Semen Kudrin

# INTELLIGENT SQL QUERY GENERATION FROM NATURAL LANGUAGE

Bachelor's thesis

Bachelor of Engineering

Information Technology

2025

South-Eastern Finland
University of Applied Sciences

South-Eastern Finland University of Applied Sciences

| | |
|---|---|
| Degree title | Bachelor of Engineering |
| Author | Semen Kudrin |
| Thesis title | Intelligent SQL query generation from natural language |
| Year | 2025 |
| Pages | 35 |
| Supervisor | Ulisses Moliterno de Camargo |


## ABSTRACT

The development of information processing technologies has traditionally aimed to simplify the interaction between humans and machines. The shift from machine code to high-level programming languages has made software development more accessible, but working with databases still demands specialized knowledge and remains inaccessible to a wide range of users.

The purpose of this thesis was to create a system that allows users to obtain data using common language queries without the need to learn specialized programming languages. This approach aimed to remove the main technical barrier, making information more accessible.

The thesis addresses several technical challenges in dealing with SQL: accurately interpreting natural language, automatically retrieving relevant data, and minimizing errors in command generation. Particular emphasis was placed on ensuring system flexibility for diverse data types and usage scenarios.

The theoretical foundation explores how people can interact with databases using everyday language. It discusses the difficulties of translating such language into correct data requests, especially when working with location-based information. The study looks at existing solutions and highlights the need for tools that help users get the right results without needing to understand technical query languages.

The result is a tool that not only simplifies access to information but also opens new opportunities for integration with analytical and visualization platforms where ease of data retrieval is fundamentally important.



**Keywords**: natural language queries, SQL generation, NL2SQL, spatial data

## CONTENTS

# 1 INTRODUCTION

In many areas of human activity, from business and science to public administration, working with large volumes of structured and unstructured data has a main role in decision-making. The relational database remains one of the most common tools for organizing and storing data. Structured Query Language (SQL) is the foundation for working with relational databases, allowing users to process, write and store data using a logical and intuitive language (Zhong et al. 2017).

Despite its simplicity, SQL is not always user-friendly. Like any programming language, it requires specific knowledge and technical skills. Even basic operations such as filtering records by a certain condition or simply counting the number of items can be difficult for someone without experience (Gan et al. 2021). This is especially true in fields where working with data is not the main task. For example, professionals in urban studies, environmental science, transport, or demographics may have strong subject knowledge but lack the technical background to write efficient SQL. As a result, their work frequently depends on the assistance from software developers or data analysts. This leads to delays and an increase in costs and the workload of technical teams.

Another issue is the risk of miscommunication. When users describe what they need in specific terms, developers must interpret these requests and translate them into database queries. This process can easily result in errors, especially when the database structure is complex.

Spatial data is particularly difficult. Unlike regular data, it requires knowledge of geographic information systems (GIS), specialized database extensions, and use of customized spatial functions (Zhang et al. 2024). Building such complex queries can be overly technical, difficult to understand, and error prone.

Due to growing complexity and availability in data types and sources, there are significant gaps between users and their data. Solving this problem

involves developing more intuitive and clear interfaces that allow users to interact with data using their natural language.

With the rapid growth of natural language processing (NLP) technologies, a new category of systems has been created, called Natural Language to SQL (NL2SQL). These systems allow users to formulate queries in natural language, and the system will automatically convert them into structured SQL commands. (Xu et al. 2017.)

At first glance, this approach appears highly promising. It offers a potential solution to the communication barrier between users and databases, enabling interaction in a form that is closer to natural language rather than programming syntax. In theory, this would allow users without technical training to retrieve data in a simple and intuitive way.

However, in practice, current NL2SQL implementations exhibit several limitations. One major issue is their low accuracy in query generation. Even advanced models often struggle with queries involving complex structures or operations across multiple tables. As a result, the generated SQL statements may contain syntax or logic errors, necessitating manual verification and correction. (Scholak et al. 2021.)

Another significant challenge is the lack of transparency in the reasoning process. Users frequently do not understand how the model arrived at a particular response based on the input query. This opacity can lead to mistrust in the system, which becomes particularly problematic in critical applications, such as those involving GPS location data, where correctness and explainability are essential. (Scholak et al. 2021.)

Additionally, most existing NL2SQL systems lack support for spatial data queries. They are generally designed to operate on standard tabular datasets and do not account for the complexities of geospatial information, such as points, lines, or polygons. This makes them unsuitable for use cases where spatial operations are essential. For example, queries such as “Find a parking

area within 500 meters of a metro station” require capabilities that these systems typically do not provide (Kamenev, 2025).

## 1.1 Objective

The objective of this thesis is to design and implement a system that enables users to interact with relational and spatial databases using natural language queries. By integrating a local large language model with semantic search and spatial SQL support (via PostGIS), the system aims to bridge the gap between non-technical users and complex data structures, particularly those involving geospatial information. The purpose is to lower the technical barrier to data access and analysis, allowing users without SQL expertise to retrieve efficiently and intuitively.

A key advantage of the proposed system is the ability to manage not only standard tabular data but also geospatial information, allowing users to ask questions about the location of objects, distances, coordinates, and similar spatial data using natural language.

In this way, the study aims to solve two significant problems at once: it improves data accessibility for non-technical users and simplifies interaction with spatial data, a task that is considered difficult without specialized knowledge.

## 1.2 Properties of the Prototype

The prototype includes the following key components. First, a natural language interface enables users interact with the system through a local large language model Mistral (Mistral models overview, 2025) deployed using the Ollama platform (Ollama documentation, 2025). This allows them to ask queries in a natural and intuitive language, such as: “*How many bicycle parking spots are there in the Kamppi district?”*, without needing to know SQL syntax. It also helps the system developer to flexibly customize how the LLM is used for better performance.

Second, the study provides automatic table mapping. The prototype does not require users to know the structure of the database. Instead, it uses semantic search through the sentence transformers library to identify relevant tables and columns based on the user's request.

Third, the prototype offers support for spatial queries. It uses PostGIS and generating spatial SQL functions like *ST_DWithin*, *ST_Intersects*, and *ST_Distance*, the system is especially suited for handling complex spatial datasets.

# 2 THEORETICAL FRAMEWORK

## 2.1 The Problem of Converting Natural Language to SQL

Modern information systems operate with great volumes of data that are stored in relational databases. There is a special language to work specifically with relational data, called Structured Query Language (SQL). SQL is commonly used to retrieve and analyze such data. However, its queries require technical knowledge, which creates a barrier for users without an experience in programming or database theory. (Xu, Liu & Song, 2017.)

The problem becomes more significant when working with geospatial data. Spatial data is the basis for geographic information systems (GIS) where spatial queries play a significant role (Zhang et al. 2024). For example, a user might want to find intersections between geometric objects, calculate distances, or identify points of interest (POIs) within a given radius. These are complex spatial operations that require GIS-specific methods. Basic SQL knowledge is not sufficient, and users also need to understand specialized spatial database systems such as PostGIS which offer customized functions such as ST_Within, ST_Distance, and ST_Intersects.

Text-to-SQL as a method offers a promising solution to these challenges (Figure 1). It allows users to write queries in natural language and have it automatically converted into valid SQL requests. This lowers the entry

threshold for interacting with databases and makes analytical tools accessible to a broader audience (Zhong, Xiong & Socher, 2017).

