Master's Thesis

Live Updates for SPARQL Endpoints Containing OpenStreetMap Data

Nicolas von Trott zu Solz

November 2, 2025

Submitted to the University of Freiburg
Faculty of Engineering
Department of Computer Science
Chair of Algorithms and Data Structures

universität freiburg

University of Freiburg
Faculty of Engineering
Department of Computer Science
Chair of Algorithms and Data Structures

Author

Nicolas von Trott zu Solz,
Matriculation Number: 5420422

Examiners

Prof. Hannah Bast,
Faculty of Engineering
Chair of Algorithms and Data Structures

Prof. Dr. Fabian Kuhn,
Faculty of Engineering
Algorithms and Complexity Group

Supervisor

Prof. Hannah Bast,
Faculty of Engineering
Chair of Algorithms and Data Structures

Declaration

I hereby declare that I am the sole author and composer of this thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.

I hereby also declare that my thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Nerterschausen, 2.11.2025
Place, Date

Nicolas von Trott
Signature

Abstract

OpenStreetMap (OSM) is a collaborative project that provides free and editable geospatial data worldwide. Representing OSM data in knowledge graphs based on the Resource Description Framework (RDF) enables the execution of expressive queries via the SPARQL query language. However, keeping such endpoints synchronized with OSM's rapidly evolving planet-scale data is computationally demanding. This thesis introduces ***osm-live-updates*** **(*olu*)**, a tool that generates SPARQL update operations from OSM change files, enabling efficient and near real-time updating of OSM data on SPARQL endpoints. When paired with a high-performance SPARQL engine like *QLever*, *olu* can generate SPARQL update operations from minute-level change files for the complete OSM Planet dataset in under 7 seconds on average, while ensuring the geometries of all OSM elements on the SPARQL endpoint remain correct. The tool is open-source and available on GitHub.

Contents

List of Figures

List of Tables

Code Index

Chapter 1

Introduction

OpenStreetMap (OSM) [1] is a map database project maintained by a global community of volunteers. The OSM database contains information on roads, trails, cafés, railway stations, and many other features. One way to access and analyze this data is through knowledge graphs, which can represent OSM information in a structured form. When built using the Resource Description Framework (RDF) [2], these graphs describe data as a collection of subject-predicate-object triples.

Tools like osm2rdf [3] can convert OSM data into RDF format, making it possible to query the data efficiently with the SQL-like SPARQL query language [4]. Endpoints such as *QLever* [5] support the execution of such queries, enabling fast and expressive search capabilities, with features like autocompletion to improve usability [6].

However, converting OSM data into RDF triples and indexing the triples for efficient querying are both computationally demanding tasks. This is especially true for large datasets like the complete OSM Planet data [7], which receives thousands of updates per minute. Keeping a SPARQL endpoint synchronized with such a rapidly evolving dataset is therefore a difficult but crucial task for applications that rely on up-to-date information.

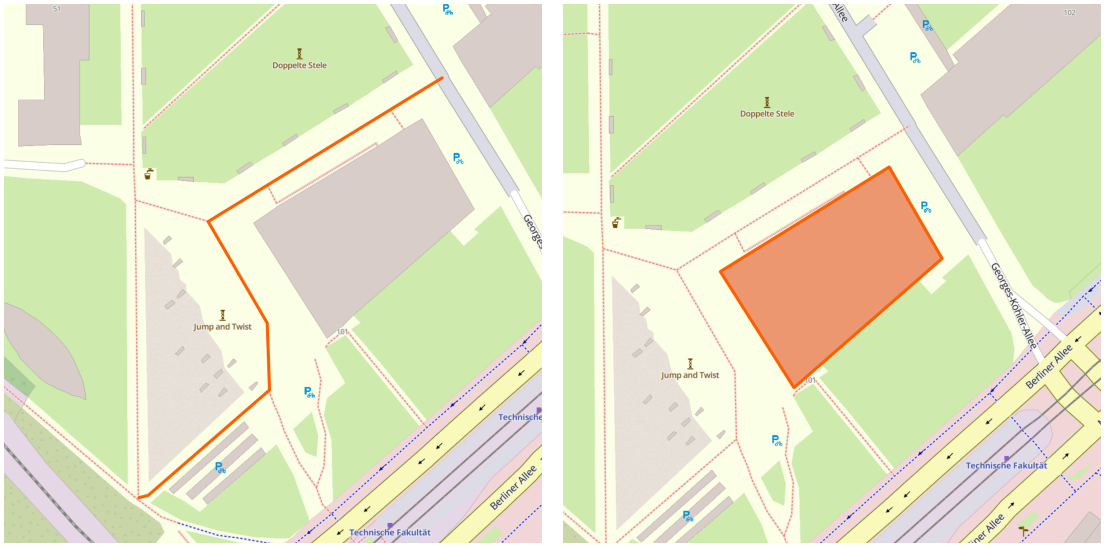
To address this problem, this thesis introduces *osm-live-updates (olu)*, a tool that provides efficient updates for SPARQL endpoints. *olu* processes OSM change files containing the edits made to the OSM database and generates the corresponding SPARQL update operations. These updates are then applied to the endpoint, keeping it synchronized with the latest OSM data. In doing so, *olu* enables near real-time querying of OSM data on SPARQL endpoints.

1.1 OpenStreetMap

OpenStreetMap (OSM), launched in 2004, has evolved into one of the largest collaborative open-data projects, with more than 10 million registered users [8]. It offers a freely editable alternative to proprietary mapping services and is widely used in applications such as navigation, geographic information systems (GIS), and location-based services. The underlying data model is structured around three fundamental element types: nodes, ways, and relations. Each element type serves a specific purpose in representing geographic features.

Node A node represents a specific geographic location and is defined by a latitude and longitude coordinate. It is the only OSM element that directly stores geographic coordinates. Nodes may denote individual points of interest, such as a bench or a restaurant, or serve as building blocks for the geometry of the other OSM elements.

Ways A way is an ordered list of nodes that can represent a linear feature, such as a road, river, or footpath (see Fig. 1.1a). If the first and the last node are the same, the way forms a closed loop and defines a polygonal area. This can represent features such as a body of water or a building footprint (see Fig. 1.1b).

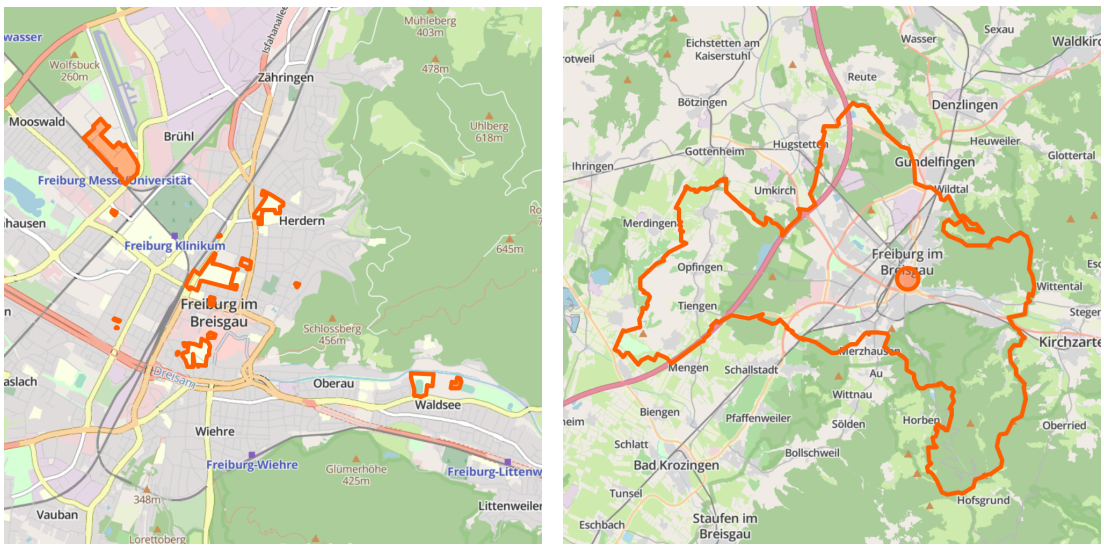


(a) Footpath (b) Building footprint

Figure 1.1: Examples that show how OSM uses ways to represent linear features, such as footpaths (a), and polygonal areas, such as building footprints (b).

* Map data copyrighted OpenStreetMap contributors and available from `https://www.openstreetmap.org`.

Relations A relation is a structured collection of nodes, ways, and/or other relations that defines the logical relationships between these elements. They can represent complex geographical features, such as multipolygons (see Fig. 1.2a), which consist of multiple polygonal elements. Relations can also be used to define areas by combining multiple ways that form closed loops. For instance, a relation could represent an administrative region by including ways that form its boundaries (see Fig. 1.2b).



(a) Multipolygon (b) Boundary

Figure 1.2: Examples illustrating how OSM relations can represent complex (multi)polygonal features. (a) A collection of polygons representing areas that are part of the University of Freiburg, and (b) a collection of ways forming the boundary of the city of Freiburg. Relations that define administrative areas often include a node marking the center of the area, shown here as a circle in the middle of area.

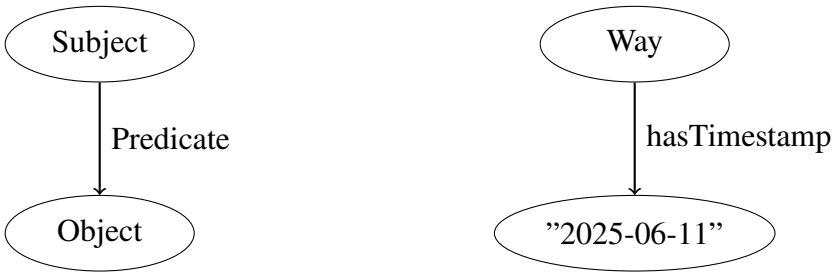
* Map data copyrighted OpenStreetMap contributors and available from `https://www.openstreetmap.org`.

Each OSM element can be annotated with tags, which are key-value pairs provided by users to describe the element's properties. Examples include road type (*highway=footway*), building function (*building=hospital*), or surface material (*surface=asphalt*). Tags enrich OSM data with semantic meaning, making them essential for interpreting and analyzing the raw geometries. In addition to these descriptive tags, each element is annotated with attributes such as a unique identifier (ID), a version number that increments with each modification, and the timestamp of the most recent change. The attributes also include metadata about the user who created or modified the element.

The OSM database currently contains around 10 billion nodes, 1 billion ways, and 100 million relations [8]. The complete planet dataset is roughly 82 GB in size when compressed, and is provided by the OSM Foundation in weekly intervals [7]. The OSM Foundation is a not-for-profit organization that supports the OSM project and is responsible for the maintenance of the central database. If working with the full planet dataset is unnecessary, it is possible to use several tools that allow for extracting subsets of OSM data. One such tool is the command-line tool *osmium* [9]. Furthermore, some providers that offer ready-to-use extracts of OSM data at different scales, ranging from entire continents down to individual cities, including *Geofabrik* [10] and *BBBike* [11].

1.2 Resource Description Framework

The Resource Description Framework (RDF) is a model for representing data in the form of directed graphs [2]. In the RDF model, information is expressed as *triples*, with each triple consisting of a subject, a predicate, and an object. The subject and object denote resources, while the predicate specifies the relationship between them. Despite its simple structure, RDF is highly expressive. Individual triples can be combined into RDF graphs that capture complex data structures. Figure 1.3 shows the graph representation of a generic RDF triple alongside an example showing how OSM data can be represented in this format.



(a) Generic RDF triple (b) RDF triple representing OSM data

Figure 1.3: (a) Graph representation of an RDF triple and (b) an example showing how it can represent OSM data, here storing the timestamp of an OSM way.

Each component of an RDF triple can be either represented as an Internationalized Resource Identifier (IRI), a literal, or a blank node. An IRI is a globally unique identifier used to reference a resource. Literals denote concrete values, such as strings or numbers, and can only appear in the object position of a triple. Blank nodes refer to anonymous resources that function as intermediate structures for grouping related

triples. An example of how a blank node can be used to model the member relationship of an OSM way and node can be seen in Appendix A.

Terse RDF Triple Language RDF is an abstract data model that can be serialized in various formats. The Terse RDF Triple Language (Turtle) is a frequently employed, human-readable serialization for RDF data, which utilizes a compact syntax for representing RDF triples [12]. In particular, Turtle employs the use of prefixes and abbreviated identifiers, which makes RDF easier to read and understand compared to more verbose formats. A concrete example of an RDF triple in Turtle format that stores the timestamp of an OSM way is provided in Listing 1.1. The prefix declarations at the top allow us to abbreviate the IRIs for the OSM way and the timestamp predicate, resulting in a more readable representation of the triple.

Listing 1.1: Example of an RDF triple encoding the timestamp of an OSM way in Turtle format. Prefix declarations are used to make the triple more compact and readable.

```
1 PREFIX osmway: <https://www.openstreetmap.org/way/>
2 PREFIX osmmeta: <https://www.openstreetmap.org/meta/>
3
4 osmway:1347584463 osmmeta:timestamp "2025-06-11T14:44:46Z"
```

1.3 Convert OSM Data to RDF triples

To store OSM data within an RDF graph, it is necessary first to convert it into RDF triples. This process involves generating triples that encode the data stored for each OSM node, way, and relation. The *osm2rdf* [3] tool performs the transformation of OSM data into RDF triples while preserving all semantic information and the geometries of the elements. Each OSM element is identified using one of the prefixes *osmnode:*, *osmway:*, or *osmrel:*, followed by its unique ID, e.g., *osmnode:1*. For each element, *osm2rdf* generates RDF triples that capture its attributes (e.g., timestamp, version), key-value pairs of associated tags, geometries, and member relationships. The geometries of OSM elements are encoded in the Well-Known Text (WKT) standard, which can representing geometric objects in a text-based form [13]. Listing 1.2 shows examples of how geometries of different OSM elements are stored in WKT format. Node geometries are represented as points, while ways are encoded as line strings or polygons when describing areas. Depending on their members, relations may form polygons or

multipolygons. If they do not define a closed area, they are represented as geometry collections containing the geometries of their member elements.

Listing 1.2: Example showing how the geometries of different OSM elements are stored in WKT format. The notation `geo:hasGeography/geo:asWKT` serves as a shortcut, facilitating the combination of two triples such that the object of the first triple becomes the subject of the second one.

