KNOWLEDGE ENGINEERING

KG OpenStreetMap

Filino Alessandro
Matricola: 7125518
E-Mail: alessandro.filino@stud.unifi.it

Macaluso Alberto
Matricola: 7115684
E-Mail: alberto.macaluso@stud.unifi.it

Data Consegna: 27/05/2023

Anno accademico 2023/24

INDICE

1 Introduzione

Lo scopo del progetto è stato automatizzare l'acquisizione e la mappatura di un grafo stradale (triplificazione) relativo a un'area geografica specifica mediante l'uso di container Docker.
Nel sistema iniziale, il grafo presentava un numero eccessivo di nodi correlati alla geometria delle strade che rallentavano significativamente il processo di creazione delle triple. Pertanto è stato necessario eliminare le informazioni non pertinenti al fine di ottimizzarlo.
Una volta generate le triple, doveva inoltre essere possibile il caricamento e la loro gestione attraverso uno store RDF.
L'utilizzo dei container Docker ha permesso di creare un ambiente isolato, scalabile e facilmente mantenibile, garantendo tutte le funzionalità necessarie per la gestione di una mappa OpenStreetMap. Ciò ha esteso la compatibilità con tutti i sistemi operativi, tra cui:

- Linux
- MacOS
- Windows

Nei capitoli successivi saranno spiegati i requisiti del sistema, nonché le funzionalità e le operazioni implementate. Successivamente, verrà presentata una descrizione completa delle varie tecnologie utilizzate e come esse cooperano tra loro.
Infine sarà descritto il processo di ottimizzazione della generazione delle triple, illustrando le modifiche apportate ai file SQL coinvolti.

Il codice sorgente è disponibile presso **repository github**: **https://github.com/AlessandroFilino/kg-open-street-map**

2 Requisiti e funzionalità

I requisiti per poter eseguire lo script e generare la triplificazione sono i seguenti:

- Python 3 (È stata testata la 3.10, non dovrebbero esserci problemi con le versioni precedenti)
    - Modulo requests (installabile tramite il comando: `pip install requests`)
- Docker Engine e Docker ComposeV2 (entrambi presenti in un'unica installazione con Docker Desktop)
- Archivio **sparqlify.zip** (Da inserire in `Dockers/scripts` se non già presente)
- Opzionali (in caso di ricompilazione di sparqlify, vedi **sez 3.1**):
    - Java: **v.**11 (openjdk)
    - Maven: **v.**3.6.3

Il sistema si basa su uno script (`triplification.py`) da eseguire a linea di comando con le dovute opzioni:

- `-h, --help` :
  Mostra l'elenco di tutte le opzioni disponibili con descrizione annessa
- `-r , --relation_name [RELATION_NAME]` :
  Permette di specificare il nome di una relazione su cui andare ad eseguire la triplificazione.
- `-o , --osm_id [OSM_ID]` :
  Permette di specificare l'osm id di una relazione su cui andare ad eseguire la triplificazione.
- `-l , --load_to_rdf [GRAPH_NAME]` :
  Permette di specificare un grafo su cui andranno caricate in modo automatico le triple generate.
  Nel caso in cui il grafo esistesse già, esso verrà resettato prima di inserire le nuove triple.
- `-f, --file_name [FILE_NAME]` :
  Permette di specificare il nome di una mappa (in formato .pbf) già scaricata in precedenza da geofabrik e posta nella directory `Dockers/maps`
- `--generate_old` :
  Permette di generare le triple anche con il vecchio sistema (senza ottimizzazione)

Note per l'utilizzo :

Lo script **triplification.py** gestisce le eccezioni mostrando a schermo i messaggi di errore, di seguito alcune note per evitare i problemi più comuni :

È necessario specificare una sola opzione tra [RELATION_NAME] e [OSM_ID] (con nessuna o entrambe lo script termina con errore).

Nel caso in cui sia stato scelto [RELATION_NAME] l'osm_id viene recuperato tramite API del server **Nominatim** e corrisponde alla relazione "più importante" corrispondente al nome indicato

(Es con `-r Firenze` si otterrà l'oms_id dell'intera provincia, non della singola città).

Se la mappa non è già presente in `Dockers/maps` o non è stato specificato [FILE_NAME] essa verrà scaricata automaticamente tramite **Overpass API**.