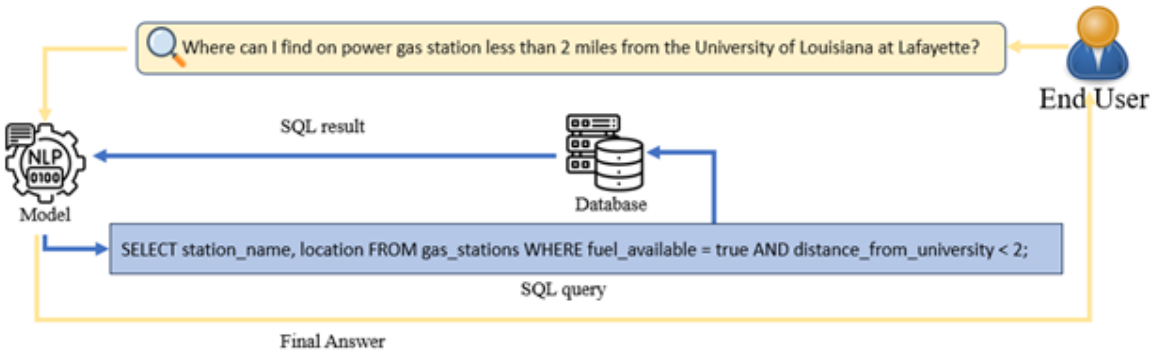



Figure 1. Text-to-SQL methods can support users in producing complex and domain specific SQL queries, lowering the barrier to utilizing relational databases

However, implementing this approach introduces key challenges:

- **Semantic ambiguity -** Natural language is inherently ambiguous. The same phrase can have different meanings depending on the context, making it difficult to create accurate SQL queries.
- **Complexity of spatial queries** - Geospatial requests rely on specialized operations that go beyond standard SQL. Functions such as *ST_Intersects* or *ST_DWithin* require precise formulation, which increase the complexity of automated query generation.
- **Limitations of existing models** - NLP-to-SQL systems often struggle to support spatial queries. As a result, their use in GIS applications is limited or impractical.

Models designed to translate text into SQL often face challenges when applied outside the domain they were trained on. For instance, a model trained on customer service data may not work effectively in fields such as healthcare, since each domain has its own structure, terminology, and typical queries that the model might not recognize or handle properly. (Y. Gan et al, 2021.)

These challenges also apply in this thesis domain and highlight the need for systems that can accurately translate natural language into SQL providing support for advanced spatial operations. Such tools can enhance users' ability to explore and analyze geospatial datasets without requiring in-depth technical expertise in SQL or GIS technologies.

## 2.2 Geospatial Data and PostGIS

Geospatial data (a.k.a. spatial data) describes objects and events that are associated with specific locations on the Earth's surface. Spatial data includes coordinates of points, routes represented by lines, boundaries shown as polygons, and other attributes such as street names, types of objects and time information. The use of geospatial data is critical in many fields, from navigation and logistics to urban planning and infrastructure management. (Kamenev, 2025.)

A key feature of spatial data is the need for specialized methods of storing, processing and analyzing it. Usually, relational databases do not have that functionality to effectively work with spatial data. PostGis was developed to solve this problem, and it significantly expands the capabilities of the PostgreSQL database management system. One of the main components of PostGIS is the use of Spatial Reference System Identifiers (SRID) which accurately specify the coordinate system for each spatial object. This allows for the correct interpretation, comparison and transformation of geodata when performing spatial operations and analysis (PostGIS Project Steering Committee).

PostGIS is a spatial extension for PostgreSQL that transforms it into a fully functional GIS at the database level. It introduces new data types (such as geometry or geography) and over 300 functions and operators that enable complex spatial queries, including:

- Search for objects within a certain radius.
- Check for spatial intersections and inclusions between geometries.
- Calculate distances.

- Transform coordinate reference systems.
- Aggregate and cluster spatial objects.

For example, a simple spatial query might look like this:

*SELECT name FROM gas_stations*
*WHERE ST_DWithin(location, ST_SetSRID(ST_Point(24.94, 60.17), 4326), 1000);*

This query returns all gas stations within 1000 meters of a given coordinate in Helsinki (using the WGS84 coordinate system).

There are several advantages to using PostGIS as the spatial extension of PostgreSQL. First, it offers full SQL integration, allowing users to perform both standard and spatial queries within the same database environment. This simplifies the overall application architecture by eliminating the need for separate tools or data layers. Second, PostGIS is widely compatible with a range of GIS software, including both open-source platforms such as QGIS and commercial solutions such as ArcGIS and GeoServer, ensuring interoperability across different systems. Finally, PostGIS delivers strong performance, leveraging spatial indexes (such as GiST) and parallel processing to support efficient query execution, even when working with datasets containing millions of spatial objects.

In this study, PostGIS server is the platform utilized to store and process both non-spatial and geospatial data. It allows users to input queries in natural language and have their request translated into SQL through AI by using PostGIS spatial functions (Kamenev, 2025). This approach allows users intuitive access to complex spatial operations without profound knowledge of SQL or GIS functions.

### 2.3 Natural Language to SQL Transformation

Natural Language to SQL (NL2SQL) is a method of natural language processing (NLP) that focuses on automatically generating valid SQL queries

from user’s natural language input. NL2SQL systems are divided into two levels of complexity: from simple pattern matching to generating syntactically and semantically correct SQL queries for arbitrary and complex database structures (Li et al, 2024).

Modern NL2SQL solutions typically comprise several key components. The first is text parsing which involves the linguistic analysis of the user’s input to identify relevant entities, actions, and conditions. This is followed by database schema mapping where metadata such as table and column names are extracted and aligned with keywords or concepts in the natural language query. The next step is SQL generation, in which a syntactically correct SQL statement is constructed based on the interpreted meaning of the input and the structure of the underlying database. Finally, validation is performed to ensure that the generated query is both syntactically and logically correct; in some cases, this step also includes providing feedback to the user when errors or ambiguities are detected.

There are two main approaches to implementing NL2SQL systems. The first is rule-based, or template-driven, relying on manually designed query templates. These methods are simple to implement but do not scale well. The second approach uses neural networks, particularly large language models (LLMs), to generate SQL queries directly from free-form natural language input, without relying on predefined templates. (Xu et al, 2017.)

However, applying these systems to real-world databases, especially those containing geospatial data, introduces additional challenges. Natural language often includes vague or context-dependent expressions such as "near," "south of," or "largest," which are difficult to translate precisely into spatial logic. Furthermore, most existing NL2SQL datasets and models lack built-in support for geospatial queries, limiting their effectiveness in this domain.

Most current text-to-SQL models - including SQLCoder, Text-to-SQL T5, GPT-4o, and Claude 3.5 - perform poorly on geospatial queries, with accuracy rates frequently falling below 60% (Kamenev, 2025).

Large language models trained on text can generate a huge range of possible outputs at each step, often choosing from tens of thousands of sub-word tokens. But when adapted to produce structured languages like SQL, they frequently generate incorrect or invalid queries, which makes the output unusable. (Scholak et al, 2021.)