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX osmnode: <https://www.openstreetmap.org/node/>
PREFIX osmway: <https://www.openstreetmap.org/way/>
PREFIX osmrel: <https://www.openstreetmap.org/relation/>

osmnode:123 geo:hasGeographie/geo:asWKT "POINT(7.84798 48.01354)"
osmway:456 geo:hasGeographie/geo:asWKT "LINESTRING (7.8534 ↩
   47.99488, 7.51577 47.99167, 7.85161 47.99096)"
osmrel:789 geo:hasGeographie/geo:asWKT "POLYGON ((7.84597 ↩
   47.9736, 7.84813 47.92621, 7.83465 47.99867, 7.84597 47.9973))"
```

osm2rdf takes an OSM dataset as input and outputs a Turtle file (`.ttl`) containing all prefix declarations and generated RDF triples in Turtle format. As OSM ways and relations do not contain geometric information themselves, all members of a way or relation must be included in the input file for *osm2rdf* to compute their corresponding geometries. Converting OSM data into RDF format is computationally expensive, particularly for large datasets. For the complete OSM planet data, the conversion process takes approximately 17 hours, generating around 135 billion triples.

1.4 Querying OSM Data in RDF graphs

The SPARQL Protocol and RDF Query Language (SPARQL) is a powerful tool for querying data in RDF graphs [4]. It enables the formulation of expressive queries over the data, including filtering, aggregation, and pattern matching operations. Listing 1.3 provides an example of a SPARQL query that retrieves the IDs of all OSM ways that describe footpaths and have been added or modified after 2024. In OSM, the tag *highway* is conventionally used to specify the road type for an OSM way. The timestamp of the ways can then be used to filter the elements for the desired dates.

Listing 1.3: SPARQL query to retrieve the IDs of all OSM ways that describe footpaths and have been created or modified after 2024. The function `YEAR` extracts the year from the timestamp as an integer, while the function `FILTER` filters the results based on the given condition.

```
PREFIX osmmeta: <https://www.openstreetmap.org/meta/>
PREFIX osmkey: <https://www.openstreetmap.org/wiki/Key:>

SELECT ?osm_id WHERE {
  ?osm_id osmkey:highway "footway" .
  ?osm_id osmmeta:timestamp ?timestamp .
  FILTER (YEAR(?timestamp) > 2024)
}
```

SPARQL Engines SPARQL engines, such as *QLever* [5], enable users to efficiently query large RDF graphs. To achieve a high query performance, it is essential to build specialized indexes, which are data structures that organize triples to allow for fast lookups. The construction of these indexes requires substantial computational resources, as they can involve parsing and processing billions of triples. This process is especially time-consuming and resource-intensive for large datasets, such as the complete OSM planet dataset, for which it takes approximately 30 hours. Table 1.1 provides an overview of the time required to convert various OSM datasets into RDF format using *osm2rdf* and index them with *QLever*.

Table 1.1: Statistics on compressed file sizes, the number of generated triples, and the conversion and indexing times for various OSM datasets. The OSM data was converted into RDF triples using *osm2rdf*. The indexing of the resulting triples was performed with *QLever*. All measurements were conducted on a machine with 16 CPU cores and 125 GB of RAM. The OSM data extracts were downloaded from *Geofabrik* [10].

OSM Dataset	File Size	# Triples	Conversion	Indexing
Planet	84.5 GB	134.8 B	17.4 h	30.8 h
Europe	31.2 GB	37.7 B	12.2 h	8.7 h
Germany	4.3 GB	6.3 B	59.4 min	61.9 min
Baden-Württemberg	0.6 GB	0.6 B	5.6 min	8.1 min
District of Freiburg	0.1 GB	0.2 B	1.3 min	1.9 min

The significant time requirements for conversion and indexing pose a challenge for applications requiring access to up-to-date OSM data, such as navigation systems, urban planning tools, or disaster response applications. Therefore, it is important to develop efficient methods for updating SPARQL endpoints with the latest OSM data without the need for a complete reprocessing of the entire dataset.

1.5 Updating OSM Data in RDF graphs

The OSM dataset is highly dynamic, with over 3 million elements created, modified, or deleted daily by the OSM community. Table 1.2 shows the number of changed OSM elements (nodes, ways, and relations) recorded per minute, hour, and day for various OSM extracts. The dynamic nature of the OSM data highlights the importance of keeping SPARQL endpoints synchronized with the latest changes to ensure accurate and up-to-date query results. As discussed in the previous section, setting up a SPARQL endpoint is computationally demanding. Therefore, it is essential to find an efficient way of updating the OSM data within an RDF graph. OSM change files provide a way to track modifications in the database. SPARQL update operations allow these modifications to be applied to existing endpoints, avoiding the need to reprocess the entire dataset from scratch. The following sections provide an overview of OSM change files and SPARQL update operations.

Table 1.2: Average number of elements that were changed by OSM contributors per minute, hour, and day across different OSM datasets. Since change file sizes vary with the activity of the OSM community, we calculated a mean value over seven days. The OSM planet change files were downloaded from the OSM replication server [7]. The data for the OSM datasets was obtained by creating regional extracts using the *osmium* tool and boundaries provided by *Geofabrik* [10].

		Number of Changed Elements		
OSM Dataset	**# Triples**	per Minute	per Hour	per Day
Planet	134.8 B	2287	0.2 M	3.2 M
Europe	37.7 B	775	43.9 K	1.0 M
Germany	6.3 B	70	4.1 K	0.1 M
Baden-Württemberg	0.6 B	10	0.6 K	14.9 K
District of Freiburg	0.2 B	2	0.1 K	2.1 K

1.5.1 OSM Change Files

OSM change files (`.osc`) are XML documents that record the differences between two states of the OSM datasets [14]. Each difference is categorized as the creation, modification, or deletion of an element. An element that exists in the later state but not the earlier state is enclosed in a *create* tag. An element that exists in the earlier state but not the later state is enclosed in a *delete* tag. An element that has changed between the two states is listed inside a *modify* tag only in its latest state. It is important to note that the change file, therefore, does not indicate what change has been made to a modified element. Listing 1.4 provides an example illustrating how a created node, a modified way, and a deleted node are represented. Within the change file, elements are typically sorted first by their type (node, way, or relation) and then by their ID. Every element in a change file is fully described, including all attributes and tags. However, as the geometry of OSM ways and relations depends on their members, it is not necessarily possible to derive their geometries from the change file alone, as the member elements are only included if they themselves were created, modified, or deleted.

Listing 1.4: Example of an OSM change file that describes the changes between two sets of OSM data. In this example, the changes include creating a node, modifying a way, and deleting a node.

```
<osmChange version="0.6">
  <create>
    <node id="2110601103" lat="48.0126985" lon="7.8347244"/>
  </create>
  <modify>
    <way id="1347584463">
      <nd ref="12465374441"/>
      <nd ref="12465374443"/>
      <nd ref="12465374440"/>
      <tag k="highway" v="footway"/>
    </way>
  </modify>
  <delete>
    <node id="1822236037" lat="48.0136829" lon="7.8344712">
      <tag k="natural" v="tree"/>
    </node>
  </delete>
</osmChange>
```

The OSM Foundation operates a replication server that provides access to change files for the entire OSM planet dataset [15]. These files are available at minutly, hourly, and daily intervals and describe changes made to the OSM database during these time periods. Each change file is accompanied by a state file containing the file's creation timestamp and a sequence number that increases with each new file. Regional change files are also available from services such as *Geofabrik* [10], although these are provided only at daily intervals. Additionally, there is a mirror[1] of the OSM planet replication server that provides minutely change files for specific regions for a limited time frame.

1.5.2 SPARQL Update Operations

SPARQL update operations provide a way to modify RDF graphs by inserting or deleting triples. They allow us to reflect the changes made to the OSM dataset directly in the RDF graph. For example, new triples can be added with an `INSERT` operation, while outdated triples can be removed with a `DELETE` operation. Listing 1.5 shows how to insert the triples that describe a newly created node into the RDF graph. Conversely, Listing 1.6 shows how to remove all triples associated with a specific OSM element, modeling its deletion. Modifications made to existing OSM elements can be expressed as a combination of both operations. First, a `DELETE` operation to remove the old triples, followed by an `INSERT` operation to add the new ones.

Listing 1.5: SPARQL update operation that inserts some triples belonging to a new OSM node into an existing RDF graph.

```
PREFIX osmmeta: <https://www.openstreetmap.org/meta/>
PREFIX osmnode: <https://www.openstreetmap.org/node/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX osm: <https://www.openstreetmap.org/>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX osm2rdfgeom: <https://osm2rdf.cs.uni-freiburg.de/rdf/geom#>

INSERT DATA {
  osmnode:1822236 rdf:type osm:node .
  osmnode:1822236 osmmeta:timestamp "2024-08-23T06:57:04" .
  osmnode:1822236 geo:hasGeometry osm2rdfgeom:osmnode_1822236 .
  osm2rdfgeom:osmnode_1822236 geo:asWKT "POINT(7.8344712 ↩
      48.0136829)" .
}
```

[1] https://download.openstreetmap.fr/replication

Listing 1.6: SPARQL update operation that deletes all triples belonging to a specific OSM way from an RDF graph. This includes triples describing member relationships linked to the OSM way via a blank node.

```
PREFIX osmway: <https://www.openstreetmap.org/way/>

DELETE WHERE {
  osmway:1347584463 ?predicate ?object .
  ?object ?memberPredicate ?memberObject .
}
```

1.6 Problem Statement and Contribution

In the previous sections, we demonstrated how OpenStreetMap (OSM) data can be converted into RDF triples and queried through SPARQL endpoints. While this enables expressive querying, the process of setting up a SPARQL endpoint is computationally demanding and time-consuming. For dynamic datasets, such as the OSM planet data, which receives millions of edits each day, the time required for this setup presents a significant challenge. Many applications require access to up-to-date data, yet rebuilding a SPARQL endpoint from scratch to update it is impractical. This problem underscores the need for a tool that can continuously synchronize a SPARQL endpoint with the latest OSM data.

The OSM Foundation publishes OSM change files at minute, hourly, and daily intervals for the complete planet data. These change files reflect the modifications made to the OSM dataset by the OSM contributors during these time periods. In principle, these change files can be used to update a SPARQL endpoint via SPARQL update operations. However, no existing tool supports this process while fully preserving the geometry of the updated OSM elements. Addressing this gap is the central contribution of this thesis.

We introduce *osm-live-updates* (*olu*), a tool that generates SPARQL update operations from OSM change files and synchronizes an existing SPARQL endpoint with the changes made to the OSM database. When paired with a high-performance SPARQL engine, such as *QLever*, *olu* can generate SPARQL update operations from minute-interval change files for the complete OSM Planet data in under 7 seconds on average.

These update operations can then be applied to a SPARQL endpoint in under 50 seconds on average, enabling near real-time querying of the complete OSM Planet data on a SPARQL endpoint.

Structure of the Thesis Following this introduction, we review existing approaches for working with up-to-date OSM data in different storage formats in Chapter 2. We then present the base implementation of *olu* in Chapter 3, explaining how SPARQL update operations can be generated from OSM change files. In Chapter 4, we provide a detailed description of *olu*'s key functionalities and outline the optimizations introduced to enhance performance. We will then evaluate the correctness and performance of *olu* in Chapter 5. In Chapter 6, we discuss current limitations and potential directions for further development. Finally, in Chapter 7, we conclude the thesis.

Chapter 2

Related Work

Depending on the user's needs and resources, several approaches are available for working with up-to-date OpenStreetMap (OSM) data. The method requiring the least complex setup is querying the *Overpass API* [16], a search engine for OSM data, which typically reflects changes to the OSM database within a few minutes. This can be done via front-end services such as *Overpass Turbo*[1]. However, the public Overpass API has limitations, including rate limits, performance issues, and the requirement to learn the Overpass Query Language to construct queries. Users or applications that require greater control and flexibility typically operate their own OSM data infrastructure. Most of these approaches rely on OSM change files (introduced in Section 1.5.1) to keep datasets up to date. This chapter reviews existing methods for applying OSM change files to different data representations, ranging from raw OSM files to relational databases and knowledge graphs.

2.1 Updating OSM Data Files

OSM data can be stored in multiple file formats. For smaller datasets, the human-readable, XML-based `.osm` format is often used. In this format, the XML file is simply a list of all OSM elements (nodes, ways, and relations), together with their attributes and tags. For larger datasets, it is common to either compress the XML files using standard algorithms (e.g., gzip or bzip2), or to use the more efficient binary PBF format (`.pbf`). This format significantly reduces storage size and read/write times [17]. The OSM planet dataset, for example, is available for download in both compressed XML and PBF formats [7].

[1] `https://overpass-turbo.eu/`

Updating OSM data files involves sequentially processing each OSM element in the file and, if the element is listed in the OSM change file, executing the specified operation. This includes inserting all elements enclosed in *create* tags, replacing elements in *modify* tags with their updated versions, and removing the elements enclosed in *delete* tags. Several tools support this, including the widely used command-line utility *osmium* [9], which provides the `apply-changes` command for applying OSM change files to OSM data files. However, this process can be inefficient because it requires scanning the entire dataset. This is especially problematic when only minor updates are needed for large datasets. Furthermore, file-based tools are good for data management and exchange but lack the advanced querying capabilities provided by databases.

2.2 Updating OSM Data in Databases

To enable efficient querying and analysis, OpenStreetMap (OSM) data is often imported into databases. That way, they provide optimized storage and indexing mechanisms. However, this approach introduces additional complexity because the raw OSM data must first be prepared for the chosen database system. Maintaining an up-to-date database by applying OSM change files adds another layer of complexity, as the updates must be transformed into a format the database can process. This section reviews existing methods for updating OSM data in relational databases and RDF graphs.

Relational Databases The most widely used relational database for OSM data is *PostgreSQL* [18]. It can be paired with the *PostGIS* extension to enable complex geospatial queries directly on OSM data [19]. The *osm2pgsql* tool can be used to import OSM data into PostgreSQL databases [20]. Many services rely on it to make OSM data available for applications such as map rendering and geocoding. *osm2pgsql* also supports updating database tables using OSM change files. For this purpose, it provides a Python script that automatically synchronizes an existing database with the latest changes from the OSM planet replication server [21]. Applying updates, however, requires importing the database in a special mode called ”slim mode”. Here, *osm2pgsql* creates additional properties in tables and indexes to ensure updates are applied correctly to all OSM elements. While this enables incremental updates, it also increases the overall database size.

RDF Graphs The use of RDF graphs for OSM data is relatively new compared to the use of relational databases. As discussed in Chapter 1.3, the *osm2rdf* tool can convert

OSM data into RDF format, which can then be queried using SPARQL endpoints such as *QLever*. However, *osm2rdf* does not provide the functionality to update an existing SPARQL endpoint. When an update to the SPARQL endpoint is needed, the endpoint must be set up from scratch using a newer OSM file.

To the best of our knowledge, *Sophox* [22] is the only tool that combines OSM-to-RDF conversion, SPARQL querying, and support for incremental updates. In *Sophox*, OSM change files are translated into SPARQL update operations, which are then applied to the RDF graph. This enables querying of up-to-date OSM data. However, the tool has several important limitations. Notably:

- Nodes without tags, which make up the majority of OSM nodes, are omitted from the output.
- Tags are only included if they contain English letters, digits, or a limited set of special characters.
- Geometries of OSM elements are not preserved, as all elements are reduced to single points.
- The project is no longer actively developed, with the last commit improving the functionality occurring several years ago[2], meaning that these limitations are unlikely to be addressed in the future.

[2]See: https://github.com/Sophox/

Chapter 3

Generating SPARQL Update Operations from OSM Change Files

We previously introduced the problem of keeping SPARQL endpoints synchronized with the latest changes from the OSM database in Section 1.6. This chapter presents the basic implementation of *osm-live-updates* (*olu*). We begin with a more precise problem definition, outlining the key challenges of maintaining synchronization between the OSM planet data and SPARQL endpoints. Section 3.2 then describes the baseline, unoptimized procedure, focusing on describing how SPARQL update operations can be generated from OSM change files.

3.1 Problem Definition

The goal of this implementation is to keep a SPARQL endpoint synchronized with the latest updates to the OSM database by applying the modifications described in OSM change files. The OSM replication server publishes these change files for the complete planet dataset. Each change file specifies the operations required to update the data: elements in *create* tags must be inserted, elements in *modify* tags replaced with their updated versions, and elements in *delete* tags removed.

These instructions, however, cannot be applied directly to a SPARQL endpoint. Instead, the changes first need to be *translated* into SPARQL update operations that the endpoint can process. This translation involves converting the OSM elements in the change file into RDF triples, and then generating the appropriate `INSERT` and `DELETE` operations that capture the changes described by the change file. In practice, this process introduces several challenges:

1. The change files do not necessarily contain sufficient information to compute the geometries of OSM ways and relations, as their members are not included if they themselves are not created or modified. This additional information must be retrieved from the updated SPARQL endpoint.
2. The geometry of an OSM way or relation may change even if the element itself is not listed in the change file. Such changes occur when the geometry of one of its member elements is modified. Detecting and updating these implicit geometry changes is therefore necessary.

3.2 Procedure

To overcome the challenges mentioned before, we implemented a four-stage process to generate SPARQL update operations from OSM change files in our base implementation of *olu*. This process involves:

1. **Collect IDs of relevant OSM elements:** In the first step, we collect the IDs of all OSM elements relevant to the update process. The relevant elements include all elements in the change file, all elements on the SPARQL endpoint that require geometry updates, and all member elements that are necessary for calculating the geometries.
2. **Construct an intermediate OSM file:** In the second step, we construct an OSM file that contains all elements identified in the previous step. For elements missing from the change file (members or elements for which the geometries need to be updated), we create lightweight dummy elements that contain only the necessary information to compute the geometries.
3. **Convert the OSM file to RDF triples:** In the third step, we use *osm2rdf* to convert the intermediate OSM file from the previous step to RDF triples.
4. **Generate SPARQL updates:** In the fourth and last step, we generate the necessary SPARQL DELETE and INSERT operations to apply the changes from the OSM change file. These generated update operations are then applied to the SPARQL endpoint.

3.2.1 Collect IDs of Relevant OSM Elements

In the first step, we collect the IDs of all OSM elements that are relevant for the update process and group them in four distinct categories:

1. **Deleted elements:** Elements inside a *delete* tag in the OSM change file.
2. **Created elements:** Elements inside a *create* tag in the OSM change file.
3. **Modified elements:** Elements inside a *modify* tag in the OSM change file, as well as OSM ways and relations that have a modified element as a member.
4. **Member elements:** All members of elements in the created or modified elements category, which are not already present in the OSM change file.

The first three categories can be extracted directly from the change file, except for OSM elements on the endpoint that are *indirectly* modified. Even if it is not explicitly listed in the change file, the geometry of an OSM way or relation may change if one of its members has its geometry modified in the change file. For instance, if the location of an OSM node is modified in the change file, the geometry of any OSM way that has this node as a member will also change, since the geometry of OSM ways and relations is determined by their members. We therefore have to update the geometry triples of every OSM element that has a modified element as a member. We query the SPARQL endpoint to detect all OSM elements that undergo such an indirect modification. Listing 3.1 shows an example query that returns the IDs of all OSM ways that have a specific OSM node as a member. We execute queries like this for each modified element to retrieve the IDs of the OSM ways and relations that reference it. It is important to first query all OSM ways with changed geometry, as these ways may be a member of another OSM relation that also needs to be updated.

Listing 3.1: SPARQL query fetching all OSM ways that have a specific OSM node as member. The `GROUP BY` modifier ensures that each OSM way appears only once in the result.