Durante la triplificazione, utilizzare **PgAdmin** per connettersi al database `maps` (dove vengono caricati i dati della mappa) produrrà un errore in quanto è necessario resettare il database ad ogni run (vedi **sez 3.3.2**).

Esempi di utilizzo :

- `$ python3 triplification.py -r Siena -l http://example.org/test`
  - Questo comando scaricherà la mappa della provincia di Siena (osm_id = 42172) in formato .osm nella cartella `Dockers/maps` e ne eseguirà la triplificazione salvando nella directory `Dockers/maps/42172` il file .n3 risultante.
    In seguito le triple generate saranno caricate su virtuoso RDF nel grafo **http://example.org/test**.

- `$ python3 triplification.py -o 42602 -f centro-latest.osm.pbf`
  - Questo comando effettuerà la triplificazione della relazione 42602 (comune di Firenze) prendendo i dati dalla mappa **centro-latest.osm.pbf** caricata dall'utente nella directory `Dockers/maps`.
    In questo caso, non essendo stato specificato il nome di un grafo, non sarà effettuato alcun caricamento su **virtuoso**.

- `$ python3 triplification.py -o 42602 -f centro-latest.osm.pbf --generate_old -l http://example.org/test`
  - Questo comando effettuerà le stesse operazioni dell'esempio precedente generando inoltre le triple con il vecchio sistema.
    Al termine del processo,nella directory `Dockers/maps/42602` avremmo quindi sia 42602.n3 (triple semplificate) che 42602_old.n3 (triple non semplificate).

