Albert-Ludwig-University Freiburg

Chair of Algorithms and Data Structures

SQL-petrimaps: Visualizing Geospatial Data using PostgreSQL and PostGIS

Tobias Bürger

A thesis submitted in Partial Fulfillment
of the Requirements for the Degree of

Bachelor of Science (B.Sc.)

Supervisor:
Prof. Dr. Hannah Bast

Advisor:
Dr. Patrick Brosi

January 14, 2025

Declaration

I hereby declare, that I am the sole author and composer of my 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 also hereby declare that my thesis has not been prepared for another examination or assignment, either in its entirety or excerpts thereof.

14.01.2025 .... Tobias Bürger ....

Place, Date Signature

Abstract

petrimaps is a tool developed by the Chair of Algorithms and Data Structures at the Albert-Ludwig-University of Freiburg. *petrimaps* is able to visualize "hundreds of millions of geospatial query results while remaining responsive. This is in contrast to other tools that slow down or become unresponsive when the number of objects in the result is large" [1]. While this is a nice property, *petrimaps* is not yet a very stable or feature-rich tool. The original paper [1] proposes to support user-written SPARQL queries, displaying results as a heatmap or as full objects, and exporting the results as CSV, TSV or GeoJSON. Later, I implemented a responsive loading bar, additional user interface to conveniently send requests to the backend and the ability to upload GeoJSON files. This thesis introduces *SQL-petrimaps* to close a major gap in the list of supported features. We look at how we can evaluate the query results of any SQL query sent to a PostgreSQL database and use *petrimaps* as the platform to visualize contained geospatial data. We benchmark *SQL-petrimaps* using multiple datasets.

Acknowledgements

I would like to thank Prof. Dr. Hannah Bast for supervising this thesis and for giving me the opportunity to complete it in this line of research. I would also like to thank Dr. Patrick Brosi for his personal support and for his suggestions. In addition, I would like to thank Dr. Rainer Ullmann, Thomas Bürger, Manuel Berger and Sofie Hofmann for proofreading this thesis.

Contents

List of Figures vi

1. Introduction 1
 1.1. Motivation . . . 1
 1.2. Problem . . . 1
 1.3. Approach . . . 2

2. Background 3
 2.1. PostgreSQL . . . 3
 2.2. PostgreSQL: OIDs . . . 3
 2.3. PostGIS . . . 4
 2.4. PostGIS: `ST_AsText` . . . 4
 2.5. SQL queries . . . 4
 2.6. Querying tables . . . 5
 2.7. libpqxx . . . 5
 2.8. Frontend and backend . . . 5
 2.9. GIS . . . 5
 2.10. pgAdmin 4 . . . 6
 2.11. osm2pgsql . . . 6

3. Related Work 7
 3.1. QGIS . . . 7
 3.2. GRASS GIS . . . 8
 3.3. OpenJUMP . . . 8

4. Methods **12**
4.1. *Expanding* select statements . . . 12
4.2. *Expanding* select statements: Example . . . 14
4.3. SQL queries in *petrimaps*: General steps . . . 15
4.3.1. Frontend: SQL query input . . . 15
4.3.2. Backend: Server . . . 16
4.3.3. Backend: SQLCache . . . 16
4.3.4. Backend: SQLRequestor . . . 16
4.3.5. Backend: Server . . . 16
4.3.6. Frontend: Map update . . . 17
4.4. SQLCache: General steps . . . 17
4.4.1. Retrieving table names and column type names . . . 18
Retrieving table names by OIDs . . . 18
Retrieving column type names by OIDs . . . 19
4.4.2. Expanding select statements . . . 20
4.4.3. Building the final query . . . 21
4.4.4. Saving RAM . . . 21
Parsing the result table in batches . . . 21
Querying non-geometry data . . . 22
4.4.5. Building the geometry count query . . . 22
4.4.6. Parsing the geometry WKTs . . . 23

5. Benchmarking **27**
5.1. Execution times . . . 28
5.1.1. Small-sized dataset . . . 28
Low-complexity queries . . . 28
Medium-complexity queries . . . 31
High-complexity queries . . . 34
5.1.2. Medium-sized dataset . . . 37
Low-complexity queries . . . 38
Medium-complexity queries . . . 39
High-complexity queries . . . 41

5.1.3. Large-sized dataset 42
Low-complexity queries 43
Medium-complexity queries 45
High-complexity queries 47
5.2. Results 49

6. Conclusion and Outlook 50
6.1. Conclusion 50
6.2. Outlook 50

7. Bibliography 52

Appendix A. Specifications 54
A.0.1. Local Machine 54
A.0.2. Software 54

List of Figures

3.1. The import dialog to visualize user-written SQL queries in QGIS 9
3.2. The import dialog to visualize tables in GRASS GIS 10
3.3. The import dialog to visualize user-written SQL queries in Open-JUMP 11

4.1. Expanding select statements: Example query result table 14
4.2. Sequence diagram displaying the general steps of *petrimaps* . . . 17
4.3. The result table of the example *Table Names* query 19
4.4. The result table of the example *Column Type Names* query . . . 20
4.5. Inspecting a Point geometry by displaying non-geometry data . 25
4.6. Example geometries and their WKTs for all geometry types . . 26
4.7. Example of parsing a MultiPoint 26

1. Introduction

1.1. Motivation

Geospatial analysis has become increasingly popular and important. As more and more large geospatial datasets become publicly available, the need to efficiently visualize large amounts of geospatial data has grown. *petrimaps* provides this functionality and is easy to use. However, it lacks important features to be versatile enough. One of these missing features is the use of SQL queries to retrieve data from an SQL database, as SQL databases have always been a popular choice for storing data sets of any size in a structured way. My contribution to this platform, *SQL-petrimaps*, fills this gap. It is capable of extracting geospatial data resulting from any valid SQL query and enables *petrimaps* to visualize around 30 000 000 geometries in 55 minutes.

1.2. Problem

By allowing the user to enter any valid SQL query, we have to deal with several problems. If the query returns a result table, we need to find out which of the selected columns contain geospatial data. We also need to retrieve that geospatial data in a format that can be parsed into a data structure used internally by *petrimaps*. Since the backend is coded in C++ for performance reasons, we use the official C++ client API for PostgreSQL, libpqxx, to communicate with a PostgreSQL database. This PostgreSQL database uses the PostGIS extension to handle geospatial data. If we send the user-written query

unmodified, the geospatial data in the result table will be encoded as binary objects that cannot be easily parsed. The official PostGIS workshop suggests that external programs should not try to use this internal representation and convert it to another format instead [2]. This means, we have to use one of the helper functions that PostGIS provides and rewrite the query to get a parsable representation in the result table. *petrimaps* uses WKT (Well-Known Text) by default, so we choose the `ST_AsText` helper function to convert the binary objects to their WKT representation. The problem with using these helper functions is that they can only take a single column as an argument. This leads to two points: Firstly, we need to know beforehand which columns contain geometry so that we can restrict calls of `ST_AsText` to them. Secondly, and this is the bigger problem, we have to rewrite *-select statements in the query because a column in the result table selected by a *-select statement can contain geometry.

1.3. Approach

SQL-petrimaps executes a series of SQL queries in order to write a final query that returns almost the same result table as the original user-written query. The only, but important, difference is that this result table of the final query contains the WKT of the geometries instead. The general idea is to guarantee that each select statement of the final query maps to exactly one column of the result table. This way, we can always call `ST_AsText` on a select statement whenever it maps to a column that contains geometry. We also know that the columns in the result table will be in the same order as the select statements in the user-written query that select them. We can use this to our advantage, when rewriting the *-select statements. We call the process of rewriting the *-select statements *expanding*, because we are expanding one *-select statement into several new select statements.

2. Background

In this chapter we will look at related background and terms and see how they relate to the work done in this thesis. This way, we can better understand the tools we use to achieve our goal.

2.1. PostgreSQL

PostgreSQL is a free and open source database system that uses and extends the SQL language. Because of its long development history and robust feature set, it has become a popular choice among available SQL database systems [6]. SQL database systems operate with tables consisting of rows and columns that can be combined to retrieve related data.

2.2. PostgreSQL: OIDs

Object IDentifiers (OIDs) are used internally by PostgreSQL as primary keys for various system tables [7]. We use them in Sec. 4.4.1 to retrieve a text representation for the names of tables and column types.

2.3. PostGIS

PostGIS is an extension for PostgreSQL. It adds support for storing, indexing and querying geospatial data. This way, geospatial data can be both stored and processed efficiently. With its rich set of spatial functions, geospatial data can be analyzed conveniently [10].

