IncRML: 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., historical versioning, 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 changing data sources and their different data change advertisement strategies. We investigate different change advertisement strategies of data sources, propose algorithms to detect implicitly advertised changes, and introduce our approach IncRML (Incremental RDF Mapping Language). IncRML combines RML and FnO functions to detect and classify changes in data sources across an RML mapping process. We also allow to optionally and automatically publish these 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, the changes are also communicated to consumers. We present our approach and evaluate it qualitatively 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 explicit 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, Knowledge Graph Construction, Incremental

1. Introduction

Most real-world datasets are not static. Whether it be the inclusion of new observations, the rectification of errors, or the evolution of data collection methodologies, datasets are, by their very nature, subject to continuous change (creates, updates, and deletes). The change frequency may vary, with some datasets undergoing rapid transformations while others experiencing a more gradual evolution. Regardless of the particular change frequency, consumers

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

of such datasets 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 face challenges such as storing multiple versions to keep historic records, having enough capacity to handle dataset's size when processing changes, and keeping up with dataset's change frequency.

Moreover, data integration processes such as Knowledge Graph (KG) generation are currently repeated *every time one of the original data sources is changed* [2, 3].

This approach is unsustainable and makes applications with real-time information (e.g., public transport routing) inefficient: (i) computing resources and time are wasted reprocessing all data even when large parts remain unchanged, (ii) history is not preserved because only the last version is made available, and (iii) the data integration process must keep up with the change frequency of the original data sources. Thus, the integrated dataset might be outdated even before it is completely regenerated. Change Data Capture approaches [2] try to address this problem by using database logs to discover changes, comparing snapshots of data dumps, etc. However, current approaches for integrating heterogeneous data sources into KGs cannot use existing Change Data Capture (CDC) approaches, as these approaches focus on a specific type of data source (e.g., relational database logs). Moreover, the type of change (create, update, delete) impacts how the change must be handled.

In this paper, we present IncRML (Incremental RDF Mapping Language): a KG generation approach that is capable of efficiently handling changing data sources and their different data change communication strategies. Our proposed approach only regenerates the parts of a KG which were changed in the original data source, thus only created, updated, and deleted data entities are regenerated. Our approach is holistic: it is designed to handle data changes from data sources which advertise their changes (e.g. relational database log) but also from data sources which do not advertise their changes (e.g. files or Web APIs). Generating a KG is performed with a mapping language (e.g. RML) and changes are detected with algorithms described by data transformation descriptions (e.g. FnO). We aligned our approach with RML and FnO to use best practices of KG generation and decouple the approach from a specific implementation. We also allow to optionally and automatically publish these changes in the form of a Linked Data Event Stream (LDES) [4], relying on the W3C Activity Streams 2.0 vocabulary [5] to describe changes semantically. Using LDES allows publishing data changes as an append-only event log that consumers can read to synchronize their KG with the latest data available. This way, the burden of history and versioning of a KG is divided among both publishers and consumers since both can decide which versions of a KG are stored by them.

We extend our previous work [6] on aligning LDES and RML where we introduced a preliminary approach for detecting changes in a *data collection*: a data source composed of a set of data members. Our previous work only handled creates (member added to data collection) and updates (member modified in data collection). In this paper, we extend that work to also detect deletions (member removed from data collection), and include an extensive evaluation with different types of real-world datasets and an extended version of the GTFS Madrid Benchmark [7]. To keep this paper self-contained, we will include the discussions of our previous work along with its extensions.

Our contributions are the following: (i) an overview of different data collection categories regarding history and change advertisement, (ii) a set of algorithms to detect changes in any of these collection categories, (iii) the integration of these algorithms in a KG construction pipeline using RML, FnO, and LDES, and (iv) an extensive evaluation on the impact of incrementally generating KGs.

By incrementally generating KGs, we optimize KG construction, using 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. Hence, we reduce the KG construction time by 0.97–4.41x. Moreover, our work is extensible. First, our implemented KG generation approach is usable for any kind of data source. Second, while we use RML as mapping language in this paper, its concepts can be expanded to any mapping language. Third, our approach can use any ontology to describe changes semantically.

Thanks to our work, data source changes can be integrated faster and with less resources in KGs – only updated members of a data collection are (re-)generated instead of the entire data collection – while keeping access to the historical records in the form of a LDES: the event stream can be used by applications that require real-time data, historical data, or both.

The remainder of this paper is structured as follows: Section 2 discusses Related Work, Section 3 provides background on RML, FnO, and LDES. Section 4 presents our approach. Section 5 shows how we implement our approach. Section 6 presents our evaluation. Section 7 discusses our results, and Section 8 concludes this paper.

2. Related Work

In this Section, we discuss related work on (i) mapping rules for declarative Knowledge Graph (KG) generation describing how a KG must be generated, (ii) Change Data Capture approaches for detecting changes in data collections, (iii) versioning strategies for KGs to store history of data collection members 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 [8] for transforming relational databases into an RDF KG [9]. R2RML was extended as the RDF Mapping Language (RML) [10, 11] to support heterogeneous data sources (e.g. JSON, XML, CSV) while keeping backwards compatibility with R2RML. Besides RML, other declarative mapping languages appeared to transform heterogeneous data sources into RDF such as xR2RML [12, 13], SPARQL-Generate [14], SPARQL-Anything [15], ShExML [16], and D-REPR [17]. RML is widely used for declarative KG generation and was extended to export RDF to various targets such as files and SPARQL endpoints [18], support for RDF Collections and Containers [13, 19], and access to Web APIs [18, 20]. Currently, the W3C Community Group on KG Construction[1] is working on standardizing RML towards a W3C Recommendation [11]. Therefore, we use RML as the mapping language in this paper.

Support for data transformations is an important requirement when generating KGs from heterogeneous data sources [21], using, e.g. the Function Ontology (FnO) [22], SPARQL Functions [23], and FunUL [24]. FnO is the most applied declarative description for describing functions to perform data transformations, and it is integrated with RML through FNML[2]. This way, RML+FnO mapping rules can not only perform the generation of a KG, but also data transformation without requiring ad hoc or use case specific scripts. We use FnO in this paper to integrate our change detection algorithms with RML to incrementally generate a KG from heterogeneous data sources.

2.2. Change Data Capture

Change Data Capture (CDC) [25, 26] refers to a technique, primarily used in databases, to identify and capture changes made to the data so that those changes can be tracked, recorded, and propagated to other systems or applications. The primary purpose of CDC is to identify and capture the creations, updates, and deletions of data in a database, enabling (near-)real-time synchronization of data across different systems. Most approaches focus on data sources able to provide some sort of change advertisement mechanism such as logs, snapshots, triggers, or timestamps. Moreover, data sources may offer different granularity regarding change advertisement, e.g. change advertisement on parts of the data source or the whole data source [27].

Log-based approaches [25, 28] rely on database logs to determine which changes were performed to the underlying data. These approaches are specific to each database, as their log system is implementation specific. *Snapshot-based approaches* [2] compare previous versions of a data source to extract changes, which requires sufficient resources to store and compare these versions. *Trigger-based approaches* [1, 29, 30] hook into a data source to execute a trigger on each change, but require support for triggers from the data source. *Timestamp-based approaches* [31]

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

[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).

analyze a last-modified timestamp of a data source to trigger the extraction of changes, but data sources need to provide timestamp-annotated data.

Although these approaches clearly have their merit, many data sources do not advertise their changes at all nor support triggers when a data record is changed (e.g. data streams, files, or Web APIs). Therefore, existing Data Change Capture approaches are insufficient to cover heterogeneous data sources which do not advertise their changes to consumers.

2.3. Versioning of Knowledge Graphs

Several approaches for versioning of (RDF) KGs have been proposed [32]. Three main RDF archive storage strategies can be identified [33]: (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; while *Independent Copies* stores a copy of the entity each time it is updated. Change-Based approaches include: R&Wbase [34], based on Git[3]; a Version Control Based RDF storage approach using patches, similar to R&Wbase by Frommhold M. et al. [35]; the works of Cassidy et al. [36], SemVersion [37], R43ples [38], and Im et al [39]. Timestamp-Based approaches for accessing different versions include: Memento (HTTP) [40] and x-RDF-3X [41] (SPARQL). Ostrich [42] and TailR [43] 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 producer.

LDES [4] uses an Independent Copies approach for versioning. However, Independent Copies suffers from scalability problems as storage is not infinite [33], LDES addresses this by dropping the oldest versions of a member in the event stream, configurable through a specific *retention policy* to only keep a certain constrained set of versions of entities. Since LDES consumers are aware of the retention policy, they can decide to store the LDES members if they need to for their particular use case.

For example, a LDES producer's retention policy is 7 days: if a consumer requires at least 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 [44]. This way, versioning is resolved on the consumer side and several versioning strategies can be applied independent of the publisher. In this work, we use LDES for describing decoratively changes and publishing these changes as an Event Stream.

