Bachelor of Science Thesis

Efficient Spatial Search for the QLever SPARQL Engine

Christoph Ullinger

January 20th, 2025

Submitted to the University of Freiburg

Department of Computer Science

Chair for Algorithms and Data Structures

universität freiburg

University of Freiburg
Department of Computer Science
Chair for Algorithms and Data Structures

Author	Christoph Ullinger, Matriculation Number: 5300285
Writing Period	October 18th, 2024 – January 20th, 2025
Examiner	Prof. Dr. Hannah Bast, Department of Computer Science Chair for Algorithms and Data Structures
Supervisor	M.Sc. Johannes Kalmbach, Department of Computer Science Chair for Algorithms and Data Structures
Declaration	I hereby declare, that I am the sole author and composer of this thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work. I hereby also declare, that my thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Freiburg, January 20th, 2025 — Place, Date

 — Signature

Abstract

We present an end-to-end workflow for efficiently performing geographic searches for nearest neighbors with the QLever SPARQL engine. Our solution significantly reduces the time and the users' effort required to combine and query spatial data from multiple sources.

QLever allows working with points in the standardized *Well-Known Text* format, which it now stores efficiently. Searching for geographically close points in data sets containing hundreds of millions of points becomes a matter of seconds using QLever's new spatial search capabilities. A fast algorithm based on a spatial index is presented as well as a proof of concept baseline algorithm. The spatial search is carefully integrated into the SPARQL syntax.

We introduce programs for the conversion of data from multiple formats (*Keyhole Markup Language*, *Comma-Separated Values* and *General Transit Feed Specification*) to RDF. Additionally, a new program allows users to construct complex spatial queries for QLever with a graphical user interface.

The retrieval of data, which would otherwise require working with many different data sets individually, is now possible in a single SPARQL query. We demonstrate the usability of our workflow using a current research question from political science.

Furthermore, we show that our efficient spatial search implementation in QLever surpasses the query performance of the popular PostgreSQL system by orders of magnitude for large inputs. Regarding all benchmarks, our implementation shows more stable running times.

Zusammenfassung

In dieser Arbeit wird ein Ende-zu-Ende-Workflow für die effiziente Suche nach geographisch nächstgelegenen Punkten in der QLever SPARQL-Engine vorgestellt. Die Zeit und der Aufwand die benötigt werden, um Daten aus verschiedenen Quellen zusammenzuführen und räumlich zu durchsuchen, verringern sich durch die Umsetzung deutlich.

QLever erlaubt die Verarbeitung von Punkten im standardisierten *Well-Known Text*-Format, welches jetzt effizient gespeichert wird. In hunderten Millionen Punkten können unter Verwendung der neuen räumlichen Suchfunktion geographisch nächstgelegene Punkte in Sekundenschnelle gefunden werden. Vorgestellt werden ein schneller Algorithmus auf der Grundlage eines räumlichen Index sowie ein Basisalgorithmus zum Vergleich. Desweitern wird die räumliche Suche sorgfältig in die SPARQL-Syntax integriert.

Die Arbeit präsentiert Programme, um die einfache Konvertierung von Daten aus verschiedenen Formaten (*Keyhole Markup Language*, *Comma-Separated Values* und *General Transit Feed Specification*) nach RDF zu ermöglichen. Außerdem wird ein Programm inklusive graphischer Benutzer:innenoberfläche eingeführt, um den Entwurf komplexer räumlicher Abfragen für QLever zu vereinfachen.

Die Gewinnung von Daten, für die sonst viele verschiedene Datensätze einzeln bearbeitet werden müssten, ist jetzt mit einer einzigen SPARQL-Abfrage möglich. Die Nutzbarkeit des dargestellten Workflows wird anhand einer aktuellen Forschungsfrage aus der Politikwissenschaft demonstriert.

Darüber hinaus wird gezeigt, dass das implementierte Verfahren der effizienten räumlichen Suche in QLever die Geschwindigkeit des weit verbreiteten PostgreSQL-Systems bei großen Eingaben um Größenordnungen übertrifft. Bei Betrachtung aller Vergleichstests zeigt das in QLever implementierte Verfahren stabilere Laufzeiten.

Contents

Acronyms

Acronym	Description
API	Application Programming Interface
CC-BY	Creative Commons Attribution License, as published in [17]
CLI	Command Line Interface
CSV	Comma-Separated Values, as defined in [62]
DELFI	Durchgängige elektronische Fahrgastinformation e.V.
EU	European Union
GiST	Generalized Search Tree
GPL	GNU General Public License, as published in [29]
GTFS	General Transit Feed Specification, as defined in [32]
GUI	Graphical User Interface
IRI	Internationalized Resource Identifier, as defined in [22]
JSON	JavaScript Object Notation, as defined in [11]
KML	Keyhole Markup Language, as defined in [50]
ODBL	Open Database License, as published in [48]
OGC	Open Geospatial Consortium
OSM	OpenStreetMap
PBF	Protocolbuffer Binary Format
PDF	Portable Document Format
POI	Point of Interest
RDF	Resource Description Framework, as defined in [71]
SPARQL	SPARQL Protocol and RDF Query Language, as defined in [73]
SQL	Structured Query Language, as defined in [41]
W3C	World Wide Web Consortium
WKT	Well-Known Text Representation of Geometry, as defined in [51]
XML	Extensible Markup Language, as defined in [70]