```
PREFIX osmway: <https://www.openstreetmap.org/way/>
PREFIX osmnode: <https://www.openstreetmap.org/node/>

SELECT ?way_id WHERE {
  ?member osmway:member_id osmnode:13101857348 .
  ?way_id osmway:member ?member
}
GROUP BY ?way_id
```

The fourth category of member elements is collected in a second pass over the change file, where we extract the member IDs of created and modified elements if they are not already listed in the change file. Additionally, we query the endpoint for the members of OSM ways and relations that require an update to their geometry. List-

ing 3.2 shows an example query that fetches all members of an OSM relation. A query like this is executed again for each OSM way or relation that requires an update to the geometry. The resulting member IDs are added to the set of member elements.

Listing 3.2: SPARQL query fetching the IDs of all members of a specific relation.

```
PREFIX osmrel: <https://www.openstreetmap.org/relation/>

SELECT ?element_id WHERE {
  osmrel:1590189 osmrel:member ?member .
  ?member osmrel:member_id ?element_id .
}
```

After performing this step, we know the IDs of all OSM elements relevant to the generation of SPARQL update operations and have them grouped by category. We can now proceed to create an intermediate OSM file that is used to generate the relevant RDF triples.

3.2.2 Construct an intermediate OSM file

In the second step, we create an OSM data file containing all OSM elements identified in the previous step, excluding deleted elements, for which we do not need to generate any RDF triples. The file is structured in the XML-based *.osm* format, which provides a human-readable representation of OSM data. It consists of XML representations of OSM nodes, ways, and relations, sorted by type and ID. The OSM elements from the change file can be incorporated directly into the OSM data file. However, some elements are not present in the change file but must still be included. The missing elements include the indirectly modified elements, whose geometries need to be updated, and the member elements, which are necessary to compute the geometries. For these OSM elements, we create *dummy* elements containing only the information required for geometry computation. Since tags and metadata are not needed for this purpose, they are omitted from the dummy elements. In the following, we describe how we create dummy elements for OSM nodes, ways, and relations.

Nodes Nodes are the only OSM elements that directly contain geographic coordinates and therefore form the basis for defining the geometries of OSM ways and relations. To compute the geometry of a way or relation, it is necessary to know the coordinates of all its member nodes. For each member node, we create a dummy element containing

the node's ID, latitude, and longitude. The coordinates are retrieved from the SPARQL endpoint using the `geo:hasGeographie/geo:asWKT` predicates. Since node locations are stored as WKT points on the endpoint, we parse the longitude and latitude from this representation. Listing 3.2.2 shows an example of a dummy element for an OSM node.

Listing 3.3: Example of a dummy element created for an OSM node. The dummy element contains the node's ID, latitude (`lat`), and longitude (`lon`).

```
1 <node id="13101857348" lat="48.0129705" lon="7.8334866"/>
```

Ways The geometry of an OSM way is determined by the ordered list of its member nodes. To compute the ways geometry, we create a dummy element for each relevant way that contains the way's ID and the ordered list of its member nodes. The predicates `osmway:member/osmway:member_id` are used to query the IDs of the member nodes of the specific ways. Listing 3.2.2 illustrates a dummy element for an OSM way with four member nodes.

Listing 3.4: Example of a dummy element created for an OSM way, which has four members. The member node is enclosed in a `nd` tag, and the `ref` attribute contains the ID of the member node.

```
1 <way id="1347584463">
2     <nd ref="12465374433"/>
3     <nd ref="12465374436"/>
4     <nd ref="12465374437"/>
5     <nd ref="12465374438"/>
6 </way>
```

Relations The geometry of an OSM relation is defined by the ordered list of its member elements, which can be nodes, ways, or other relations. Each member also has a *role* attribute. For example, in a multipolygon relation, the role can distinguish between inner and outer boundaries. Additionally, relations have a mandatory *type* tag that determines how their geometry is computed in *osm2rdf*: Relations of type *multipolygon* generate multipolygons, relations of type *boundary* define polygons, and other relation types are represented as collections of their member geometries. For each relevant relation, we create a dummy element containing the relation's ID and type, as well as the ordered list of members with their roles. The member IDs and roles are queried from the SPARQL endpoint using the predicates `osmrel:member`,

`osmrel:member_id`, and `osmrel:member_role`, and the relation type is obtained via the predicate `osmkey:type`. Listing 3.5 shows an example dummy element for an OSM relation, representing a multipolygon with one OSM way that defines the outer boundary and another OSM way that defines an inner boundary, which could, for example, model a lake with an island.

Listing 3.5: Example of a dummy element created for an OSM relation, that describes a multipolygon. The members of the relation are enclosed in `member` tags. For each member the type (node, way or relation), ID (`ref`) and role is stored.

```
<relation id="539979">
    <member type="way" ref="4099376" role="outer"/>
    <member type="way" ref="4885940" role="inner"/>
    <tag k="type" v="multipolygon"/>
</relation>
```

The dummy nodes, ways, and relations are first written to separate XML files and then merged with the original OSM change file using the *osmium* library [23]. This approach ensures that the elements are properly ordered by type and ID. After this step is finished, the resulting OSM file contains all elements necessary to update the SPARQL endpoint. In the next step, this OSM file is converted into RDF triples.

3.2.3 Convert OSM Data to RDF Triples

In this step, the intermediate OSM file created previously is converted into RDF triples using the *osm2rdf* tool. *osm2rdf* natively supports the `.osm` format and generates triples for all included OSM elements, which can then be used to update the SPARQL endpoint. *osm2rdf* provides configurable options, such as excluding metadata, adding additional geometry triples, e.g., the centroid or envelope, or setting the precision of WKT strings. It is therefore important to maintain consistency with the triples already stored on the endpoint. To ensure consistency, we extended *osm2rdf* to include these options that were used as metadata triples[1]. Listing 3.6 shows an example of how the WKT precision option is stored as a triple on the SPARQL endpoint. The relevant *osm2rdf* options are queried from the endpoint and reused when osm2rdf is run to convert the OSM data file into RDF triples.

[1] See pull request: `https://github.com/ad-freiburg/osm2rdf/pull/116`

Listing 3.6: Example of a triple storing an *osm2rdf* option, in this case the precision of the WKT strings.

```
PREFIX osm2rdfmeta: <https://osm2rdf.cs.uni-freiburg.de/rdf/meta#>

osm2rdfmeta:option osm2rdfmeta:wkt-precision "6"
```

As discussed in the previous section, the intermediate OSM file also contains dummy elements created for geometry calculations. Triples generated from these dummy elements are irrelevant for the update process and must be filtered out. To do this, we retain only triples whose subject IDs belong to the created or modified categories identified in Section 3.2.1. While filtering, blank nodes, which lack explicit IDs, have to be treated differently. We need to keep a mapping for a blank node that belongs to a relevant OSM element and also include the triples where this blank node appears as the subject.

The filtered *osm2rdf* output now represents the latest state of the OSM elements as described in the OSM change file. These triples are then used for generating SPARQL insert operations in the next step.

3.2.4 Generating SPARQL Update Operations

In this step, we generate the SPARQL update operations [24] that are used to synchronize the SPARQL endpoint with the changes from the OSM change file. Each change action in the file corresponds to one or both of the two basic SPARQL update operations: `DELETE` and `INSERT`. Specifically, elements in *delete* tags are handled with `DELETE` operations, which remove all triples belonging to the deleted elements. For elements in *create* tags, we use `INSERT` operations to add the newly generated triples to the endpoint. Elements in *modify* tags require both operations: we first delete all triples associated with the modified element and then insert the updated triples. In summary, we delete all triples associated with deleted and modified elements, and insert new triples for created and modified elements, i.e., the triples generated in the previous step.

DELETE Operations We use SPARQL delete operations to remove all triples associated with deleted and modified elements from the endpoint. These operations must cover triples where the OSM element appears directly as the subject, and all connected triples through intermediate resources, such as blank nodes for member relationships and geometry triples. Listing 3.7 provides a concrete example of this process, showing

how all triples belonging to an OSM way can be removed. We execute the query for each deleted or modified element for which the IDs were collected in the first step.

Listing 3.7: Example of an SPARQL update operation that deletes all triples with the subject `osmway:1347584463`, as well as all triples connected via one intermediate resource, such as member relationships or geometries.

```
PREFIX osmway: <https://www.openstreetmap.org/way/>

DELETE WHERE {
  osmway:1347584463 ?predicate ?object .
  ?object ?memberPredicate ?memberObject .
}
```

INSERT Operations After the `DELETE` operations are performed, the newly generated triples for all created or modified elements are inserted. Since these triples were filtered in the previous conversion step (Section 3.2.3), they can be added directly without further checks. Listing 3.8 shows an example of an `INSERT DATA` operation that inserts some triples belonging to an OSM node at the SPARQL endpoint. We execute this operation for each generated triple.

Listing 3.8: Example of an update operation that inserts generated triples for an OSM node with the ID 1822236.