2.4. Versioned generation of Knowledge Graphs

Ontologies (Change Detection Ontology [45, 46]) and benchmarks (EvoGen Benchmark Suite [47], BEAR benchmark [33]) were proposed for generating versioned KGs, but they are tight to a specific ontology or focus on querying the versions while we focus on the generation in this work. The EvoGen Benchmark Suite [47] allows generating synthetic versioned RDF data for benchmarking purposes. BEAR benchmark [33] 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 [45, 46] allows describing changes inside the original data sources as a changelog and is used in the MQ framework [48] 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 [21] as seen in tools such as Morph-KGC [49] and SDM-RDFizer [50]. Both approaches 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

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

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 Morph-KGC and SDM-RDFizer, existing work on incremental KG generation is either not implemented [51] or does not consider heterogeneous data sources [52, 53].

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 during KG construction (FnO), and publishing the KG changes as an event stream (LDES) that allows replication and synchronization of such KGs.

3.1. RDF Mapping Language (RML)

RDF Mapping Language (RML) [10, 11][4] is an extension of W3C Recommendation R2RML [8] 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 as IRIs are generated from the data source 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 .
```

[4]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).

3.2. Function Ontology (FnO)

The Function Ontology (FnO) [22] semantically describes and declares implementation-independent functions and their relations to related concepts such as parameters, outputs, mappings to concrete implementations, and executions. The alignment between RML and FnO (via FNML) specifies how a data transformation must be performed by specifying the function to execute (Listing 2: line 5) and the function's 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 a RML Term Map. Therefore, an FnO function can be used as Subject Map, Predicate Map, Object Map, or other RML Term Maps.

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.

```
 1 <#FunctionMap> a fnml:FunctionMap
 2   fnml:functionValue [
 3     rr:predicateObjectMap [ a rr:PredicateObjectMap;
 4       rr:predicate fno:executes;
 5       rr:object grel:toUppercase;
 6     ];
 7     rr:predicateObjectMap [ a rr:PredicateObjectMap;
 8       rr:predicate grel:inputString;
 9       rr:objectMap [ a rr:ObjectMap;
10         rml:reference "name";
11       ];
12     ];
13   ];
14 .
```

3.3. Linked Data Event Streams (LDES)

Linked Data Event Stream (LDES) is a RDF data publishing approach fostered by the EU Semantic Interoperability Community (SEMIC)[5] and officially adopted as a standard specification by the Flemish government through its Flemish Smart Data Space project[6]. LDES defines datasets in terms of a collection of immutable objects (aka. members) such as versioned entities or observations (Listing 3), where every member must have its own unique IRI [4]. 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 [54][7] to semantically describe hypermedia relations among subsets or fragments of the data (see Figure 1). These hypermedia relations can be configured in multiple ways, e.g. by publishing fragments organized by time or by version. TREE also allows further describing the content of each member in an LDES collection through a SHACL shape (`tree:shape`, Listing 3: line 4), which allows consumers to understand the type and properties of the LDES members (`tree:member`, Listing 3: line 5), 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). 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 versions of a member. Consumers may poll or subscribe to an LDES to obtain new versions of members, which then can be processed and materialized in the consumer's local environment to remain up-to-date.

[5]https://joinup.ec.europa.eu/collection/semic-support-centre/linked-data-event-streams-ldes
[6]https://www.vlaanderen.be/vlaamse-smart-data-space-portaal
[7]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;
.
```

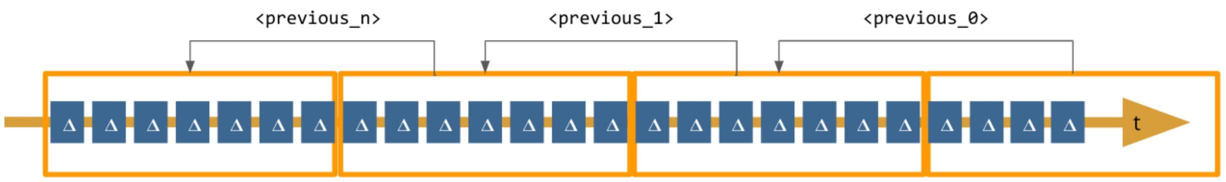

Fig. 1. LDES's structure allows to traverse the collection by clients with semantic descriptions of the collection's relations.

4. Approach

In this Section, we discuss the different types of change advertisement strategies identified in a set of heterogeneous data collections, and introduce our approach for incrementally detecting and integrating such changes into a Knowledge Graph (KG). Section 4.1 introduces the general idea for incremental KG generation. Section 4.2 discusses the various change advertisement strategies, which we studied along 3 dimensions: history, change advertisement, and change types. Section 4.3 describes how we detect both implicit and explicit changes in data collections. Lastly, Section 4.4 shows how we implemented our approach to incrementally generate and publish a KG.

4.1. IncRML

IncRML (Incremental RML) refers to 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 advertised explicitly or implicitly, and (ii) enriching the original target ontology mapping to include additional metadata that makes explicit which triples 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, as only the changes between consecutive data source(s) versions are materialized and integrated into a KG. Each time a change is detected during the KG generation process, we (re-)generate the RDF triples/quads of all the entities/members 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 collections modelled by an ontology with the semantics and expressivity to describe member changes, or extensions thereof. This allows consumers to synchronize their local version of a KG with the original producer, by interpreting the change semantics present in the LDES and performing the corresponding create, update, and delete operations instead of fetching and regenerating a complete version of the KG. Currently, triplestores lack support for ingesting LDES data directly. An additional step in the pipeline would be needed to integrate the LDES members directly into a triplestore. Therefore, in this work we focus on studying the impact of our approach on the generation aspect of an LDES-based continuous KG publishing system, and deem out of scope the ingestion aspect into triplestores.

4.2. Change advertisement strategies

Several change advertisement strategies can be identified if we analyze data collections along the following 3 dimensions: (i) availability of historical records, (ii) how changes are communicated to consumers, and (iii) type of change

History We identified 3 different types of history availability:

- *latest state*: Latest state refers to data collections that publish only the latest version of all its members on every change.
- *latest changes*: Latest changes refers to data collections that publish only changed members (a.k.a. delta updates). Thus, consumers must have access to an initial complete version of the collection upfront and reconcile updates over it.
- *full history*: Full history refers to data collections that are published including both historical and current versions of its members. Depending on the data publisher not all previous versions might be available.

Change communication We identified 2 change communication strategies:

- *explicit*: Data collection 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) and its type (e.g., create, update, or delete).
- *implicit*: Data collection 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 data collection. The member did not exist before in the data collection.
- *update*: An existing member of the data collection is modified. New properties are added to the member or existing properties are modified.
- *delete*: An existing member of the data collection 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 advertisement strategy combinations with respect to history availability, change communication, and change types identified on the set of real-world data collections analyzed in this work.

Table 1
Change advertisement strategies according to their history availability and change communication. Change communication can differ per type of change.

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

4.3. Implicit and explicit change detection

Data collections communicate their changes mainly in 2 ways, regardless of historic records availability: (i) explicitly through uniquely identified members, logs, etc., and/or (ii) implicitly by *silently* creating, updating, or deleting data collection 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 approaches [2] to handle both explicit and implicit data collection changes. In our approach, explicit changes are detected by relying on uniquely identified data collection members (e.g., via subject IRIs that depend on last modified timestamps), while implicit changes are detected by comparing consecutive snapshots of data collection members, checking their subject IRIs and (a subset of) their corresponding properties for changes.

Important to note that implicit deletion communication is not possible when combined with full history or latest changes. *Full history* data collections 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 present in the data collection. Even when a data consumer can tell apart historical records from new data, if a member stops occurring in a full history data collection, it could be interpreted either as being deleted or simply 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* data collections are similar to *full history* regarding deletions. If such data collections do not explicitly communicate that a member is deleted, consumers cannot determine with full certainty if a member was deleted or simply remains unchanged.