Figure Index

Table Index

Code Index

1 Introduction

In this thesis we introduce an end-to-end workflow for efficiently performing spatial searches with the QLever SPARQL engine[1]. More specifically, QLever is enabled to support a fast search for geographically close points that can originate from arbitrary other query operations. In order to realize this feature, an efficient point representation, baseline and index-based nearest neighbors search algorithms and an integration into SPARQL syntax are introduced.

To facilitate the workflow from data to query result, we present converters for the Keyhole Markup Language (KML), Comma-Separated Values (CSV) and General Transit Feed Specification (GTFS) formats to transform data into RDF as understood by QLever. Additionally, we introduce a program and graphical user interface to automatically generate large SPARQL queries for spatial search using an uncluttered configuration.

1.1 Motivation

Typically, when researching a more complex empirical question, data from multiple third-party sources is required. A very common scenario in this case is manually arranging data in a spreadsheet program, writing small helper scripts, importing some parts of the data into a database system and spending much time plugging all parts together.

A concrete example is the current research question in political science, how the availability of public infrastructure influences political polarization. In order to produce a full data set for this question for two German elections, ten different data sets from multiple sources have to be combined. Especially, there are multiple spatial data sets: the OpenStreetMap (OSM) data to provide residential buildings and points of interest (POIs), files in GTFS format to provide information on public transport and in KML format to provide electoral districts not present in OSM. Then multiple spatial searches have to be performed to measure the distance

[1] Detailed explanations follow in chapter 2 on background.

from residential buildings to different public services and infrastructure. Finally this information needs to be aggregated spatially using the electoral districts and brought together with the computation of a polarization index and socioeconomic structural data. Using existing tools one can easily spend weeks on building the dataset and hours for computation of the spatial search. Assuming we want to apply the analysis for another election again, the effortful process would have to be repeated.

Using the software presented in this thesis, the process is streamlined to running a program for each data set to convert it to RDF, constructing a single complete SPARQL query and running QLever to perform the spatial searches and assemble all required data. With the fast implementation, the individual spatial searches on millions of buildings and hundreds of thousands of POIs complete in seconds, the entire complex query in a few minutes.

1.2 Related Work

In the following, we discuss already existing works and implementations.

The popular Blazegraph SPARQL engine implements a spatial search operation [65]. It uses a special SPARQL `SERVICE` query, similar to what we propose in our approach. However, Blazegraph can only collect all points limited by a range and cannot perform a search for an exact amount of nearest neighbors with arbitrary distance. Also it requires using its own literal data type not the standardized WKT format.

Another popular SPARQL engine, Apache Jena Fuseki supports many features from the GeoSPARQL standard as well as custom functions, for example `spatialF:nearby(?geometry1, ?geometry2, ?distance, ?unit)` for nearby points with a maximum distance [1]. A nearest neighbors search is not available.

While not being a SPARQL engine but a relational database management system built on SQL, PostgreSQL and its spatial extension PostGIS are widely used. PostGIS provides a very large set of spatial features including a nearest neighbors search as required for the queries we intend to be able to run [60]. Furthermore, PostgreSQL implements a generalized search tree as an index data structure [37, 42], which can be used for spatial indices.

Another approach to spatial indexing is implemented as part of the s2geometry library [35]. It operates on a three-dimensional unit sphere and provides a fast index data structure. The index is based on a mapping of the sphere's surface to one-dimensional identifiers, locality-preservingly allocated with a space-filling curve. Since s2geometry is a library, it cannot be used directly to query data but will be integrated into QLever in this thesis.

Similar to our approach, the embedding of spatial information in integer identifiers and the use of a space-filling curve have already been discussed [67] for the RDF-3X system [47]. The authors have also implemented a nearest neighbors search. However the software and source code of this spatial extension to the RDF-3X system is not publicly available.

Besides the mentioned programs, QLever is another SPARQL engine, which is faster than RDF-3X or Blazegraph in general query evaluation benchmarks [9] and publicly available as free software. Up until now QLever does not support a nearest neighbors search yet.

The remaining subtask of this thesis' topic, converting spatial data to RDF, has also been addressed previously. For the conversion of OSM data to RDF, the osm2rdf [6] tool is available. The program also precomputes spatial relations such as intersections between geometries on the map using the spatialjoin library [8].

A program called GeoTriples is available for the conversion of KML data among many other formats [43]. However it is large and complex and thus not well-suited for our intention of a simple workflow. For converting GTFS data to RDF, the Open Transport Working Group has provided a reference implementation, which is unfortunately no longer maintained and therefore currently unusable due to dependency errors.