3 Descrizione del sistema

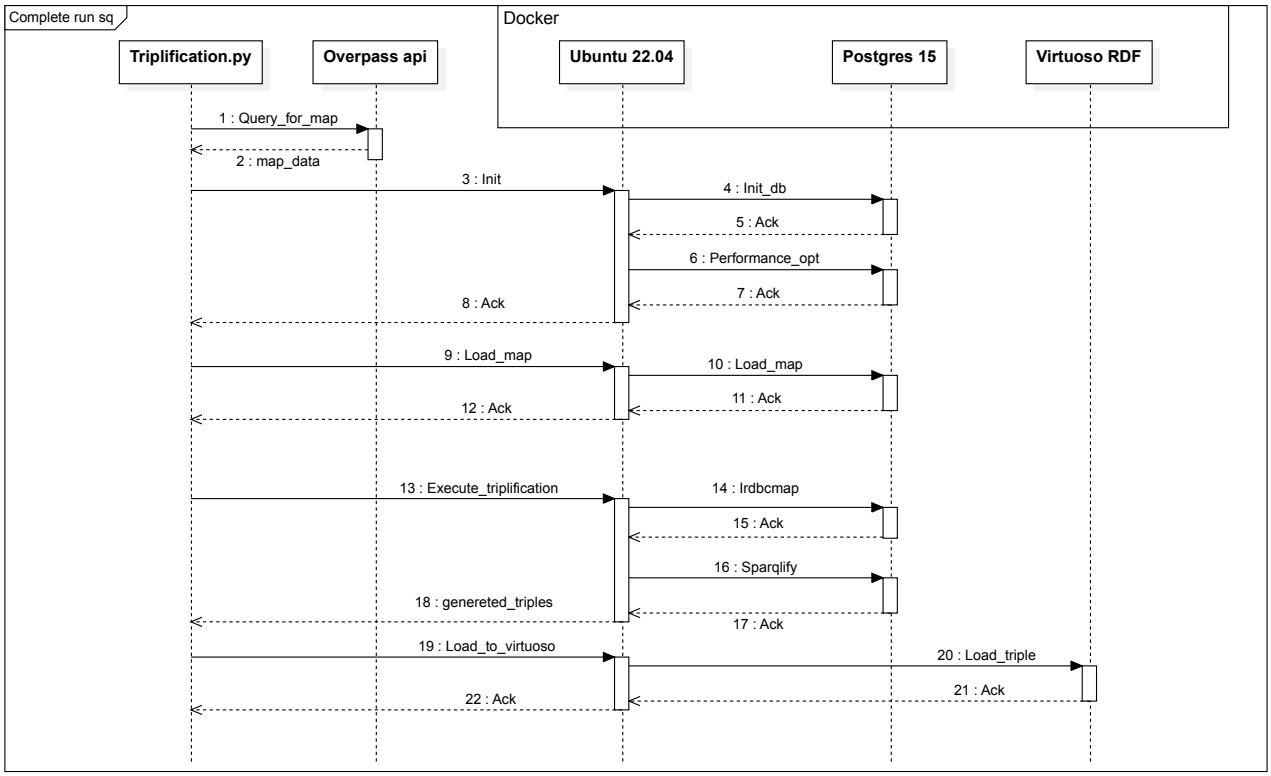

Figure 1: Diagramma di flusso di una run completa

Il diagramma rappresenta il flow delle richieste tra i vari componenti del sistema, i quali saranno illustrati esaustivamente in seguito.
In questa sezione invece andremo a descrivere il comportamento del sistema a più alto livello.

Una volta avviato lo script `triplification.py`, in base alle opzioni specificate, verra scaricata la mappa richiesta e sarà eseguito il compose up dei conteiners.
Una volta inizializzati (potrebbe essere necessario un po' di tempo per fare il pull delle immagini durante il primo avvio) lo script inizierà a comunicare con il container **Ubuntu**. Quest'ultimo andrà a eseguire una serie di operazioni sul container **Postgres** quali la creazione del database e la sua preparazione per poter contenere i dati scaricati da OpenStreetMap.
Successivamente viene effettuato l'upload della mappa sul database e sempre tramite **Ubuntu** viene avviata una query atta alla trasformazione delle tabelle per la triplificazione. Questo è uno dei processi fondamentali del sistema e il tempo di esecuzione può variare molto in base alla grandezza della relazione e alla quantità dei dati contenuti in essa. Una volta completata la query avverrà la triplificazione vera e propria per mezzo del tool **Sparqlify**.
Infine se è stato specificato un grafo, si procederà con la copia delle triple in locale e il successivo caricamento su **Virtuoso RDF** nel grafo indicato.

3.1 Sparqlify

All'interno della directory `Dockers/scripts` è stata inserito uno zip contenente **Sparqlify** nella sua ultima versione **v.**0.9.1.
È possibile ottenere il medesimo risultato anche compilando manualmente il codice fornito nella loro **repository Github**.
Per poter eseguire correttamente la compilazione del codice è necessario:

- Installare JAVA: `$ sudo apt-get install openjdk-11-jdk`
- Installare Maven: `$ sudo apt-get install maven`
- All'interno della directory `Dockers/scripts`, eseguire il comando :
  `git clone https://github.com/SmartDataAnalytics/Sparqlify`
- Aprire il file Main.java posto in:
  `Dockers/scripts/Sparqlify/sparqlify-cli/src/main/java/org/aksw/sparqlify/web/Main.java`
- Inserire alla riga 2010 (circa), prima del comando:
  `logger.info("Database product: " + dbProductName);`
  le seguenti stringhe:

```
1        if(dbProductName.equals("PostgreSQL")) {
2    dbProductName = "Postgresql";
3  }
4
```

- All'interno della directory `Dockers/scripts/Sparqlify` eseguire i comandi:
  - `$ mvn clean install`
  - `$ cd sparqlify-cli`
  - `$ mvn assembly:assembly`
- Concluse queste operazioni, all'interno della directory:
  `Dockers/scripts/Sparqlify/sparqlify-cli/target` troveremo il file jar da utilizzare.

N.B: Nel caso in cui si volesse utilizzare una versione diversa da quella indicata sarà necessario aggiornare il file bash: `Dockers/scripts/sparqlify.sh`.

3.2 Osmosis

Osmosis è un tool da riga di comando che permette di processare dati OpenStreetMap.
Nel nostro caso è stato utilizzato per eseguire il dump della mappa (dai formati .osm o .pbf) in appositi file di testo ottimizzati per il caricamento sul database. Inoltre è stato impiegato un filtro basato sul `BoundingBox` della relazione di interesse nel caso di mappa fornita dall'utente. In questo modo, evitando di caricare parti non necessarie alla triplificazione, è stato possibile ridurre notevolmente il tempo di esecuzione.