```
PREFIX osmmeta: <https://www.openstreetmap.org/meta/>
PREFIX osmnode: <https://www.openstreetmap.org/node/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX osm: <https://www.openstreetmap.org/>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX osm2rdfgeom: <https://osm2rdf.cs.uni-freiburg.de/rdf/geom#>

INSERT DATA {
  osmnode:1822236 rdf:type osm:node .
  osmnode:1822236 osmmeta:timestamp "2024-08-23T06:57:04" .
  osmnode:1822236 geo:hasGeometry osm2rdfgeom:osmnode_1822236 .
  osm2rdfgeom:osmnode_1822236 geo:asWKT "POINT(7.8344712 ↩
     48.0136829)" .
}
```

Depending on user preference, the generated SPARQL update operations can either be applied directly to the SPARQL endpoint or written to a file. When applied directly, the `DELETE` and `INSERT` operations are sent via HTTP requests. For insertions, the Graph Store Protocol [25] is used. This way the triples are included in Turtle format within the request body. If the operations are written to a file, the SPARQL `INSERT DATA` syntax is employed (as shown in Listing 3.8), allowing the file to be executed on the SPARQL endpoint at a later time.

Summary In this chapter, we presented the base implementation of *osm-live-updates* (*olu*) and detailed the four-step pipeline for generating SPARQL update operations from OSM change files. Using this procedure, a SPARQL endpoint can be kept synchronized with the latest updates from the OSM database. While the described approach ensures correctness, it is not yet optimized for performance or scalability. The next chapter builds on this foundation by introducing optimizations that improve efficiency, enabling *olu* to process larger change files and support near real-time updates. Furthermore, we will provide a technical overview and describe some key features of *olu* in more detail.

Chapter 4

Implementation Details and Optimizations

One of the key challenges of implementing *olu* is processing OSM change files quickly enough to update the SPARQL endpoint with changes from the OSM replication server in near-real time. Since replication files are released at fixed intervals (minutely, hourly, or daily), the tool must complete the update process within the corresponding time window, which is particularly demanding given the large number of changes continuously committed to the OSM database. To meet this requirement, *olu* applies a series of optimizations to the baseline implementation presented in the previous chapter. First, we will provide a technical overview of our implementation. Next, we will discuss the optimizations employed to enhance usability and performance.

4.1 Technical Overview

osm-live-updates is implemented as a command-line tool that requires two mandatory parameters: (1) the URL of the SPARQL endpoint to be updated and (2) either a directory containing OSM change files or the URL of an OSM replication server. When a replication server is used, *olu* automatically downloads the required change files to update the endpoint (see Section 4.2). Based on the user-provided or automatically downloaded change files, the tool generates SPARQL update operations. The update operations are by default directly applied to the SPARQL endpoint, but they can also be written to a file using the `output` option. A complete list of available command-line options is provided in Appendix B. *olu* is implemented in *C++* (using features from the *C++23* standard) and built with *CMake*. The source code is available under the

GPL-3.0 license on GitHub[1]. A *Dockerfile* is included for containerized builds, and a pre-built Docker image is available at `docker.io/adfreiburg/olu`.

olu is compatible with any SPARQL endpoint that supports the SPARQL 1.1 standard [4]. The tool is particularly optimized for use with *QLever*, where it can additionally report update statistics (e.g., number of inserted and deleted triples or processing time on the *QLever* endpoint). OSM data is converted into RDF triples using the *osm2rdf* tool [3], which is included as a dependency in the repository. For compatibility, a fork of *osm2rdf*[2] is used that has been adapted for integration with *olu*.

Additionally, *olu* integrates with the *qlever-control* project, which provides management scripts for *QLever* endpoints. Using a valid configuration file (*QLeverfile*), setting up a continuously updated SPARQL endpoint requires only a few commands, as illustrated in Listing 4.1.

Listing 4.1: Example of how a continuously updated *QLever* endpoint can be setup using *qlever-control*.

```
qlever get-data # downloads OSM data and converts it via osm2rdf
qlever index # builds the indexes
qlever start # starts the QLever endpoint
qlever update-osm # starts continuous updates with olu
```

Adding the option `granularity` to the `qlever update-osm` command allows users to select the desired update interval (minutely, hourly, or daily).

4.2 Determining the Start Sequence Number

olu can automatically determine which change files to download and apply to the SPARQL endpoint when an OSM replication server is specified with the `replication-server` option. This task requires determining the current state of the endpoint and mapping it to the corresponding state on the replication server. As an indicator of the endpoint's state, we use the latest timestamp of any OSM element stored on the SPARQL endpoint. The SPARQL query in Listing 4.2 retrieves this timestamp.

[1] `https://github.com/ad-freiburg/osm-live-updates`
[2] `https://github.com/nicolano/osm2rdf/tree/olu_compatibility`

Listing 4.2: SPARQL query fetching the latest timestamp of any OSM elements on the SPARQL endpoint. Since timestamps in OSM use ISO 8601 format, their lexicographical order corresponds to chronological order. The `MAX` function therefore returns the most recent timestamp.

```
PREFIX osmmeta: <https://www.openstreetmap.org/meta/>

SELECT (MAX(?timestamp) AS ?latestTimestamp) WHERE {
  ?subject osmmeta:timestamp ?timestamp .
}
```

Each change file on an OSM replication server is uniquely identified by a sequence number that increases incrementally with each replication interval[3]. Alongside every change file, the server provides a state file that links the sequence number to a timestamp marking the file's creation time. To determine the correct starting point for updating the SPARQL endpoint, we compare the latest timestamp stored on the endpoint with the timestamps recorded in the state files. The first state file whose timestamp is greater than or equal to the endpoint's latest timestamp yields the correct sequence number. This sequence number then serves as the starting point for downloading all subsequent change files up to the most recent one.

Performing this lookup each time *olu* is executed can be inefficient. The SPARQL query used to obtain the latest timestamp is relatively costly, taking around one minute for the complete OSM planet dataset on a *QLever* endpoint. Downloading and scanning the state files introduces additional overhead. This becomes particularly problematic if the update interval is short (for example, for minutely updates) and the SPARQL endpoint has not been updated for an extended period. Determining the starting sequence number for an endpoint that is one week out of sync with a minutely replication server would, for example, require checking 10,080 state files. To address this issue, we implemented two optimizations that significantly improve the efficiency of determining the starting sequence number.

Estimating the Start Sequence Number When using the official OSM planet replication server, change files are provided at minute, hourly, or daily intervals. This allows us to estimate the starting sequence number by comparing the latest timestamp on the

[3]For example, the minutely change file with the sequence number 6755760 can be downloaded from `https://planet.openstreetmap.org/replication/minute/006/755/760.osc.gz`.

SPARQL endpoint with the current timestamp of the replication server. The elapsed time (in minutes, hours, or days, depending on the chosen granularity) corresponds to the number of change files that have been created since the SPARQL endpoint was created or last updated. Subtracting this number from the latest sequence number on the replication server yields an initial estimate. For example, suppose the endpoint's latest timestamp is `2025-09-01T12:00:00Z` and the replication server's current timestamp is `2025-09-08T16:00:00Z`. In that case, 7 days have elapsed between the creation of the endpoint and the current timestamp. When using a daily update interval and the latest sequence number on the replication server is 4503, we can estimate the starting sequence number as 4503 - 7 = 4496. Using this method, we only need to download the state file of the most recent change file, the state file of the estimated sequence number, and, to avoid rounding errors, one state file immediately before and after the estimate.

Read Start Sequence Number from Endpoint For SPARQL endpoints that have already been updated, *olu* can skip the sequence number estimation by storing metadata about previous updates directly on the endpoint. Specifically, it records the latest applied sequence number using the predicate `osm2rdfmeta:updatesCompleteUntil`. Since sequence numbers differ between replication servers (e.g., daily and minute-interval files use separate numbering), *olu* stores the URL of the used replication server with the predicate `osm2rdfmeta:replicationServer`. After each update, *olu* also adds a timestamp triple with the predicate `osm2rdfmeta:dateModified`, which can be helpful in database management tasks. Listing 4.3 shows an example of these three metadata triples.

Listing 4.3: Example of the three metadata triples that are inserted after a successful update. In this case, the last change file has a sequence number of 4503 and was downloaded from a replication server provided by *Geofabrik*.

```
1 PREFIX osm2rdfmeta: <https://osm2rdf.cs.uni-freiburg.de/rdf/meta#>
2
3 osm2rdfmeta:info osm2rdfmeta:updatesCompleteUntil (Sequence ↩
    number: 4,503, Timestamp: 2025-08-11T20:21:31Z)
4 osm2rdfmeta:info osm2rdfmeta:replicationServer ↩
    https://download.geofabrik.de/europe/germany/bremen-updates/
5 osm2rdfmeta:info osm2rdfmeta:dateModified 2025-08-12T07:59:22
```

When *olu* is executed again, it retrieves these metadata triples from the SPARQL endpoint. It then checks whether the stored replication server URL matches the one

currently in use. If that is the case, *olu* can use the sequence number stored in the `osm2rdfmeta:updatesCompleteUntil` triple as the starting point for the update process, avoiding state file lookups on the replication server. This optimization substantially reduces startup overhead, particularly for endpoints that are updated in minute-intervals.

4.3 Merging OSM Change Files

When updating a SPARQL endpoint, it is often necessary to process multiple OSM change files simultaneously. Rather than applying each file separately, it is generally more efficient to merge them into a single, combined file. Merging change files reduces the overall number of changed OSM elements, since these elements can undergo multiple updates within a given time span. Processing only the most recent state avoids applying intermediate updates that would be overwritten anyway. Similarly, OSM elements created or modified in an earlier change file may be deleted in a later one, rendering the intermediate updates unnecessary. Depending on the time intervals and OSM data extracts used, this can reduce the overall number of changed elements that need to be processed by up to 10 %. An additional benefit of merging change files and processing them together is that it reduces the overall number of member elements that are fetched from the SPARQL endpoint. When multiple elements across different change files share the same OSM nodes, ways, or relations as members, merging the change files avoids repeated lookups that would have been necessary if the change files were processed separately.

We use the *osmium* library to merge the change files. Each file is loaded into an `osmium::memory::Buffer`, and lightweight pointers to the objects are stored in an `osmium::ObjectPointerCollection`. This has the advantage that only the pointers need to be sorted and copied, while the actual objects remain in their respective buffers. The pointers to the objects are sorted by type (node, way, or relation) and ID. To ensure that only the most recent version of each element is retained, we also sort by the *version* attribute, which increments with each modification in the OSM database. Deleted objects are treated as having the highest version so that they appear first in the sorted list. After sorting, we use the *C++* function `std::unique_copy` to extract only the first occurrence of each OSM element, which represents its most recent version, and write them to a combined change file.

Merging change files in this manner is efficient because it involves sorting and copying only pointers rather than full objects. This approach allows large numbers of change files, such as the daily change files for the full OSM planet dataset, which can exceed 100 MB, to be merged with moderate resource usage.

Besides the performance benefits of merging change files, sorting the change files is also an important validation step to ensure that each OSM element appears only once in the resulting change file. OSM elements can appear multiple times in a change file when they are updated multiple times within the same time interval. This happens, for example, regularly in the minute-interval change files from the OSM planet replication server. It also ensures that the OSM elements in the change files are sorted in the correct order (by type and ID), which is a requirement for the subsequent processing steps in *olu*. The sorting step is therefore always performed, even when only a single change file is processed.

4.4 Optimizing Updates for OSM Extracts

When working with OSM data limited to a specific geographic area, it is often advantageous to set up a SPARQL endpoint using an extract of the complete OSM planet data, as this reduces system resource requirements. In such cases, change files from the OSM planet replication server may contain information that is irrelevant for the update process. One alternative is to use a replication server that provides change files specific to the extract. However, these servers often come with limitations. For example, *Geofabrik* provides replication servers for country extracts, but only updates them at daily intervals. Another option is a mirror of the OSM planet server[4], which provides minute-level updates but retains the files only for a short period. To our knowledge, no public replication server currently provides change files with full detail across all update intervals in the same way as the OSM planet replication server. Additionally, some OSM extracts may not align with administrative boundaries and are therefore not covered by any public replication server.

In such cases, OSM planet change files must be filtered before applying them to extract-based SPARQL endpoints, because they can include information that is irrelevant for updating the elements inside the extract's boundaries. For this task, we use

[4] `http://download.openstreetmap.fr/replication/`

the *osmium* command-line tool [9], which provides the `extract` command for extracting elements within a bounding box or polygon, provided in either polygon (`.poly`) or GeoJSON (`.geojson`) format. The tool supports different extraction strategies, which determine how elements that intersect the boundary are handled. For example, one could choose to include all members of ways and relations that cross the boundary of the extract, even if the members themselves are not within the boundary. These strategies vary in terms of memory requirements and the number of times the change file must be read. For details, see the *osmium* documentation[5]. To maintain consistency with the initial OSM import, *olu* provides the `extract-strategy` option to specify the same strategy when filtering change files. The used bounding box or polygon can be provided to *olu* using the `bbox` or *polygon* options, respectively.

Depending on the number and size of change files, as well as the size of the OSM extract, the most efficient order for merging, sorting, and creating the extracts can vary. When working with a small number of change files, it is, in most cases, faster to first extract the relevant elements from each change file and then merge and sort the resulting smaller files. However, as the *osmium* extract command does not work for some extract strategies if an OSM element appears multiple times in the change file, we always perform the merging and sorting step first to ensure that change files can always be processed correctly.

4.5 Minimizing Updates for Modified OSM Elements

Updating a SPARQL endpoint with OSM change files often requires the insertion and deletion of millions of RDF triples at the SPARQL endpoint, which can represent a significant fraction of the total update time. Although the baseline approach described in Chapter 3 guarantees correctness, it does not optimize the number of triples that need to be updated. While we can not reduce the number of triples that need to be updated for newly created or deleted OSM elements, we can reduce the number of updated triples for modified elements. In the baseline approach, modified OSM elements are handled by first deleting all triples associated with the element and then inserting the newly generated triples. While this method is straightforward, it is suboptimal because a considerable number of triples remain unchanged yet are redundantly deleted and reinserted. To reduce this inefficiency, we investigate optimization strategies designed

[5] `https://osmcode.org/osmium-tool/manual.html`

to minimize the number of triples that need to be updated at the SPARQL endpoint.

4.5.1 Minimize Updates for Changing Geometries

The geometry of an OSM element must be updated on the SPARQL endpoint when it has another OSM element that has been modified in the change file as a member. In the baseline implementation of *olu*, all triples of such referencing elements are deleted and reinserted, even though only the geometry triples need to be updated. For instance, if the geometry of an OSM node that is a member of an OSM way changes, the OSM way's attributes, member lists, and tags remain unaffected. Therefore, instead of deleting and reinserting all triples that belong to the way, we can update only the triples describing its geometry.

To implement this, *olu* keeps a dedicated list of ways and relations whose geometries require updating. A targeted delete operation removes only geometry-related triples (such as `geo:asWKT` or `osm2rdf:hasCompleteGeometry`) for these OSM elements. The number and type of the geometry triples depend on the *osm2rdf* options used during the initial data dump. *olu* retrieves these options directly from the endpoint to generate the corresponding delete operations. When inserting the updated triples, our filter step described in Section 3.2.3 includes only the relevant geometry triples for the OSM elements that are updated on the endpoint.

Since geometry triples make up only a minor portion of the triples that are generated by *osm2rdf* for OSM ways and relations, and the number of OSM elements that need to be updated can be high, this approach significantly cuts down the number of triples that need to be updated on the SPARQL endpoint and achieves better performance at no additional cost.

4.5.2 Minimizing Updates by Checking Node Location Changes

When a node appears in a *modify* tag in the change file, it is only necessary to update the geometries of referencing OSM elements if the node's location has actually changed. For example, the geometries of OSM ways or relations do not change if a member node is modified by adding a tag to it. Updating geometry triples in these cases would be unnecessary. To implement this optimization, we retrieve the coordinates of all modified nodes from the SPARQL endpoint. Then, we can compare them with the coordinates

provided in the change file. If the coordinates are identical, we can skip updating the geometry triples of the referencing elements.

Although nodes with tag-only modifications constitute a small fraction of all OSM nodes (approximately 1 % of modified nodes) in change files, this optimization can still yield performance benefits. Modified nodes account for roughly 25 % of all elements in change files, and comparing coordinates is computationally cheap. In practice, this optimization can reduce the total update time. However, the actual performance gain depends on the specific change file and SPARQL endpoint in use, and it is relatively small compared to the total update time.

4.5.3 Evaluating Member-Change Checks for Ways and Relations

Since the geometry of an OSM way or relation is defined by its member list, updating the geometry of referencing elements is only necessary when the member list of the modified element changes. Similar to the node location check discussed previously, we explored minimizing the number of triples that need to be updated by verifying whether the member lists of OSM ways was modified.

To implement this check, we retrieve the member lists of modified elements from the SPARQL endpoint and compare them with the member lists in the change file. For ways, we fetch the member elements using the predicates `osmway:member_id` and `osmway:member_pos`. For relations, we also have to include the member roles via `osmrel:member_id`, `osmrel:member_pos`, and `osmrel:member_role`, as they also have an influence on how the geometry is calculated. The retrieved member IDs are stored in arrays sorted by position and compared against the corresponding arrays from the change file. If the members are identical, updating the geometry and member triples for all referencing elements can be skipped.

In practice, this approach, however, did not improve overall performance. Although it is relatively common for OSM ways and relations to be modified without changing their members (approximately 50 % of ways and 25 % of relations), these elements together account for only about 5 % of all elements in a change file. Moreover, unlike the node location check, comparing member lists is computationally expensive, requiring fetching, parsing, and sorting large numbers of triples. The resulting reduction in inserted triples was minimal (around 2 % at max), and in some cases, the extra

computation even degraded performance. For these reasons, this optimization was not included in the final implementation of *olu*.

4.6 String Parsing

Parsing large strings, such as OSM change files and SPARQL endpoint responses, is a key component of our implementation of *olu*. To maximize performance, we employed specialized libraries and optimized low-level string operations. This includes using the *osmium* library [23] for parsing OSM change files, the *simdjson* library [26] for parsing SPARQL JSON responses, and custom routines for high-performance string handling.

Parsing OSM Change Files The daily change files from the OSM planet replication server can reach 100 MB in size. Processing several files at once requires working with hundreds of megabytes of data. When first implementing *olu*, we started using standard *C++* XML libraries that are not optimized for OSM data, such as *Boost.PropertyTree*[27] (which internally uses *RapidXml* [28]). This library loads the entire XML document into memory and constructs a full tree structure. This approach leads to high memory consumption and slow performance, making it unsuitable for large OSM files. To fix this issue, *olu* now uses the *osmium* library [9], which allows the OSM change files to be read in buffered chunks, avoiding full in-memory loading. Its handler interface allows custom logic to be applied to each OSM element in the change file, enabling efficient processing even on systems with limited memory.

Parsing Responses from the SPARQL Endpoint We request JSON responses from the SPARQL endpoints to retrieve the necessary information for processing the change files. The *simdjson* library[6] is used to parse these JSON responses efficiently. It is designed to handle large JSON documents quickly and with low memory overhead, making it suitable for our use case. *simdjson* uses SIMD (single instruction, multiple data) instructions to accelerate parsing, allowing it to parse gigabytes of JSON per second [26].

Optimizing Low-Level String Parsing In addition to specialized libraries, we optimized frequently executed string operations. We used `std::regex` in an initial implementation to parse element IDs, coordinates, and other data. However, regular expressions are slow and memory-intensive when applied to large datasets. We replaced them

[6] https://simdjson.org

with custom parsing routines that manipulate character pointers directly. For instance, our optimized method extracts a node ID from an IRI in 7.8 ns, compared to 616 ns with `std::regex`. Operations like this can be executed millions of times per *olu* run. By avoiding regular expressions in heavily repeated operations, we significantly improved both speed and memory efficiency.

4.7 Batching of SPARQL Queries and Operations

Sending one SPARQL query per OSM element would introduce excessive overhead. To avoid this, we use the `VALUES` syntax in SPARQL, allowing a single query to handle multiple lookups simultaneously. Listing 4.4 illustrates how a single query can retrieve the geometries of multiple OSM nodes at once. The `VALUES` clause specifies a list of values for a variable, which the query then uses to filter results.

Listing 4.4: Example for a SPARQL query fetching multiple nodes locations at once.