2.4. PostGIS: `ST_AsText`

As mentioned in Sec. 1.2, we have to use a PostGIS helper function to retrieve geometry in a parsable format. We can choose between the WKB (Well-Known Binary) and WKT (Well-Known Text) representation. We can also retrieve a so-called SRID (Spatial Reference IDentifier) for each geometry that tells us how coordinates within the geometry have to be interpreted in terms of projection [16]. Because *petrimaps* uses WKT by default and expects coordinates as latitude and longitude, we choose the PostGIS helper function `ST_AsText` which converts geometry to WKT and omits the SRID [5].

2.5. SQL queries

Using SQL queries, we can communicate with our PostgreSQL database. They are mostly used to create new tables, alter existing tables or to retrieve data from existing tables. In *SQL-petrimaps* we assume that we have a PostgreSQL database instance running that already stores all the data we need. Thus, we focus on user-written queries that retrieve data.

2.6. Querying tables

In order to retrieve data, we have to query tables. This means, that we have to specify which columns of which tables we want to select under which restrictions. The SQL standard offers a rich syntax to do this, so that we can always retrieve the data we want [8]. The retrieved data itself is again represented as a table that we call *result table*. We call all statements that select columns *select statements*.

2.7. libpqxx

Because *petrimaps* is being developed using C++, we also need a C++ library that allows us to send SQL queries to our PostgreSQL database and evaluate the result table. libpqxx is the official C++ client API for PostgreSQL and thus an obvious choice [13].

2.8. Frontend and backend

In a piece of software we differentiate between frontend and backend. The frontend is the part that is presented to the user, while the backend processes data in the background. In *petrimaps* we regard the website accessed through the browser as the frontend and the server we send requests to as the backend [14].

2.9. GIS

GIS stands for Geographic Information System. Any system that can handle geospatial data is a GIS [15]. In this thesis we consider a GIS to be a piece of software.

2.10. pgAdmin 4

pgAdmin 4 is an open source administration and development platform for PostgreSQL [12]. It provides a rich user interface to communicate with PostgreSQL. Within this thesis we use it primarily to execute SQL queries and to inspect their result tables.

2.11. osm2pgsql

"Osm2pgsql is an open source tool for importing OpenStreetMap (OSM) data into a PostgreSQL/PostGIS database" [11]. We use it to import the data of the large-sized dataset into PostgreSQL that we use in Sec. 5.1.3.

3. Related Work

There are only a few theses or papers available that are related to querying a PostgreSQL database in order to visualize the results. They usually also use other GIS to communicate with an SQL database and thus leave out most of the information on how the results are being retrieved. So instead, we look at those other relevant GIS and see how they perform when given the problem we want to solve in this thesis. There are several GIS available that can communicate with a PostgreSQL database that uses the PostGIS extension. But many of them do not allow user-written queries or expect a specific query structure (e.g. GeoServer and MapServer). Additionally, it is very difficult or impossible to gather information on how these GIS communicate with PostgreSQL. Most of them keep all the result data in RAM, which becomes a huge problem if this data becomes too big. We try to visualize the results of some user-written queries. Specifically, we try to visualize the table `planet_osm_line` from the large-sized dataset we use later on which contains more than 17 million geometries. Specifications of the local machine used for these tests are listed in Appx. A.

3.1. QGIS

QGIS allows us to connect to a PostgreSQL database, execute user-written SQL queries, and view the result table of the query. We can convert the result table to a new layer so that we can visualize the geometry data it contains. This conversion requires us to select exactly one column name from which to

extract the geometry. This process allows us to filter out unwanted geometry, and by creating multiple layers, we can still visualize all of the geometry. The downside is extra work. By default, *SQL-petrimaps* wants to assume that all selected columns are intentionally selected. There is no feedback about the importing progress and when trying to import `planet_osm_line`, there was not enough RAM available. The import dialog can be seen in Fig. 3.1 below.

3.2. GRASS GIS

GRASS GIS also allows us to connect to a PostgreSQL database and execute user-written SQL queries using `db.select`. But here, the geometry columns are omitted. We can create new layers using `v.external` to visualize entire database tables. We can see a console that displays the current importing progress. `planet_osm_line` was imported after roughly 22 minutes, but the user interface became too unresponsive after trying to inspect them. Overall, there is no functionality to visualize user-written SQL queries. The import dialog to visualize whole tables can be seen in Fig. 3.2 below.

3.3. OpenJUMP

OpenJUMP allows us to connect to a PostgreSQL database and execute user-written SQL queries. A new layer is created that visualizes the extracted geometries and we can inspect the result table of individual geometries. When trying to import `planet_osm_line`, there was not enough RAM available. Again, the import dialog can be seen in Fig. 3.3 below.

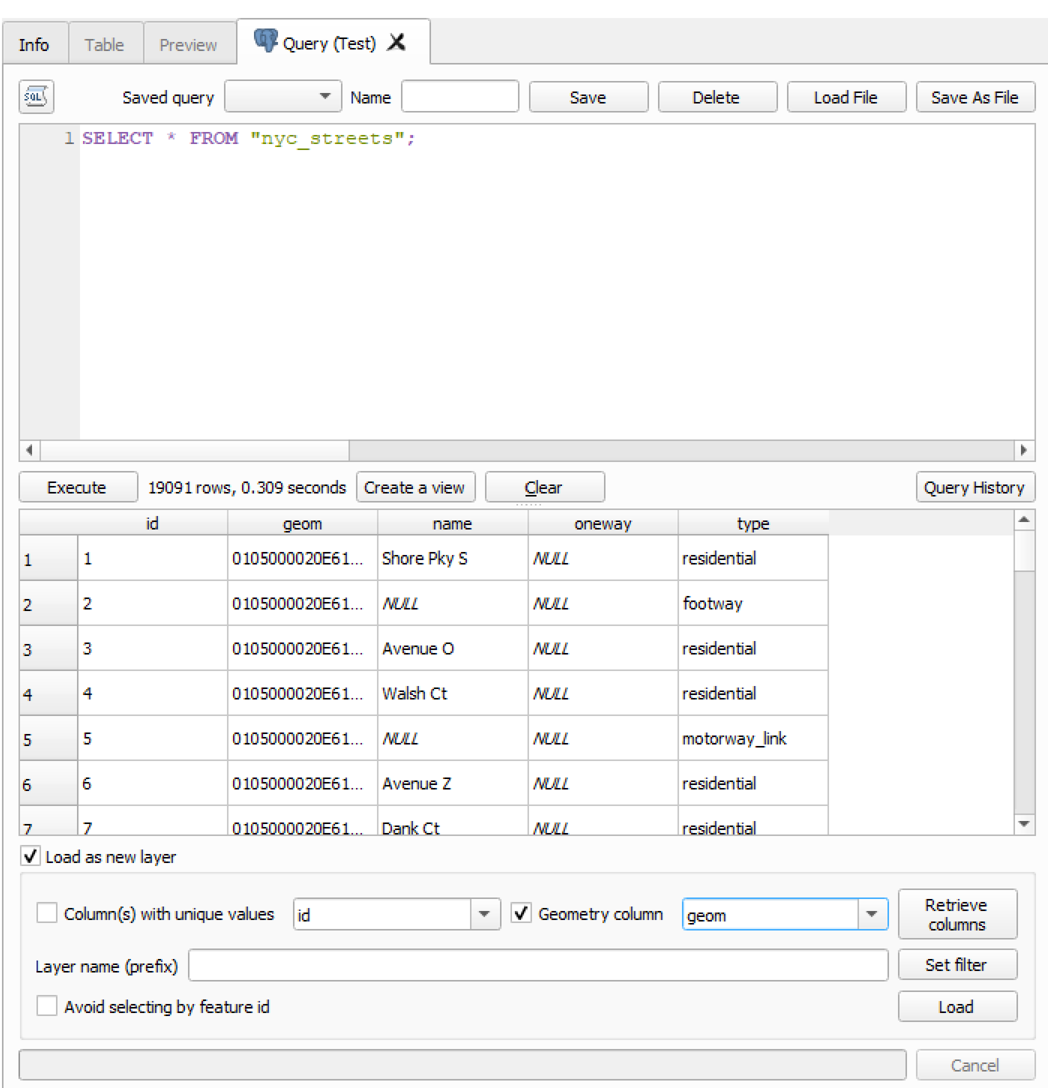

Figure 3.1: The import dialog to visualize user-written SQL queries in QGIS. We can see the user-written query in the top field. After we execute the user-written query, we can see the result table below. Then we can select a single column that contains geometry and load the geometry on a new layer.