These problems require model fine-tuning and the include of domain-specific data, such as geographic identifiers. A new model worth mentioning is AINO AI V1 This is the first generation of a domain-specific LLM designed for geospatial analysis, also referred to as Text2Map, developed by AINO WORLD. (Kamenev, 2025.)

AINO AI is trained to recognize geospatial entities and transform natural language queries into map-based data operations, ranging from basic extraction to analysis and visualization. It integrates SQL generation with spatial output in the form of maps or analytic reports. The system combines the power of large language models, spatial databases such as PostGIS, and a cartographic rendering engine, enabling interactive GIS interfaces. (Kamenev, 2025.)

However, AINO AI is not used in this study due to security concerns, closed-source nature of the code, and lack of necessary computing power. While it excels on more conceptual problems, it is insufficient to be used in real-world applications.

Instead, this study presents a light prototype system that follows similar principles: parsing user input, generating SQL, and executing it in a PostgreSQL and PostGIS environment. This approach allows us to demonstrate the applicability of NL2SQL to geographical and spatial queries and lay the foundation for future integration with more complex models such as AINO AI.

## 2.4 The Raise of Large Language Models (LLMs)

In recent years, large language models (LLMs) have shown impressive results in a wide range of natural language processing (NLP) tasks, including code generation and SQL query generation. Their ability to analyze context, recognize structure, and produce syntactically correct text makes them a promising tool for natural language to SQL transformation (Zhu et al., 2024).

One of the main advantages of LLMs is their flexibility. Language models can operate with any data structure if they are provided with the appropriate information as a prompt*,* a textual request that specifies the context and format of the expected response. This enables LLMs to make SQL generation without retraining the model for each individual database, relying instead on prompt engineering techniques and instructions (Hong et al, 2024).

In this study, a local deployment of Mistral, a modern open-source LLM, is used via the Ollama framework. Ollama is a lightweight, developer-oriented platform for running and managing large language models locally on consumer hardware (Ollama documentation). It provides a unified interface for downloading, configuring, and serving models such as Mistral without relying on cloud-based APIs. The approach enables user queries without relying on external APIs or transmitting data to third-party servers, which is particularly important for privacy, data protection and system autonomy.

However, despite their advantages, LLM also have notable limitations, especially when used with only prompt engineering techniques. One major issue is their insufficient support for geospatial queries. Most open-source LLMs are not trained on spatial data and struggle to understand SQL functions from extensions like PostGIS, which limits their ability to handle geospatial contexts accurately (Kamenev, 2025).

Another limitation with LLM their low precision without well-crafted prompts. Without clear and specific context, LLMs may produce SQL queries that are syntactically valid but semantically incorrect, failing to capture the user’s intent. In addition, LLMs require access to the database schema to generate

relevant queries. This means that table names, fields, and relationships must be dynamically included in the prompt, adding complexity to the system's overall design.

Using LLM services such as Mistral allows the user to build a functional base to convert natural language into SQL, avoiding some of the problems cited above. However, to expand it to geo-spatial data, either additional training of the models or the construction of a separate logic layer responsible for working with spatial data is required.

# 3 DEVELOPMENT OF THE PROTOTYPE

This section presents the prototype system built during the practical phase of the project. It describes the implementation process, explains the workflow architecture, and discusses the main components, challenges, and results.

## 3.1 The Study Case and Context

This study was developed in collaboration with Aino World**.** Aino World is a company exploring innovative applications of artificial intelligence in the field of geospatial data analytics (Kamenev, 2025)**.** The company focuses on building intelligent tools for spatial data processing, urban analysis, and automation of geodata workflows.

The specific case addressed in this study is the automation of natural language interaction with geospatial databases. The system enables users to input queries in English and receive results by converting these into corresponding spatial SQL queries. This is achieved through a locally deployed Mistral language model integrated via the Ollama framework, working in tandem with PostGIS, an extension for PostgreSQL supporting spatial data types and operations.

During development, a locally deployed version of the Mistral language model was used due to its availability and ease of integration. In the planned production setup, a custom language model will be hosted on the server side to ensure better scalability, maintainability, performance, and security.

The study presented here is part of a larger vision. Aino plans to develop a full-featured geospatial application-comparable to Google Maps-augmented by an integrated language model. Unlike traditional GIS platforms, this system would allow users to interact with maps via natural language, automating complex spatial queries, visualizations, and even decision-making processes. For example, users could ask, *“Show me all parking spots within one kilometer of the city center,”* and the system would autonomously query the database and render the results on a map.

In this context, the study reflects an important trend toward natural language interfaces for spatial data analysis. This direction aligns with the rise of LLM-based no-code platforms, which lower the technical barrier for users in areas such as urban planning, logistics, and environmental monitoring.

### 3.2 The Prototype

The prototype serves as a proof of concept for a geospatial data query system powered by natural language and large language models (LLMs), with extensibility for broader applications in spatial information services.

The prototype justifies the concept of using LLMs for natural language interaction with spatial databases. While limited in scope, the system successfully:

- Accepts user datasets and metadata.
- Enables semantic table selection.
- Generates spatial SQL queries.
- Handles geometry and coordinate data.
- Returns results through a web interface.

This implementation forms the core of a broader intelligent mapping service envisioned in the future work section.

### 3.3 System Architecture

The study was developed entirely in Python, using a set of modular and easily integrable tools. Its architecture comprises several core components that work together to enable a complete natural language querying pipeline. The frontend was built using Flask, a minimal Python web framework that manages HTTP requests, file uploads, and user queries.

For data processing, the system uses Pandas and GeoPandas to load and manipulate both tabular and geospatial data from CSV and Excel files. The database layer is based on PostgreSQL, extended with PostGIS to support both conventional and spatial queries.

In order to access to database, SQLAlchemy is employed in its core expression mode, which offers fine-grained control over SQL generation without relying on the full ORM layer. The psycopg2 driver is used for direct communication with the PostgreSQL database.

Language model integration is achieved through a locally hosted instance of the Mistral large language model, deployed via the Ollama platform. This component is responsible for translating natural language queries into executable SQL commands.

In addition, semantic search functionality is implemented using the all-MiniLM-L6-v2 model from the sentence-transformers library. This model encodes both user queries and table metadata into vector embeddings, enabling the system to identify relevant tables based on semantic similarity.

The overall purpose of this architecture is to support a full query processing pipeline, from natural language input to semantic interpretation, SQL generation, and spatial result delivery, making data access intuitive and efficient, even for non-technical users.

### 3.4 Functionality

The study provides two primary functionalities for the user: table upload and natural language search. Users can upload their own tables into the system via the web interface. They must provide the following:

- A file (CSV or Excel)
- A target table name (used in the database)
- Tags (comma-separated keywords describing the dataset)

For example, uploading a table named *rivers_and_lakes_of_helsinki* with tags such as *river* and *lake* enables future semantic retrieval of this table via related queries.

The backend processes the file and stores the dataset in the PostgreSQL/PostGIS database. Metadata, including the table name and tags, is saved in auxiliary tables-*sources* and *source_tags*-which enables semantic search and filtering.

For performing natural language search, users can enter queries such as, e.g., “Show me lakes within 5 km from Suutarila district.” This initiates a multi-stage processing pipeline involving semantic tag matching, SQL generation via LLM, and result display. Queries may include spatial filters, aggregations, or logical conditions.