```
1 PREFIX geo: <http://www.opengis.net/ont/geosparql#>
2 PREFIX osm2rdfgeom: <https://osm2rdf.cs.uni-freiburg.de/rdf/geom#>
3
4 SELECT ?value ?location WHERE {
5   VALUES ?value { osm2rdfgeom:osmnode_1 osm2rdfgeom:osmnode_2 ↩
        osm2rdfgeom:osmnode_3 }
6   ?value geo:asWKT ?location .
7 }
```

Using the `VALUES` clause drastically reduces the number of queries sent to the SPARQL endpoint. However, including too many values in a single query may cause the endpoint to run out of memory or reject the request. To address this, we implemented a batching mechanism that splits large lists of values into smaller chunks, each of which is sent as a separate query. This batching strategy is also applied to SPARQL update operations. This is important because deletion operations are memory-intensive, and the HTTP requests that carry the triples that need to be inserted as a payload can reach hundreds of megabytes, too large for some endpoints to process in a single request.

In general, performance improves with fewer batches, but the optimal batch size depends on the endpoint's capabilities. To provide flexibility, *olu* includes the command-line option `batch-size`, allowing users to configure the number of values per batch or triples per update operation.

Chapter 5

Evaluation

In this chapter, we evaluate our implementation of *olu* with respect to two key aspects: correctness and performance. Correctness is crucial for ensuring that the SPARQL endpoint accurately reflects the current state of the OSM data. To this end, we verify that all relevant triples are inserted, updated, or deleted correctly during the update process. Our performance evaluation focuses on the efficiency of the update mechanism, specifically the time required to generate SPARQL update operations from OSM change files and the time required to process these operations at the endpoint. We use the publication intervals of OSM planet replication files (minute, hour, and day) as practical benchmarks to determine whether *olu* can keep pace with real-world update frequencies. Together, these evaluations demonstrate both the reliability and scalability of *olu* in handling continuous OSM updates to an SPARQL endpoint.

5.1 Correctness of the Update Process

An update process can be defined as correct when it accurately reflects the current state of the data after an update has been applied. In the context of updating a SPARQL endpoint with *olu*, this means that after applying a series of OSM change files, the SPARQL endpoint should contain the same information as if it had been created from scratch using the latest OSM data dump. Consequently, to verify the correctness of the update process, we compare a graph on the SPARQL endpoint after applying the change files with a graph that was created from scratch using the latest OSM data dump. If the two graphs are identical, the update process can be considered correct.

A central challenge in testing RDF graph equality is handling blank nodes. Since blank nodes do not have fixed or unique values, determining equality between two blank

nodes in different graphs necessitates establishing a mapping between them. Calculating such a mapping for blank nodes in two different graphs is known to be an NP-complete task [29]. However, this complexity can be avoided by eliminating blank nodes from the graphs. Once removed, RDF graphs can be treated as sets of triples, and we can test the equality using simple set operations. Section 5.1.1 discusses *Skolemization*, a technique for eliminating blank nodes. Section 5.1.2 explains how equality for RDF graphs without blank nodes can be verified using a SPARQL query.

5.1.1 Skolemization

osm2rdf represents member relationships in OSM ways and relations using blank nodes (see Appendix A). Although blank nodes are ideal for modeling intermediate resources, they complicate the comparison of RDF graphs, as mentioned before. To avoid the need for an algorithm to map the blank nodes, we employ a technique called *Skolemization*. Here, blank nodes are replaced with globally unique URIs. This transformation turns previously anonymous resources into regular resources, thereby eliminating the need to find a mapping between the two graphs.

For this purpose, we extended *osm2rdf* with the ability to use unique URIs in the `genid` namespace instead of blank nodes when describing member relationships[1]. The skolemized identifier is constructed from the ID of the member, the entity it belongs to, and a letter indicating the object type (`r` for relations, `w` for ways, `n` for nodes). Members can appear multiple times in the member list, for example, in closed ways that define an area where the first and last nodes are the same. To guarantee uniqueness, we also append the member position to the identifier (e.g. `p3`). Listing 5.1 shows how a blank node describing the membership of an OSM way and an OSM node is replaced with such a deterministic URI. This strategy ensures that the resulting identifiers are globally unique and are equal across different data dumps.

Listing 5.1: Example of replacing a blank node representing the member relationship of an OSM way and a node with the unique URI `genId:w5362n2473p3`.

```
1 PREFIX genid: <http://osm2rdf.cs.uni-freiburg.de/.well-known/genid/>
2
3 osmway:5362 osmway:member genId:w5362n2473p3
4 genId:w5362n2473p3 osmway:member_id osmnode:2473
5 genId:w5362n2473p3 osmway:member_pos "3"
```

[1] https://github.com/ad-freiburg/osm2rdf/pull/114

5.1.2 Proof of Equality

By applying Skolemization to remove blank nodes from the RDF graphs, each resource is assigned a deterministic and globally unique identifier. Consequently, if two triples convey the same information, they will be identical in both graphs. This property allows us to treat RDF graphs as sets of triples and verify graph equality using set theory.

Set Equality Formally, two sets A and B are equal if and only if they contain exactly the same elements

$$A = B \iff (A \subseteq B) \land (B \subseteq A)\,. \tag{5.1}$$

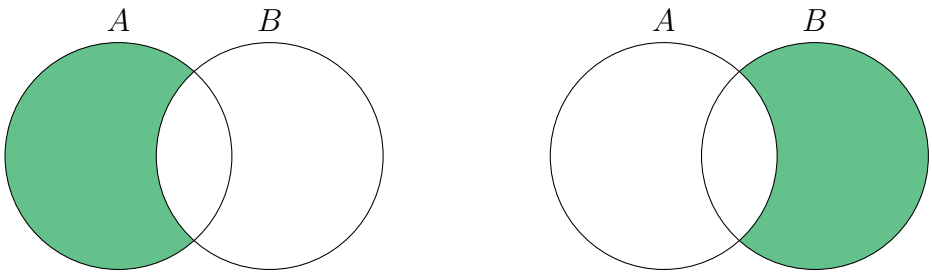
If A and B are equal, then A is a subset of B and therefore all elements of A are contained in B, e.g.

$$A \subseteq B \iff A - B = \emptyset\,. \tag{5.2}$$

Similarly, if B is a subset of A, then $B - A = \emptyset$. Thus, equality holds precisely when both set differences are empty

$$(A - B = \emptyset) \land (B - A = \emptyset) \iff A = B\,. \tag{5.3}$$

Figure 5.1 provides a visual illustration of these set differences.



(a) $A - B$ (b) $B - A$

Figure 5.1: Visualization of the set differences between two sets *A* and *B*. If both green areas are empty, i.e., $A - B = \emptyset$ and $B - A = \emptyset$, the sets are equal.

Graph Equality For our equality check, we compare two RDF graphs: G_{latest}, created from the most recent OSM data dump, and G_{updated}, obtained by applying OSM change files to a SPARQL endpoint build from an older OSM data dump. If the update process is correct, both graphs must contain the same triples. Using Equation 5.3, equality can be verified by computing

$$G_{\text{latest}} - G_{\text{updated}} = \emptyset \quad \wedge \quad G_{\text{updated}} - G_{\text{latest}} = \emptyset\,, \tag{5.4}$$

or equivalently

$$(G_{\text{latest}} - G_{\text{updated}}) \cup (G_{\text{updated}} - G_{\text{latest}}) = \emptyset\,. \tag{5.5}$$

Listing 5.2 shows how this equation can be implemented as a SPARQL query. The `MINUS` operator computes the set differences, while `UNION` combines them. In SPARQL, each graph is identified by an IRI: `<http://example.com/updated>` represents the updated graph (G_{updated}), and `<http://example.com/latest>` represents the graph created from the latest data dump (G_{latest}). An empty query result would confirm that the two graphs are identical. As this SPARQL query is memory-intensive, it can be split up. For example, triples with specific, frequently occurring predicates can be filtered out and checked separately.

Listing 5.2: SPARQL query to compute the difference between two RDF graphs. The `GRAPH` keyword ensures that triples are queried from the specified graph.