1.3 Contribution

Considering the existing work, our contributions in this thesis include the following:

- We present an efficient representation of geographic points in QLever that speeds up all operations on points.
- We implement a proof of concept baseline algorithm for nearest neighbors search for reference and correctness testing.

- We implement a fast index-based algorithm for nearest neighbors search using the s2geometry library.
- We integrate the nearest neighbors search into QLever's query parsing and query planning using a special `SERVICE` in SPARQL syntax. Therefore a nearest neighbors search becomes possible for the first time in QLever. Additionally, we implement further configuration and optimization functionalities for this integration.
- We extend QLever to support further useful SPARQL functions for exponentiation and standard deviation.
- We introduce new programs for the conversion of data in various formats (KML, CSV, GTFS) to RDF. Unlike existing solutions, the programs are free software and have no external dependencies. They can be easily used in a workflow with QLever.
- We also introduce a new program and user interface to simplify the process of creating SPARQL queries for QLever that contain many spatial searches.
- We demonstrate the use of all implemented software with a current use case from political science.
- We evaluate the implementations extensively and compare them to the widely-used PostgreSQL system.

2 Background

In this chapter we present the concepts that are relevant to understand the approach and implementation.

2.1 RDF

The Resource Description Framework (RDF) standardizes a terminology and a data model for knowledge graphs [71, 72]. An RDF dataset is an edge-labeled, directed graph.

Definition 2.1 (Edge-labeled graph)**.** An edge-labeled graph G is defined as $G = (V, Lab, E)$. V is a finite set of vertices. Lab is a finite set of labels. $E \subseteq V \times Lab \times V$ is a finite set of triples representing directed, labeled edges [2].

In practice, an RDF graph is usually stored as a sequence of triples. Each triple corresponds to an element of the edge set of the graph. The triples are structurally inspired by simple human sentences of the form *subject – predicate – object*. In addition to the Definition 2.1 of edge-labeled graphs, RDF differentiates three kinds of vertices, also called nodes. Nodes can be blank, Internationalized Resource Identifiers (IRIs) [22] or literals. Literals may carry a data type or, only for strings, a language tag. In RDF, both IRIs and Literals are described with the terms *resources* and *entities*. They represent "something in the world [...] including physical things, documents, abstract concepts, numbers and strings" [71]. Outgoing edges of a node are called *properties* of a node. The entire triple representing an edge is a *statement*.

A simple example of an RDF knowledge graph is given as triples in a text representation in Code 2.1. The same data is visualized as an edge-labeled graph in Figure 2.1.

```
<bear> <is-a> <family> .
<brown-bear> <is-a> <species> .
<brown-bear> <subclass-of> <bear> .
<polar-bear> <is-a> <species> .
<polar-bear> <subclass-of> <bear> .
<giant-panda> <is-a> <species> .
<giant-panda> <subclass-of> <bear> .
<jiao-qing> <is-a> <giant-panda> .
```

Code 2.1: Example RDF knowledge graph as triples in RDF turtle format

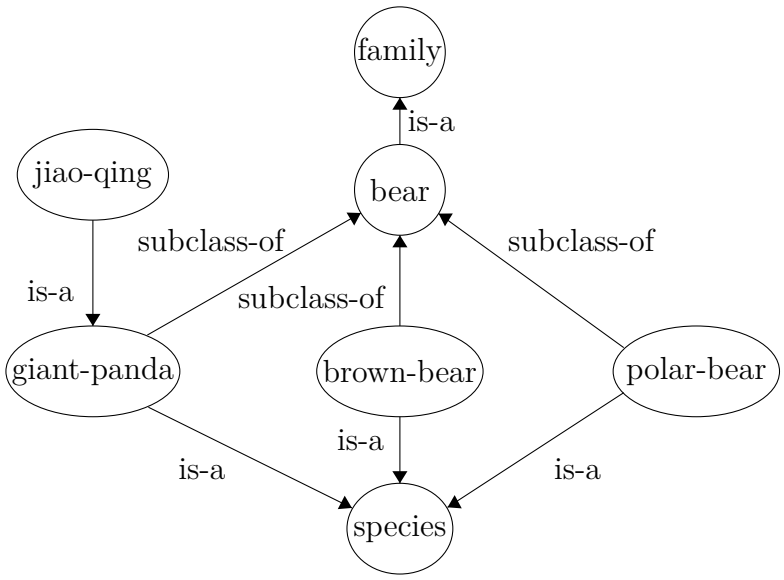


Figure 2.1: Visualization of the RDF knowledge graph from Code 2.1

RDF knowledge graphs are commonly stored in *turtle* format, which is just a text file containing the triples separated by space and ended by a dot, like Code 2.1. In practice however, many IRIs contain long string prefixes. For example all entities from the Wikidata project begin with `http://www.wikidata.org/entity/`. To avoid repeating these strings RDF allows for *prefix declarations* that abbreviate prefixes by a user-defined shorthand as shown in Code 2.2.

```
@prefix wd: <http://www.wikidata.org/entity/> .
```

Code 2.2: Example prefix declaration in RDF turtle format

2.2 SPARQL and GeoSPARQL

The World Wide Web Consortium (W3C) standardizes the "SPARQL Protocol and RDF Query Language", SPARQL for short [73]. It is a declarative query language for RDF knowledge graphs. While the SPARQL syntax is roughly inspired by the Structured Query Language (SQL) for relational databases, the semantics and features of both languages differ largely.