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

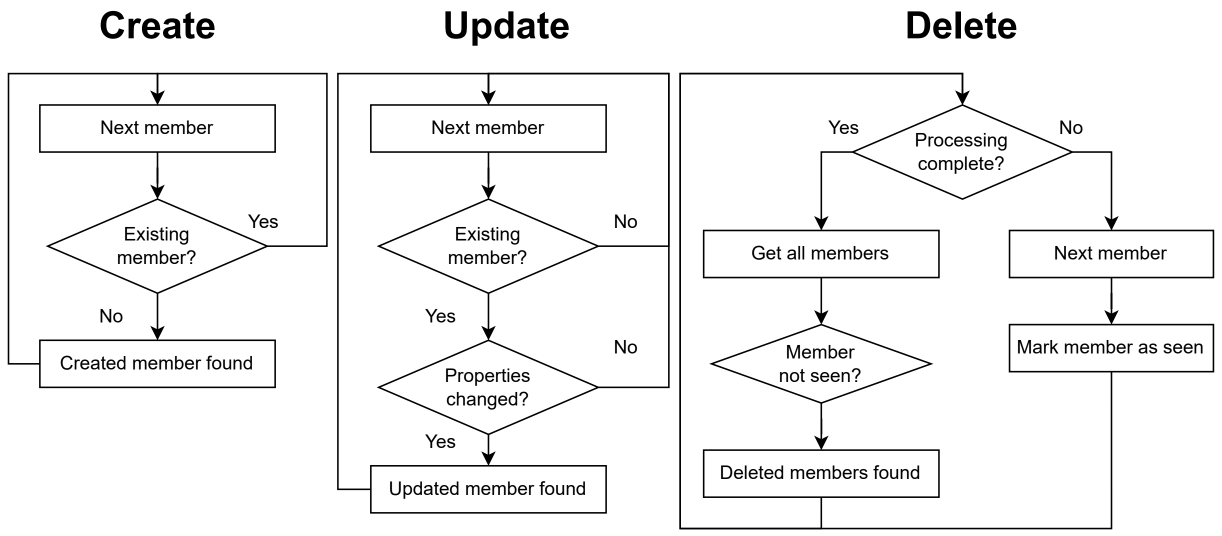

Fig. 2. Implicit updates must be detected by a Change Data Capture algorithm. Creates and updates are detected by checking if the member already exists in a previous snapshot. Members in a data collection are identified through their generated IRI. If the IRI does not exist in a previous snapshot, we 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 data collection.

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

Implicit Updates (Figure 2, middle) Similar to creates, the algorithm checks if the member was already present through its subject IRI and 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*. If the member's subject IRI was present and its watched properties' values were changed, a *updated* member is found.

Implicit Deletes (Figure 2, right) At KG generation time, the algorithm iterates over each member and marks it as seen. Once all members have been processed in the current version of the data collection, it identifies the members which were not seen in the current execution, but that were present in the previous KG generation execution (if any): these are marked as the *deleted* members in the current version of the data collection.

4.4. Incremental KG construction and publishing pipeline

We provide an implementation of our approach by bringing together (i) RML for declaratively defining generation rules for a KG; (ii) FnO for detecting changes across data source(s) versions (Section 4.3; and (iii) LDES to optionally publish these changes as an event stream for consumers. Figure 3 presents a schematic view of a data processing pipeline using our IncRML approach to incrementally construct and publish a KG. If changes are detected, the corresponding RML mapping(s) is/are executed to incorporate only the parts that have changed into the KG. In practice, a member is generated by a RML Triples Map (`rr:TriplesMap`[8].) with one Subject Map (`rr:SubjectMap`) and zero or more Predicate Object Maps (`rr:PredicateObjectMap`). Our Change Data Capture 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 is materialized and published as a typed event in an LDES. We rely on the W3C Activity Streams 2.0 vocabulary to annotate if a member was created (`as:Create`[9]), updated (`as:Update`), or deleted (`as:Delete`). Created and updated members are republished while deleted members only publish a tombstone (a simplified version of the member which indicates that the member was deleted without any other properties). By annotating these changes with the type of change, consumers can incorporate these changes on top of their copy of the KG.

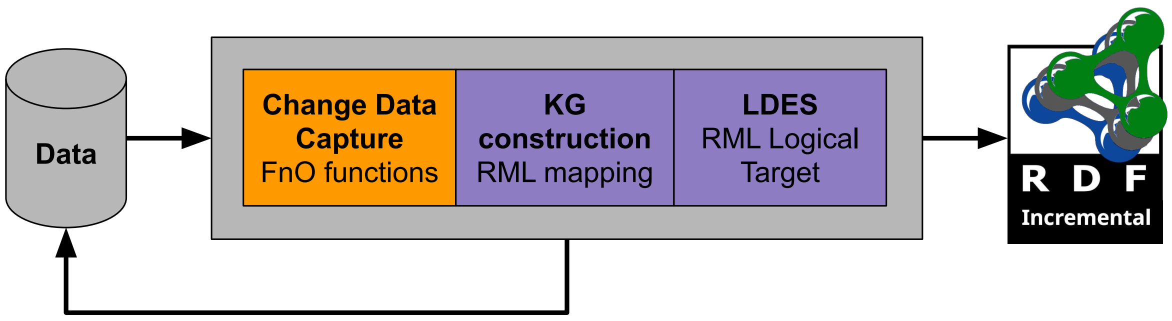

Fig. 3. Incremental KG construction and publishing pipeline using RML. The pipeline is executed repeatedly to detect changes with FnO functions and incorporate changes into the incrementally generated KG. The changes are published as an event stream using LDES.

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

5. Implementation

In this Section, we describe how we implement our proposed approach (Section 4) with Change Data Capture FnO functions, RML mapping rules to construct a Knowledge Graph (KG), and LDES for publishing an incremental 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 CDC FnO functions, Section 5.3 explains how we implemented LDES Logical Target for generating an LDES, Section 5.4 demonstrates how our implementation is applied on an example data collection, and Section 5.5 discusses the extensions we applied to the GTFS Madrid Benchmark for generating multiple data collection versions of the GTFS Madrid Benchmark.

5.1. Change Data Capture FnO functions

We implement our Change Data Capture algorithms from Section 4.3 [10] as FnO functions. A dedicated FnO function was implemented for each change type and change communication strategy (Table 2). In this way, we semantically describe which type of detection is applied on the data collection, even if the underlying detection algorithms operate similarly. Listing 4 shows the FnO description of a Change Data Capture FnO function for detecting implicit updates in a data collection. Based on the subject IRI of a member (`idlab-fn:iri`) and a set of watched properties (`idlab-fn:watchedProperty`), we detect implicit updates. The function keeps a state of the properties per member in the directory 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.

Listing 4: RML usage of a Change Data Capture FnO function for detecting implicit updates in a data collection.

```
<#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; ];
    ];
  ];
.
```

[10]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.

Each function requires an `rr:template` to construct the IRI of a member that is being checked for changes. The functions for explicitly communicated changes all implement the same functionality: checks if a member IRI was already generated in the past. This is due to each IRI being guaranteed to be unique when changes are explicit, therefore making it unnecessary to materialize previously seen IRIs. `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) for each member, to be compared with the previous version of the data collection and determine if anything changed. `idlab-fn:implicitDelete` keeps a registry (e.g. hashmap) of all seen member IRIs. This registry is used at the end of every processing round to detect missing – thus, deleted – members from the data collection. It returns a list with all the deleted member IRIs.

5.2. RML and CDC FnO functions integration in the Knowledge Graph construction pipeline

We integrate our FnO function 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 data collection. A separate Triples Map is used to generate additional metadata in the form of an event log to describe which type of changes took place in the data collection. A possible ontology to describe these change types is the W3C Recommendation Activity Streams 2.0 [5], but other ontologies can be used as well.

5.3. LDES Logical Target

We use LDES (Linked Data Events Stream) for publishing a stream of events about data collection member changes. LDES allows publishing only the changed members as a stream which can be consumed by third-parties to replicate and synchronize with a data collection. LDES consumers synchronize their KG locally by fetching or subscribing to the LDES. 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 a few additional properties such as `rmlt:ldesBaseIRI`, `rmlt:ldes`, and `rmlt:generateImmutableIRI`. The Event Stream Target provides unique and immutable IRIs for each LDES member as required by the LDES specification[11]. This way, consumers can uniquely identify individual changes in a data collection over time. We use an Event Stream Target for all member change types and one for generating an event log allowing consumers to subscribe to all members or only members of a certain change type.

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

Listing 5 provides an example of our alignment between LDES and RML with Change Data Capture FnO functions. In this example, we use `as:Create`, `as:Update`, and `as:Delete` of W3C Activity Streams 2.0 [5] to advertise the type of change to LDES consumers in the event log. Each change type has a dedicated RML Triples Map with an FnO function (Listing 5: lines 50-64, 65-84, 85-96) which outputs the generated members to a LDES Event Stream Target (Listing 5: lines 11-20). The W3C Activity Streams 2.0 event log is outputted to another LDES Event Stream Target (Listing 5: lines 1-10).