```
SELECT ?s ?p ?o WHERE {
  {
    {
      GRAPH <http://example.com/updated> { ?s ?p ?o . }
    } MINUS {
      GRAPH <http://example.com/latest> { ?s ?p ?o . }
    }
  } UNION {
    {
      GRAPH <http://example.com/latest> { ?s ?p ?o . }
    } MINUS {
      GRAPH <http://example.com/updated> { ?s ?p ?o . }
    }
  }
}
```

5.1.3 Testing the Correctness of the Update Process

As described in the previous section, to verify the correctness of the update process we set up a SPARQL endpoint containing two graphs: one graph that contains the triples from the latest OSM data dump which we call G_{latest} and another graph that contains triples from a data dump that is several days older which we call G_{updated}. The OSM data dumps were created with a custom fork[2] of *osm2rdf* with the options `no-blank-nodes` and `write-ogc-triples none`[3] enabled. We also reduce the precision of WKT points in *osm2rdf* with the option `wkt-precision` to six decimal places, instead of the seven decimal places that the OSM data can represent. This is done because *QLever* currently stores only six decimal places for node coordinates. The `wkt-precision` option affects only the output for node locations, not the precision used during geometry computation of OSM ways and relations. We must therefore take additional measures to ensure consistency between the updated and latest graphs. For this purpose, we created and used a Python script that reduces the coordinate precision to six decimal places in the initial OSM data files and the OSM change files before running *olu*. This way, *osm2rdf* always uses only six decimal places to compute the geometries of the OSM elements. We then executed *olu* to update G_{updated}, specifying its URI via the command-line option `graph`.

This experiment was repeated with multiple OSM extracts, however, the largest correctness experiment was conducted on the OSM Germany dataset, which contains around 4.8 billion triples. In this experiment, we downloaded the latest OSM Germany data dump from *Geofabrik*[4] dated Oct 23, 2025, and a one-week-old dump dated Oct 17, 2025. Before the update, the SPARQL query in Listing 5.2 returned around 5 million triples, indicating the differences between the two graphs. We then used the OSM change files published between these two dates to update the older graph with *olu*. The generated SPARQL update operations deleted around 17 million triples and inserted 19 million triples at the SPARQL endpoint. After applying all updates, the SPARQL query returned around 50 triples. In the following we list some reasons why the response was not empty:

- The *osm2rdf* metadata triple (`osm2rdf:dateDumped`) that stores the date when the dump was created is different between both OSM datadumps

[2] `https://github.com/nicolano/osm2rdf/tree/olu_compatibility`

[3] *olu* does not support the update of spatial relations like `ogc:sfContains` or `ogc:sfIntersects`. See Section 6.2 for details.

[4] `https://download.geofabrik.de/europe/germany.html`

- The *olu* metadata triple (`osm2rdf:dateModified`) that stores the date of the update is only present on the updated graph
- The timestamp of the OSM data file used to create graph G_{latest} does not necessarily coincide with that of the latest OSM change file. Therefore, it is possible that some changes are applied to the updated graph, G_{updated}, that are not present in the latest graph, G_{latest}. As these files are most commonly created at night, the number of cases in which this occurs is small, particularly when working with regional extracts such as OSM Germany data. In such cases, we verified the correctness manually.
- When working with regional extracts of OSM data created using the *osmium* tool and the extract strategy *smart*, the geometry of OSM relations that cross the boundaries can differ between the two graphs. This is because only OSM relations that are of type *multipolygon* are reference complete, meaning that all referenced OSM ways and nodes are included in the extract. In rare cases, a referenced OSM element may be present in one of the OSM change files that was used for updating the graph G_{updated}. This can occur if the reference was modified while it was part of a multipolygon at some point between updates.
- When the key of an OSM element tag contains a UTF-8 codepoint that is invalid according to the Turtle grammar, *osm2rdf* uses a blank node to model the tag and stores the key as a literal in the object of a triple. Our Skolemization does not handle such cases, so the SPARQL query returns these triples because the blank nodes differ across the graphs. However, this only occurred for two tags, so it was possible to manually verify that the tags were the same in the updated and latest graphs.

Apart from these rare edge cases, the experiment showed that the SPARQL update operations generated by *olu* correctly update a SPARQL endpoint containing large OSM datasets, such as the OSM Germany data.

5.2 Performance of the Update Process

For *olu* to be used in a production environment, it must be able to efficiently handle the high volume of updates continuously made to the OSM database. The OSM planet replication server publishes change files that capture these updates at three intervals: minute, hour, and day. These files therefore provide a realistic benchmark for *olu*'s performance. To enable continuous updates for the SPARQL endpoint, the change files must be processed within their respective publication intervals.

The results of the performance evaluation are summarized in Table 5.1. The table reports the mean time required to generate SPARQL update operations with *olu* and the mean time required to apply them to a SPARQL endpoint. Measurements were taken for various OSM datasets differing in size, characterized by their triple count N_{triple}: the district of Freiburg (FR), Baden-Württemberg (BW), Germany (DE), Europe (EU), and the complete OSM Planet data (P). Evaluations were conducted for change files corresponding to minutely (t_{min}), hourly (t_{hour}), and daily (t_{day}) updates. The number of elements in a change file and, thus, the update time can vary substantially. For this reason, we computed an average time for each dataset and publication interval. The average was computed using change files that covered one week of OSM database updates (i.e., 10,080 minutely, 168 hourly, and 7 daily change files). Due to increasing runtimes, we used one day of OSM database updates (i.e., 1,440 minutely, 24 hourly and one daily change files) for the OSM Planet dataset. Unfortunately, we were unable to report a value for the application of the generated SPARQL update operations for daily change files for the OSM Planet data due to an out-of-memory issue that occurred when applying the updates to the *QLever* endpoint.

Experimental Setup All experiments were executed on a machine equipped with an AMD Ryzen 9 7950X 16-Core Processor with 32 Threads, 125 GB of RAM, and a 4 TB NVMe SSD running Ubuntu 22.04.2 LTS. We used *QLever* as the SPARQL endpoint and *qlever-control* to set up the endpoint using the configuration from the default *Qleverfile* for the OSM planet data[5]. The official OSM Planet replication server [7] was used to pre-download the change files for each measured dataset and publication interval. For the smaller OSM datasets, generating update operations therefore requires an additional step of extracting the relevant information from the planet change files. The SPARQL endpoint was reset to its initial state before each measurement.

[5]https://github.com/ad-freiburg/qlever-control/blob/main/src/qlever/Qleverfiles/Qleverfile.osm-planet

Table 5.1: Mean time required to generate SPARQL update operations from OSM change files with *olu*, and the mean time for a *QLever* endpoint to apply these updates. Measurements were conducted for different OSM datasets characterized by triple count N_{triple} and change file intervals of one minute (t_{min}), one hour (t_{hour}), and one day (t_{day}). The change files were downloaded from the OSM Planet replication server.

		Generating Updates			**Applying Updates**		
Dataset	N_{triple}	t_{min}	t_{hour}	t_{day}	t_{min}	t_{hour}	t_{day}
P	134.8 B	6.7 s	73.0 s	21.6 min	48.5 s	180.0 s	-
EU	37.7 B	1.3 s	19.1 s	5.4 min	10.2 s	41.5 s	26.1 min
DE	4.8 B	0.3 s	5.5 s	74.9 s	0.8 s	4.5 s	92.2 s
BW	0.6 B	0.1 s	2.1 s	33.0 s	40 ms	0.5 s	9.6 s
FR	0.2 B	68 ms	1.5 s	27.9 s	7 ms	88 ms	1.4 s

Results The results demonstrate that, on average, *olu* processes all OSM change files for the evaluated OSM datasets within the publication interval of the change files. While individual change files may occasionally require more processing time due to variations in file size and the number of contained changes, this suggests that *olu* can ultimately reflect the update frequency of the OSM Planet replication server.

However, the average total update time for minute-interval change files for the complete OSM Planet data is only slightly shorter than the publication interval itself. Therefore, on systems with limited computational resources, it is advisable to use hourly or daily change files to ensure updates can be performed consistently within the required timeframe. These findings emphasize the importance of continuing to optimize the *olu* update process.

5.2.1 Scalability of the Update Process

Based on the results of our performance evaluation in Table 5.1, we plotted the mean total update time t_{update} (i.e. the time taken to generate and apply the update operations at the endpoint) for minute-interval change files against the number of triples in the OSM dataset in Figure 5.2.1. To empirically determine the computational complexity of *olu*, we fitted the data points using the following power law model

$$y(x) = a \cdot x^b + c \, . \tag{5.6}$$

While a power law exponent of $b \approx 1$ would suggest a linear runtime, an exponent of $b > 1$ would indicate a non-linear, and therefore polynomial, runtime. For comparison, we have also added a linear fit using the model

$$y(x) = a \cdot x + b \, . \tag{5.7}$$

We obtained the following fitted parameters using the power law model from Equation 5.6

$$a = 8.96 \times 10^{-10}\,\text{ms}, \quad b = 1.24, \quad c = 125\,\text{ms} \, . \tag{5.8}$$

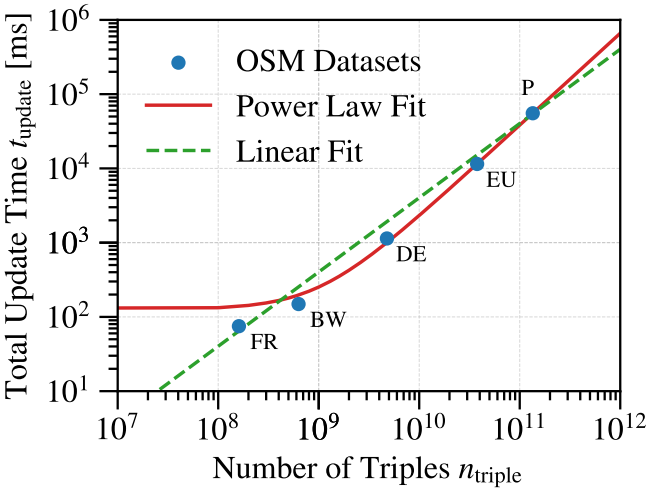


Figure 5.2: Relationship between OSM dataset size n_{triple} and the total update time t_{update} for minute-interval change files. The annotated data points represent the total update time for various OSM datasets. The red curve shows a power law fit using Equation 5.6. For comparison, we also added a linear fit using Equation 5.7 with the restriction that $b > 0$, to avoid negative values for the total update time.

The experimental results, shown in Figure 5.2.1, therefore indicate a super-linear runtime of *olu*, meaning the runtime is polynomial but smaller than $O(n^2)$. Using these parameters, we can derive a formula that can be used to estimate the total update time for minute-interval change files t_{min}, in seconds, as a function of the OSM dataset size n_{triple} (in billions of triples)

$$t_{\text{min}}(n_{\text{triple}}) \approx 0.13\,\text{s} \cdot n_{\text{triple}}^{1.24} \, . \tag{5.9}$$

By performing the same power law fitting on the results of the total update time for

hour- and day-interval change files, we can derive the following formulas to estimate the runtime of *olu* for hour- and day-interval change files as a function of the OSM dataset size n_{triple} (in billions of triples)

$$t_{\text{hour}}(n_{\text{triple}}) \approx 0.57\,\text{s} \cdot n_{\text{triple}}^{1.27} \tag{5.10}$$

$$t_{\text{day}}(n_{\text{triple}}) \approx 19.2\,\text{s} \cdot n_{\text{triple}}^{1.26} \tag{5.11}$$

While these formulas should only be used to provide rough estimates of runtime, particularly for smaller OSM datasets, the statement that *olu* has a super-linear runtime also applies to larger update intervals and, consequently, larger runtimes.

To understand the reason for *olu*'s super-linear runtime, we can examine Figure 5.3, which shows how the average number of elements n_{elem} per minute-interval change file and the average number of triples n_{update} that are inserted and deleted on the SPARQL endpoint per minute-interval change file develop as the dataset size n_{triple} grows. Using the linear fit from Equation 5.7, we can see that both variables grow linearly with the size of the dataset.

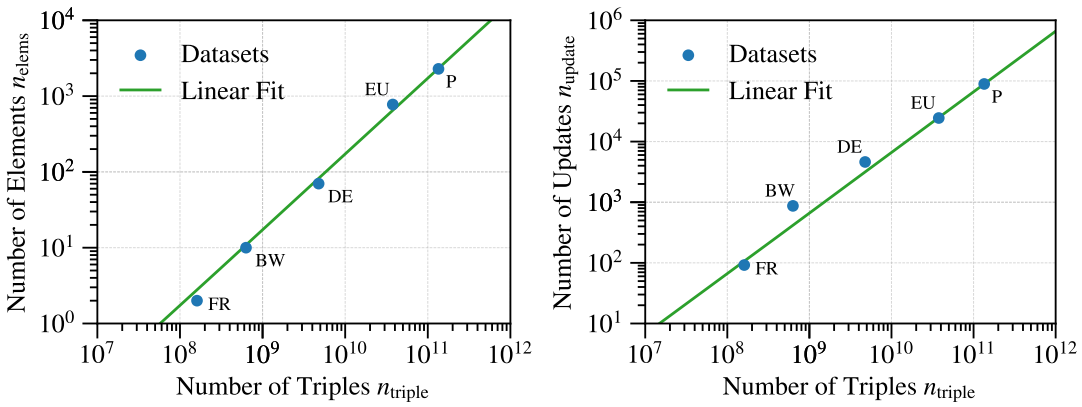


Figure 5.3: Relationship between OSM dataset size n_{triple} and the average number of elements in a minute-interval OSM change file n_{elem} with a linear fit, using Equation 5.7 with $b = 0$.

As the dataset grows, each change file contains more OSM elements to process, leading to an increased number of SPARQL queries that have to be performed and triples that have to be updated on the SPARQL endpoint. Moreover, the execution time of individual SPARQL queries and updates also grows with the dataset size. We believe

that this is the main reason for the non-linear runtime of *olu*. To verify this claim, we plot the time spent on the SPARQL endpoint t_{QLever} per minute-interval change file[6] against and the remaining runtime of *olu* t_{olu}, i.e. the total update time minus the time spent on the *QLever* endpoint, against the dataset size n_{triple}.

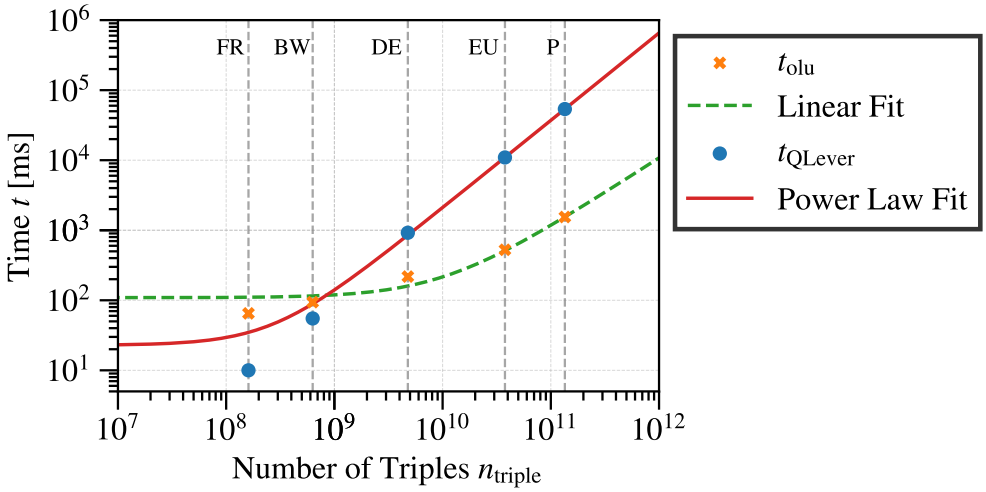


Figure 5.4: Relationship between OSM dataset size n_{triple} and the time spent on the *QLever* endpoint t_{QLever} as well as the remaining runtime from *olu* t_{olu} for various OSM datasets. The red curve shows a power law fit to the values of t_{QLever} using Equation 5.6 with power law exponent $b = 1.25$. The dotted green line shows a linear fit to the values of t_{olu} using Equation 5.7.

We can see that while the remaining runtime of *olu* t_{olu} shows a linear runtime, the time spent on the *QLever* endpoint t_{QLever} shows a super-linear runtime, similar to the one reported for the total update time.

Over 1.5 billion new OSM elements are added to the OSM database each year, resulting in a steady increase in dataset size [30]. Using this fact, we can assume that the OSM dataset grows by approximately 18 billion triples annually[7]. The total number of triples in the OSM database is therefore expected to double within roughly eight years. Based on the super-linear runtime of *olu*, we can project how the total update time will evolve in the coming years. Using our prediction model in Equation 5.9, the total update time for minute-interval change files of the complete OSM Planet dataset will increase from under one minute in 2025 to over two minutes by 2032. This increase will render minutely updates of the complete OSM planet dataset infeasible on the same

[6]This time was calculated from the timing statistics that is returned by the *QLever* endpoint for each query and update operation.

[7]On average, one OSM element corresponds to about 12 RDF triples.

machine. This projection highlights the importance of further optimizing *olu* to ensure it can continue to handle OSM's rapidly growing dataset in the future.

5.2.2 Breakdown of the Update Time

We equipped *olu* with detailed logging functionality to decompose the total update time into its major components. When executed with the `statistics` option, the tool produces a comprehensive report on the time spent in each phase of the update process. When used with a *QLever* endpoint, *olu* can also report detailed information about statistics related to the query and update operation execution on the endpoint. This breakdown enables the identification of potential bottlenecks within the update pipeline and provides valuable insights into possible optimization opportunities.

Table 5.2 presents the average percentage share of the total update time for each significant process step of olu. We listed the results for processing minutely change files for various OSM dataset sizes. The listed steps roughly correspond to the procedure described in Section 3.2, with two additional preprocessing steps for the change files: merging and sorting (see Section 4.3) and applying the boundaries for regional extracts (see Section 4.4). The percentage share was computed from the results of our performance evaluation, which was presented in Table 5.1.

Table 5.2: Breakdown of the total update time into the significant processing steps of *olu*. The table shows the average percentage share of each step for different OSM dataset sizes. The values were calculated from the values for minute change files in Table 5.1.

	Percentage Share of the Total Update Time				
Process Step	FR	BW	DE	EU	P
Merging and sorting change files	22 %	11 %	1 %	>1 %	>1 %
Apply boundaries to change files	40 %	23 %	6 %	1 %	0 %
Collect IDs of OSM Elements	2 %	12 %	7 %	4 %	4 %
Construct intermediate OSM file	2 %	6 %	7 %	4 %	6 %
Convert OSM Data to RDF Triples	6 %	14 %	6 %	1 %	>1 %
Apply Delete Operations	4 %	15 %	53 %	72 %	77 %
Apply Insert Operations	5 %	18 %	17 %	17 %	13 %

The primary observation is that as the dataset size increases, the dominant portion of the total update time shifts from preprocessing the change files to applying the

generated SPARQL update operations at the endpoint. Another noteworthy finding is that generating SPARQL update operations from the preprocessed OSM change files accounts for only a minor fraction of the total update time (at most 32 % for the OSM Baden-Württemberg dataset, but only 9 % for the OSM Planet dataset). The following discussion explores these observations in more detail.

Preprocessing of the OSM Change Files The preprocessing of the OSM change files consists of two main steps: (1) merging and sorting the change files, and (2) applying boundaries for regional extracts. Merging and sorting are both optimization and validation steps that ensure each OSM element appears only once in the resulting change file and that the change file is correctly sorted. For this reason, this step is performed even when processing a single change file.

The extraction of relevant OSM entities for regional datasets is an additional optimization step that limits updates to the target region, thereby avoiding processing of OSM elements outside the regional extract. This step is not required when working with the complete OSM planet dataset.

The time required for sorting the change files remains constant across datasets, as all experiments use the same change files from the OSM planet replication server. In contrast, the time needed to apply regional boundaries increases with dataset size, since larger datasets contain more OSM elements to extract. Nevertheless, the relative share of preprocessing time decreases as the dataset size grows, because subsequent processing stages scale more steeply.

Preprocessing the change files could be omitted if the provided change files already contain only the relevant OSM elements for the target region and are correctly sorted. This could be achieved using change files from a third-party replication server that provides pre-filtered regional extracts. However, this would shift the preprocessing workload to the replication server, potentially introducing higher delays. Therefore, this approach is only advisable when local resource savings are prioritized over total update time efficiency.

Generating SPARQL Update Operations Generating SPARQL update operations from the preprocessed OSM change files involves several sub-steps: collecting the IDs of the changed OSM elements, constructing an intermediate OSM file, converting the

OSM data to RDF triples, and finally generating SPARQL delete and insert operations. The time required for these steps is relatively small compared to the overall update time, accounting for only a minor fraction of the total processing time. This indicates that *olu* is efficient in generating the necessary update operations, even for larger datasets. There is, however, still optimization potential. We will discuss in Section 6.5 how parallelization could further improve the performance of this step.

Applying Updates to the SPARQL Endpoint The time taken to apply the generated SPARQL update operations to the SPARQL endpoint increases with the size of the dataset, eventually accounting for the majority of the total update time for larger datasets (around 90% for the OSM Planet dataset). This is an expected trend, as larger datasets typically involve more changes, resulting in a greater number of triples being inserted and deleted. While under 100 triples are deleted and inserted at the SPARQL endpoint per minute for the OSM Freiburg dataset, around 200,000 triples are deleted and inserted for the OSM Planet dataset.

The time required to apply the generated SPARQL update operations to the endpoint is influenced by several factors, but is mainly driven by the SPARQL endpoint used. In our experiments, we used *QLever* as the SPARQL endpoint. However, it is important to note that the performance of other SPARQL endpoints may differ.

Although *olu* has no influence on the performance of the SPARQL endpoint itself, it can affect the efficiency of the generated SPARQL update operations. One way to do this is to minimize the number of triples that need to be inserted and deleted on the SPARQL endpoint. This would significantly improve the overall update performance. We will discuss potential strategies for optimizing the update operations for modified OSM elements in Section 6.4.

Chapter 6

Discussion and Future Work

In the previous chapter, we demonstrated that our approach correctly and efficiently updates a SPARQL endpoint with OSM data, enabling us to keep up with the changes made to the OSM database. However, there are still some areas that could be improved or extended in future work, which we will discuss in the following sections.

6.1 Different Namespace for Tagged and Untagged Nodes

The majority of elements in the OSM database are untagged nodes, which are primarily used to describe the geometries of OSM ways and relations. As tags carry the semantic meaning of OSM elements, such untagged nodes cannot be used to represent points of interest. This has implications for querying OSM data. For example, if a user wants to find all the restaurants in a city, we only need to search the tagged nodes for information if they represent restaurants. For such a question, it would be efficient to use a SPARQL query for which the SPARQL endpoint only has to search the indexes of tagged nodes. However, this is not possible if tagged and untagged nodes share the same prefix. For this reason, *osm2rdf* was recently updated to allow users to specify a separate prefix for untagged nodes[1].

Unfortunately, this is not supported with the current version of *olu* because, among other reasons, the prefixes of OSM elements are hard-coded. Future work is required to enable *olu* to handle a variable namespace for OSM elements:

- While it is possible to hard-code the use of separate prefix in *olu*, this would not solve the underlying problem. A more holistic approach should instead be used

[1] See PR: https://github.com/ad-freiburg/osm2rdf/pull/120

to avoid hard-coding prefixes in the future. For example, the output of *osm2rdf* could be extended to include a triple indicating the namespace for a given ID of an OSM element. This would enable the queries used by *olu* to avoid hard-coded prefixes.

- Currently, it is not necessary to check for created dummy objects for OSM nodes, if they are tagged or untagged. This changes when separate prefixes are used. Therefore, to support separate prefixes, it must be checked if a node is tagged or untagged.
- In the generated triples for OSM ways, the members are identified by the triple with the predicate `osmway:member_id`, or `osmrel:member_id` for members of OSM relations. The IRI of the OSM element is used as the object. Previously, updating the tags of a member had no effect on these triples, but now the object of the triple can change if, for example, an untagged node is modified with a new tag. Therefore, the triples that store the member IRIs of a modified OSM element in the change file must be updated.

These changes could be implemented within days. However, *osm2rdf* would also need to be updated for this to be possible.

6.2 Updating Spatial Relation Triples

In addition to triples representing the geometry of an OSM element, *osm2rdf* can generate triples encoding spatial relations, such as `sfIntersects`, `sfContains`, or `sfTouches`. These triples allow efficient spatial queries, for example, to identify all buildings within a city. When an element is modified, *olu* deletes all triples associated with it, including all spatial relation triples, but does not regenerate or update them. We based this decision on the fact that the correctness of the spatial relations cannot be guaranteed after an update. Moreover, if the geometry of an element changes due to modifications in one of its members, only the geometry triples are updated, leaving the spatial relation triples potentially outdated. For instance, if a building is moved, its `geo:sfWithin` triple might no longer be correct. Consequently, after an update, spatial relation triples may be outdated and incorrect.

The current implementation of *olu* makes it infeasible to update the spatial relation triples incrementally. Recomputing all spatial relations for a dataset as large as the OSM planet data requires approximately 1.5 hours [31]. Although it might be possi-

ble to recompute the spatial relations for the subset of OSM elements that change their geometry during an update, identifying these elements and recalculating their spatial relations efficiently remains challenging, especially for short update intervals.

A practical alternative is to periodically (e.g., daily or weekly) recompute and update spatial relation triples using a dedicated tool. This approach ensures that spatial relation triples remain somewhat consistent with the latest OSM data while maintaining the performance of incremental updates. Such a tool could easily be implemented. However, there will always be periods during which the spatial relation triples may be outdated.

Another promising approach is to integrate spatial capabilities directly into the SPARQL endpoint, allowing queries to operate on geometry triples without relying on precomputed spatial relationships. Work in this direction has been carried out for engines such as *QLever* [32], [33], demonstrating that on-demand spatial querying can be executed efficiently. However, these implementations do not provide the full range of spatial predicates generated by *osm2rdf*, and supporting them would require further research and development.

6.3 Database Consistency

In Section 5.1, we demonstrated that our approach can correctly update a SPARQL endpoint. However, if the process is interrupted, whether due to a crash, lost network connection, or manual termination, not all generated SPARQL update operations may be applied to the endpoint. This could leave the database in an inconsistent state. Furthermore, correctness is only ensured after all update operations have been applied. During the update process, the database state may also be inconsistent. The following sections discuss these scenarios and outline strategies for handling them.

6.3.1 Interrupted Update Process

We can distinguish two types of interruptions that can occur during the update process: those that occur before any SPARQL update operation is sent to the endpoint, and those that arise while updates are being executed on the SPARQL endpoint. The former interruptions are harmless because no modifications have been applied to the SPARQL endpoint. In contrast, interruptions during the application of the update operations can

leave the database in an inconsistent state. For instance, if the process crashes after the delete operations have been issued, but before corresponding insert operations are completed, triples for modified elements may be removed without being reinserted. Fortunately, rerunning *olu* will restore the consistency of the OSM data on the SPARQL endpoint because the most recent versions of all modified elements are contained in the change files and no information will be deleted from the SPARQL endpoint that is needed for the update process.

At present, however, *olu* handles these interruptions inefficiently. The tool does not support resuming an update from the point of failure. Instead, it must reprocess all change files, regenerate the SPARQL update operations, and reapply them to the database. In the worst case, this means that the database remains inconsistent until the full update is reapplied.

Future Work Extending *olu* with support for resumable updates would mitigate this problem. A checkpointing mechanism could be introduced to allow the update process to continue from well-defined stages. These stages could include after downloading and merging change files, converting them into RDF triples, or after each successfully applied SPARQL update operation. Implementing these checkpoints would necessitate maintaining metadata about the update process's current state, including confirmation of update success from the SPARQL endpoint. With checkpoints in place, *olu* could resume from the last completed step, thus avoiding redundant work and minimizing the time during which the database remains inconsistent. Such a feature could probably be implemented in a matter of days.

6.3.2 Inconsistent State During Update

As discussed in the previous section, interruptions can leave the SPARQL endpoint in an inconsistent state. However, even without crashes, temporary inconsistencies may arise due to the sequential execution of update operations. In the current approach, all delete operations are issued before the new triples are inserted. During a time period between these two steps, queries may return incomplete results: the old triples of a modified element have already been removed, while the new triples have not yet been inserted. Consequently, the endpoint may briefly expose a view of the data that corresponds to no valid OSM state. This temporary inconsistency can be problematic for applications that require reliable information at all times.

Future Work This issue can be resolved by using the `DELETE/INSERT` construct. Unlike sequential operations, it allows deletions and insertions to be combined into one update request. If the SPARQL endpoint executes such a request atomically, as recommended by the SPARQL 1.1 specification (though not strictly required[2]), the inconsistency window can be reduced to zero. Listing 6.1 shows how to update the timestamp of an OSM node by deleting the old timestamp triple and inserting the new one in a single atomic operation.

Listing 6.1: SPARQL DELETE/INSERT example to update the timestamp of an OSM node in a single operation. This involves deleting the triple containing the old timestamp and inserting a triple containing the new timestamp.