A SPARQL query in its simplest form is a `SELECT … WHERE { … }` statement containing a list of triples. The user can replace any part of a triple by a variable prefixed with `?`. Multiple occurrences of the same variable constitute a conjunction: one value of the variable must satisfy each statement. An example for the knowledge graph from 2.1 is shown in Code 2.3.

```
SELECT * WHERE {
    ?a <is-a> ?s .
    ?s <is-a> <species> .
    ?s <subclass-of> <bear> .
}
```

Code 2.3: Simple SPARQL example query

A simple SPARQL query, as described, is equivalent to a directed, edge-labeled query graph. Each variable and literal in the SPARQL query corresponds to one vertex. Additionally, for each individual triple in the SPARQL query there is an edge in the query graph. For the query in Code 2.3, the query graph is visualized in Figure 2.2.

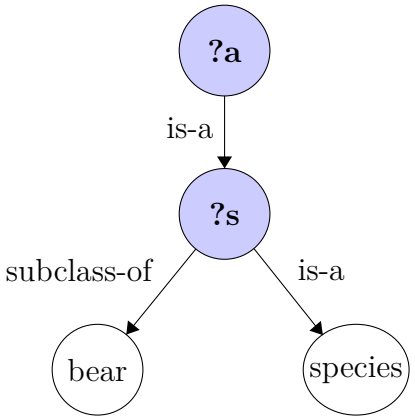


Figure 2.2: Visualization of the SPARQL query from Code 2.3 as a query graph

The task of a SPARQL engine is to answer a query. For this purpose, it is necessary to find all mappings of variable vertices in the query graph to vertices in the knowledge graph that constitute a homomorphism (Definition 2.2) between both graphs. Of course, literals in the query must be mapped to the equivalent literal in the knowledge graph. All combinations of values for the variables in the query that satisfy these criteria are part of the answer.

Definition 2.2 (Graph-homomorphism)**.** Let $G_1 = (V_1, Lab_1, E_1)$ and $G_2 = (V_2, Lab_2, E_2)$ be edge-labeled graphs. A function $f : V_1 \to V_2$ is called a graph-homomorphism between G_1 and G_2 if and only if $(u, l, v) \in E_1 \Rightarrow (f(u), l, f(v)) \in E_2$ [2].

It is important to note that SPARQL works on bag semantics. The sets in the mathematical definitions are therefore in practice lists or multisets.

Furthermore, SPARQL supports many additional features such as aggregation, path queries (recursion), filters, applying functions, subqueries, set union, set minus and more.

Beyond the SPARQL standard's features, there is a standardization for processing geographic information with SPARQL: GeoSPARQL [49]. It is maintained by the Open Geospatial Consortium (OGC) together with the standard for *Well-Known Text* [51] (WKT). A string representation of geometries like points, polygons or lines in this standard format is called a WKT literal. An example point literal is shown in Code 2.4.

```
1 "POINT(7.84129473 47.9977308)"
  ↪  ^^<http://www.opengis.net/ont/geosparql#wktLiteral>
```

Code 2.4: Example WKT point literal for "Freiburg (Breisgau) Hbf"

GeoSPARQL allows for queries like the following (Code 2.5), which searches for all restaurants in Freiburg. This query uses OSM data, which is described in 2.4.