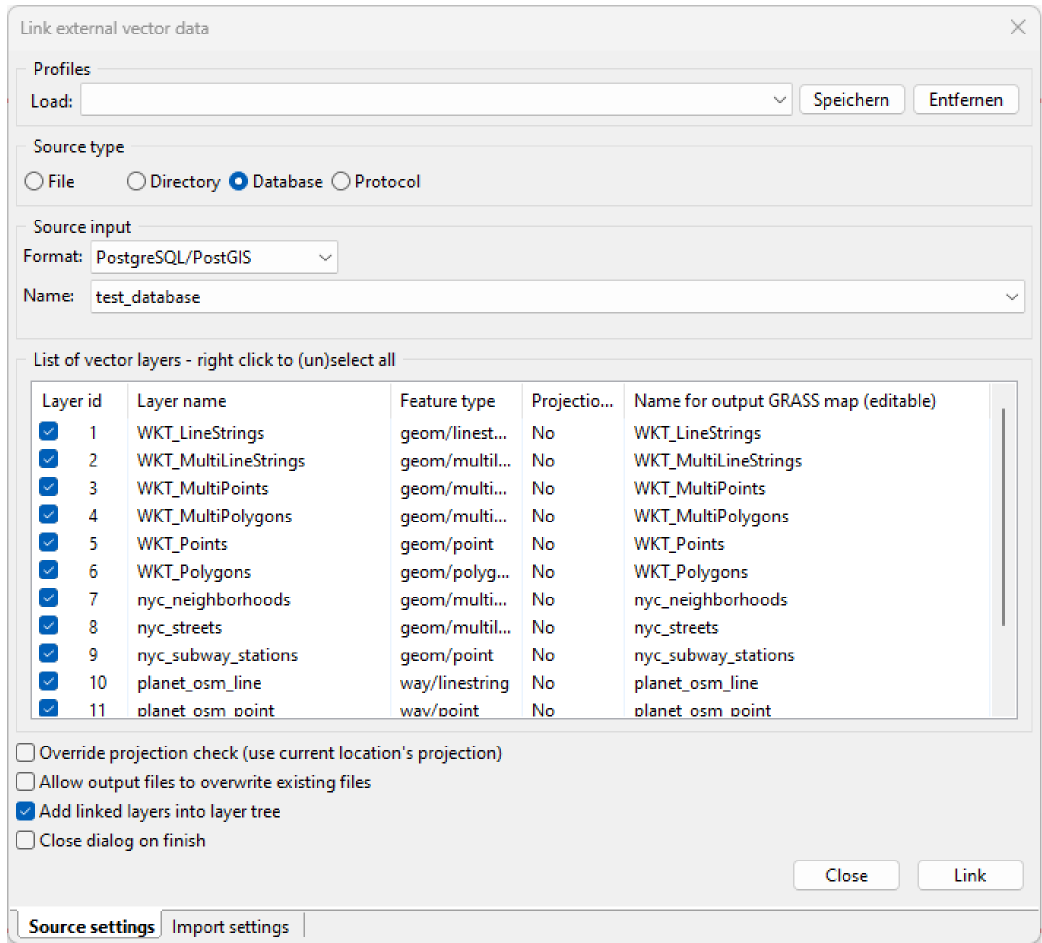

Figure 3.2: The import dialog to visualize tables in GRASS GIS. After we select a PostgreSQL database that uses PostGIS, we can see a list of all available tables. We can now select the tables we wish to visualize.

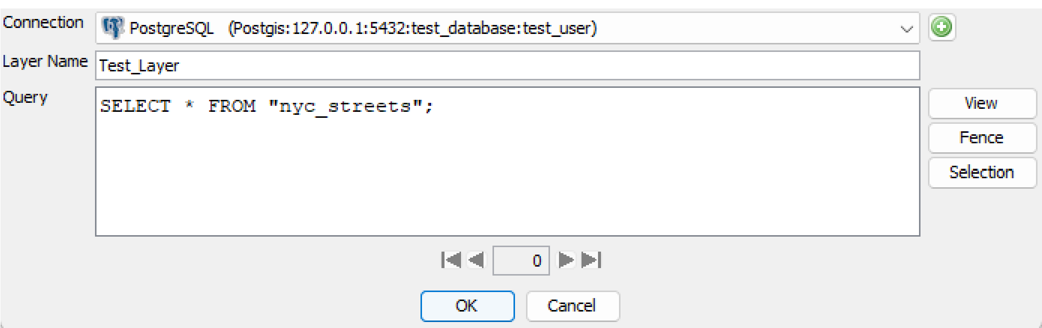

Figure 3.3: The import dialog to visualize user-written SQL queries in OpenJUMP. We have to select a connection to a PostgreSQL database that uses PostGIS, enter a name for the layer that will be created and enter a user-written query.

4. Methods

4.1. *Expanding* select statements

Our goal is to guarantee that one select statement maps to exactly one column name. This way, we know that we can safely call `ST_AsText` on a select statement if the column it maps to contains geometry. Therefore, we want to show that given a function m that maps the select statements of the user-written query to possibly multiple columns, we can obtain a new function that remaps equivalent select statements to exactly one column each. With equivalent select statements, we refer to select statements that collectively select the same columns as the select statements in the user-written query.

We define q to be a valid user-written query and Sel to be the select statements within q. Further, let Col be the column names of the result table we obtain by executing q. To stay analogous to vectors used in C++, we define them as tuples to keep order and to allow duplicates. We see that $|\text{Col}| \geq |\text{Sel}|$, because each select statement maps to at least one column. Thus, we define m to be a mapping from the index of a select statement in Sel to the indices of the column names in Col. There exist two cases now:

1. $\forall i \in \{0, 1, \ldots, |\text{Sel}| - 1\} : |m(i)| = 1$:
   Every select statement maps to exactly one column name. In this case m is the desired mapping.

2. $\exists i \in \{0, 1, \ldots, |\text{Sel}| - 1\} : |m(i)| > 1$:
There is at least one select statement that maps to more than one column. Thus, we have to construct a new mapping $\tilde{m}$. Because of the case condition, we know that $\forall j \in m(i) : \text{Col}(j)$ is not an alias. This is because SQL only allows aliases for select statements that map to a single column. Thus, we define $\ell_i := |m(i)|$ and can replace $\text{Sel}(i)$ in the query by $\text{Col}(m(i)_0))$, $\text{Col}(m(i)_1)$, $\ldots$, $\text{Col}(m(i)_{\ell-1})$. Knowing this, we can create equivalent select statements $\tilde{\text{Sel}}$:

$\forall i \in \{0, 1, \ldots, |\text{Sel}| - 1\}$:

a) if $|m(i)| = 1$:
$\tilde{\text{Sel}}(i) := (\text{Sel}(i))$

b) if $|m(i)| > 1$:
$\forall j \in m(i) : \tilde{\text{Sel}}(i)_j := \text{Col}(m(i)_j)$
Please note that a select statement could be prefixed with a table name that will be omitted in Col. If that is the case, we also add the table name as a prefix to $\text{Col}(m(i)_j)$ to prevent ambiguity.

$\tilde{\text{Sel}}$ has the same structure as m, so we can also regard m as a mapping from Sel to $\tilde{\text{Sel}}$. Because of how we defined $\tilde{\text{Sel}}$, we know that every select statement maps to exactly one column name. We also know that we have as much select statements in $\tilde{\text{Sel}}$ as we have column names in Col. This means that we can flatten $\tilde{\text{Sel}}$ to a tuple that does not contain tuples itself. This process can be seen in Sec. 4.2 below. We obtain our desired mapping $\tilde{m}$:
$\forall i \in \{0, 1, \ldots, |\tilde{\text{Sel}}_{\text{flat}}| - 1\} : \tilde{m}(i) := i$.

4.2. *Expanding* select statements: Example

Let us take a simple query q as an example:

```
SELECT "WKT_Points".*, "WKT_Points".geom AS geometry,
"WKT_LineStrings".* FROM "WKT_Points", "WKT_LineStrings";
```

The result table of q can be seen in Fig. 4.1.

	id integer	geom geometry	geometry geometry	id integer	geom geometry
1	0	01010000202...	01010000202...	0	01020000202...

Figure 4.1: The result table of the example query q shows how select statements are being expanded.

In this case we have:

$$\begin{aligned}
\text{Sel} &= (\texttt{"WKT_Points".*}, \texttt{"WKT_Points".geom AS geometry}, \\
&\qquad \texttt{"WKT_LineStrings".*}) \\
\text{Col} &= (\texttt{id}, \texttt{geom}, \texttt{geometry}, \texttt{id}, \texttt{geom}) \\
m &= ((0,1),(2),(3,4))
\end{aligned}$$

Because we have at least one select statement, two in this case, that map to more than one column name, we construct our new select statements $\tilde{\text{Sel}}$:

$$\begin{aligned}
\tilde{\text{Sel}} = (&(\texttt{"WKT_Points".id}, \texttt{"WKT_Points".geom}), \\
&(\texttt{"WKT_Points".geom AS geometry}), \\
&(\texttt{"WKT_LineStrings".id}, \texttt{"WKT_LineStrings".geom}))
\end{aligned}$$

We flatten $\tilde{\text{Sel}}$ to obtain:

$$\begin{aligned}\tilde{\text{Sel}}_{\text{flat}} = (&\texttt{"WKT_Points".id}, \texttt{"WKT_Points".geom},\\ &\texttt{"WKT_Points".geom AS geometry},\\ &\texttt{"WKT_LineStrings".id}, \texttt{"WKT_LineStrings".geom})\end{aligned}$$

Now we can define the trivial mapping $\tilde{m}$:
$\forall i \in \{0, 1, \ldots, |\tilde{\text{Sel}}_{\text{flat}}| - 1\} : \tilde{m}(i) \coloneqq i$.