3.3 Docker

Il setup del gruppo di container è definito nel file `Dockers/docker-compose.yml` e comprende :

- Le immagini dei container :
  - `postgis/postgis:15-3.3` (postgres 15 con l'estenzione per l'utilizzo di **Postgis**)
  - `ubuntu:22.04` (Per questo container è stato utilizzato un docker file apposisto, vedi **sez 3.3.1**)
  - `openlink/virtuoso-opensource-7`
  - `dpage/pgadmin4:7.1`
- Il mapping delle porte dei servizi utilizzati (sono state lasciate quelle di defaults)
- Le configurazioni per le autenticazioni :
  - **Postgres** :
    `POSTGRES_USER=admin, POSTGRES_PASSWORD=admin`
  - **Virtuoso** :
    `DBA_PASSWORD=admin`
  - **PgAdmin** :
    `PGADMIN_DEFAULT_EMAIL=test@gmail.com, PGADMIN_DEFAULT_PASSWORD=admin`
- Il mount di alcuni volumi quali:
  - Su **Postgres** e **Virtuoso** le directory contenenti i database :
    `/postgresDB/postgres-data:/var/lib/postgresql/data, /virtuoso_data:/database`
  - Su **Ubuntu** alcune directory di appoggio per le mappe e per gli script da eseguire :
    `/maps:/home/maps, /scripts:/home/scripts`

3.3.1 Ubuntu

Per l'immagine di **Ubuntu** è stata utillizzata l'immagine **v.**22.04.
Il container è previsto di alcune utility utilizzate per la predisposizione del sistema e successivamente per la sua inizializzazione. Queste sono state definite all'interno di un dockerfile (`Dockers/Dockerfile_ubuntu`).
Durante la fase di avvio, il container esegue in ordine:

- Lo script **init.sh** posto nella directory `Dockers/scripts` che esegue il setup (in caso di prima esecuzione) delle directory necessarie. Lo stesso script è incaricato anche di estrarre lo zip contenente sparqlify (vedi **sez 3.1**) ed eseguire il setup del database postgres (vedi **sez 3.3.2**). Infine esegue i file sql necessari per l'esecuzione di osmosis (vedi **sez 3.2**).
- Lo script **load_map.sh** posto nella directory `Dockers/scripts` che esegue il caricamento della mappa all'interno del database postgres.
- Lo script **irdbcmap.sh** posto nella directory `Dockers/scripts` che genera le triple nella modalità ottimizzata o classica (vedi **sez 2**). Questo è possibile attraverso lo script **sparqlify.sh** che esegue il file jar (vedi **sez 3.1**).

- Lo script **load_to_virtuoso.sh** posto nella directory `Dockers/scripts` che, come si evince dal nome, esegue il caricamento delle triple nel grafo indicato in fase di setup.

Tutti gli script bash sono previsti di un **handle** che in caso di problemi durante l'esecuzione, sono in grado di informare l'utente in quale punto specifico si è verificato l'errore fornendo anche una breve descrizione. Inoltre sono stati resi globali i riferimenti del nome utente, password, nome del database e il grafo su cui caricare le triple. In questo modo sarà più facile ridefinire questi parametri in futuro senza dover apportare troppe modifiche al codice.

3.3.2 Postgres e PgAdmin

Il container **Postgres** contiene l'immagine `postgis/postgis:15-3.3` ossia le ultime versioni attualmente disponibili per un database **PostgreSQL** (**v.**15) e **PostGis** (**v.**3.3.2).
Il databse dovrà quindi contenere le informazioni estratte dalla mappa e attraverso l'estensione postgis sarà possibile ottenere un miglior supporto per la la ricerca, indicizzazione e l'esecuzione delle spatial queries.
Per fare queste operazioni, attraverso l'esecuzione del file `triplification.py` vengono eseguiti dal container **Ubuntu** gli script bash (vedi **sez.** 3.3.1) di inizializzazione del database.
In particolare troviamo **init.sh** che:

- Controlla se esiste già un database con il nome indicato dalla variabile `DB_NAME`. In caso affermativo lo cancella e lo crea.
- Abilita nel database indicato l'estensione per l'utilizzo di PostGis (per una gestione ottimizzata delle geometrie) e hstore (che consente di memorizzare insiemi di coppie chiave/valore all'interno di un singolo elemento).
- Esegue i file sql per il caricamento dei dati della mappa.

Assieme a Postgres, è previsto anche un container che comprende **PgAdmin**. Attraverso questo tool, collegandosi all'indirizzo `loaclhost:5050` è possibile:

- Eseguire Query sul database.
- Controllare graficamente lo stato di funzionamento.
- Visionare la geometria della mappa caricata.