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX osmkey: <https://www.openstreetmap.org/wiki/Key:>
PREFIX ogc: <http://www.opengis.net/rdf#>
PREFIX osmrel: <https://www.openstreetmap.org/relation/>
SELECT ?restaurant ?geometry WHERE {
  ?restaurant osmkey:amenity "restaurant" .
  ?restaurant geo:hasGeometry/geo:asWKT ?geometry .
  osmrel:62768 ogc:sfContains ?restaurant .
}
```

Code 2.5: Example query using GeoSPARQL to find all restaurants in the city of Freiburg

2.3 QLever

QLever[1] [9] is a very fast implementation of a SPARQL engine in the C++ programming language. It is being developed at the Chair for Algorithms and Data Structures at the University of Freiburg. Unlike other SPARQL engines, QLever can handle very large RDF knowledge graphs with tens of billions of triples on a consumer computer. QLever is free software[2] available to the public under the permissive Apache License 2.0 [3].

One factor, that speeds up QLever's query evaluation speed is the usage of carefully crafted integer identifiers instead of strings. These *ValueIds* preserve the natural order of the values they encode. As much of the query processing as possible is performed only on ValueIds without reading the full strings from disk. Since each ValueId is only a 64-bit integer, even large intermediate results can be materialized in memory. A closely related important optimization of QLever is *folding* literals into ValueIds where possible: integers, doubles and dates are embedded directly in the ValueId. Thus literals of these datatypes never require loading and parsing strings from disk during query processing. The index data structures used by QLever also operate on ValueIds. Strings are stored separately in a *vocabulary*.

A second fundamental idea is building indices for permutations of the input data's triples. For each triple consisting of subject (S), predicate (P) and object

[1] GitHub repository: https://github.com/ad-freiburg/qlever

[2] For an overview on the term "free software", see: https://www.gnu.org/philosophy/free-sw.html

(O), QLever can build an index containing the triples in SPO, SOP, POS, PSO, OSP and OPS order. The benefit of such indices is that for each possible position of variables in the query statement, there is an index where the appropriate constants are first in the index' search keys. Thus the set of candidate triples for the result can be narrowed down faster. Examples are shown in Code 2.6.

```
1 ?var1 <predicate> <object> .   # POS or OPS
2 ?var1 <predicate> ?var2 .      # POS or PSO
3 <subject> ?var1 ?var2 .        # SPO or SOP
4 <subject> ?var1 <object> .     # SOP or OSP
```

Code 2.6: Example statements, where indices over differently permuted triples are beneficial

Additionally, QLever supports full-text indexing and combined search in natural language annotated by knowledge graph entities. However these features are beyond the scope of the topic of this thesis. In this thesis we present additions to QLever that extend its spatial querying capabilities.

2.4 OpenStreetMap and osm2rdf

OpenStreetMap[3] is a crowd-sourced effort to build a map of the world as an entirely free and open database. The data is publicly available under the Open Database License [48] (ODBL).

The basics of the OSM data model [53] follow a simple concept. The map data consists of three types of elements. Each element is identified by an integer. *Nodes* are simple points consisting of the longitude and latitude coordinates. *Ways* are collections of nodes that represent lines on the map like roads or railway tracks. *Relations* can combine ways to more complex geometries like polygons or sets thereof. For example, they are used to represent shapes of buildings. All three types of elements can be associated with an arbitrary number of *tags*. Tags are key — value pairs that represent features of the geographic element, for example `amenity=restaurant` or `railway=station`.