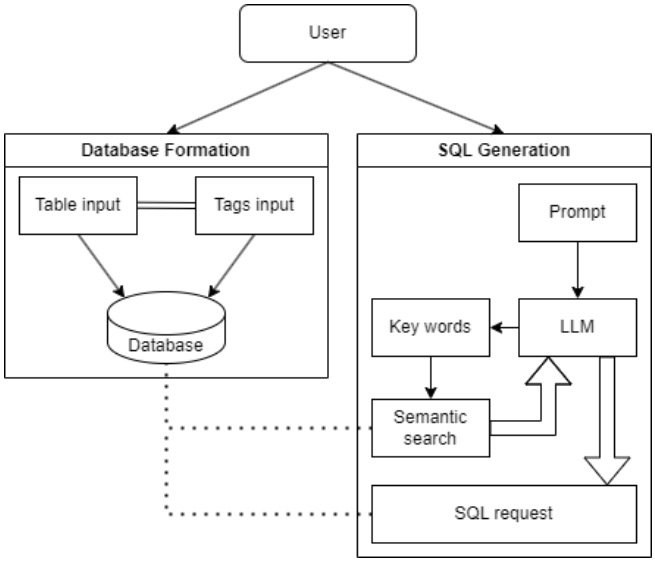



Figure 2. Scheme of how prototype functions work

Figure 2 schematically shows the purpose of the available functions. As mentioned above, the user has access to two functions: forming a database from own tables and generating SQL queries. When forming a database, the user adds their table with tags to the common database and when generating an SQL query, the user's prompt is sent to the model where it is divided into keywords. A semantic search is performed by keywords, where a match of keywords and tags is searched for. The found tags and tables linked to these tags are sent back to the model. The model, having access to the prompt, tags and table schemas, creates an SQL query which is sent to the database.

### 3.5 Methods

The study was developed with a combination of tools selected for their flexibility, ease of integration, and support for spatial data processing. All components were implemented in Python, chosen for its readability, extensive ecosystem, which allowed greater focus on application logic rather than low-level implementation details.

The web interface was built using Flask, a lightweight Python web framework well-suited for handling HTTP requests and rendering HTML pages in small to medium-scale applications. Flask was selected for its simplicity and minimal overhead which aligned well with the scope of the project.

For data manipulation, Pandas was employed to load and process tabular data from CSV and Excel files. In cases where datasets included spatial components (e.g. coordinates or geometries in WKT format), GeoPandas was used as a drop-in replacement. GeoPandas extends the functionality of Pandas to support spatial operations and geometries, thereby enabling efficient handling of geographic data.

Interaction with the PostgreSQL database was managed with use SQLAlchemy, a SQL toolkit and Object Relational Mapper (ORM) for Python. In this paper, only the core SQL expression language was used, without leveraging the full ORM capabilities, to retain closer control over query generation. The psycopg2 driver was used to facilitate communication with the PostgreSQL database. pgAdmin was utilized as a graphical user interface for managing database content, inspecting table structures, and running SQL queries during development.

A central feature of the study is the automatic generation of SQL queries from natural language questions. This functionality was implemented using a locally hosted Mistral language model running on the Ollama platform. The model receives as input a representation of the target table schema - including column names and data types - along with a user query expressed in natural language. It then generates a corresponding SQL statement, enabling non-technical users to retrieve data without writing SQL manually.

In order to assist in selecting the appropriate table based on user intent, semantic similarity matching was employed using the *sentence-transformers* library, specifically the *all-MiniLM-L6-v2* model. Both the user query and table metadata (such as tags or descriptions) were embedded into a shared vector space, and cosine similarity was used to identify the most relevant table for query generation.

This integrated toolchain resulted in a system capable of handling diverse data types - including geospatial content - while supporting intuitive user interaction through natural language interfaces.

# 4 RESULTS

## 4.1 System Overview

### 4.1.1 Creating a Database from User Tables

The architecture of the data upload process begins when the user uploads a file via the web interface and assigns it a table name (table_name) and descriptive tags (tags). As shown in Figure 3, only files in CSV or Excel (XLSX) format are supported. The file format is validated during the upload stage to ensure compatibility.

```
file = request.files['file']
table_name = request.form['table_name']
tags = request.form['tags']

# Validate file format
if not (file.filename.endswith('.csv') or file.filename.endswith('.xlsx')):
    return jsonify({"status": "error", "message": "Unsupported file format. Please upload a CSV or XLSX file."}), 400
```

Figure 3. File format validation for uploaded CSV or XLSX files

Once a file is uploaded, it is read into a Pandas DataFrame based on the file extension. If a geometry column containing Well-Known Text (WKT) is detected, the data is converted into a GeoSeries and further into a GeoDataFrame. Otherwise, the data remains in a standard DataFrame format. This process is illustrated in Figure 4, which shows the conversion logic used when spatial data is present.

```
# Read the file into a Pandas DataFrame
if file.filename.endswith('.csv'):
    df = pd.read_csv(file, sep=',')
    if 'geometry' in df.columns:
        df = gpd.GeoDataFrame(df, geometry=gpd.GeoSeries.from_wkt(df['geometry']))
elif file.filename.endswith('.xlsx'):
    df = pd.read_excel(file)
    if 'geometry' in df.columns:
        df = gpd.GeoDataFrame(df, geometry=gpd.GeoSeries.from_wkt(df['geometry']))
```

Figure 4. Reading user-uploaded files and converting to GeoDataFrame if a geometry column is present

After the file is read and processed, the data is written to a PostgreSQL database extended with PostGIS. As demonstrated in Figure 5, an

SQLAlchemy engine is created using a DATABASE_URI connection string to establish the connection.

```
# Connect to the database
engine = create_engine(DATABASE_URI)
```

Figure 5. Creating *an* SQLAlchemy engine

The system then attempts to write the table to the database. As shown in Figure 6, if the data contains spatial information, the to_postgis() method from GeoPandas is used to create a spatial column in PostGIS automatically. Should the initial write attempt fail, the process is retried with a fallback based on the presence of a geometry attribute (hasattr(df, 'geometry')). For non-spatial data, the standard to_sql() method is used instead.

```
try:
    if hasattr(df, 'geometry'):
        df.to_postgis(table_name, engine, if_exists='replace', index=False)
    else:
        df.to_sql(table_name, engine, if_exists='replace', index=False)
except Exception as e:
    print(f"Error writing via to_postgis: {e}. Falling back to to_sql...")
    df.to_postgis(table_name, engine, if_exists='replace', index=False)
```

Figure 6. Writing table to PostgreSQL with fallback logic for spatial and non-spatial data

Once the table is successfully stored in the database, associated metadata is saved in two separate service tables. Figure 7 illustrates how metadata and tags are inserted into the sources and source_tags tables. The sources table contains the table name and a placeholder for the description, while the source_id generated during this step is linked to each user-defined tag in the source_tags table.

```
with engine.connect() as connection:
    # Insert into 'sources' and get the generated ID
    result = connection.execute(text(
        "INSERT INTO sources (source_path, description) VALUES (:source_path, :description) RETURNING id"
    ), {"source_path": table_name, "description": ""})
    source_id = result.fetchone()[0]  # Get the auto-generated ID

    # Split tags into a list and insert into 'source_tags'
    tag_list = [tag.strip() for tag in tags.split(",")]
    for tag in tag_list:
        connection.execute(text(
            "INSERT INTO source_tags (source_id, tag) VALUES (:source_id, :tag)"
        ), {"source_id": source_id, "tag": tag})

    connection.commit()
```

Figure 7. Inserting metadata and tags into sources and source_tags tables

Special considerations are made when handling geospatial data. It is often necessary to transform coordinate systems or convert geometry from WKT format into proper spatial objects. This process is largely automated by GeoPandas, which performs the necessary conversions internally. For instance, although a GeoDataFrame can contain multiple geometry columns, only one spatial column can be retained for storage in PostGIS; any additional columns are converted to WKT before saving.