```
PREFIX osmnode: <https://www.openstreetmap.org/node/>
PREFIX osmmeta: <https://www.openstreetmap.org/meta/>
DELETE {
  osmnode:123 osmmeta:timestamp ?timestamp .
}
INSERT {
  osmnode:123 osmmeta:timestamp "2022-09-05T08:03:29"
}
WHERE {
  osmnode:123 osmmeta:timestamp ?timestamp .
}
```

To implement this in *olu*, deletions and insertions for modified elements would need to be handled together rather than as separate operations. Since this is not currently the case, introducing such a change would likely require some refactoring of the codebase. Nonetheless, the necessary functionality is already present in *olu*, and we estimate that this change could be implemented within a few days to a week. It is important to note, however, that not all SPARQL endpoints support atomic execution of `DELETE/INSERT` operations. As a result, this approach may not be universally applicable.

6.4 Minimizing Triple Updates for Modified Elements

While the amount of triples that need to be inserted for newly created OSM elements and the amount of triples that need to be deleted for deleted OSM elements in the change

[2] See: https://www.w3.org/TR/sparql11-update/#updateServices

file cannot be reduced, there is potential for optimization when it comes to modified OSM elements. In the current implementation of *olu*, all triples associated with a modified element are deleted and then reinserted, regardless of whether they have actually changed. This approach guarantees correctness but can lead to unnecessary deletions and insertions, especially when only a small portion of an element's data has changed. One example is the modification of a single tag of an OSM element. Around a quarter of all elements in the change file are modified, yet they account for the majority of triples deleted from the SPARQL endpoint. We therefore see potential for optimizations in this area. Our experiment to verify the correctness of our update process in Section 5.1.3 emphasizes this point. Before the update was processed, 5 million triples were different between the old and the new OSM data. *olu* inserted and deleted around 35 million triples on the SPARQL endpoint. Therefore, in this example, the number of deleted and inserted triples could theoretically be reduced by almost 85 %.

We already discussed in Section 4.5 optimizations that we made to our implementation to reduce the number of updated triples. However, these optimizations target only specific cases and only speed up the update process marginally. In future work, a more general approach could be developed.

Future Work A straightforward method to minimize triple updates for modified elements is to compare the old and new versions of each OSM element and identify the differences, subsequently updating only the triples that have really changed. This could be done at two stages of our implementation: either before or after converting the OSM elements to RDF triples.

Comparing OSM elements before conversion would require fetching the existing triples for each modified OSM element from the SPARQL endpoint, parsing them, and comparing the extracted information to the data in the corresponding XML element of the change file. However, this approach would introduce substantial performance overhead and significantly increase the complexity of the implementation. For example, our optimization that checked each OSM way in the change file for modifications to its member list did not noticeably reduce the overall update time.

A more promising approach would be to perform the comparison after the conversion to RDF triples, i.e., using the *new* triples generated for the modified OSM elements. For this approach, the *old* triples of the modified elements would have to be retrieved

from the SPARQL endpoint. Leveraging existing approaches, such as a signature-based mapping algorithm with $O(n \log n)$ complexity [29], we could efficiently compute the difference between these two sets, enabling us to generate precise DELETE and INSERT operations affecting only the changed triples and thus minimizing unnecessary updates. However, this approach should be carefully evaluated. It remains to be determined whether the additional computational overhead introduced by fetching the old triples for the modified elements from the endpoint and calculating the set of differences would be offset by the reduced workload on the SPARQL endpoint when applying the updates.

6.5 Parallelization of the Update Process

Currently, parallelization in *olu* is limited to downloading and processing change files. Other stages of the update pipeline could also benefit from concurrent execution. As discussed in Section 4.7, SPARQL queries and operations are grouped into batches of configurable size but are processed sequentially: the next batch is sent only after the previous one completes. If the SPARQL endpoint supports concurrent requests, multiple batches could be executed in parallel. This would significantly accelerate parts of the implementation that involve frequent endpoint interaction, such as fetching IDs of relevant OSM elements or creating dummy objects. While no new functionality would be required for this, substantial code refactoring would be necessary to support parallel query execution. Implementing this change would likely require several days of development effort.

Another promising candidate for parallelization is the filtering of converted RDF triples. In the current implementation, each triple is examined sequentially to determine whether it should be included in the final update set. By partitioning the triples into smaller chunks and processing them in parallel, this step could be significantly accelerated, particularly for large datasets. However, care must be taken to preserve dependencies between triples involving blank nodes. All triples connected to the same blank node must be processed together to ensure that the blank node is consistently linked to its corresponding OSM element. Only then can a decision be made about whether the OSM element, and by extension, the associated blank node, is relevant for the update process. Compared to parallel query execution, implementing parallelization for triple filtering would require less effort and could likely be done in a few hours.

However, as the filtering step currently constitutes only a small fraction of the total processing time (under 1 %), the overall performance gain from this change would be limited.

6.6 Caching

Another potential optimization for *olu* is to use caching to store frequently accessed data, such as member of OSM elements retrieved from the SPARQL endpoint or the dummy objects created for them. In the current implementation, each execution of *olu* is independent and does not reuse information from previous runs. This can result in redundant work when *olu* is executed consecutively. For example, when the SPARQL endpoint is updated in minutely intervals. Changes made by OSM contributors often affect the same geographic areas and thus involve the same referenced OSM elements.

With caching, *olu* could avoid making repeated queries to the endpoint and prevent the creation of duplicate dummy objects. This would presumably speed up the generation of the SPARQL update operations. However, designing such a mechanism would require careful balancing of memory usage, cache invalidation, and consistency with the most recent OSM state. Even a basic caching layer could provide significant performance improvements, especially in scenarios involving frequent, minor updates. However, it should first be verified that the overhead of managing the cache does not outweigh its benefits. Implementing an effective caching strategy would likely require several days of development effort.

Chapter 7

Conclusion

In this thesis, we introduced *osm-live-updates (olu)*, a tool designed to efficiently update SPARQL endpoints containing OpenStreetMap (OSM) data. We explored the challenges of maintaining synchronization with the OSM database, a rapidly evolving global dataset, and proposed a solution that generates SPARQL update operations from OSM change files. Our approach follows a structured four-stage pipeline, which we described in detail along with key optimizations that enable high performance even for large-scale datasets such as the complete OSM Planet data.

We evaluated *olu* with respect to both correctness and performance. The correctness evaluation confirmed that the generated update operations accurately reflect the modifications described in the OSM change files, ensuring that the SPARQL endpoint remains consistent with the OSM database over time. The performance evaluation demonstrated that *olu* can efficiently process the large volume of changes made to the OSM database, including those from the full planet dataset.

We also outlined several potential directions for future work. These include methods for stronger database consistency during updates and additional performance enhancements, such as minimizing the number of triples that need to be updated on the SPARQL endpoint. Implementing these improvements could further increase the robustness and scalability of *olu*.

In summary, this work introduces *olu*, the first tool capable of keeping SPARQL endpoints synchronized with the continuously changing OSM database, while preserving the geometries and information of all OSM elements on the endpoint. By enabling incremental updates, *olu* provides applications and users using SPARQL endpoints with up-to-date access to one of the world's largest open geospatial data sources.

Chapter 8

Acknowledgments

The Work on *olu* began as part of my master's project at the *Chair of Algorithms and Data Structures*.

I would like to express my sincere gratitude to *Prof. Dr. Hannah Bast* for giving me the opportunity to work on this project. Her continuous support, valuable feedback, and insightful discussions have been invaluable throughout the course of this thesis.

I am also grateful to *Prof. Dr. Fabian Kuhn* for kindly agreeing to examine this thesis.

Finally, I want to thank my family and friends for their unwavering support and encouragement, not only during the writing of this thesis but throughout my entire academic journey. Their patience, understanding, and constant belief in me have carried me through the challenges of this long process, and I am deeply grateful for them.

Chapter 9

Additional Resources

DeepL was used to translate some technical terms and to improve the language of this thesis.

Grammarly was used to correct spelling or grammatical errors.

The *Python* library *Matplotlib* was used to create the plots in this thesis and the *Scipy* library was used to calculate the linear and power law fits used in these plots.

Bibliography

[1] OpenStreetMap contributors, *Openstreetmap*, https://www.openstreetmap.org, Accessed: 2025-08-15, 2025.

[2] R. Cyganiak, D. Wood, and M. Lanthaler, *Rdf 1.1 concepts and abstract syntax*, W3C Recommendation, W3C Recommendation published on 25 February 2014, Feb. 2014. [Online]. Available: https://www.w3.org/TR/rdf11-concepts/.

[3] H. Bast, P. Brosi, J. Kalmbach, and A. Lehmann, "An efficient rdf converter and sparql endpoint for the complete openstreetmap data," in *Proceedings of the 29th International Conference on Advances in Geographic Information Systems*, ser. SIGSPATIAL '21, Beijing, China: Association for Computing Machinery, 2021, pp. 536–539, ISBN: 9781450386647. DOI: 10.1145/3474717.3484256.

[4] W3C SPARQL Working Group, *SPARQL 1.1 Query Language*, https://www.w3.org/TR/sparql11-query/, W3C Recommendation, 21 March 2013. Accessed: 2025-08-15, 2013.

[5] H. Bast and B. Buchhold, "Qlever: A query engine for efficient sparql+text search," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, ser. CIKM '17, Singapore, Singapore: Association for Computing Machinery, 2017, pp. 647–656, ISBN: 9781450349185. DOI: 10.1145/3132847.3132921.

[6] H. Bast, J. Kalmbach, T. Klumpp, F. Kramer, and N. Schnelle, *Efficient sparql autocompletion via sparql*, Accessed: 2025-08-03, 2021. arXiv: 2104.14595 [cs.DB]. [Online]. Available: https://arxiv.org/abs/2104.14595.

[7] OpenStreetMap contributors, *Planet dump retrieved from https://planet.osm.org*, https://www.openstreetmap.org, Accessed: 2025-08-15, 2025.

[8] OpenStreetMap contributors, *Openstreetmap stats*, https://www.openstreetmap.org/stats/data_stats.html, Accessed: 2025-08-15, 2025.

[9] J. Topf, *Osmium tool, a multipurpose command line tool based on the osmium library*, Accessed: 2025-08-15, 2025. [Online]. Available: https://osmcode.org/osmium-tool/.

[10] G. GmbH, *Geofabrik downloads*, Accessed: 2025-08-15, 2025. [Online]. Available: https://download.geofabrik.de/.

[11] BBBike.org, *Bbbike openstreetmap extracts*, Accessed: 2025-08-15, 2025. [Online]. Available: https://download.bbbike.org/.

[12] D. Beckett, T. Berners-Lee, E. Prud'hommeaux, and G. Carothers, "Rdf 1.1 turtle," *World Wide Web Consortium*, vol. 25, pp. 18–31, 2014.

[13] J. Herring et al., "Opengis® implementation standard for geographic information-simple feature access-part 1: Common architecture [corrigendum]," 2011, Accessed: 2025-08-22.

[14] OpenStreetMap, *Openstreetmap change files*, https://wiki.openstreetmap.org/wiki/Change_file, Accessed: 2025-08-18.

[15] OpenStreetMap contributors, *Openstreetmap - replication diffs*, https://planet.openstreetmap.org/replication/, Accessed: 2025-08-18.

[16] R. Olbricht et al., "Overpass api," *Anwenderkonferenz für Freie und Open Source Software für Geoinformationssysteme*, 2011, Accessed: 2025-08-15.

[17] OpenStreetMap, *Pbf format*, https://wiki.openstreetmap.org/wiki/PBF_Format, Accessed: 2025-09-01.

[18] The PostgreSQL Global Development Group, *Postgresql*, Accessed: 2025-09-01, 2025. [Online]. Available: https://www.postgresql.org/.

[19] PostGIS Project Steering Committee, *Postgis*, Accessed: 2025-09-01, 2025. [Online]. Available: https://postgis.net/.

[20] J. Burgess and K. Krueger, *Osm2pgsql: Openstreetmap data to postgresql converter*, Accessed: 2025-08-18, 2025. [Online]. Available: https://osm2pgsql.org/.

[21] J. Burgess and K. Krueger, *Osm2pgsql manual: Updating an existing database*, Accessed: 2025-08-18, 2025. [Online]. Available: https://osm2pgsql.org/doc/manual.html#updating-an-existing-database.

[22] S. Contributors, *Sophox: A collection of services exposing osm data*, Accessed: 2025-08-18, 2025. [Online]. Available: https://sophox.org/.

[23] Osmium Developers, *Osmium library*, `https://osmcode.org/libosmium/`, Fast and flexible C++ library for working with OpenStreetMap data, supporting nodes, ways, relations, and change files., 2025.

[24] W3C SPARQL Working Group, "Sparql 1.1 update," 2013. [Online]. Available: `https://www.w3.org/TR/sparql11-update/`.

[25] W3C SPARQL Working Group, *Sparql 1.1 graph store http protocol*, `https://www.w3.org/TR/sparql11-http-rdf-update/`, Accessed: 2025-09-16, 2013.

[26] G. Langdale and D. Lemire, "Parsing gigabytes of json per second," *The VLDB Journal*, vol. 28, no. 6, pp. 941–960, 2019.

[27] B. C. Libraries, *Boost.propertytree*, Accessed: 2025-09-03, Boost C++ Libraries, 2025. [Online]. Available: `https://www.boost.org/doc/libs/release/doc/html/property_tree.html`.

[28] D. Novatchev, *Rapidxml*, Accessed: 2025-09-03, SourceForge, 2025. [Online]. Available: `http://rapidxml.sourceforge.net/`.

[29] Y. Tzitzikas, C. Lantzaki, and D. Zeginis, "Blank node matching and rdf/s comparison functions," in *International Semantic Web Conference*, Springer, 2012, pp. 591–607.

[30] P. Neis, *Osmstats*, Accessed: 2025-10-24. [Online]. Available: `https://osmstats.neis-one.org/?item=elements`.

[31] H. Bast, P. Brosi, J. Kalmbach, and A. Lehmann, "Efficient spatial joins for large sets of geometric objects," in *Proceedings of the 32nd ACM International Conference on Advances in Geographic Information Systems*, 2024.

[32] C. Ullinger, "Efficient spatial search for the qlever sparql engine," 2025.

[33] J. Zeller, "Implementing efficient geo queries for the sparql engine qlever," 2025.

Appendix A

Blank Node Example

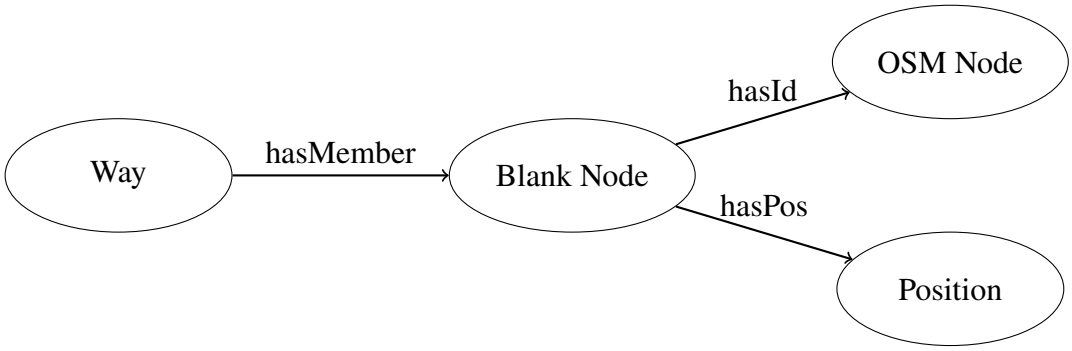


Figure A.1: Illustration of how a blank node can be used to represent the ordered list of members in an OSM way by explicitly associating each member with its corresponding position in the list. This member relationship is expressed through three RDF triples (see Listing A).

Listing A.1: The member relationship between an OSM way and a node is expressed through three RDF triples (see Figure A.1). A blank node is introduced to link the node to its specific position within the ordered sequence of way members.