In order to query OSM data with SPARQL, it needs to be transformed to an RDF knowledge graph. This task can be accomplished using osm2rdf[4] [6, 8], a free

[3] https://www.openstreetmap.org
[4] GitHub repository: https://github.com/ad-freiburg/osm2rdf

and open-source tool developed at the Chair for Algorithms and Data Structures of the University of Freiburg. It is available under the GNU General Public License version 3 [29] (GPL). The tool reads data in OSM's own *Protocolbuffer Binary Format* [55] (PBF) and outputs RDF turtle.

Differing from other similar-purposed programs like *osm2pgsql* [39], osm2rdf's output contains all OSM geometries and tags by default. This is possible due to the flexibility of the RDF data model. The target format of osm2pgsql is PostgreSQL, which limits the number of columns in a table. That limitation can be circumvented in newer versions using `hstore` columns at a performance loss [4, 38]. In contrast, RDF natively does not have restrictions on the number of predicates.

A second important advantage of osm2rdf is its precalculation of geometric relations. For all geometries from OSM the program outputs GeoSPARQL triples `ogc:sfContains`, `ogc:sfIntersects` and more. Such triples can be calculated efficiently for the entire planet on a conventional computer [8]. Using precomputed geometric relations instead of computing them ad hoc can speed up spatial queries on OSM by orders of magnitude.

2.5 S2Geometry

In this thesis we take advantage of the s2geometry[5] [35] library developed by Google. It provides efficient indexing features for geographic data on a special data model. Unlike other geometric libraries, s2geometry does not operate on two-dimensional projections but on the surface of a three-dimensional unit sphere. This way, the varying distortion of a two-dimensional projection and special cases for poles can be avoided.

For fast access, s2geometry divides the sphere's surface into rectangles, called *cells* [33]. They can be used similarly to a bounding box. A hierarchy of 31 different sizes of cells from large to small is used: from dividing the surface into 6 cells to dividing it into $7 * 10^{18}$ cells [34]. Cells of the smallest size are called *leaf cells* and represent approximately $0.74\,\mathrm{cm}^2$ on the earth's surface. An example with cells on different levels that contain each other is shown in Figure 2.3.

[5] GitHub repository: https://github.com/google/s2geometry

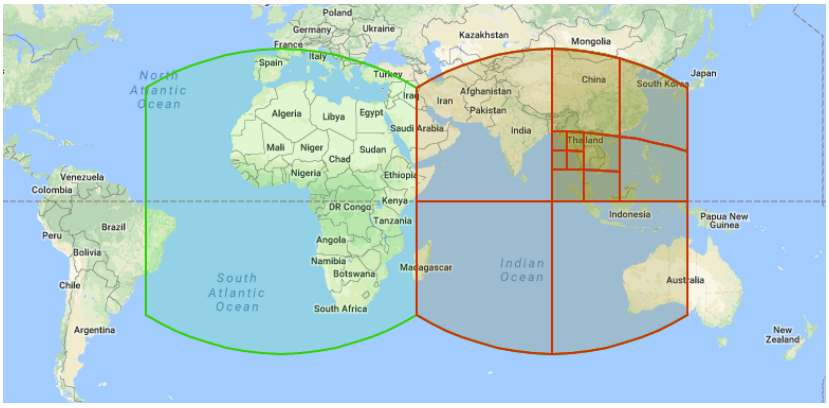


Figure 2.3: Areas addressed by cells are rectangles on the sphere's surface, but not in a two-dimensional projection, Source: [33]

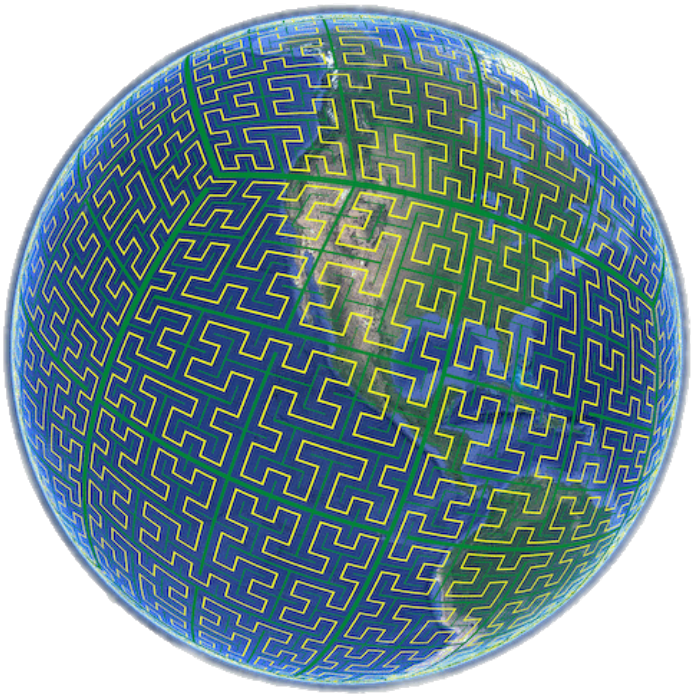


Figure 2.4: Visualization of the space-filling curve used for numbering of cells, Source: [33]

The key to efficient queries on s2's cells is the *S2CellId*. The library makes use of a space-filling curve for assigning one-dimensional 64-bit integers to each cell on the sphere's surface. The use of a fractal curve, as shown in Figure 2.4, preserves the locality. This means that geometrically close cells have numerically close identifiers. The mapping of cells to cell IDs reduces the dimension for addressing the cells, which allows the use of fast index data structures for one-dimensional data.

The s2geometry library provides support for various geometries, like points, lines and polygons. In this thesis, however, we only consider points, which suffice for many applications. For points, s2geometry offers a specific index data structure called *S2PointIndex* based on a balanced search tree, also called *B-tree* [10]:

```
164   using Map = s2internal::BTreeMultimap<S2CellId, PointData>;
```

Code 2.7: `s2point_index.h`: Type declaration of the used index data structure

For each point in an S2PointIndex, the corresponding tree stores the S2CellId of the leaf cell containing the point as a search key. Additionally the original point and a user-defined payload is stored. A multimap is used because one cell could contain multiple points. The balanced search tree is especially favorable here, because it allows fast range queries and access to sibling nodes.

3 Approach and Implementation

Multiple subproblems need to be solved to allow users to perform efficient spatial searches using the QLever SPARQL engine. While a fast implementation of the search algorithm itself might be scientifically interesting, the algorithm is only practically useful if the users' workflow is thought through from beginning to end.

For this reason, we do not only consider the efficient internal data representation and search for nearest neighbors as a part of the problem to be solved. Users should be able to quickly retrieve the data they wish in its entirety using a single SPARQL query. To accomplish this, it is equally as important to have the ability for import of all required data. Further subproblems for a maximum in usability are an assistive tool to construct complex queries and utility functions to use in these queries.

The approaches to each of these subproblems are discussed in the following sections.

3.1 Efficient Representation of Geographic Points

Currently, QLever supports the three GeoSPARQL functions given in Code 3.1.

```
1 PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
2 geof:latitude(?point)
3 geof:longitude(?point)
4 geof:distance(?point1, ?point2)
```

Code 3.1: GeoSPARQL functions supported by QLever

Each of the functions expects point geometries as WKT literals. When one of these functions is called, the ValueId of each literal is resolved into a string using the vocabulary. Then a precompiled regular expression is applied to extract the latitude and longitude coordinates from the literal string. Each of the substrings

representing coordinates is then parsed as a double precision floating point number (`double`).

The described procedure of loading strings from the hard disk and parsing the points from these strings repeatedly for every query is not efficient. Thus, we implement a faster procedure for computing query results containing calculations on geographic points. As mentioned in section 2.3, QLever already stores boolean, integer, double and date-time literals directly within the 64 bits of the ValueId. The same will now happen for geographic points from WKT literals. The literals will only be parsed from strings during the precomputation of QLever's index and if the query contains a point literal directly. The parser decides for each WKT literal it encounters if it can be folded into the ValueId. This is the case for all valid geographic points on earth, but not for other types of geometries. If the literal cannot be folded, it remains stored as a string.

With this new strategy, ValueIds containing points' coordinates are now available directly during every query processing step without any additional overhead for loading or parsing. To make this possible, we implement a new `class GeoPoint` for the internal representation of points during query processing. Similarly to the previous implementation, it stores points as two doubles.

```
35 class GeoPoint {
36  private:
37   double lat_;
38   double lng_;
```

Code 3.2: `GeoPoint.h`: Definition of attributes in the new `GeoPoint` class

The new implementation enforces an invariant: the new exception type `struct CoordinateOutOfRangeException : public std::exception {...}` is thrown for invalid latitude or longitude values that do not exist on earth. Aside parsing from string literals, GeoPoint now also allows converting points from and to the binary representation for folding in a ValueId.

Since each ValueId has a size of 64 bits, but requires 4 bits to determine the data type, 30 bits remain for storing each of the two coordinates.

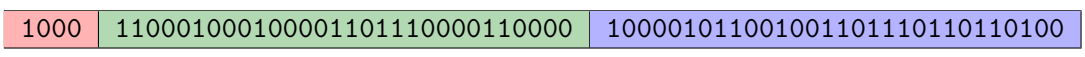


Figure 3.1: Example 64-bit *ValueId* for "Freiburg (Breisgau) Hbf" with `Datatype::GeoPoint`, latitude 47.9977308 and longitude 7.84129473

In order to convert a point given by two doubles to a ValueId it is therefore necessary to reduce the coordinates' precision to 30 bits. This should be done with the minimal possible loss of information. For this reason we scale the coordinates linearly from their range of [-90, 90] and [-180, 180] respectively to a range of [0, $2^{30} - 1$]. To obtain the new latitude value lat$'$ from a standard-scaled value lat, we apply

$$\text{lat}' = \frac{\text{lat} + 90}{180} * (2^{30} - 1).$$

Analogously to obtain the new longitude value lng$'$, we apply

$$\text{lng}' = \frac{\text{lng} + 180}{360} * (2^{30} - 1).$$

The resulting value of lat$'$ and lng$'$ has to be rounded to nearest integer to avoid unnecessary precision loss by flooring. To obtain the binary representation, the new 30 bit latitude coordinate is shifted 30 bits to the left and the new longitude coordinate is then embedded using a bitwise or operation. The data type in the 4 highest-order bits is then embedded with another or operation.

The inverse operation for decoding a geographic point from the binary representation is similar. First, the latitude coordinate is retrieved using a bitwise and operation with a bit mask which has the bits 5-34 set to 1 and all others set to 0. The result is shifted 30 bits to the right, resulting in the 30 bit integer representation lat$'$, as described above. The scaling is reversed using:

$$\text{lat} = \frac{\text{lat}'}{2^{30} - 1} * 180 - 90.$$

The longitude coordinate can be obtained using a bitwise and operation with a bit mask which has bits 35-64 set to 1 and the remaining set to 0. As we have selected the lowest-order bits, shifting is not required in this case. Therefore we can directly reverse the scaling:

$$\text{lng} = \frac{\text{lng}'}{2^{30} - 1} * 360 - 180.$$

The question, whether the precision of this approach suffices for exact results, can be answered positively. The earth's equatorial radius measures 6378.137 kilometers [69]. Therefore the maximum circumference in east-west direction

(longitude) under the simplifying assumption of earth being a perfect ellipsoid is

$$C_{lng} = 2\pi * 6378.137 \text{ km} \approx 40075 \text{ km}.$$

With earth's polar radius of 6356.752 kilometers [69] the maximum circumference in north-south (latitude) direction is

$$C_{lat} = 2\pi * 6356.752 \text{ km} \approx 39941 \text{ km}.$$

Using the range of $[0,\ 2^{30} - 1]$, each coordinate in our representation can hold $2^{30} = 1073741824$ different integer values. Therefore, in combination with the linear scaling, the range of coordinates in the standard scaling addressed by the same integer in our representation has in the worst case a size of:

$$P_{lng} = \frac{C_{lng}}{2^{30}} \approx \frac{40075 * 10^5 \text{ cm}}{2^{30}} \approx 3.7323 \text{ cm},$$

$$P_{lat} = \frac{C_{lat}}{2^{30}} \approx \frac{39941 * 10^5 \text{ cm}}{2^{30}} \approx 3.7198 \text{ cm}.$$

This means, since earth curvature can be ignored in the range of centimeters, we can approximate the precision that we can address on the earth's surface with points folded into ValueIds by a grid of cells with a maximum size of

$$P_{lat} * P_{lng} \approx 13.8833 \text{ cm}^2.$$

It can be assumed that this precision is sufficient for all civilian applications.

In accordance with the folding of points into ValueIds, we also update the implementation of the supported GeoSPARQL functions (see Code 3.1) to work with GeoPoint objects. While the optimization of folding points into ValueIds does not change the asymptotic runtime behavior or asymptotic memory consumption of any algorithm used in QLever, it creates a practically relevant and measurable speed-up. This is analyzed in the evaluation chapter.

3.2 Nearest Neighbors Spatial Search

The core problem addressed in this thesis is the efficient spatial search for geographic points close to other geographic points. In plain SPARQL and with only the `geof:distance(?point1, ?point2)` function available, this requires a cartesian product. An example for this is given in Code 3.3.