Overall, as demonstrated throughout this workflow, the use of GeoPandas greatly simplifies spatial data handling. With a single function call, to_postgis() enables the seamless upload of spatial tables to the database. Once stored, the new table is visible in the database schema and is integrated into the tagging system for efficient future retrieval.

### 4.1.2 Natural Language Query Processing

To allow users to interact with the database using plain English, the system translates natural language questions into SQL queries. This process involves several steps, including semantic search, schema generation, and SQL code creation by a language model as presented below.

First, the user formulates a question in plain language. This query is then passed through a SentenceTransformer model, which has been pre-trained to extract semantic embeddings from natural language. Based on this semantic representation, relevant keywords and tags are extracted.

The system proceeds by identifying the most relevant data sources. As illustrated in Figure 8, the user's query is received in JSON format through the /search route, where it is validated and processed.

```
try:
    data = request.get_json()
    query = data.get('query', '').strip().lower()

    if not query:
        return jsonify({"status": "error", "message": "Query cannot be empty."}), 400

    result_df = locate_sources(query)
```

Figure 8. Receiving and validating user query from JSON request

The core logic of source identification is handled by the locate_sources function. Figure 9 demonstrates how this function operates using semantic similarity between the query and the existing tag embeddings.

```
def locate_sources(prompt):

    query = "SELECT * FROM source_tags"
    df = pd.read_sql(query, engine)

    unique_tags = list(set(list(df['tag'])))

    required_tags = []
    for tag in unique_tags:

        vec1, vec2 = tag_model.encode([prompt, tag])
        sim = cosine(vec1, vec2)

        if sim < 0.6:
            required_tags.append(tag)

    source_ids = list(
        set(list(df[df['tag'].isin(required_tags)]['source_id'])))

    query = f"SELECT id, source_path, description FROM sources WHERE id IN ({','.join(map(str, source_ids))})"
    res = pd.read_sql(query, engine)

    return res
```

Figure 9. Locating relevant data sources using semantic similarity of tags

All tags are read from the database. For each tag, cosine similarity is calculated between the query embedding and the tag embedding (using the all-MiniLM-L6-v2 model).

Tag filtering is done through vector comparison of embeddings (usually cosine similarity). The query is converted into a vector, which is then compared with the stored vector representations of table tags. Tables with similarity above a

specified threshold are considered relevant and included in the analysis. If the similarity is greater than a threshold (sim > 0.6), the tag is considered relevant, and its source_id is saved. Using the list of source_ids, metadata (table name, description) is retrieved from the sources table.

Once the relevant tables are selected, the create_schema function generates a textual representation of their structure, listing columns and data types. This schema is used as contextual input for the large language model (LLM), allowing it to understand how to formulate a valid SQL query.

Next, the SQL query itself is generated. This is done using a local model such as CodeLlama deployed via the Ollama platform. The model receives both the user's question and the schema context, and outputs an SQL statement based on predefined templates.

The generated SQL code is executed in a PostgreSQL database, and the resulting data is returned to the user. In the event of an error-either in syntax or during execution-the system handles it gracefully using try-except blocks and responds with a JSON object that includes an informative error message.

Once the relevant tables are found, SQL query generation and execution are handled by the process_sql_queries function (see Figure 10).

```
def process_sql_queries(result_df, query):
    results = []

    for i, row in result_df.iterrows():
        try:
            schema = create_schema(row['source_path'])
            full_prompt = add_prompt(schema, query)
            raw_response = get_sql(full_prompt)
            sql_data = json.loads(raw_response)
            if sql_data.get('status') != 'ok':
                results.append({'sql': sql_data, 'data': None})
                continue
            data = get_source_data(sql_data['sql'])
            results.append({'sql': sql_data, 'data': data})

        return results
```

Figure 10. SQL query processing with json parsing and error management

The first step involves retrieving the structure of each selected table using the create_schema function. This function queries the database for column names and data types without loading any actual data (Figure 11).

```python
def create_schema(table_name):
    query = f"SELECT * FROM {table_name} LIMIT 0"
    table = pd.read_sql(query, engine)
    dtypes = table.dtypes.to_dict()
    columns = []
    for col_name in table.columns:
        columns.append({
            'column_name': col_name,
            'column_format': str(dtypes.get(col_name, 'unknown'))
        })

    schema = {'table_name': table_name, 'columns': columns}
    return schema
```

Figure 11. Create schema function extracting table structure from database

Next, the original user query is integrated with the schema. This is done through the add_prompt function, where the table schema and user request are combined into a single prompt string (Figure 12). This prompt serves as input for the language model.

```python
def add_prompt(schema, prompt):
    schema['user_prompt'] = prompt
    return str(schema)
```

Figure 12. Add user prompt to schema and convert to string

The SQL query itself is generated using the get_sql function. As illustrated in Figure 13, the prompt is passed to the CodeLlama model via Ollama, which returns a JSON response containing the SQL query and a status flag. The system is configured to enforce strict SQL formatting: it uses ILIKE for case-insensitive search, wraps string values in single quotes, and escapes potentially unsafe input. These measures reduce the risk of SQL injection. If errors occur during parsing or execution, the system handles them gracefully and returns a structured JSON error message for debugging.

```
def get_sql(prompt):

    instruction = '''

    You are an AI assistant to create SQL for a user request.
Take name of table and list of columns and return a valid SQL request inside the "sql" key in JSON format.
If SQL creation is successful, add key "status" with value "ok"; if it fails, add "status" with value "fail".

Always:
- Enclose all column names in double quotes.
- Treat table and column names as case-insensitive.
- For string filters, use ILIKE for case-insensitive comparisons.

Geospatial support:
- If the request involves coordinates (latitude, longitude), generate SQL using PostGIS spatial functions.
- Use ST_DWithin for proximity search.
- Always use SRID 4326 for geographic data (WGS 84).
- If geometry columns may have undefined SRID (e.g., 0), wrap them in ST_SetSRID(column, 4326) before using in operations.
- When comparing geometry with a point, make sure both have the same SRID.
- Use ST_SetSRID(ST_MakePoint(lon, lat), 4326) to create the point.
- When using meters for distance, cast both geometries to geography: `::geography`.
- Avoid mixing geometry and geography types in the same function call.

Do not escape quotes (do NOT use backslashes like \\" or \\').

                    '''
    msgs = [
        {"role": "system",  "content": instruction},
        {"role": "user", "content": prompt}]

    output = ollama.chat(model="codellama", messages=msgs, format='json')
    return output["message"]["content"]
```