Il container PgAdmin dipende dal container Postgres. Questo ci consente di instaurare un collegamento diretto tra i due senza ricercare ad ogni esecuzione gli indirizzi ip specifici che potrebbero ogni volta variare. In questo modo possiamo accedere all'interfaccia di PgAdmin e collegarci al server Postgres inserendo semplicemente nel campo Hostname/address la sigla `postgres` (ossia il nome del container assegnato nel docker-compose al server SQL).

3.3.3 Virtuoso

Il container **Virtuoso** contiene l'immagine `openlink/virtuoso-opensource-7` ed è accessibile all'indirizzo `localhost:8890`.
Dopo l'esecuzione del file `triplification.py` e generato il file .n3, quest'ultimo viene copiato all'interno del

volume dedicato a Virtuoso.
Successivamente, è possibile caricare il contenuto nel grafo selezionato eseguendo lo script:
`Dockers/scripts/load_to_virtuoso.sh`.
Durante questa fase, viene verificata l'esistenza di un grafo con il nome specificato:

- Se esiste già, viene eliminato e creato uno nuovo.
- Altrimenti, viene creato direttamente un nuovo grafo con il nome indicato.

Nello script bash viene anche effettuato un controllo aggiuntivo: ogni volta che si desidera creare o aggiornare un grafo, viene verificato se ne era stato precedentemente creato uno utilizzando le stesse triple specificate. Se si verifica questa condizione, vengono eliminate tutte le voci correlate al file .n3 specificato dalla lista `DB.DBA.LOAD_LIST` di Virtuoso. Senza questa operazione preliminare, non sarebbe possibile creare grafi che condividono un file .n3 con lo stesso nome o aggiornare quelli già esistenti.

4 Ottimizzazione delle triple

Dopo aver implementato l'intero sistema su docker, un altro degli obiettivi del progetto è stato quello di eseguire una semplificazione delle triple prodotte.
Gli elementi che compongono i file `.n3` relativi alle mappe si basano sul **knowledge model** di **Km4City** e in particolare la nostra analisi è stata incentrata sui `RoadElement` di cui mostriamo di seguito un esempio.

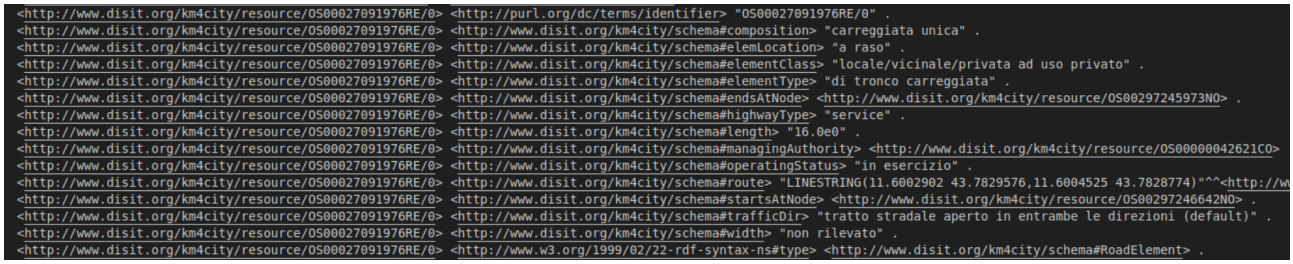

Figure 2: Estratto di un RoadElement

Quello rappresentato in figura è l'estratto di un `RoadElement` relativo ad un singolo segmento di una strada (in particolare il primo segmento di `OS00027091976RE`) e le righe che lo descrivono di conseguenza sono ripetute per ogni tratto di ogni strada. Questo porta ad numero molto elevato di righe (e quindi di spazio) per ogni file `.n3`, cosa che potrebbe essere evitata modificando opportunamente i `Route`.
Questi sono campi dei `RoadElement` che contengono i `Linestring` ossia informazioni geometriche che permettono di unire i nodi consecutivi andando a rappresentare il percorso della strada.

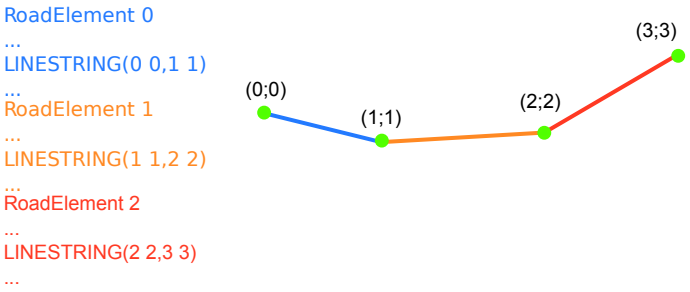

Figure 3: Estratto di RoadElementRoute senza ottimizzazione

Nella situazione di partenza viene generato un `RoadElement` per ogni segmento formato da due nodi successivi, come è visibile in figura.