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.

```
 1 # Logical Target for outputting W3C ActivityStreams 2.0 event log as an LDES
 2 <#LDESLogicalTargetAS> a rmlt:EventStreamTarget;
 3     rmlt:target [ a void:Dataset; void:dataDump <file:///eventlog.nq>; ];
 4     rmlt:serialization formats:N-Quads;
 5     rmlt:ldes [ a ldes:EvenStream;
 6       ldes:timestampPath dct:created; ldes:versionOfPath dct:isVersionOf;
 7       tree:shape <https://example.org/shape/>;
 8     ];
 9     rmlt:ldesBaseIRI <https://example.org/ldes/eventlog/>;
10     rmlt:ldesGenerateImmutableIRI "true"^^xsd:boolean .
11 # Logical Target for outputting data collection member changes as an LDES
12 <#LDESLogicalTargetMember> a rmlt:EventStreamTarget;
13     rmlt:target [ a void:Dataset; void:dataDump <file:///members.nq>; ];
14     rmlt:serialization formats:N-Quads;
15     rmlt:ldes [ a ldes:EvenStream;
16       ldes:timestampPath dct:created; ldes:versionOfPath dct:isVersionOf;
17       tree:shape <https://example.org/shape/>;
18     ];
19     rmlt:ldesBaseIRI <https://example.org/ldes/members/>;
20     rmlt:ldesGenerateImmutableIRI "true"^^xsd:boolean .
21 # Input CSV file as datasource
22 <#DataSource> a rml:LogicalSource;
23   rml:source "data.csv";
24   rml:referenceFormulation ql:CSV .
25 # W3C ActivityStreams 2.0 eventlog generation of created members
26 <#TriplesMapASCreate> a rr:TriplesMap;
27   rml:logicalSource <#DataSource>;
28   rr:subjectMap [
29     rr:constant "http://blue-bike.be/event/create"; rr:class as:Create;
30     rml:logicalTarget <#LDESLogicalTargetAS>;
31   ] .
32 # W3C ActivityStreams 2.0 eventlog generation of updated members
33 <#TriplesMapASUpdate> a rr:TriplesMap;
34   rml:logicalSource <#DataSource>;
35   rr:subjectMap [
36     rr:constant "http://blue-bike.be/event/update"; rr:class as:Update;
37     rml:logicalTarget <#LDESLogicalTargetAS>;
38   ] .
39 # W3C ActivityStreams 2.0 eventlog generation of deleted members
40 <#TriplesMapASDelete> a rr:TriplesMap;
41   rml:logicalSource <#DataSource>;
42   rr:subjectMap [
43     rr:constant "http://blue-bike.be/event/delete"; rr:class as:Delete;
44     rml:logicalTarget <#LDESLogicalTargetAS>;
45   ] .
46 # Data collection member
47 <#PersonName> a rr:PredicateObjectMap;
48   rr:predicate schema:name;
49   rr:objectMap [ rml:reference "name"; rr:datatype xsd:string; ] .
50 # Detection of member creations with FnO function
51 <#TriplesMapObjectCreate> a rr:TriplesMap;
52   rml:logicalSource <#DataSource>;
53   rr:subjectMap [
54     fnml:functionValue [
55       rr:predicateObjectMap [ rr:predicate fno:executes; rr:object idlab-fn:explicitCreate; ];
56       rr:predicateObjectMap [
57         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 member updates with FnO function
<#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 member deletions with FnO function
<#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 data collection which consists of an initial version (Table 3a) and an implicitly updated version (Table 3b). The resulting RDF quads of the initial (Listing 6) and updated (Listing 7) data collections consist of 3 named graphs: `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 [5][12].

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

(a) *Initial* data collection 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* data collection has 2 updated, 1 unchanged, 1 created and 1 deleted member(s).

Table 3
Example data

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

Listing 6: The materialized KG in TriG of the *initial* data collection. All data collection members are materialized because this is the first time the data collection is processed. Thus, they are part of the Create named graph.

```
:Created {
 <http://ex.org/Mbr0#0> 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> 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> 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> 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* data collection. 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.

```
: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> .
```

5.5. *GTFS Madrid Benchmark extension*

We extend the GTFS Madrid Benchmark [7] data generator[13] to allow generating different versions of the benchmark data by supporting creates, updates, and deletes of the GTFS data model. We implemented creates, updates, and deletes in the GTFS Madrid Benchmark similar to how real-world GTFS datasets are changed from 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. Creates are applied by adding GTFS routes and their associated GTFS trips, stops, stoptimes, and services entities. For example: 25% creates will provide 25 % additional new routes with respect to the original amount of routes. 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 for applying our creates, updates, and deletes.

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

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 creates 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 deletes must be applied on the generated data.

6. Evaluation

In this Section, we describe our methodology to evaluate our approach on several use cases using both synthetic and real-world data. First, we discuss our use cases and how they can be classified according to the different change advertisement strategies identified in this work (Section 6.1). Afterwards, 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 advertisement 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 generating Knowledge Graphs (KG) incrementally; and (iii) 5 different real-world use cases (Section 6.1.3) to investigate its performance and resource impact on different types of datasets (Section 4.2),

6.1.1. Functionality

We evaluate 3 change advertisement strategy dimensions (history availability, change communication, and change type (Section 4.2)) through a set of test cases (Table 4). We combined the 3 types of history availability (latest state, latest changes, and full history) with the 2 types of change advertisement (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 the implementation, which yields additionally 6 more test-cases (24 in total). Since 2 combinations are not possible (Section 4: implicit deletion advertisement for full history and latest changes), we removed these from the test cases, bringing the number of test cases to 22.

We validate that all combinations of these dimensions are possible and use these test cases to verify if our implementation covers all possible scenarios. The test cases are available on GitHub[14].

6.1.2. Extended version of the GTFS Madrid Benchmark

We aim on analyzing the impact of our approach on data collections of varying size and change characteristics, by measuring execution time and resource usage (storage, CPU time, and RAM usage). 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 our approach may achieve, might be affected by the amount of changes, type of changes, and the data collection 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 data collection size scales. Besides scaling the data size, we also scale the change percentage (0%, 25%, 50%, 75%, and 100%), equally divided among the different change types, with a fixed data size scale of 100. We also experiment with the type of change by using a fixed change percentage of 50% 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 data collection which we apply as a change over an initial base version.

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

Table 4
22 test cases for detecting changes in data collections. Achieving 100% coverage in our reference implementation extending the RMLMapper proves the feasibility of our approach.

ID	History availability	Advertisement	Change type	Purpose
RMLLDES0001a	Latest State	Explicit	No Change	Unchanged data produces no output
RMLLDES0001b	Latest State	Explicit	Create	Create to the data produces an added entity
RMLLDES0001c	Latest State	Explicit	Update	Update to the data produces a modified entity
RMLLDES0001d	Latest State	Explicit	Delete	Delete in the data produces a deleted entity
RMLLDES0001e	Latest State	Implicit	No Change	Unchanged data produces no output
RMLLDES0001f	Latest State	Implicit	Create	Create to the data produces an added entity
RMLLDES0001g	Latest State	Implicit	Update	Update to the data produces a modified entity
RMLLDES0001h	Latest State	Implicit	Delete	Delete in the data produces a deleted entity
RMLLDES0002a	Latest Changes	Explicit	No Change	Unchanged data produces no output
RMLLDES0002b	Latest Changes	Explicit	Create	Create to the data produces an added entity
RMLLDES0002c	Latest Changes	Explicit	Update	Update to the data produces a modified entity
RMLLDES0002d	Latest Changes	Explicit	Delete	Delete in the data produces a deleted entity
RMLLDES0002e	Latest Changes	Implicit	No Change	Unchanged data produces no output
RMLLDES0002f	Latest Changes	Implicit	Create	Create to the data produces an added entity
RMLLDES0002g	Latest Changes	Implicit	Update	Update to the data produces a modified entity
RMLLDES0003a	Full History	Explicit	No Change	Unchanged data produces no output
RMLLDES0003b	Full History	Explicit	Create	Create to the data produces an added entity
RMLLDES0003c	Full History	Explicit	Update	Update to the data produces a modified entity
RMLLDES0003d	Full History	Explicit	Delete	Delete in the data produces a deleted entity
RMLLDES0003e	Full History	Implicit	No Change	Unchanged data produces no output
RMLLDES0003f	Full History	Implicit	Create	Create to the data produces an added entity
RMLLDES0003g	Full History	Implicit	Update	Update to the data produces a modified entity

6.1.3. Real-world use cases

We apply our approach on 5 real-world use cases: (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 use cases stand as representative examples for all the identified change advertisement strategies (Table 5) regarding change communication, change types, and history availability. For each use case, we collect released versions of each data collection during 24 hours, with the exception of De Lijn and NMBS, since they only provide a new version per day. We use a time-frame instead of the number of versions to have different frequencies when new versions of the data collections are published e.g. VVC is changed more frequent (every 2-3s) compared to KMI (every 10 mins). In this case, we collect new versions during a week. Next we describe in detail each of the analyzed data collections, which are also summarized in Table 5.

BlueBike & JCDecaux These data collections provide information related to bike-sharing services including data about stations and currently available bikes. They are publicly accessible via HTTP APIs[15]. Both data collections 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 data collection. Regarding history availability, these data collections follow the *latest state* strategy, overwriting the whole data collection with the current state of each collection mem-

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

ber on every new version. We use a GBFS-based vocabulary[16] to transform this bike-sharing data into RDF and retrieve the data every minute to check for changes.