```
1        ex:osmway ex:hasMember _:blankNode
2        _:blankNode ex:hasId ex:osmNode
3        _:blankNode ex:hasPos ex:position
```

Appendix B

Command-Line Options

Table B.1: Overview of command-line options for *osm-live-updates*.

Option name	**Description**
access-token	The access token of the SPARQL endpoint.
batch-size	The number of values or triples that should be sent in one batch to the SPARQL endpoint.
bbox	Specify a bounding box (LEFT,BOTTOM,RIGHT,TOP) to cut out a specific area from the change files.
debug	If set, all SPARQL queries are written to the output file.
endpoint-uri-updates	Specify a separate URI for sending SPARQL update requests.
extract-strategy	Specify the extract strategy to use.
graph	Specify the URI of a specific graph that you want to update.
input	A directory containing OSM change files
max-sequence-number	The sequence number where to stop the update process.
polygon	Specify a (multi)polygon file (.poly) to cut out a specific area from the change files.
replication-server	The base URL of the replication server.
sequence-number	The sequence number to start the update process from.
sparql-output	Specify the file to which the SPARQL updates should be written.
sparql-response-output	Specify the file to which the SPARQL endpoint responses should be written.
statistics	Specify if detailed statistics should be added to the output.
timestamp	The time stamp to start the update process from.
tmp	Specify the location of the directory for the temporary files.
qlever	Specify if the SPARQL endpoint is QLever. More statistics will be added to the output.