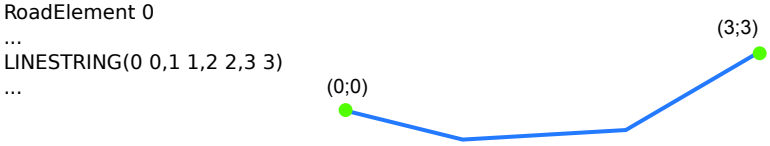

Figure 4: Estratto di RoadElementRoute con ottimizzazione

L' obiettivo è stato quello di conservare esclusivamente i nodi di inizio e fine strada (o di incrocio), mantenendo comunque le informazioni geometriche dei nodi intermedi rimossi all'interno di un unico `Linestring`, come

mostrato in fig. 5.

Figure 5: Esempio modello iniziale

Nell'immagine è possibile vedere un esempio di un tracciato reale di una strada intersecato dal confine della relazione.
La rappresentazione è relativa allo stato iniziale del sistema (senza ottimizzazione). Le operazioni da svolgere sono quindi la rimozione dei nodi (in verde) non necessari, ricostruendo i `Linestring` (in blue) senza apporre variazioni al percorso.

Le modifiche per realizzare la semplificazione sono state apportate al file `Dockers/scripts/irdbcmap.sql` contentente una lista di query atte alla generazione delle tabelle per la triplificazione. Esso viene eseguito dal container `Ubuntu` con lo script `Dockers/scripts/irdbcmap.sh`.

Per conseguire quanto appena descritto è stato necessario modificare e adattare le query esistenti:

1. **Operazioni relative ai nodi da mantenere per ogni strada**.
   Le query che seguono andranno a modificare la tabella `way_nodes` che contiene informazioni sui nodi su cui si basano tutte le query successive nel file `Dockers/scripts/irdbcmap.sql` .
   Quello che vogliamo fare è rimuovere tutti i nodi che non sono di inizio/fine strada, incroci o immediatamente precedenti/successivi ai bordi del confine. I nodi rimasti saranno gli estremi dei segmenti che andranno a formare i nuovi `RoadElement`.
   Per non perdere informazioni, tutti i nodi rimossi sono stati copiati e salvati in apposite tabelle, nel dettaglio:

   - Salvataggio ed eliminazione dei nodi che non sono all'interno del confine della relazione.

```
CREATE TABLE way_nodes_out AS
    SELECT * FROM way_nodes wn
    WHERE wn.node_id NOT IN (
```

```
        SELECT id
        FROM nodes
        JOIN extra_config_boundaries boundaries
        ON ST_Covers(boundaries.boundary, nodes.geom));

DELETE FROM way_nodes wn
WHERE wn.node_id IN (SELECT wo.node_id FROM way_nodes_out wo);

```

- Salvataggio di tutti i nodi caricati (esclusi quelli eliminati al passo precedente).

```
CREATE TABLE all_way_nodes AS
    SELECT * from way_nodes;
```

- Rimozione dei nodi che non sono incroci tra due strade o nella stessa strada (es rotonde).

```
CREATE TEMPORARY TABLE tmp_node_to_delete AS
    SELECT w2.node_id
    FROM way_nodes w2 EXCEPT
        SELECT w2.node_id
        FROM way_nodes w2
        INNER JOIN way_nodes w3
        ON w2.way_id != w3.way_id AND w2.node_id = w3.node_id EXCEPT
            SELECT w4.node_id
            FROM way_nodes w4
            GROUP BY w4.node_id
            HAVING COUNT(*) > 1;

DELETE FROM way_nodes w1
WHERE w1.node_id IN (SELECT w2.node_id FROM tmp_node_to_delete w2);
```