NMBS & De Lijn These data collections contain public transport timetables in GTFS format, which are updated on daily basis. They are publicly accessible as data dumps[17]. In GTFS timetables, creations are explicitly communicated by adding 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[18] to transform the GTFS timetables into RDF, based on the RML mappings of the GTFS-Madrid-Bench [7].

OpenStreetMap (OSM) This is a crowd-funded geographical data collection which provides changed data collections 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 [55], which provides an ontology for OpenStreetMap's Nodes and Ways. OSM follows the *latest changes* approach regarding history availability. Since OpenStreetMap already explicitly provides unique IDs for each change, no additional processing is needed by our approach. Thus, semantically publishing the data collection changes can be achieved with RML mappings without applying the CDC functions on the data collections. We include this data collection in this work to show the wide range of data collection types regarding change advertisement, history availability, and change types.

Vlaams Verkeerscentrum (VVC) This data collection contains 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 use case due to the high update frequency of this data collection, compared to the other use cases: the RDF generation must keep up with this high frequency to avoid high latency. We use the DATEX II v3 VMS ontology[19] for the content of the dynamic message board messages and collect the data every 3 seconds. This data collection 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 use case, but republished as new message instances instead. Therefore, there are no updates, only creations and deletions. Deletes 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 data collection. If the board is put in service again, it is considered as a creation.

Koninklijk Meteorologisch Instituut van België (KMI) This data collection is published by the Belgian meteorological institute which allows access to the measurement history of their weather sensors. The data collection 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 data collection. We use the W3C Semantic Sensor Network ontology (SSN) [20] to transform the sensors' measurements into RDF and collect the data every 10 minutes.

[16]https://github.com/jiaoxlong/gbfs-json-schema/tree/gbfs-ld/GBFS-LD/v2.3
[17]NMBS: https://gtfs.irail.be/nmbs/gtfs/latest.zip, De Lijn: https://gtfs.irail.be/de-lijn/de_lijn-gtfs.zip
[18]http://vocab.gtfs.org/terms#
[19]https://datex2.eu/vocab/3/Vms/
[20]https://www.w3.org/TR/vocab-ssn/

Table 5
Real-world use cases addressed in our evaluation. Each dimension of data collection types is covered: change advertisement and history availability. Some datasets only provide certain change types, inapplicable change types are marked as Not Applicable (NA).

Use case	**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

6.2. *Quantitative evaluation setup*

We evaluate our approach on the described use cases and a modified version of the GTFS-Madrid-Benchmark by measuring the following metrics: (i) execution time to materialize the set of changes of the data collection 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 [21] to investigate how our approach impacts the measured metrics during RDF KG generation.

For each experiment, we materialize a given data collection and its corresponding updated versions as RDF quads. 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 collection is available.
- `CHANGE` is our novel approach to versioned 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 collection, not the complete KG. Thanks to the LDES Logical Target, we add additional metadata to advertise semantically 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 data collection version. Since our focus is the generation, we do not include the ingestion of the generated RDF quads into a triplestore for querying. Moreover, triplestores have various approaches to insert new quads: SPARQL's `UPDATE`, bulk loading, etc. Each of these ingestion approaches has its own benefits and drawbacks depending on the triplestore and vendor. Thus, we only compare the amount of generated quads for each data collection 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 [22].

[21]Our approach is engine-agnostic, thus can be implemented in any KG construction engine that supports FnO.
[22]DOI: https://doi.org/10.5281/zenodo.10171157

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). Afterwards, 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 use cases (Section 7.3). In Section 7.4 we discuss the impact of our approach on the measured metrics.

7.1. Functionality

We integrated our set of 22 test cases (Table 4 of Section 6.1.1) in the RMLMapper v6.3.0[23] which **has 100% coverage** for these test cases. With these test cases, we confirm that our approach is implemented correctly in the RMLMapper and covers all possible combinations regarding change advertisement and history availability of the different data collection types.

7.2. GTFS Madrid Benchmark

In this section, 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 upon the initial version (Table 7). We execute each experiment by starting with the base data collection to generate the initial KG and apply the corresponding data collection updates on top of the base data collection. Each experiment is executed 5 times from which the median value is taken for each metric of each data collection version. We report the average of these median metrics e.g. execution time, CPU time, and memory usage across multiple data collection updates in Tables 6 and 7. Storage usage is reported as the sum of KG sizes generated from the base data collection and all its updated data collections (Table 8).

Results show that our approach **heavily 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 a slight increase in execution time, storage, and CPU time, while memory usage is mostly unaffected, only the tracking of data collection members by our approach has a slight increase in memory for `CHANGE`. Our approach remains unaffected by the amount of changes or type of change, being solely impacted by the data collection size.

Table 6
Initial execution results for all GTFS Madrid Benchmark for strategies `ALL` (no change detection) and `CHANGE` (with change detection). GTFS$_{\text{CHANGE}}$ and GTFS$_{\text{TYPE}}$ all use the same data scale (scale 100) which results into the same storage usage for the initial generation.

Scenario	**Execution time (s)**		**CPU time (s)**		**Peak memory (GB)**	
	ALL	**CHANGE**	**ALL**	**CHANGE**	**ALL**	**CHANGE**
GTFS$_{\text{SCALE}}$ 1	12.29	16.18	26.80	36.31	4.25	4.44
GTFS$_{\text{SCALE}}$ 10	82.15	124.84	129.45	206.05	7.57	14.00
GTFS$_{\text{SCALE}}$ 100	922.43	1943.38	2 260.62	4626.85	46.30	51.78
GTFS$_{\text{CHANGE}}$ 0%	919.25	2 736.11	2 259.27	5 002.98	45.29	52.30
GTFS$_{\text{CHANGE}}$ 25%	914.98	1 575.10	2 303.53	4 521.50	46.38	51.64
GTFS$_{\text{CHANGE}}$ 50%	922.43	1 943.38	2 260.62	4 626.85	46.30	51.78
GTFS$_{\text{CHANGE}}$ 75%	924.12	2 357.18	2 224.46	4603.18	46.77	52.09
GTFS$_{\text{CHANGE}}$ 100%	912.37	1 420.53	2 336.65	4 431.36	46.32	51.70
GTFS$_{\text{TYPE}}$ CREATE	990.30	1 397.49	2 448.92	4 414.65	48.07	51.73
GTFS$_{\text{TYPE}}$ UPDATE	924.16	1 910.59	2 239.66	4 507.61	47.13	51.64
GTFS$_{\text{TYPE}}$ DELETE	929.74	1 671.17	2 302.00	4 702.38	46.58	52.30

[23]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 data collection updates are included, initial execution is not included.

Scenario	**Execution time (s)**		**CPU time (s)**		**Peak memory (GB)**	
	ALL	**CHANGE**	**ALL**	**CHANGE**	**ALL**	**CHANGE**
GTFS$_{\text{SCALE}}$ 1	12.20	8.74	27.40	26.58	4.25	4.25
GTFS$_{\text{SCALE}}$ 10	83.41	44.93	132.63	88.73	7.53	12.58
GTFS$_{\text{SCALE}}$ 100	928.93	473.55	2 306.50	1 083.96	47.41	47.01
GTFS$_{\text{CHANGE}}$ 0%	927.82	470.73	2 292.39	1 072.47	47.19	47.33
GTFS$_{\text{CHANGE}}$ 25%	912.50	481.62	2 303.02	1 090.74	46.86	47.29
GTFS$_{\text{CHANGE}}$ 50%	928.93	473.55	2 306.50	1 083.96	47.41	47.01
GTFS$_{\text{CHANGE}}$ 75%	929.53	466.87	2 292.81	1 056.88	47.21	47.19
GTFS$_{\text{CHANGE}}$ 100%	914.68	460.30	2 301.77	1 063.46	47.05	47.35
GTFS$_{\text{TYPE}}$ CREATE	1 021.04	454.06	2 386.46	1 063.90	48.55	46.61
GTFS$_{\text{TYPE}}$ UPDATE	953.32	479.87	2 298.29	967.63	46.77	47.32
GTFS$_{\text{TYPE}}$ DELETE	922.04	464.78	2 266.11	1 056.56	46.69	47.33

Table 8
Storage usage for initial and all updates per strategy of the GTFS Madrid Benchmark. GTFS$_{\text{CHANGE}}$ and GTFS$_{\text{TYPE}}$ all use the same data scale (scale 100) which results into the same storage usage for the initial generation. Total storage usage is the sum of the base collection and all applied updates upon it.

