Semantics (WIMS14)*, WIMS '14, Association for Computing Machinery, 2014. doi:10.1145/2611040.2611082.

[55] V.M.P. Vidal, M.A. Casanova and D.S. Cardoso, Incremental Maintenance of RDF Views of Relational Data, in: *On the Move to Meaningful Internet Systems: OTM 2013 Conferences*, 2013, pp. 572–587.

[56] X. Pu, J. Wang, Z. Song, P. Luo and M. Wang, Efficient incremental update and querying in AWETO RDF storage system, *Data & Knowledge Engineering* (2014), 55–75. doi:https://doi.org/10.1016/j.datak.2013.11.003.

[57] P. Colpaert, Building materializable querying interfaces with the TREE hypermedia specification, in: *Proceedings of the 8th Workshop on Managing the Evolution and Preservation of the Data Web (MEPDaW) co-located with the 21st International Semantic Web Conference (ISWC 2022), Virtual event, October 23rd, 2022*, D. Graux, F. Orlandi, E. Niazmand, G. Ydler and M. Vidal, eds, CEUR Workshop Proceedings, Vol. 3339, CEUR-WS.org, 2022, pp. 8–18. https://ceur-ws.org/Vol-3339/paper2.pdf.

[58] Colpaert, Pieter and Abelshausen, Ben and Rojas Melendez, Julian Andres and Delva, Harm and Verborgh, Ruben, Republishing OpenStreetMap's roads as linked routable tiles, in: *SEMANTIC WEB: ESWC 2019 SATELLITE EVENTS*, Vol. 11762, Hitzler, Pascal and Kirrane, Sabrina and Hartig, Olaf and de Boer, Victor and Vidal, Maria-Esther and Maleshkova, Maria and Schlobach, Stefan and Hammar, Karl and Lasierra, Nelia and Stadtmüller, Steffen and Hose, Katja and Verborgh, Ruben, ed., Springer, 2019, pp. 13–17. ISSN 0302-9743. ISBN 9783030323264. http://doi.org/10.1007/978-3-030-32327-1_3.

[59] Conde-Herreros, Diego and Stork, Lise and Pernisch, Romana and Poveda-Villalón, María and Corcho, Oscar and Chaves-Fraga, David, Propagating Ontology Changes to Declarative Mappings in Construction of Knowledge Graphs, in: *Proceedings of the 5th International Workshop on Knowledge Graph Construction co-located with 21th Extended Semantic Web Conference (ESWC 2024)*, CEUR Workshop Proceedings, Vol. 3718, CEUR-WS.org, 2024, pp. 1–16. https://ceur-ws.org/Vol-3718/paper1.pdf.

[60] B. De Meester, S. Jozashoori, P. Maria, D. Chaves-Fraga and A. Dimou, RML-FNML, Technical Report, Knowledge Graph Construction Community Group, 2023. https://kg-construct.github.io/rml-fnml/spec/docs/.

[61] A. Iglesias-Molina, D. Van Assche, J. Arenas-Guerrero, B. De Meester, C. Debruyne, S. Jozashoori, P. Maria, F. Michel, D. Chaves-Fraga and A. Dimou, The RML Ontology: A Community-Driven Modular Redesign After a Decade of Experience in Mapping Heterogeneous Data to RDF, in: *Submited to ISWC2023*, 2023.

[62] S. Jozashoori, D. Chaves-Fraga, E. Iglesias, M.-E. Vidal and O. Corcho, FunMap: Efficient Execution of Functional Mappings for Knowledge Graph Creation, in: *International Semantic Web Conference*, 2020, pp. 276–293.

[63] D. Van Assche, J.A. Rojas Meléndez, B. De Meester and P. Colpaert, Change Data Capture for continuous knowledge graph generation from heterogeneous data, in: *Submitted to ISWC2023 P&D*, 2023.

[64] M. Brambilla, R. Chbeir, F. Frasincar and I. Manolescu (eds), Web Engineering, 21st International Conference, ICWE 2021, Biarritz, France, May 18–21, 2021, in: *Web Engineering*, Lecture Notes in Computer Science, Vol. 12706, Springer, Cham, 2021. ISBN 9783030742966.

[65] D. Vrandečić, K. Bontcheva, M.C. Suárez-Figueroa, V. Presutti, I. Celino, M. Sabou, L.-A. Kaffee and E. Simperl (eds), The Semantic Web – ISWC 2018: 17th International Semantic Web Conference, Monterey, CA, USA, October 8–12, 2018, Proceedings, Part I, in *Lecture Notes in Computer Science*, Vol. 11136, Springer, Cham, 2018. ISBN 978-3-030-00671-6. doi:10.1007/978-3-030-00671-6.