- Inserimento dei nodi di inizio e fine di ogni strada.

```
CREATE TEMPORARY TABLE temp_way_nodes AS
    SELECT way_id, MAX(sequence_id) AS max_sequence_id, MIN(sequence_id) AS
    min_sequence_id
    FROM all_way_nodes
    GROUP BY way_id;

INSERT INTO way_nodes
    SELECT w1.*
    FROM all_way_nodes w1
    JOIN temp_way_nodes w2 ON w1.way_id = w2.way_id
    LEFT JOIN way_nodes wn ON w1.node_id = wn.node_id
    WHERE wn.node_id IS NULL
    AND (w1.sequence_id = w2.max_sequence_id OR w1.sequence_id = w2.min_sequence_id);
```

- Inserimento dei nodi immediatamente precedenti o successivi ai bordi del confine

```
DROP TABLE IF EXISTS way_nodes_conf;
CREATE TABLE way_nodes_conf as
SELECT aw.way_id, aw.node_id, aw.sequence_id  from all_way_nodes aw
JOIN way_nodes_out next_node ON aw.way_id = next_node.way_id
    AND aw.sequence_id = next_node.sequence_id - 2
```

```
JOIN all_way_nodes aw1 ON aw1.way_id = aw.way_id
    AND aw1.sequence_id = aw.sequence_id + 1
UNION
SELECT aw2.way_id, aw2.node_id, aw2.sequence_id FROM all_way_nodes aw2
JOIN way_nodes_out prev_node ON aw2.way_id = prev_node.way_id
    AND aw2.sequence_id = prev_node.sequence_id + 1;

INSERT INTO way_nodes select *
FROM way_nodes_conf
ON CONFLICT DO NOTHING;

CREATE TABLE way_nodes_old as
SELECT * FROM way_nodes;
```

- Aggiornamento degli id delle sequenze dei nodi in modo che risultino consecutivi nella tabella.

```
CREATE OR REPLACE VIEW tmp_view AS
    SELECT way_id,node_id,
    (ROW_NUMBER() over(partition by way_id ORDER BY sequence_id) -1) AS sequence_id
    FROM way_nodes;

SELECT *
INTO tmp_db
FROM tmp_view;

DELETE FROM way_nodes;

INSERT INTO way_nodes
SELECT *
FROM tmp_db;
```

2. **Operazioni relative ai** `Linestring`.

   Le query successive andranno a modificare la tabella `RoadElementRoute` da cui derivano i campi `Route` dei `RoadElement` nel file **.n3**.

   Tramite una serie di viste di appoggio, abbiamo tracciato i segmenti delle strade (basandoci sui nodi descritti precedentemente) andando ad includere anche le coordinate dei nodi intermedi rimossi.

   Nel dettaglio:

   - Creazione (tramite viste di appoggio) di un array per ogni strada contenente le coordinate dei nodi che la compongono.

```
CREATE OR REPLACE VIEW tmp_way_array AS
    SELECT way_id, array_agg(points) AS points
    FROM tmp_node_coord0
    GROUP BY way_id;
```

   - Selezione degli indici degli elementi degli array relativi ai nodi del punto precedente.

```
CREATE TABLE tmp_node_idx AS
    SELECT t1.way_id, t1.sequence_id AS start_idx, t2.sequence_id AS end_idx
    FROM (SELECT way_id, sequence_id,
    ROW_NUMBER() OVER (ORDER BY way_id, sequence_id) AS row_number
```

```
    FROM way_nodes_old) t1
    JOIN
        (SELECT way_id, sequence_id, ROW_NUMBER() OVER (ORDER BY way_id, sequence_id)
        AS row_number
        FROM way_nodes_old) t2
        ON t2.row_number = t1.row_number + 1 AND t2.way_id = t1.way_id;
```

- Filtraggio degli indici in modo da evitare di tracciare segmenti `Linestring` al di fuori del confine della relazione.

```
UPDATE tmp_node_idx AS ar
SET end_idx = ar.start_idx
FROM
    (SELECT ar1.way_id, ar1.start_idx, ar1.end_idx
     FROM tmp_node_idx ar1
        JOIN way_nodes_conf wc
            ON wc.way_id = ar1.way_id
            AND wc.sequence_id = ar1.start_idx
      JOIN way_nodes_conf wc1
            ON wc1.way_id = ar1.way_id
            AND wc1.sequence_id = ar1.end_idx)
     AS subquery
WHERE ar.way_id = subquery.way_id
    AND ar.start_idx = subquery.start_idx
    AND ar.end_idx = subquery.end_idx;
```