Scenario	**Initial storage usage (kb)**		**Total storage usage (kb)**	
	ALL	**CHANGE**	**ALL**	**CHANGE**
GTFS$_{\text{SCALE}}$ 1	92 537.13	116 662.92	1 023 192.74	125 962.43
GTFS$_{\text{SCALE}}$ 10	732 959.66	1 168 374.14	8 100 043.47	1 234 254.86
GTFS$_{\text{SCALE}}$ 100	7 347 846.47	11 701 971.43	81 178 874.08	12 323 348.17
GTFS$_{\text{CHANGE}}$ 0%	7 347 846.47	11 701 971.43	80 826 311.20	11 703 851.89
GTFS$_{\text{CHANGE}}$ 25%	7 347 846.47	11 701 971.43	81 005 737.97	12 019 442.49
GTFS$_{\text{CHANGE}}$ 50%	7 347 846.47	11 701 971.43	81 178 874.08	12 323 348.17
GTFS$_{\text{CHANGE}}$ 75%	7 347 846.47	11 701 971.43	81 355 130.62	12 631 882.64
GTFS$_{\text{CHANGE}}$ 100%	7 347 846.47	11 701 971.43	81 530 274.97	12 934 632.82
GTFS$_{\text{TYPE}}$ CREATE	7 347 846.47	11 701 971.43	82 031 098.47	13 530 545.51
GTFS$_{\text{TYPE}}$ UPDATE	7 347 846.47	11 701 971.43	80 826 311.20	11 712 313.42
GTFS$_{\text{TYPE}}$ DELETE	7 347 846.47	11 701 971.43	80 689 654.29	11 727 476.53

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 4). However, the initial generation of the KG from the base data collection has a longer execution time and higher resource consumption since all data collection 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 data collection changes, the overhead of detecting changes and LDES event stream metadata is mostly noticeable with GTFS$_{\text{SCALE}}$ 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$_{\text{SCALE}}$ 10, the memory usage increases since the Java JVM does not perform garbage collection as the Java heap space still has enough free space. This is not the case for GTFS$_{\text{SCALE}}$ 100 where memory usage is similar again compared to no change detection.

Scaling amount of changes **Increasing the amount of changes increases the storage usage** with our approach can achieve, while it has no impact on the resource consumption (Figure 5) because each data collection member

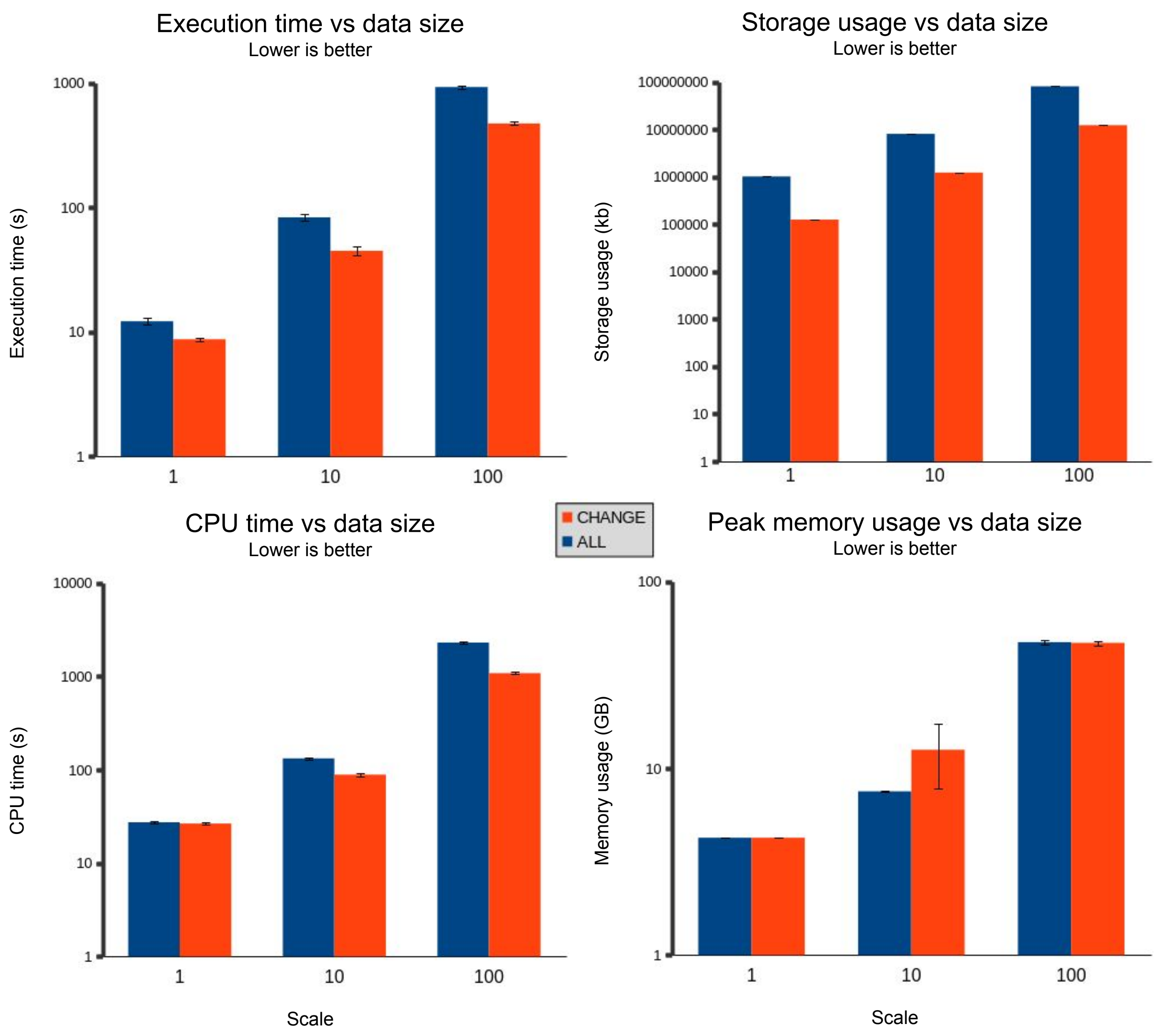

Fig. 4. GTFS Madrid Benchmark results for different data scales with a fixed amount of changes (50%). Only results of data collection updates are included, initial execution is not included.

is evaluated regardless of the amount of changes. Similar to scaling the data size, the initial KG generation from the base data collection introduces overhead to initialize the state for tracking the data collection members. When only considering the data collection updates, if no changes are found between 2 versions (0%), only the overhead of generating LDES event stream metadata is affecting storage usage because no members are materialized, 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 data collection member, for each change type (Figure 6). 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 as GTFS trips, shapes, etc., must be generated as new triples. Deletions only keep a tombstone of the member, e.g.,