4.3. SQL queries in *petrimaps*: General steps

We will now take a closer look at the approach used to obtain the geospatial data from any user-written query. But first, to understand where this takes place in the *petrimaps* architecture, we will take a look at the general steps that must be taken from receiving a user-written query to displaying the results on a map in chronological order. Fig. 4.2 also shows a sequence diagram that visualizes the process. Please note: these steps focus only on the most relevant aspects. Explaining everything in detail would go beyond the scope of this thesis.

4.3.1. Frontend: SQL query input

When the user opens the website, a menu will appear with three tabs. One for SPARQL queries, one for SQL queries and one for uploading GeoJSON files. By clicking on the SQL query tab, a small text editor and an execute button show up. When the user clicks on this button, the text inside the editor will be sent to the *petrimaps* backend as an SQL query request.

4.3.2. Backend: Server

In the backend, an instance of the Server class receives this request and handles it. A geometry cache SQLCache is created that receives the SQL query.

4.3.3. Backend: SQLCache

On creation, the SQLCache establishes a new connection to the PostgreSQL database using libpqxx. The credentials needed to establish this connection are hardcoded. Now, depending on the structure of the received SQL query, the SQLCache executes additional SQL queries and rewrites the original query such that it can parse the geometries in the final result table. Afterwards, the SQLCache uses multiple datastructures to store the parsed geospatial data. We will take an in-depth look at this step in Sec. 4.4.

4.3.4. Backend: SQLRequestor

The server creates a requestor SQLRequestor with a reference to the SQLCache that now holds the relevant data obtained from the query. This requestor is the bridge between the server and the SQLCache. It provides the relevant data from the SQLCache in a format that the server expects.

4.3.5. Backend: Server

The server returns a session ID to the frontend. This ID is used to reference the SQLRequestor. The frontend includes this session ID in further requests.

4.3.6. Frontend: Map update

Using the session ID, the frontend sends a heatmap request to the server. The server now renders a PNG image of a heatmap or the full objects and returns it. The frontend displays this image on top of the map.

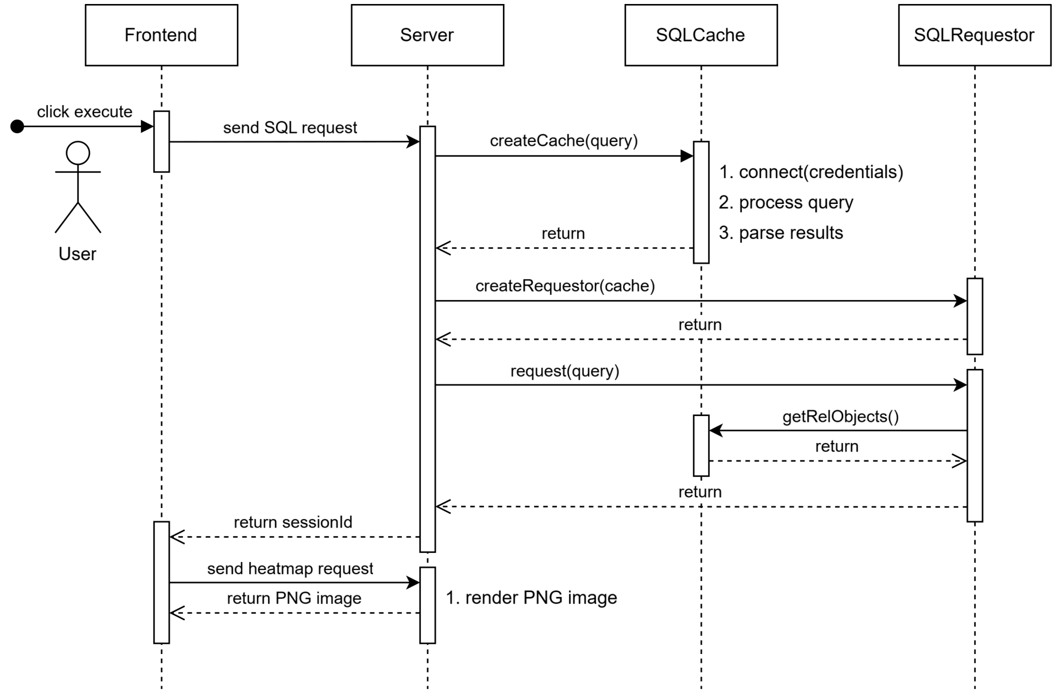

Figure 4.2: Sequence diagram displaying the general steps of *petrimaps*.

4.4. SQLCache: General steps

We will now take a look at the general steps on how the SQLCache processes the user-written SQL query. This corresponds to Sec. 4.3.3 above.
Our goal is to expand the select statements in the query so that we can build a new query that uses `ST_AsText`. But only expanding is not enough, because it doesn't tell us which select statements to call `ST_AsText` on. Thus, we also

have to find out which columns in the result table will contain geometry. We need table names to expand the select statements, and column type names to know which columns contain geometry. We will also see how we can alleviate the problem of not having enough RAM to store all the data from the result table.

4.4.1. Retrieving table names and column type names

Because we want the result table of the final query to contain the same columns as the result table of the user-written query, we can use the user-written query to find out which columns in the result table will contain geometry. Thus, we modify and execute the user-written query such that the result table contains no rows. We call this query *Table OIDs* query. For each column in the result table we can retrieve the OID of the table the column was selected from and the OID of the column type. For both we have to run an additional query to obtain a useful text-representation. Thus, we create a vector for each type of OID which takes linear time in the number of columns in the result table.

Retrieving table names by OIDs

PostgreSQL manages a table called `pg_class` internally. Using this table, we can obtain the table name for a given OID. This means that we have to write a query that results in a result table that contains exactly the table names of the OIDs we retrieved. We call this query *Table Names* query:

```
SELECT oid, relname FROM pg_class WHERE oid IN (OIDs);
```

This gives us a result table whose first column contains the OID, and second column the table name it refers to. We can now iterate all rows to create a mapping that maps an OID to a table name. This takes linear time in the number of unique OIDs we obtained, which is equivalent to the amount of unique table columns.

Example

Let us say we retrieved the table OIDs 65591, 65558, 65599. Our query then would be:

```
SELECT oid, relname FROM pg_class WHERE oid IN (65591, 65558, 65599);
```

The result table of this example query can be seen in Fig. 4.3.

	oid [PK] oid	relname name
1	65599	WKT_Polygons
2	65558	WKT_LineStrings
3	65591	WKT_Points

Figure 4.3: The result table of the example *Table Names* query.

Retrieving column type names by OIDs

Analogously, PostgreSQL also manages a table called `pg_type` internally. This time we write a query to obtain the column type names. We call this query *Column Type Names* query:

```
SELECT oid, typname FROM pg_type WHERE oid IN (OIDs);
```

This gives us a result table whose first column contains the OID, and second column the column type name it refers to. Again, we iterate all rows to create a mapping that maps an OID to a column type name. This also takes linear time in the number of unique OIDs we obtained.

Example
Let's say we retrieved the column type OIDs 23, 16392 and 1043. Our query then would be:

```
SELECT oid, typname FROM pg_type WHERE oid IN (23, 16392, 1043);
```

The result table of this example query can be seen in Fig. 4.4.

	oid [PK] oid	typname name
1	23	int4
2	1043	varchar
3	16392	geometry

Figure 4.4: The result table of the example *Column Type Names* query.

4.4.2. Expanding select statements

This corresponds to Sec. 4.1. *-select statements are the only select statements that can map to several columns in the result table in SQL. Thus, they are the only ones we have to expand. We filter them out using a regular expression. Then we build a query to execute them individually, again with zero rows in the result table, and use the column names in the result table as the new select statements in the final query. We call each of these queries *Expand Select Statement* query.

4.4.3. Building the final query

Using the expanded select statements and the column types, we can now build the final query. We iterate the expanded select statements and wrap `ST_AsText` around the ones where the column type is geometry. We keep everything except the select statements as it was in the user-written query.