- Mapping tra gli indici dei nodi e le rispettive posizioni nell'array della vista di appoggio creata in precedenza (`tmp_way_array`).

```
CREATE OR REPLACE VIEW tmp_sequence_to_idx AS
    SELECT subquery.way_id, subquery.local_id, subquery.idx
    FROM (SELECT tc.way_id, tc.local_id ,
        ROW_NUMBER() OVER(partition by tc.way_id order by tc.local_id)
        AS idx
        FROM tmp_node_coord0 tc) subquery
    ORDER BY subquery.local_id;

CREATE OR REPLACE VIEW tmp_array_idx AS
    SELECT ai.way_id, old.local_id as old_idx,
        (SELECT idx FROM tmp_sequence_to_idx WHERE way_id = ai.way_id
         AND local_id = ai.start_idx) AS start_idx,
        (SELECT idx FROM tmp_sequence_to_idx WHERE way_id = ai.way_id
         AND local_id = ai.end_idx) AS end_idx
    FROM tmp_node_idx ai
    JOIN tmp_sequence_to_idx old
        ON old.way_id = ai.way_id AND old.local_id = ai.start_idx
    ORDER BY start_idx;
```

- Generazione effettiva dei `Linestring` tramite l'utilizzo degli indici(`tmp_array_idx`) a partire dagli array di coordinate precedentemente creati(`tmp_way_array`).

```
CREATE OR REPLACE VIEW tmp_linestrings AS
    SELECT 'OS' || lpad(ar.way_id::text,11,'0') || 'RE/' || wn.sequence_id AS id,
```

```
    ar.start_idx, ar.end_idx, ST_MakeLine(t1.points[ar.start_idx : ar.end_idx])
    AS linestring
    FROM tmp_way_array t1
    LEFT JOIN tmp_array_idx ar
        ON t1.way_id = ar.way_id
    JOIN way_nodes_old AS wno
        ON ar.way_id = wno.way_id AND wno.sequence_id = ar.old_idx
    JOIN way_nodes AS wn
        ON wn.way_id=wno.way_id AND wn.node_id = wno.node_id
    ORDER BY id;

CREATE OR REPLACE VIEW tmp_cleaned AS
    SELECT t1.id, t1.linestring
    FROM tmp_linestrings t1
    LEFT JOIN tmp_linestrings t2
        ON t1.linestring = t2.linestring AND t1.id > t2.id
    WHERE t2.id IS NULL AND t1.linestring <> '';
```

- Composizione finale dei `RoadElementRoute`.

```
INSERT INTO RoadElementRoute
    SELECT CONCAT('http://www.disit.org/km4city/resource/OSM/', :OSM_ID),
    id, linestring
    FROM tmp_cleaned;
```

Una volta applicate le modifiche sopra esposte è stato ottenuto il risultato visibile in figura.

Figure 6: Esempio modello finale

5 Conclusioni

Con l'elaborato svolto, oltre a realizzare un sistema portabile con un interfaccia unica a linea di comando, siamo riusciti a introdurre anche alcune migliorie dal punto di vista delle performance.
In primo luogo l'ottimizzazione delle triple prodotte. La rimozione dei `RoadElement` non necessari nelle triple ha portato alla diminuzione del numero di righe totali di circa 6/7 volte quelle iniziali. L'ottimizzazione effettiva comunque dipende molto dal numero di strade e incroci presenti nella relazione.
Tutte le componenti del sistema sono state aggiornate rispetto al modello iniziale e sono state svolte le relative operazioni di adattamento. I cambiamenti più incisivi sono stati il passaggio da postgres 9.6 a postgres 15 e la ricompilazione di `Sparqlify` all'ultima versione disponibile (vedi **sez 3.1**).
Sono state applicati anche alcuni miglioramenti in relazione alle query sul database, quali la rimozione dei nodi al di fuori della relazione e l'ordinamento dei `RoadElement`.
Inoltre sono state ottimizzate le operazioni svolte tramite `Osmosis` sui dati OpenStreetMap (vedi **sez 3.2**). In particolare è stato impiegato l'utilizzo dei dump per velocizzare il caricamento sul database ed è stato applicato un filtro per velocizzare le operazioni successive.

Per come è stato strutturato lo script python e il parsing, dovrebbe risultare abbastanza semplice aggiungere nuove opzioni e ampliare ulteriormente il progetto.