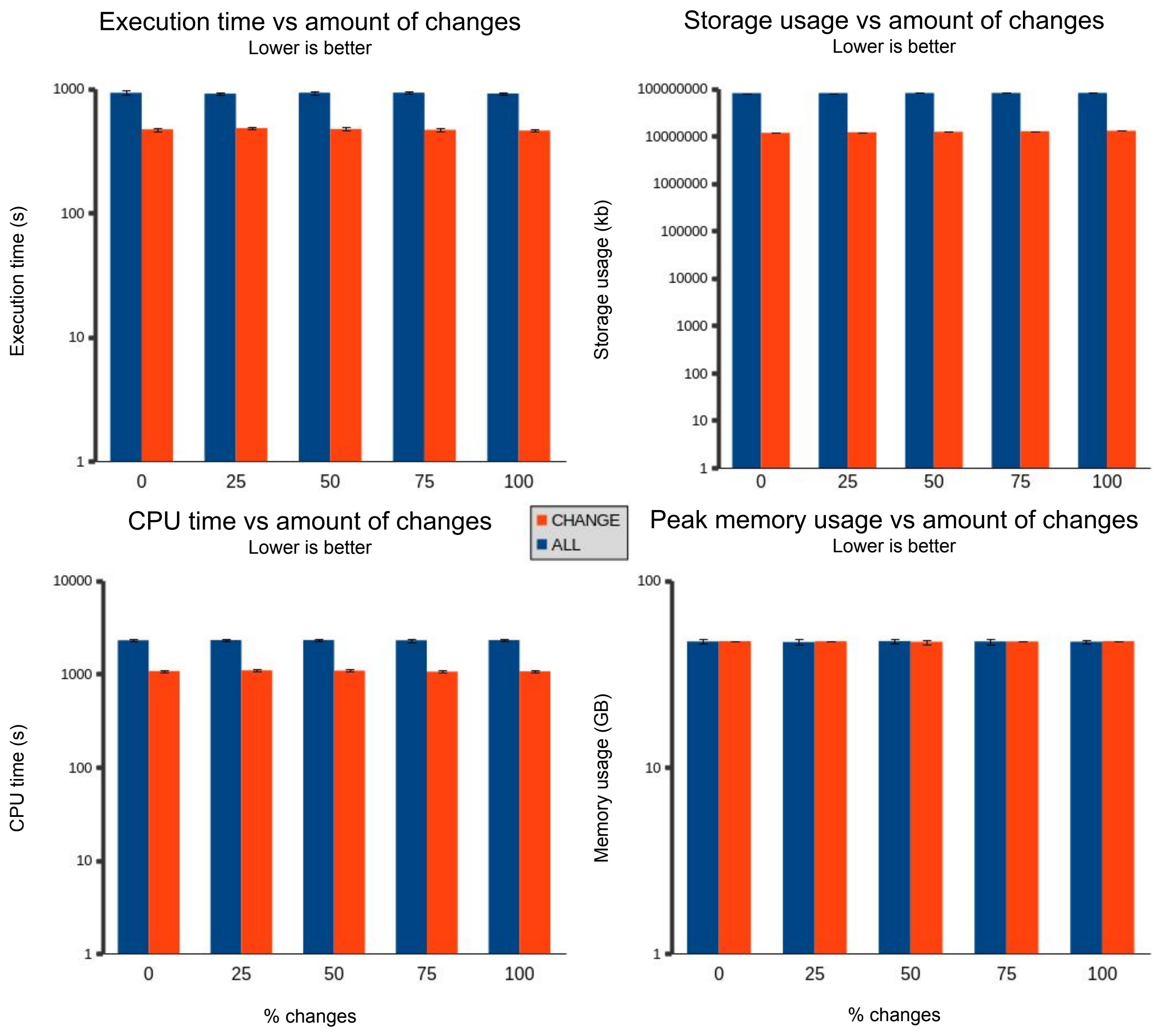

Fig. 5. GTFS Madrid Benchmark results for different amount of changes with a fixed data size (scale 100). Only results of data collection updates are included, initial execution is not included.

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 data collection 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 data collection.

7.3. Real-World use cases

In this section, we discuss the results obtained from applying our approach over the set of real-world use cases described in Section 6.1.3, with respect to storage usage, execution time, CPU time, and memory usage. For each use case, we measure 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 multiple data collection updates on the measured metrics. We execute each experiment by starting with the base

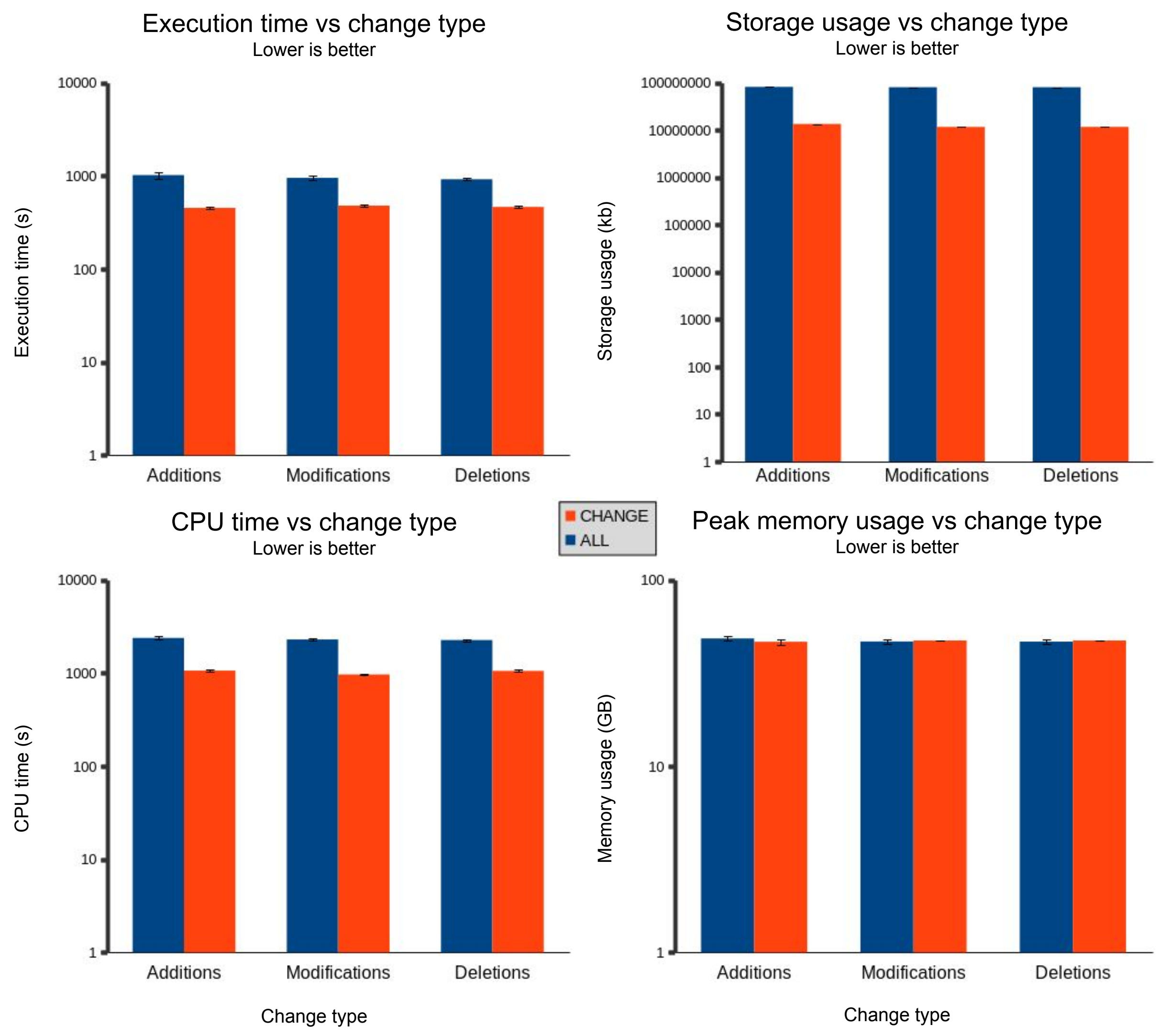

Fig. 6. GTFS Madrid Benchmark results for different change types with a fixed data size (scale 100) and amount of changes (50%). Only results of data collection updates are included, initial execution is not included.

data collection to generate the initial KG and apply the corresponding data collection updates on top of the base data collection. Each experiment is executed 5 times from which the median value is taken for each metric of each data collection update. We report the average of these median metrics e.g., execution time, CPU time, and memory usage across multiple data collection updates in Tables 9 and 10. Storage usage is reported as the sum of KG sizes generated from the base data collection and all its updated data collections (Table 11). OSM advertises explicitly all changes, thus there is no $\mathrm{OSM}_{\mathrm{ALL}}$.

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).

Scenario	Execution time (s)		CPU time (s)		Peak memory (GB)	
	ALL	CHANGE	ALL	CHANGE	ALL	CHANGE
BlueBike	2.59	2.80	7.87	9.12	2.39	2.42
JCDecaux	3.73	4.65	16.82	21.63	2.13	2.33
NMBS	151.02	177.11	232.11	292.61	17.41	23.58
De Lijn	1 156.67	1 542.23	3 101.31	6 142.43	49.98	50.09
KMI	534.93	637.28	850.56	1 147.12	36.11	38.86
VVC	19.94	17.94	44.67	43.09	7.61	6.46
OSM	NA	3.10	NA	11.22	NA	2.67

Table 10
Execution results for all real-life use cases for strategies `ALL` (no change detection) and `CHANGE` (with change detection). Only results of data collection updates are included, initial execution is not included.

Scenario	Execution time (s)		CPU time (s)		Peak memory (GB)	
	ALL	CHANGE	ALL	CHANGE	ALL	CHANGE
BlueBike	2.56	2.63	7.68	8.21	2.39	2.40
JCDecaux	3.79	3.78	16.74	19.77	2.13	2.16
NMBS	146.98	86.17	227.29	144.21	20.08	27.88
De Lijn	1 072.17	385.53	2523.12	1 268.39	48.88	48.61
KMI	527.95	119.63	845.98	184.25	34.03	31.14
VVC	17.47	6.36	41.10	22.73	7.35	4.87
OSM	NA	3.56	NA	16.64	NA	2.92

Table 11
Storage usage for initial and all updates per strategy of the real-world use cases. Total storage usage is the base collection and the applied updates upon it.

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

Storage usage **Our approach reduces storage usage by a reduction factor between 3.24 and 315.83 depending on the data collection** (Figure 7). We observe a significant reduction in storage use in *full history* data collections such as KMI, since 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 data collection. The same principle applies for data collections that have few changed members on every new version (e.g. VVC). Data collections containing a large number of changes between versions, e.g. NMBS, have a lower reduction in storage usage. The initial KG generation has a higher storage usage for all use cases because all members are generated next to the additional the metadata indicating that each member was created.

CPU time Similar to execution time, **our approach reduces CPU time depending on the data collection size** (Figure 7), with a factor up to 4.59. On larger data collections (e.g. KMI, De Lijn, NMBS, or VVC) more members avoid unnecessary materialization, thus obtaining a higher reduction in CPU time. However, smaller data collections (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 use cases, as the state for each use case is initialized to track each data collection member.