Example

$$
\begin{aligned}
& \texttt{SELECT * FROM "WKT_Points";} \\
\text{Sel} &= (\texttt{*}) \\
\tilde{\text{Sel}}_{\text{flat}} &= (\texttt{"WKT_Points".id}, \texttt{"WKT_Points".geom}) \\
\text{final} &= \texttt{SELECT "WKT_Points".id,} \\
& \texttt{ST_AsText("WKT_Points".geom)} \\
& \texttt{FROM "WKT_Points";}
\end{aligned}
$$

4.4.4. Saving RAM

As we have seen in Ch. 3, it is important to save RAM in order to ensure that even huge amounts of data can be processed. We will take a look at two measures we take in order to save a lot of RAM while processing a user-written query.

Parsing the result table in batches

By default, libpqxx creates an object that represents the entire result table of the user-written query. We use this object to parse the geometries contained within. But we don't need the entire result table available at all times in

order to parse the geometries. Therefore, we split the result table into several batches and only keep one batch in memory at a time. We use the class `stateless_cursor` libpqxx provides to achieve this.

Querying non-geometry data

When the user wants to inspect a geometry, we have to send the non-geometry data of the result table row where we obtained the geometry from to the frontend. If we stored all of this non-geometry data for every geometry in RAM, we would need a lot of space, especially if we had to deal with millions of geometries. Instead, we don't save this data at all and request it from PostgreSQL on demand. This comes with a problem: when there is no specific order defined within a query, PostgreSQL does not guarantee that the rows of the result table will always be in the same order for this query. Therefore, when we want to query a specific row, we cannot be sure if the row data we obtain really belongs to the geometry the user wants to inspect.

To solve this problem we add `ROW_NUMBER() OVER ()` to the select statements of the final query. It adds an extra column to the result table that enables us to uniquely identify each row by a row number. When we parse the result table of the final query we can retrieve this row number for every geometry. Later, when the user wants to inspect a geometry, we can send another query that filters out the row that contains the row number for that geometry. Each row always receives the same row number when the final query is executed.

4.4.5. Building the geometry count query

petrimaps displays a loading bar to the frontend that shows how much geometry we have to load in total and how much we have already loaded. In order to provide this information, we have to execute an additional query that tells us the amount of geometry we will load in total. We do this before we execute the final query. In the result table of any SQL query, all columns will always

have the same number of rows. This means we can just count the amount of rows of the result table of the final query and multiply it by the number of columns that contain geometry. We do this by using the SQL `COUNT` function which counts the rows of a column.

Example

```
              SELECT *, geom FROM "WKT_Points";
q_geomCount = SELECT COUNT(*)
              FROM (SELECT *, geom FROM "WKT_Points")
              AS finalQuery;
```

In this case we count one row. Because we have two columns containing geometry, we know that there are two geometries in total.

4.4.6. Parsing the geometry WKTs

After we execute the final query we obtain a result table containing the WKT of all geometries. For each row we want to parse every WKT into a datastructure that *petrimaps* uses internally to represent geometry of a specific type. We do not use the non-geometry data here, but we still want to be able to retrieve it with an additional query later on. We call this query *Non-geometry* query. This way, we can display it to the user as seen in Fig. 4.5.

There are three types of geometry: Points, LineStrings and Polygons. For every type of geometry, there is also a corresponding "Multi"-version. It can store multiple geometries of the corresponding geometry type. Additionally, there is also GeometryCollection which serves as a collection of geometries of any types. Within the WKT of a geometry the coordinates are stored in parenthesis, each axis separated by a whitespace. In *petrimaps* we only consider 2D-coordinates.

A point consists of one coordinate, a linestring of several points and a polygon of at least one closed linestring. Example geometries and WKTs can be seen in Fig. 4.6.

We iterate the WKT and parse the coordinates as 2D-points used internally by *petrimaps*. For every geometry type we have a method that interprets the obtained coordinates differently. For a "Multi"-version we reuse the method that parses a single geometry until we find a closing parenthesis. In the end, we only have to iterate the WKT once: each method returns the position in the WKT where it finished parsing a geometry. The next method continues from there. So the time needed to parse a WKT is linear in its length.

Example

Let us take a look at an example. In Fig. 4.7 we can see a MultiPoint geometry consisting of 4 coordinates. Because the WKT starts with the substring "MULTIPOINT", we call the method `parseMultiPoint`. The "10" corresponds to the position where this substring ends and thus tells the method where to start. Because a MultiPoint consists of several Points, we now call the method `parsePoint` until we find a closing parenthesis at the very end. `parsePoint` tells `parseMultiPoint` where it stopped parsing.

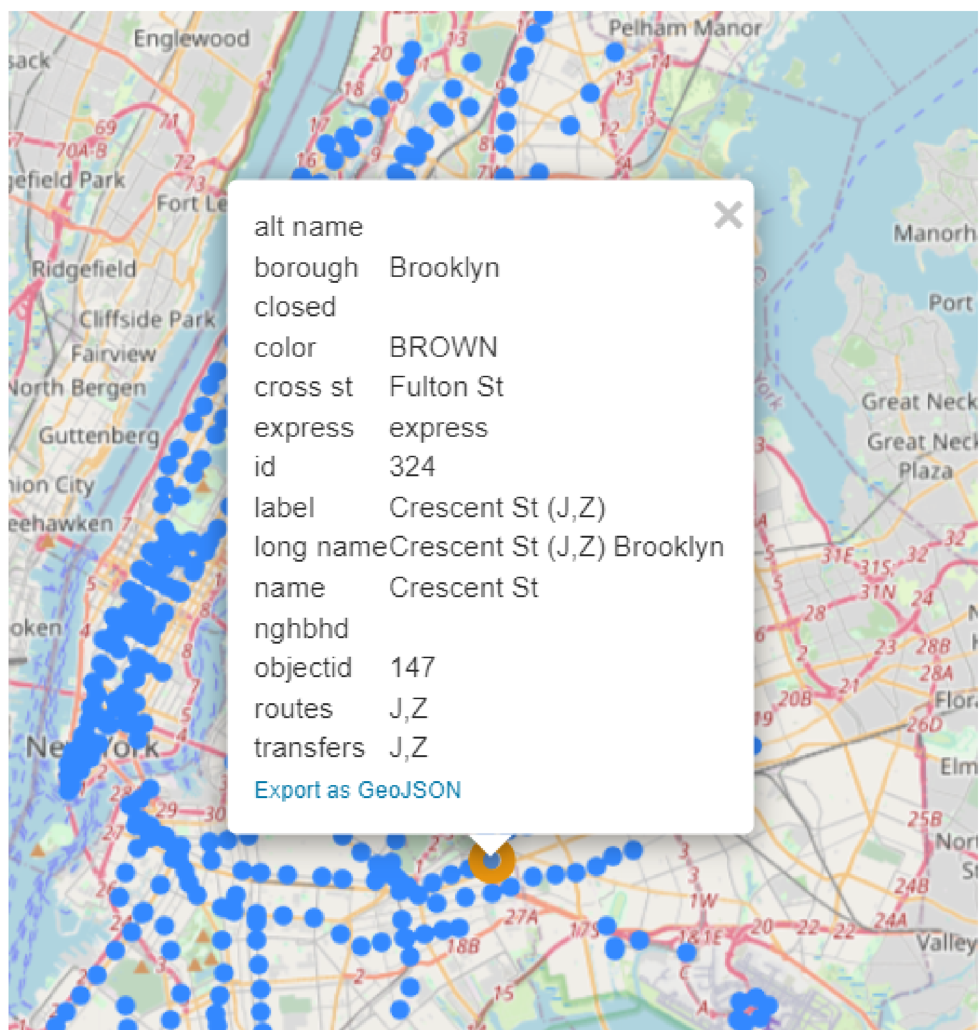

Figure 4.5: Inspecting a Point geometry by displaying non-geometry data. For each column that does not contain geometry we can see the column name on the left and the value of that column in the row of the geometry on the right.

Geometry primitives (2D)

Type	Examples
Point	POINT (30 10)
LineString	LINESTRING (30 10, 10 30, 40 40)
Polygon	POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))
	POLYGON ((35 10, 45 45, 15 40, 10 20, 35 10), (20 30, 35 35, 30 20, 20 30))

Multipart geometries (2D)

Type	Examples
MultiPoint	MULTIPOINT ((10 40), (40 30), (20 20), (30 10))
	MULTIPOINT (10 40, 40 30, 20 20, 30 10)
MultiLineString	MULTILINESTRING ((10 10, 20 20, 10 40), (40 40, 30 30, 40 20, 30 10))
MultiPolygon	MULTIPOLYGON (((30 20, 45 40, 10 40, 30 20)), ((15 5, 40 10, 10 20, 5 10, 15 5)))
	MULTIPOLYGON (((40 40, 20 45, 45 30, 40 40)), ((20 35, 10 30, 10 10, 30 5, 45 20, 20 35), (30 20, 20 15, 20 25, 30 20)))
GeometryCollection	GEOMETRYCOLLECTION (POINT (40 10), LINESTRING (10 10, 20 20, 10 40), POLYGON ((40 40, 20 45, 45 30, 40 40)))

Figure 4.6: Example geometries and their WKTs for all geometry types [16].