Figure 13. Generate SQL query using AI assistant with geospatial support and strict formatting rules

After that, the SQL query is executed via the get_source_data function. The query string is first cleaned (e.g., escape sequences removed), then executed using SQLAlchemy. The result is returned to the application in the form of a Pandas DataFrame (Figure 14).

```
def get_source_data(sql):
    try:
        if isinstance(sql, dict):
            sql_query = sql.get('sql', '')
        else:
            sql_query = str(sql)
        if not sql_query:
            raise ValueError("Empty SQL query")
        clean_sql = sql_query.replace("\\'", "'").replace('\\"', '"')
        with engine.connect() as conn:
            result = conn.execute(text(clean_sql))
            return pd.DataFrame(result.fetchall(), columns=result.keys())
```

Figure 14. Extract source data from the database by executing a cleaned SQL query and returning it as a data frame

Overall, the process is designed as a robust pipeline: natural language is analyzed, relevant tables are identified using semantic similarity, table structure is extracted, SQL is generated via an LLM, the query is executed, and results are returned. This pipeline is modular and fault-tolerant, making it easy to scale and adapt by adding new tags and tables without changing the core logic.

# 5 DISCUSSION

## 5.1 Assessment and Validation of The Results

The key components of the system enabling conversion of natural language queries into SQL including and providing support for geospatial data were successfully implemented. The evaluation of the system's performance and functionality was carried out in several distinct stages, each yielding specific results and insights.

### Database transfer and tag management

Database transfer and setup were completed without significant issues. Specifically, the scripts used to copy tables and store tags (metadata about the structure and content of the tables) executed successfully. The resulting tag table became a key reference point for generating SQL queries based on semantic elements extracted from user input.

### Sql query generation setup

The main challenge during development was the accurate generation of SQL queries using the language model. Initial versions of the prompts failed to yield stable and correct results. The model did not recognize the table structure, often produced syntactically incorrect queries, or omitted necessary geospatial functions.

After several iterations of prompt engineering, a final version was developed. It included detailed instructions, table descriptions, and example queries. Only then did the Mistral model began to consistently interpret natural language input and generate correct SQL statements.

Figures 15 and 16 show the old and latest versions of the instructions for generating SQL requests.

```python
def get_sql(prompt):

    instruction = '''You are an AI assistant to create sql for a user request.
    Take name of table and list of columns and return valid sql request with json in key "sql".
    If request is successfull add key "status" with value "ok", if you fail with creating of sql add key "status" with value "fail"
    If you filter by string value use LIKE operator. Always put names of columns in resulting sql in double quotes.
    Always put names of columns in resulting sql in double quotes.
    Ignore case sensitivity in column names and table names - treat them as case-insensitive.
    When comparing string values, use ILIKE instead of LIKE for case-insensitive comparison where supported,
    otherwise use UPPER() or LOWER() functions to ensure case-insensitive matching.
    Never escape quotes (do NOT add backslashes like \\" or \\').
                '''
    msgs = [
        {"role": "system",  "content": instruction},
        {"role": "user", "content": prompt}]

    output = ollama.chat(model="codellama", messages=msgs, format='json')
    return output["message"]["content"]
```

Figure 15. Old version of strict formatting rules

```python
def get_sql(prompt):

    instruction = '''

    You are an AI assistant to create SQL for a user request.
Take name of table and list of columns and return a valid SQL request inside the "sql" key in JSON format.
If SQL creation is successful, add key "status" with value "ok"; if it fails, add "status" with value "fail".

Always:
- Enclose all column names in double quotes.
- Treat table and column names as case-insensitive.
- For string filters, use ILIKE for case-insensitive comparisons.

Geospatial support:
- If the request involves coordinates (latitude, longitude), generate SQL using PostGIS spatial functions.
- Use ST_DWithin for proximity search.
- Always use SRID 4326 for geographic data (WGS 84).
- If geometry columns may have undefined SRID (e.g., 0), wrap them in ST_SetSRID(column, 4326) before using in operations.
- When comparing geometry with a point, make sure both have the same SRID.
- Use ST_SetSRID(ST_MakePoint(lon, lat), 4326) to create the point.
- When using meters for distance, cast both geometries to geography: `::geography`.
- Avoid mixing geometry and geography types in the same function call.

Do not escape quotes (do NOT use backslashes like \\" or \\').

                    '''
```

Figure 16. *Latest* version of strict formatting rules

### 5.1.1 Successful Use Cases

The system successfully processed several test phrases. Examples include:

**Prompt:** *Show me parking spaces where parking for electric cars are more than 0*

**Generated SQL:**
*SELECT * FROM "access_to_parking_spaces_test" WHERE CAST("electric_car" AS INTEGER) > 0;*

This query selects all records from the relevant table where the number of electric car parking spots is greater than zero. The CAST function is used to ensure numerical comparison, in case the values are stored as text. This example illustrates the system's ability to extract conditions and apply appropriate filtering logic in SQL. Figure 17 shows the result of a successful request.

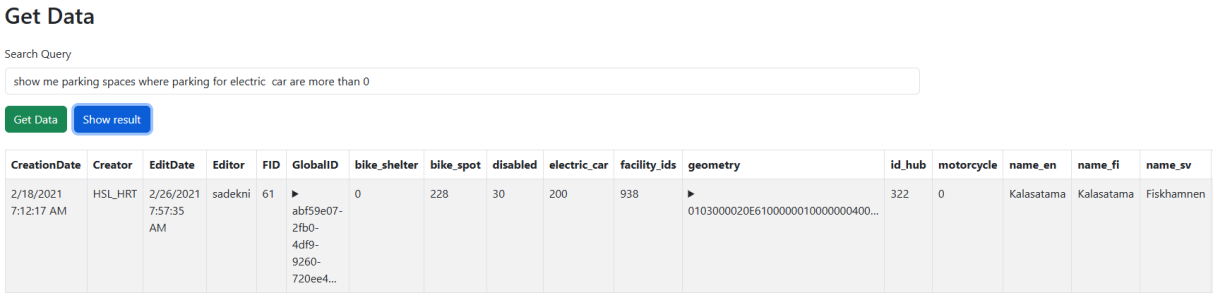



Figure 17. Successful query result with prompt "*Show me parking spaces where parking for electric car are more than 0*"

**Prompt:** *Could you please show Berga parking*

**Generated SQL:**

*SELECT * FROM "access_to_parking_spaces_test" WHERE name_fi ILIKE '%berga%' OR name_sv ILIKE '%berga%' OR name_en ILIKE '%berga%';*

The system identified "Berga" as a location and generated a query that searches for it across multiple language-specific name columns using case-insensitive matching (ILIKE). This ensures broader coverage regardless of the column language. Figure 18 shows the result of a successful request.