Memory usage Results show that **our approach does not have a significant impact on the memory usage** (Figure 7) 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 JVM does not execute garbage collection while processing this data collection since the heap space is not limited yet. Bigger data collections, 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 use cases. VVC use case 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 data collection updates because the same amount of memory is needed to evaluate each data member of the base data collection and the corresponding data collection updates.

7.4. Discussion

Functionality RMLMapper v6.3.0 has **100% coverage for all the test cases** (Table 4). RMLMapper can detect all change types, both explicit and implicit with all identified history advertisement types.

GTFS Madrid Benchmark Our **approach reduces the storage (~6x), execution time (~2x), CPU time (~2x), memory consumption is mostly affected** 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 data collection is rather small which results in a bigger impact of the overhead from tracking data collection members. Therefore, CPU time usage remains the same while execution time has a smaller reduction (-30%) compared the $\text{GTFS}_{\text{SCALE}}$ 10 and $\text{GTFS}_{\text{SCALE}}$ 100. Only for $\text{GTFS}_{\text{SCALE}}$ 10 the memory usage increases because the Java JVM does not perform garbage collection as the Java heap space still have enough free memory. $\text{GTFS}_{\text{SCALE}}$ 100 is larger which causes the Java JVM to trigger garbage collection, thus lower 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 involved (11.70GB to 12.93GB) since more new versions of data collection members are generated. CPU time usage and memory usage are unaffected because the number of data collection members remains unchanged, each data collection member is evaluated 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 data collection need to be created. Deletions reduce the storage usage, but not heavily because of deleting a GTFS Route only affects a subset of the data collection. Moreover, a tombstone is still kept to indicate that the member was deleted.

Real-world use cases Depending on the data size, **our approach generates a KG up to 4.41 times faster** by only materializing the actual changes into the new version of the generated KG. Besides faster KG 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 data collection size. **Memory usage is not impacted** because the increase of memory usage from 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 data collections in practice. The type of change does not

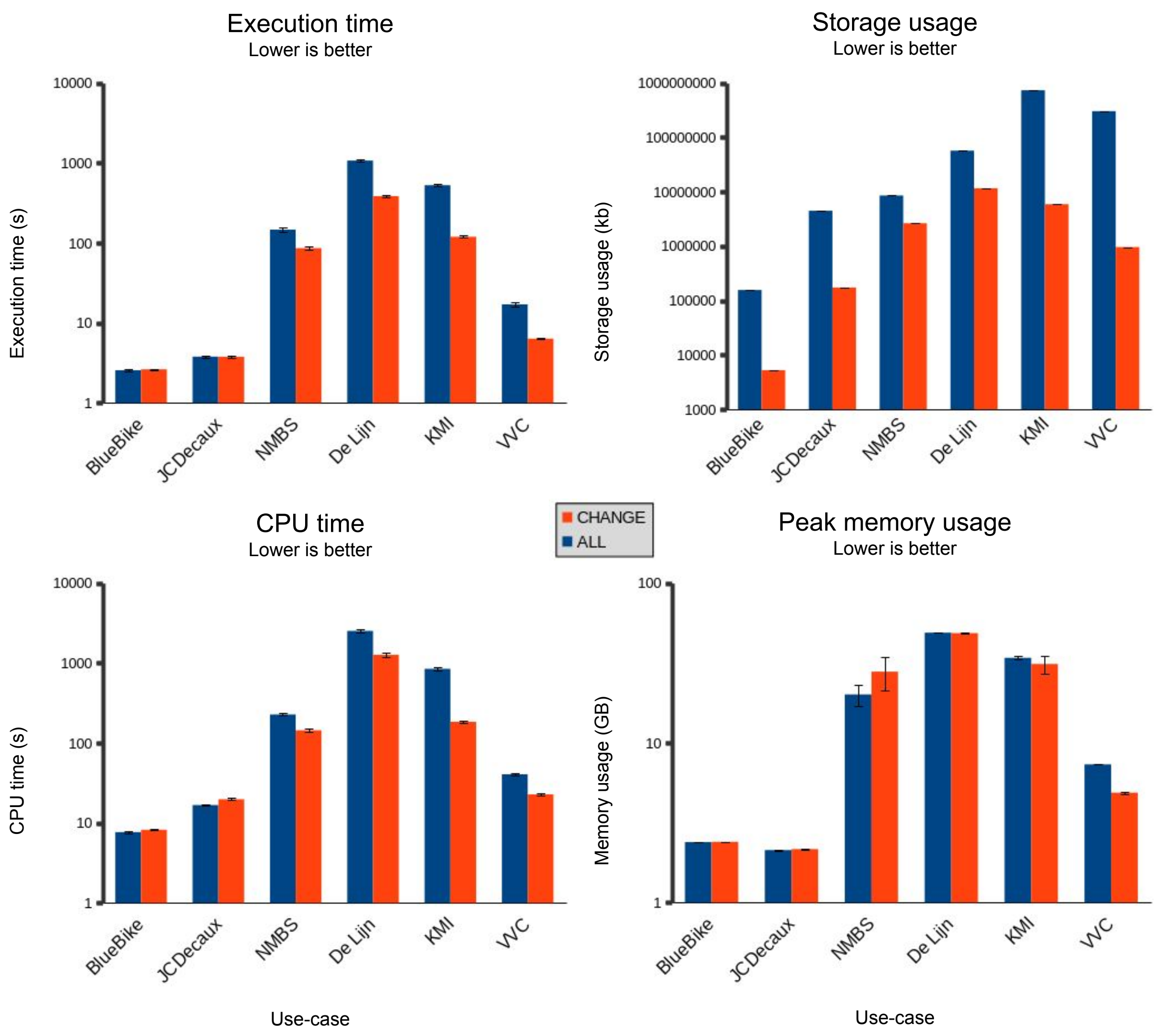

Fig. 7. Our approach reduces the necessary resources to generate different versions of a KG. Storage is reduced 3.24–315.83 times depending on dataset type. Execution time is reduced between 0.97–4.41 times depending on the dataset type. CPU time usage is reduced 0.85–4.59 times. Memory usage is mostly unaffected.

affect our approach in execution time, CPU and memory usage because every data collection member is evaluated. However, storage usage is affected since additional created members need to be store, deleted members release occupied storage, and updated members could increase or decrease storage usage based on the change.

For smaller data collections (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 still reduced (-96.14% – -96.65%**. For the initial KG generation from the base data collection, the execution time is longer and more resources are consumed because the internal state must be initialized for tracking all data collection members. Once the data collection 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 data collections more than larger ones. Storage usage is still heavily reduced. Nevertheless, clients consuming the semantically annotated stream of changes in the form of an LDES, could benefit from knowing precisely which parts of the data are created, updated and deleted in the KG. Clients could potentially optimize their consumption based on the type of change.

For larger data collections (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 data collection as a KG. Note that memory consumption has an increase for NMBS (+38.8%) because the Java JVM did not perform garbadge collection as still enough heap memory is free for this use case. Other use cases are impacted by -0.55% – -31.31%.

8. Conclusion

In this paper, we investigated how to detect and materialize only data collection updates towards establishing an incremental publishing approach for Knowledge Graphs (KG). We show how our approach reduces execution time and computing resources usage (CPU, memory, and storage).

With 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.

We evaluated our approach on a wide spectrum of different real-world data collections, ranging from weather sensor data to public transport timetables which indicated a reduction in storage (up to 315.83x less), CPU time (up to 4.59x less) and memory usage (up to 1.51x less) while also materializing updates faster into a KG (up to 4.41 times lower execution time). Our approach covers different change communication strategies (explicitly by the data source, or implicitly by silently changing the data), change types (creations, updates, and deletions), and history availability by the data source (latest state, latest changes, or full history data collections).

Thanks to our approach, we provide the means to generate semantically annotated data and metadata from both implicitly and explicitly changed data collections. Therefore, we help to bring more transparency/provenance, in particular to implicitly advertised data collections. We rely on LDES to publish a semantically and structurally described stream of events that could enable data consumers to replicate and continuously synchronize with a data collection, whether their changes are explicitly or implicitly advertised. Furthermore, our approach facilitates and reduces the cost of storing and publishing historic data, which is vital for e.g., data analysis and machine learning applications.

Further research includes expanding the pipeline to investigate the impact on end-to-end performance with triple-stores and multiple consumers, investigating optimizations for the proposed change detection algorithms, exploring windowing techniques for streaming data, and involving querying of incremental KGs.

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] 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.

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

[6] 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.

[7] 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.

[8] 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/.

[9] 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/.

[10] 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.

[11] 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.

[12] 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.

[13] 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/.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[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] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

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

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] R. Taelman, M. Vander Sande and R. Verborgh, Versioned Querying with OSTRICH and Comunica in MOCHA 2018, in: *Proceedings of the Mighty Storage Challenge*, 2018.

[43] 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.

[44] 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.

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

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

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

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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/.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.