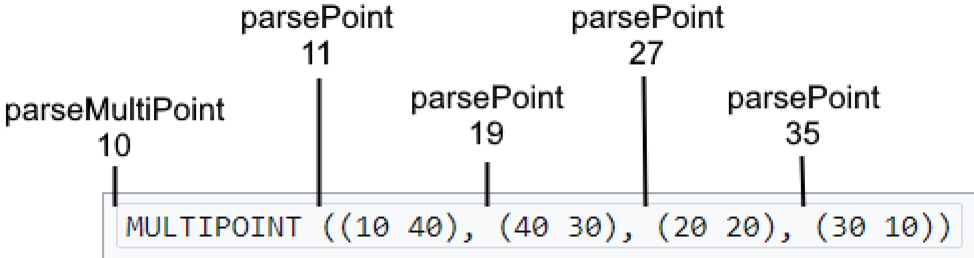

Figure 4.7: Example of parsing a MultiPoint. On top we can see the methods we call with the positions in the WKT.

5. Benchmarking

We benchmark *SQL-petrimaps* using different SQL queries on three different datasets. The queries vary from low complexity to high complexity, while the datasets vary primarily in the amount of geometries per table. We want to compare the time PostgreSQL needs to execute all queries with the time needed to only execute the user-written query. We omit the *Geometry Count* query, because it has to be executed in either case. Additionally, we merge the *Table Names* and *Column Type Names* query to save space and also separately include the *Non-geometry* query execution time. We use the PostgreSQL statement `EXPLAIN` with the arguments `ANALYZE` and `TIMING` for the query analysis. This way, we can retrieve the actual time needed for execution.

Furthermore, we also want to show that *SQL-petrimaps* remains useable, even when processing huge amounts of data. For this, we measure the time needed from when the SQLCache receives the SQL query until the geometries have been parsed. We do this by comparing two timestamps. We also take a look at the total RAM usage. Because *petrimaps* and PostgreSQL run on WSL, this is the total RAM WSL takes up during that process. Because there are a lot of other processes within WSL that use RAM apart from *petrimaps* and PostgreSQL, the measured values are only supposed to give us an idea on how the RAM usage scales with the amount of data we query. Before executing a user-written query we always restart WSL to ensure that there is no RAM still occupied because of the previous query. During these tests, *petrimaps* and PostgreSQL run on the same local machine whose specifications you can find in Appx. A. The machine used to perform these tests has a major influence on the results.

Please note that the measured execution times are only typical outcomes. The query execution times can vary for each query execution depending on the query plan PostgreSQL builds to handle it. This variation scales with the time measured. Therefore, we have to expect variations from a few milliseconds for queries that run fast up to variations of a few seconds for queries that run slow. We also have to keep in mind that the measured execution times for the SQLCache depend on all query execution times and will therefore accumulate the variations. In addition, the measured SQL query times are measured separately and not when the queries are executed through the SQLCache due to technical reasons.

5.1. Execution times

5.1.1. Small-sized dataset

The small-sized dataset contains a table for each type of geometry and each table contains a few sample geometries. These are the geometries we used as examples earlier and which can be found on Wikipedia [16].

Low-complexity queries

```
SELECT * FROM "WKT_Points";
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
0.006 ms	1.001 ms	0.046 ms	1.001 ms	0.013 ms	2.061 ms	51.08 ms

Total RAM usage: 2.715 GB
Total geometries: 1
Non-geometry query: 0.015 ms

`SELECT * FROM "WKT_LineStrings";`

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
0.006 ms	1.001 ms	0.03 ms	1.001 ms	0.013 ms	2.045 ms	27.09 ms

Total RAM usage: 2.689 GB
Total geometries: 1
Non-geometry query: 0.017 ms

`SELECT * FROM "WKT_Polygons";`

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
0.004 ms	1.001 ms	0.03 ms	1.001 ms	0.014 ms	2.046 ms	24.97 ms

Total RAM usage: 2.673 GB
Total geometries: 2
Non-geometry query: 0.015 ms

```
SELECT * FROM "WKT_MultiPoints";
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
0.004 ms	1.001 ms	0.027 ms	1.001 ms	0.013 ms	2.042 ms	28.98 ms

Total RAM usage: 2.679 GB

Total geometries: 8

Non-geometry query: 0.015 ms

```
SELECT * FROM "WKT_MultiLineStrings";
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
0.007 ms	1.001 ms	0.03 ms	1.001 ms	0.015 ms	2.047 ms	18.37 ms

Total RAM usage: 2.639 GB

Total geometries: 2

Non-geometry query: 0.014 ms

```
SELECT * FROM "WKT_MultiPolygons";
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
0.002 ms	1.001 ms	0.032 ms	1.001 ms	0.014 ms	2.048 ms	26.18 ms

Total RAM usage: 2.643 GB
Total geometries: 4
Non-geometry query: 0.018 ms

Medium-complexity queries

```
SELECT "WKT_Points".*, "WKT_LineStrings".geom FROM "WKT_Points"
JOIN "WKT_LineStrings" ON "WKT_Points".id = "WKT_LineStrings".id;
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
0.008 ms	2.001 ms	0.061 ms	2.001 ms	0.015 ms	4.078 ms	20.68 ms

Total RAM usage: 2.639 GB
Total geometries: 2
Non-geometry query: 0.018 ms

```
SELECT "WKT_LineStrings".*, "WKT_Polygons".geom FROM
"WKT_LineStrings" JOIN "WKT_Polygons"
ON "WKT_LineStrings".id = "WKT_Polygons".id;
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
0.01 ms	2.001 ms	0.061 ms	2.001 ms	0.016 ms	4.079 ms	22.8 ms

Total RAM usage: 2.641 GB
Total geometries: 2
Non-geometry query: 0.025 ms

```
SELECT "WKT_Polygons".*, "WKT_MultiPoints".geom FROM "WKT_Polygons"
JOIN "WKT_MultiPoints" ON "WKT_Polygons".id = "WKT_MultiPoints".id;
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
0.01 ms	2.001 ms	0.06 ms	2.001 ms	0.026 ms	4.088 ms	24.25 ms

Total RAM usage: 2.639 GB
Total geometries: 10
Non-geometry query: 0.045 ms

```
SELECT "WKT_MultiPoints".*, "WKT_MultiLineStrings".geom
FROM "WKT_MultiPoints" JOIN "WKT_MultiLineStrings"
ON "WKT_MultiPoints".id = "WKT_MultiLineStrings".id;
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
0.01 ms	2.001 ms	0.059 ms	2.001 ms	0.02 ms	4.081 ms	19.37 ms

Total RAM usage: 2.639 GB
Total geometries: 6
Non-geometry query: 0.02 ms

```
SELECT "WKT_MultiLineStrings".*, "WKT_MultiPolygons".geom
FROM "WKT_MultiLineStrings" JOIN "WKT_MultiPolygons"
ON "WKT_MultiLineStrings".id = "WKT_MultiPolygons".id;
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
0.01 ms	2.001 ms	0.06 ms	2.001 ms	0.017 ms	4.079 ms	24.55 ms

Total RAM usage: 2.639 GB
Total geometries: 4
Non-geometry query: 0.034 ms

```
SELECT "WKT_MultiPolygons".*, "WKT_Points".geom
FROM "WKT_MultiPolygons" JOIN "WKT_Points"
ON "WKT_MultiPolygons".id = "WKT_Points".id;
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
0.011 ms	2.001 ms	0.086 ms	2.001 ms	0.017 ms	4.105 ms	21.5 ms

Total RAM usage: 2.627 GB
Total geometries: 3
Non-geometry query: 0.044 ms

High-complexity queries

```
SELECT PointsEvenID.*, "WKT_LineStrings".geom
FROM (SELECT * FROM "WKT_Points" WHERE id % 2 = 0)
AS PointsEvenId FULL JOIN "WKT_LineStrings"
ON PointsEvenID.id = "WKT_LineStrings".id;
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
0.015 ms	2.001 ms	0.076 ms	2.001 ms	0.029 ms	4.122 ms	19.31 ms

Total RAM usage: 2.635 GB
Total geometries: 2
Non-geometry query: 0.051 ms

```
SELECT LineStringsEvenId.*, "WKT_Polygons".geom
FROM (SELECT * FROM "WKT_LineStrings" WHERE id % 2 = 0)
AS LineStringsEvenId FULL JOIN "WKT_Polygons"
ON LineStringsEvenId.id = "WKT_Polygons".id;
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
0.04 ms	2.001 ms	0.063 ms	2.001 ms	0.03 ms	4.095 ms	27.07 ms