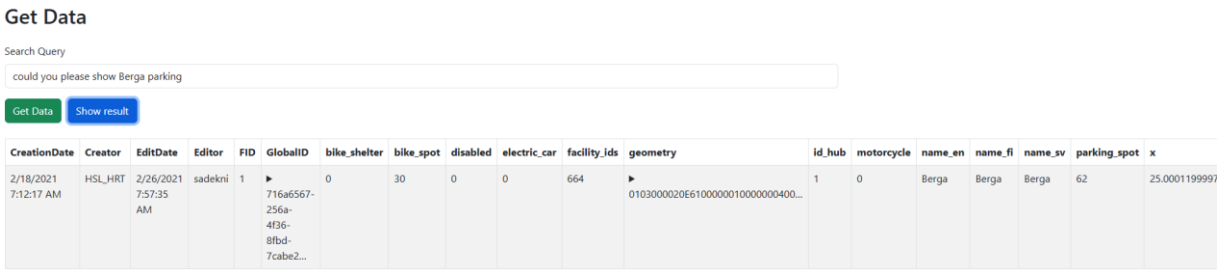



Figure 18. Successful query result with prompt "*Could you please show Berga parking*"

### 5.1.2 Issues With Geometry and SRID

One of the key technical challenges involved the use of an incorrect SRID (Spatial Reference System Identifier) in the source tables. Errors were caused by incompatibilities between the library used for converting X/Y coordinates to geometries and PostGIS requirements. This resulted in failures when performing spatial operations.

Use of unsupported coordinate system EPSG:3857 with geography type:

*SQL execution failed: Only lon/lat coordinate systems are supported in geography.*

This error occurs due to a mismatch in coordinate systems (SRID). The geography type in PostGIS only supports longitude/latitude (e.g., EPSG:4326), so using EPSG:3857 is not allowed.

SRID mismatch between geometry and the query point:

*SQL execution failed: LWGEOM_dwithin: Operation on mixed SRID geometries (Polygon, 0) != (Point, 4326)*

This  part of the error means geometries in the query have different SRIDs - they must be transformed to the same SRID before comparison.

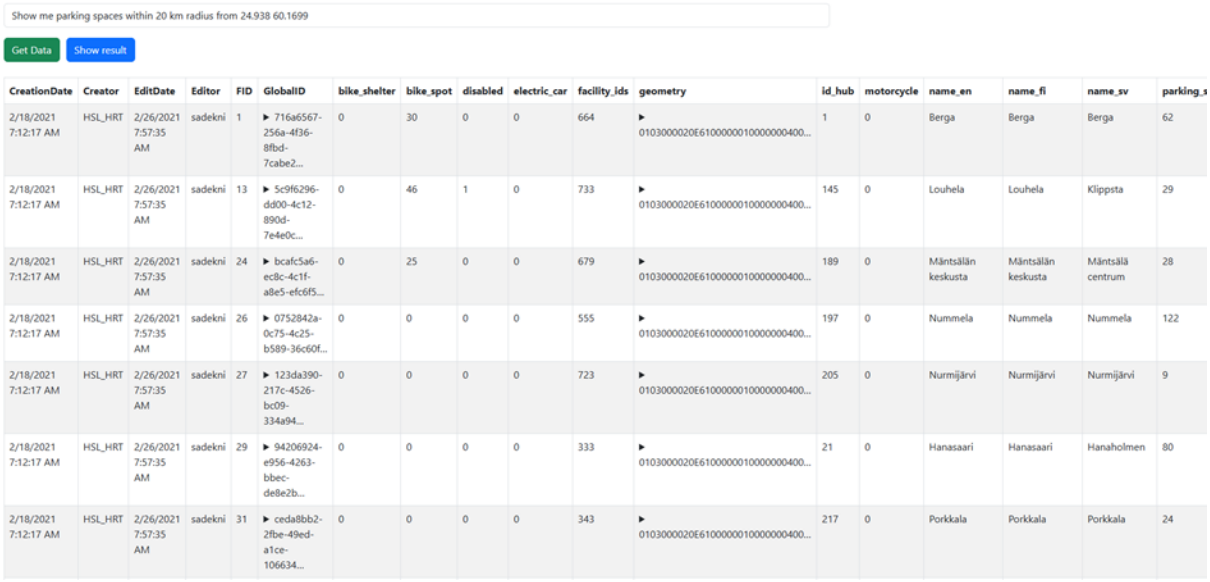
The solution was to manually fix the SRID values and recreate the geometry using the correct coordinate system (EPSG:4326) for the existing tables, and change the X and Y coordinate conversion script for the following tables. After that, spatial queries operated correctly.

**Prompt:** *Show me parking spaces within 20 km from 24.9384, 60.1699*

**Generated SQL:**

*SELECT * FROM "access_to_parking_spaces_test"*
*WHERE ST_DWithin("geometry", ST_SetSRID(ST_MakePoint(24.9384, 60.1699), 4326), 20000);*

The SQL finds parking spaces whose geometry is within 20 km of the given coordinates by using *ST_DWithin* with a point defined at SRID 4326.




| CreationDate | Creator | EditDate | Editor | FID | GlobalID | bike_shelter | bike_spot | disabled | electric_car | facility_ids | geometry | id_hub | motorcycle | name_en | name_fi | name_sv | parking_s |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2/18/2021 7:12:17 AM | HSL_HRT | 2/26/2021 7:57:35 AM | sadekni | 1 | ▸ 716a6567-256a-4f36-8fbd-7cabe2... | 0 | 30 | 0 | 0 | 664 | ▸ 0103000020E6100000010000000400... | 1 | 0 | Berga | Berga | Berga | 62 |
| 2/18/2021 7:12:17 AM | HSL_HRT | 2/26/2021 7:57:35 AM | sadekni | 13 | ▸ 5c9f6296-dd00-4c12-890d-7e4e0c... | 0 | 46 | 1 | 0 | 733 | ▸ 0103000020E6100000010000000400... | 145 | 0 | Louhela | Louhela | Klippsta | 29 |
| 2/18/2021 7:12:17 AM | HSL_HRT | 2/26/2021 7:57:35 AM | sadekni | 24 | ▸ bcafc5a6-ec8c-4c1f-a8e5-efc6f5... | 0 | 25 | 0 | 0 | 679 | ▸ 0103000020E6100000010000000400... | 189 | 0 | Mäntsälän keskusta | Mäntsälän keskusta | Mäntsälä centrum | 28 |
| 2/18/2021 7:12:17 AM | HSL_HRT | 2/26/2021 7:57:35 AM | sadekni | 26 | ▸ 0752842a-0c75-4c25-b589-36c60f... | 0 | 0 | 0 | 0 | 555 | ▸ 0103000020E6100000010000000400... | 197 | 0 | Nummela | Nummela | Nummela | 122 |
| 2/18/2021 7:12:17 AM | HSL_HRT | 2/26/2021 7:57:35 AM | sadekni | 27 | ▸ 123da390-217c-4526-bc09-334a94... | 0 | 0 | 0 | 0 | 723 | ▸ 0103000020E6100000010000000400... | 205 | 0 | Nurmijärvi | Nurmijärvi | Nurmijärvi | 9 |
| 2/18/2021 7:12:17 AM | HSL_HRT | 2/26/2021 7:57:35 AM | sadekni | 29 | ▸ 94206924-e956-4263-bbec-de8e2b... | 0 | 0 | 0 | 0 | 333 | ▸ 0103000020E6100000010000000400... | 21 | 0 | Hanasaari | Hanasaari | Hanaholmen | 80 |
| 2/18/2021 7:12:17 AM | HSL_HRT | 2/26/2021 7:57:35 AM | sadekni | 31 | ▸ ceda8bb2-2fbe-49ed-a1ce-106634... | 0 | 0 | 0 | 0 | 343 | ▸ 0103000020E6100000010000000400... | 217 | 0 | Porkkala | Porkkala | Porkkala | 24 |

Figure 19. Successful query result with prompt "*Show me parking spaces within 20 km from 24.9384, 60.1699*" after fixing errors

## 6 CONCLUSION

This study addressed the challenge of simplifying user interaction with geospatial databases by enabling the translation of natural language queries into structured SQL statements. This functionality is particularly valuable in scenarios where intuitive data access is essential, and where end users may lack technical expertise in database systems or formal query languages. By abstracting the complexity of SQL, the system promotes broader accessibility and usability of spatial data.