Total RAM usage: 2.631 GB
Total geometries: 3
Non-geometry query: 0.133 ms

```
SELECT PolygonsEvenId.*, "WKT_MultiPoints".geom
FROM (SELECT * FROM "WKT_Polygons" WHERE id % 2 = 0)
AS PolygonsEvenId FULL JOIN "WKT_MultiPoints"
ON PolygonsEvenId.id = "WKT_MultiPoints".id;
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
0.017 ms	2.001 ms	0.062 ms	2.001 ms	0.023 ms	4.087 ms	19.35 ms

Total RAM usage: 2.629 GB
Total geometries: 9
Non-geometry query: 0.074 ms

```
SELECT MultiPointsEvenId.*, "WKT_MultiLineStrings".geom
FROM (SELECT * FROM "WKT_MultiPoints" WHERE id % 2 = 0)
AS MultiPointsEvenId FULL JOIN "WKT_MultiLineStrings"
ON MultiPointsEvenId.id = "WKT_MultiLineStrings".id;
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
0.017 ms	2.001 ms	0.062 ms	2.001 ms	0.024 ms	4.088 ms	17.18 ms

Total RAM usage: 2.631 GB
Total geometries: 6
Non-geometry query: 0.068 ms

```
SELECT MultiLineStringsEvenId.*, "WKT_MultiPolygons".geom
FROM (SELECT * FROM "WKT_MultiLineStrings" WHERE id % 2 = 0)
AS MultiLineStringsEvenId FULL JOIN "WKT_MultiPolygons"
ON MultiLineStringsEvenId.id = "WKT_MultiPolygons".id;
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
0.017 ms	2.001 ms	0.068 ms	2.001 ms	0.025 ms	4.095 ms	30.92 ms

Total RAM usage: 2.629 GB
Total geometries: 6
Non-geometry query: 0.057 ms

```
SELECT MultiPolygonsEvenId.*, "WKT_Points".geom
FROM (SELECT * FROM "WKT_MultiPolygons" WHERE id % 2 = 0)
AS MultiPolygonsEvenId FULL JOIN "WKT_Points"
ON MultiPolygonsEvenId.id = "WKT_Points".id;
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
0.018 ms	2.001 ms	0.092 ms	2.001 ms	0.023 ms	4.117 ms	21.92 ms

Total RAM usage: 2.627 GB
Total geometries: 3
Non-geometry query: 0.053 ms

5.1.2. Medium-sized dataset

The medium-sized data set uses data from the data bundle obtained from the official PostGIS Workshop [3]. This data bundle contains data about New York City. We import it into PostgreSQL and obtain three tables called `nyc_subway_stations`, `nyc_streets` and `nyc_neighborhoods`. `nyc_streets` approximately contains 19 000 geometries.

Low-complexity queries

```
SELECT * FROM "nyc_neighborhoods";
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
0.028 ms	1.001 ms	0.078 ms	1.001 ms	0.613 ms	2.693 ms	31.07 ms

Total RAM usage: 2.629 GB

Total geometries: 159

Non-geometry query: 0.549 ms

```
SELECT * FROM "nyc_streets";
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
1.609 ms	1.002 ms	0.087 ms	1.001 ms	15.2 ms	17.29 ms	0.254 s

Total RAM usage: 2.637 GB

Total geometries: 19 091

Non-geometry query: 19.21 ms

```
SELECT * FROM "nyc_subway_stations";
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
0.044 ms	1.001 ms	0.069 ms	0.002 ms	0.222 ms	1.294 ms	23.05 ms

Total RAM usage: 2.643 GB
Total geometries: 491
Non-geometry query: 0.508 ms

Medium-complexity queries

```
SELECT "nyc_neighborhoods".*, "nyc_streets".geom
FROM "nyc_neighborhoods" JOIN "nyc_streets"
ON "nyc_neighborhoods".id = "nyc_streets".id;
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
0.101 ms	2.002 ms	0.106 ms	2.001 ms	0.669 ms	4.778 ms	26.44 ms

Total RAM usage: 2.641 GB
Total geometries: 288
Non-geometry query: 0.696 ms

```
SELECT "nyc_streets".*, "nyc_subway_stations".geom
FROM "nyc_streets" JOIN "nyc_subway_stations"
ON "nyc_streets".id = "nyc_subway_stations".id;
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
3.287 ms	2.002 ms	0.1 ms	2.002 ms	3.829 ms	7.993 ms	34.56 ms

Total RAM usage: 2.643 GB
Total geometries: 982
Non-geometry query: 4.046 ms

```
SELECT "nyc_subway_stations".*, "nyc_neighborhoods".geom
FROM "nyc_subway_stations" JOIN "nyc_neighborhoods"
ON "nyc_subway_stations".id = "nyc_neighborhoods".id;
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
0.167 ms	2.001 ms	0.118 ms	2.002 ms	0.666 ms	4.787 ms	24.89 ms

Total RAM usage: 2.645 GB
Total geometries: 284
Non-geometry query: 0.756 ms

High-complexity queries

```
SELECT NeighborhoodsEvenId.*, "nyc_streets".geom
FROM (SELECT * FROM "nyc_neighborhoods" WHERE id % 2 = 0)
AS NeighborhoodsEvenId FULL JOIN "nyc_streets"
ON NeighborhoodsEvenId.id = "nyc_streets".id;
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
3.357 ms	2.003 ms	0.099 ms	2.001 ms	18.89 ms	22.99 ms	0.251 s

Total RAM usage: 2.641 GB
Total geometries: 19 160
Non-geometry query: 18.639 ms

```
SELECT StreetsEvenId.*, "nyc_subway_stations".geom
FROM (SELECT * FROM "nyc_streets" WHERE id % 2 = 0)
AS StreetsEvenId FULL JOIN "nyc_subway_stations"
ON StreetsEvenId.id = "nyc_subway_stations".id;
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
3.234 ms	2.003 ms	0.11 ms	2.002 ms	11.1 ms	15.22 ms	0.147 s

Total RAM usage: 2.637 GB
Total geometries: 10 036
Non-geometry query: 11.532 ms

```
SELECT SubwayStationsEvenId.*, "nyc_neighborhoods".geom
FROM (SELECT * FROM "nyc_subway_stations" WHERE id % 2 = 0)
AS SubwayStationsEvenId FULL JOIN "nyc_neighborhoods"
ON SubwayStationsEvenId.id = "nyc_neighborhoods".id;
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
0.35 ms	2.002 ms	0.101 ms	2.001 ms	0.836 ms	4.94 ms	25.12 ms

Total RAM usage: 2.635 GB
Total geometries: 406
Non-geometry query: 0.872 ms

5.1.3. Large-sized dataset

Last but not least, we use the South America dataset from GeoFabrik as the third dataset [9]. It contains a lot of data from the OpenStreetMap database. This data is available under the Open Database License [4]. We import the data into PostgreSQL using osm2pgsql and obtain four tables called `planet_osm_line`, `planet_osm_point`, `planet_osm_polygon` and `planet_osm_roads`. `planet_osm_polygon` approximately contains 25 million geometries.

Low-complexity queries

```
SELECT * FROM "planet_osm_line";
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
18.42 s	1.001 ms	0.084 ms	1.001 ms	44.21 s	44.21 s	8 m 13 s

Total RAM usage: 23.252 GB
Total geometries: 17 383 566
Non-geometry query: 48.55 s

```
SELECT * FROM "planet_osm_point";
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
0.63 s	1.001 ms	0.108 ms	1.001 ms	5.49 s	5.49 s	1 m 3 s

Total RAM usage: 5.730 GB
Total geometries: 10 682 528
Non-geometry query: 5.82 s

```
SELECT * FROM "planet_osm_polygon";
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
18.27 s	1.001 ms	0.07 ms	1.001 ms	48.46 s	48.46 s	53 m 58 s

Total RAM usage: 23.116 GB
Total geometries: 25 351 494
Non-geometry query: 1 m 6 s

```
SELECT * FROM "planet_osm_roads";
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
0.14 s	1.001 ms	0.112 ms	1.001 ms	6.85 s	6.85 s	1 m 25 s

Total RAM usage: 6.238 GB
Total geometries: 1 193 908
Non-geometry query: 6.28 s

Medium-complexity queries

```
SELECT "planet_osm_line".*, "planet_osm_point".way
FROM "planet_osm_line" JOIN "planet_osm_point"
ON "planet_osm_line".id = "planet_osm_point".id;
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
16.69 s	2.001 ms	0.15 ms	2.001 ms	35.02 s	35.02 s	6 m 39 s

Total RAM usage: 21.867 GB
Total geometries: 21 365 056
Non-geometry query: 1 m 56 s

```
SELECT "planet_osm_point".*, "planet_osm_polygon".way
FROM "planet_osm_point" JOIN "planet_osm_polygon"
ON "planet_osm_point".id = "planet_osm_polygon".id;
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
40.17 s	2.001 ms	0.291 ms	2.001 ms	47.73 s	47.73 s	13 m 14 s

Total RAM usage: 20.337 GB
Total geometries: 21 365 056
Non-geometry query: 2 m 26 s

```
SELECT "planet_osm_polygon".*, "planet_osm_roads".way
FROM "planet_osm_polygon" JOIN "planet_osm_roads"
ON "planet_osm_polygon".id = "planet_osm_roads".id;
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
1.48 s	2.001 ms	0.143 ms	2.001 ms	10.38 s	10.39 s	9 m 12 s

Total RAM usage: 16.151 GB
Total geometries: 2 387 816
Non-geometry query: 11.79 s

```
SELECT "planet_osm_roads".*, "planet_osm_line".way
FROM "planet_osm_roads" JOIN "planet_osm_line"
ON "planet_osm_roads".id = "planet_osm_line".id;
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
10.37 s	2.001 ms	0.071 ms	2.001 ms	18.6 s	18.6 s	2 m 18 s

Total RAM usage: 14.379 GB
Total geometries: 2 387 816
Non-geometry query: 21.69 s

High-complexity queries

```
SELECT LineEvenId.*, "planet_osm_point".way
FROM (SELECT * FROM "planet_osm_line" WHERE id % 2 = 0)
AS LineEvenId FULL JOIN "planet_osm_point"
ON LineEvenId.id = "planet_osm_point".id;
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
30.88 s	2.001 ms	0.114 ms	2.001 ms	50.23 s	50.23 s	5 m 35 s

Total RAM usage: 19.837 GB
Total geometries: 19 374 311
Non-geometry query: 57.8 s

```
SELECT PointEvenId.*, "planet_osm_polygon".way
FROM (SELECT * FROM "planet_osm_point" WHERE id % 2 = 0)
AS PointEvenId FULL JOIN "planet_osm_polygon"
ON PointEvenId.id = "planet_osm_polygon".id;
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
1 m 51 s	2.001 ms	0.09 ms	2.003 ms	2 m 31 s	2 m 31 s	55 m 32 s

Total RAM usage: 24.668 GB
Total geometries: 30 692 758
Non-geometry query: 3 m 35 s

```
SELECT PolygonEvenId.*, "planet_osm_roads".way
FROM (SELECT * FROM "planet_osm_polygon" WHERE id % 2 = 0)
AS PolygonEvenId FULL JOIN "planet_osm_roads"
ON PolygonEvenId.id = "planet_osm_roads".id;
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
24.6 s	2.001 ms	0.172 ms	2.002 ms	56.1 s	56.1 s	42 m 17 s

Total RAM usage: 22.449 GB
Total geometries: 13 869 655
Non-geometry query: 1 m 21 s

```
SELECT RoadsEvenId.*, "planet_osm_line".way
FROM (SELECT * FROM "planet_osm_roads" WHERE id % 2 = 0)
AS RoadsEvenId FULL JOIN "planet_osm_line"
ON RoadsEvenId.id = "planet_osm_line".id;
```

User-written	Table OIDs	Names	Expand	Final	Total	SQL Cache
36.6 s	2.001 ms	0.106 ms	2.001 ms	1 m 39 s	1 m 39 s	9 m 49 s

Total RAM usage: 24.751 GB
Total geometries: 17 980 520
Non-geometry query: 1 m 42 s

5.2. Results

As we can see from the queries we used to benchmark *SQL-petrimaps*, we are able to visualize all sorts of SQL queries. The execution times we measured for the SQL queries suggest that apart from the final query, there is no query that has a significant impact on the total time needed. Depending on the query, the final query can take much longer than the user-written query because we have to convert all geometries to their corresponding WKTs.

The large difference between the time the SQLCache needs in total and the total SQL query execution time arises mostly from how the geometries are being parsed. The SQLCache has to iterate every WKT, retrieve the points that make up a geometry for every geometry and also project every point. Additionally, *petrimaps* also densifies all lines in a geometry. During the tests I found that an SQLCache would sometimes be stuck on a single geometry for minutes.

6. Conclusion and Outlook

6.1. Conclusion

We have successfully implemented support for visualizing SQL queries that can be used to query a PostgreSQL database that uses the PostGIS extension. We have seen that we can efficiently rewrite the user-written query and retrieve the data contained within the result table in a parseable format without using tons of RAM. We have also seen how we can achieve this without having to make any assumptions on how the user-written query has to look like. This is in contrast to other GIS that provide similar functionality where we either ran out of RAM or weren't able to properly inspect the results because the app became too unresponsive. Still, especially if we have to deal with millions of geometries, this whole process can take up both a lot of RAM and a lot of time.

6.2. Outlook

There can still be done a lot, mostly in terms of stability and usability. The main goal should be trying to speed up the processing process of the SQLCache, especially the parsing of geometries. For this, we should look into retrieving and parsing the WKB representation of geometries instead of the WKT representation, because it stores the information more compact. Alternatively, we could also try to parse the representation PostGIS uses internally. This way, we would also not have to expand *-select statements, because we could read

the geospatial data from the result table of the user-written query directly. This would require more maintenance, because the representation could vary in different versions. Additionally, we would also have to figure out how to parse the data in the first place. Furthermore, we should also look into how a PostgreSQL database can be set up to manage huge amounts of data more efficiently. Maybe there is also a way to preprocess the tables within a database in order to be able to retrieve the results faster.

For usability the main issue is that the user currently has no feedback on what data they are able to query. They don't even know the names of the available tables. We should provide a list of those table names and add a way to visualize sample rows of the result table of a user-written query. This way, the user could easily understand the data they can work with by writing simple queries in advance. Additionally, it would also make sense to allow the user to query multiple databases. This way, we would not have to put every table in the same database.

7. Bibliography

[1] Hannah Bast, Patrick Brosi, Johannes Kalmbach, and Axel Lehmann. "Efficient Interactive Visualization of Very Large Geospatial Query Results". In: *SIGSPATIAL*. 2023.

[2] Boundless, subsequent revisions by Paul Ramsey, and others. *Geometry Input and Output*. https://postgis.net/workshops/postgis-intro/geometries.html#geometry-input-and-output. Accessed: 2024-04-12.

[3] Boundless, subsequent revisions by Paul Ramsey, and others. *Introduction to PostGIS*. https://postgis.net/workshops/postgis-intro. Accessed: 2024-10-07.

[4] *Copyright and License*. https://www.openstreetmap.org/copyright/en. Accessed: 2024-10-16.

[5] The PostGIS Development Group. *ST_AsText*. https://postgis.net/docs/ST_AsText.html. Accessed: 2024-10-03.

[6] The PostgreSQL Global Development Group. *About PostgreSQL*. https://www.postgresql.org/about. Accessed: 2024-10-01.

[7] The PostgreSQL Global Development Group. *Object Identifier Types*. https://www.postgresql.org/docs/14/datatype-oid.html. Accessed: 2024-10-03.

[8] The PostgreSQL Global Development Group. *Querying a Table*. https://www.postgresql.org/docs/14/tutorial-select.html. Accessed: 2024-10-02.

[9] *OpenStreetMap Data Extracts*. https://download.geofabrik.de. Accessed: 2024-10-23.

[10] PostGIS PSC & OSGeo. *About PostGIS*. https://postgis.net. Accessed: 2024-10-01.

[11] *OSM2PGSQL*. https://osm2pgsql.org. Accessed: 2024-10-23.

[12] *pgAdmin*. https://www.pgadmin.org. Accessed: 2024-10-16.

[13] *The C++ connector for PostgreSQL*. pqxx.org/libpqxx. Accessed: 2024-10-02.

[14] Wikipedia volunteers. *Frontend and backend*. https://en.wikipedia.org/wiki/Frontend_and_backend. Accessed: 2024-10-02.

[15] Wikipedia volunteers. *Geographic information system*. https://en.wikipedia.org/wiki/Geographic_information_system. Accessed: 2024-10-02.

[16] Wikipedia volunteers. *Well-known text representation of geometry*. https://en.wikipedia.org/wiki/Well-known_text_representation_of_geometry. Accessed: 2024-10-03.

Appendix A. Specifications

We list the specifications of the hardware and software used within this thesis.

A.0.1. Local Machine

1. Processor: AMD Ryzen 5 5600 6-Core 3.50 GHz
2. Installed RAM: 32 GB
3. WSL usable RAM: 28 GB
4. Graphics Card: NVIDIA GeForce RTX 3060
5. Operating System: Windows 11 Pro

A.0.2. Software

1. PostgreSQL database: Version 14.13
2. PostGIS: Version 3.4
3. libpqxx: Version 6.4.5
4. pgAdmin 4: Version 8.9
5. QGIS: Version 3.34.8-Prizren
6. GRASS GIS: Version 8.3.2
7. OpenJUMP: Version 2.3.0
8. osm2pgsql: Version 1.6.0