A prototype system was developed based on the locally hosted Mistral language model and PostGIS for executing spatial queries. During the

experiments, several common challenges were identified in generating SQL from natural language instructions: difficulties in interpreting spatial parameters, errors in specifying SRID, and ambiguities in query phrasing that reduced the accuracy of generated statements. These findings formed the basis for further analysis and refinement of the prototype.

In order to improve the prototype's performance, a more powerful large language model could be utilized, which might reduce processing time and increase the precision of SQL generation. However, this would require more advanced hardware, particularly greater GPU memory. Additionally, automating the creation of semantic tags for database tables could enhance context understanding, again relying on a more capable LLM. As a smaller, immediate improvement, the instructions for Mistral could be refined through better prompt engineering, but this would require significant effort and provide only a modest benefit.

The study is part of a broader initiative to develop an intelligent geoinformation service. Within this initiative, the concept and initial training of a specialized language model named Text2Map, intended for geospatial data processing, were conducted.

# 7 REFERENCES


Gan, Y., Chen, X. & Purver, M., 2021. Exploring underexplored limitations of cross-domain text-to-SQL generalization. arXiv preprint. Available at: https://arxiv.org/abs/2109.05157 [Accessed 19 May 2025].

Guo, J. et al., 2019. Towards complex text-to-SQL in cross-domain database with intermediate representation. arXiv preprint. Available at: https://arxiv.org/abs/1905.08205 [Accessed 19 May 2025].

Hong, Z., Yuan, Z., Chen, H., Zhang, Q., Huang, F., & Huang, X., 2024. Knowledge-to-SQL: enhancing SQL Generation with data expert LLM. arXiv preprint. Available at: https://arxiv.org/abs/2402.11517 [Accessed 19 May 2025].

Hugging Face, n.d. Web page. Available at: https://huggingface.co/docs/transformers [Accessed 19 May 2025].

Kamenev, A., 2025. Aino AI v1: First-generation LLM for geospatial analysis. Web page. Available at: https://medium.com/aino-world/aino-ai-v1-first-generation-llm-for-geospatial-analysis-61e23c54d0c5 [Accessed 19 May 2025].

Li, B., Luo, Y., Chai, C., Li, G., & Tang, N., 2024. The Dawn of Natural Language to SQL: Are We Fully Ready? arXiv preprint. Available at: https://arxiv.org/abs/2406.01265 [Accessed 19 May 2025].

Mistral models overview, n.d. Web page. Available at: https://ollama.com/library/mistral [Accessed 19 May 2025].

Ollama documentation, n.d. Web page. Available at: https://ollama.com/library [Accessed 19 May 2025].

Pandas documentation, 2024. Web page. Available at: https://pandas.pydata.org/docs/ [Accessed 19 May 2025].

pgAdmin documentation, 2025. Web page. Available at: https://www.pgadmin.org/docs/ [Accessed 19 May 2025].

PostGIS project steering committee. 2023. PostGIS manual (version 3.1): Chapter 4 – PostGIS usage. Web page. Available at: https://postgis.net/docs/manual-3.1/postgis_usage.html [Accessed 19 May 2025].

PostgreSQL global development group. 2023. Geometric functions and operators. PostgreSQL 15 documentation. Web page. Available at: https://www.postgresql.org/docs/15/functions-geometry.html [Accessed 19 May 2025].

Qi, J. et al., 2022. RASAT: Integrating relational structures into pretrained seq2seq model for text-to-SQL. arXiv preprint. Available at: https://arxiv.org/abs/2205.06983 [Accessed 19 May 2025]

Rajkumar, N., Li, R. & Bahdanau, D., 2022. Evaluating the text-to-SQL capabilities of large language models. arXiv preprint. Available at: https://arxiv.org/abs/2204.00498 [Accessed 19 May 2025].

Scholak, T., Schucher, N. & Bahdanau, D., 2021. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. arXiv preprint. Available at: https://arxiv.org/abs/2109.05093 [Accessed 19 May 2025].

SentenceTransformers., 2024. Web page. Available at: https://sbert.net/ [Accessed 19 May 2025].

SQLAlchemy project., 2025. SQLAlchemy 2.0 documentation. Web page. Available at: https://docs.sqlalchemy.org/ [Accessed 19 May 2025].

Tai, C.-Y. et al., 2023. Exploring chain of thought style prompting for text-to-SQL. arXiv preprint. Available at: https://arxiv.org/abs/2305.14215 [Accessed 19 May 2025]

Wang, B. et al., 2020. RAT-SQL: Relation-aware schema encoding and linking for text-to-SQL parsers. arXiv preprint. Available at: https://arxiv.org/abs/1911.04942 [Accessed 19 May 2025].

Xu, X., Liu, C. & Song, D., 2017. SQLNet: Generating structured queries from natural language without reinforcement learning. arXiv preprint. Available at: https://arxiv.org/abs/1711.04436 [Accessed 19 May 2025].

Yu, T. et al., 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. arXiv preprint. Available at: https://arxiv.org/abs/1809.08887 [Accessed 19 May 2025].

Yu, T. et al., 2018. SyntaxSQLNet: Syntax tree networks for complex and cross-domain text-to-SQL task. arXiv preprint. Available at: https://arxiv.org/abs/1810.05237 [Accessed 19 May 2025].

Yu, T. et al., 2018. TypeSQL: Knowledge-based type-aware neural text-to-SQL generation. arXiv preprint. Available at: https://arxiv.org/abs/1804.09769 [Accessed 19 May 2025].

Yu, T. et al., 2019. CoSQL: A conversational text-to-SQL challenge towards cross-domain natural language interfaces to databases. arXiv preprint. Available at: https://arxiv.org/abs/1909.05378 [Accessed 19 May 2025].

Zhang, X. et al., 2024. GS-SQL: Modeling spatial semantics in spatial text-to-SQL. Proceedings of IJCNN 2024. Available at: https://dblp.org/rec/conf/ijcnn/ZhangXYZ24 [Accessed 19 May 2025].

Zhong, V., Xiong, C. & Socher, R., 2017. Seq2SQL: Generating structured queries from natural language using reinforcement learning. arXiv preprint. Available at: https://arxiv.org/abs/1709.00103 [Accessed 19 May 2025].

Zhu, X., Li, Q., Cui, L. & Liu, Y., 2024. Large language model enhanced text-to-SQL generation: A survey. arXiv preprint. Available at: https://arxiv.org/abs/2410.06011 [Accessed 19 May 2025].