```
PREFIX ogc: <http://www.opengis.net/rdf#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
PREFIX osmrel: <https://www.openstreetmap.org/relation/>
PREFIX osmkey: <https://www.openstreetmap.org/wiki/Key:>

SELECT ?stop_name ?stop_geo
       (MIN(?dist) * 1000 AS ?min_dist_meters)
WHERE {
  # All supermarkets with location and name
  ?shop osmkey:shop "supermarket" ;
        osmkey:name ?shop_name ;
        geo:hasCentroid/geo:asWKT ?shop_geo .
  # All public transport stops with location and name
  ?stop osmkey:public_transport "platform" ;
        osmkey:name ?stop_name ;
        geo:hasCentroid/geo:asWKT ?stop_geo .
  # They must be within the city of Freiburg
  osmrel:62768 ogc:sfContains ?shop , ?stop .
  # Calculate distance
  BIND (geof:distance(?shop_geo, ?stop_geo) AS ?dist)
}
GROUP BY ?stop ?stop_name ?stop_geo
```

Code 3.3: SPARQL query to find the minimum distance from every public transport stop in Freiburg to a supermarket using a cartesian product

With larger inputs the quadratic runtime of the cartesian product quickly exceeds feasible dimensions. The example query can be answered in a fair amount of time for the city of Freiburg, it is slow but possible for the state of Baden-Württemberg, but impossible within reasonable time for Germany as a whole. Running times in detail are discussed in the evaluation chapter (5). Still, nearest neighbors searches like the one shown in the example should be efficiently possible not only for Freiburg but also for entire Germany or even the planet.

In general we understand a nearest neighbors search as a type of *join operation* between the *join column* of a *left table* and the *join column* of a *right table*. Join operations are an important concept in databases and information systems. In a join, the rows from the two given tables are combined into rows in a single result table holding the columns of both tables according to a given criteria. The simplest example is a join on the equivalence of a left and right join column. This is demonstrated in Table 3.1.

User	First Name	Last Name
105	Alice	Example
235	Bob	Demo
300	Carol	Reader

(a) Left join table

Post	User	Text
1	235	Hello World
2	105	Good Morning
3	235	How's it going?

(b) Right join table

User	First Name	Last Name	Post	Text
105	Alice	Example	2	Good Morning
235	Bob	Demo	1	Hello World
235	Bob	Demo	3	How's it going?

(c) Corresponding result table

Table 3.1: A simple example of an equivalence join. Join column: User

In a nearest neighbors search, the join columns of both tables are required to have a geographic point as a value in each row. The result of the search operation contains rows with the columns from both tables, a subset of the cartesian product. To determine which rows are included, the user may choose two search parameters: the maximum number of results and the maximum distance. The search algorithm should then find the geographically closest points from the right join column for each point from the left join column such that the parameters are satisfied. For each of these point pairs, a result row containing the column values from both input tables should be added. For better understanding of the upcoming in-depth explanations, an example is provided. Considering the input tables in Table 3.2, a nearest neighbors search for a maximum of 2 results and with a maximum distance of 8.5 is performed. The search is visualized in Figure 3.2. Here the colored points represent the points from the left join table. The black points represent the points from the right join table. The colored areas include the result points for their respective point from the left table. Finally the result table corresponding to the depicted search is given in Table 3.3.

#	Name	Point
1	Alice' House	(-4,-2)
2	Bob's House	(-2, -3)
3	Carol's House	(4, 6)

(a) Fictional left join table

#	Type	Name	Point
1	Station	R	(1,-5)
2	Supermarket	S	(1,-2)
3	Supermarket	S'	(2,-4)
4	Bakery	B	(-4,2)

(b) Fictional right join table

Table 3.2: Input tables for the nearest neighbors search example

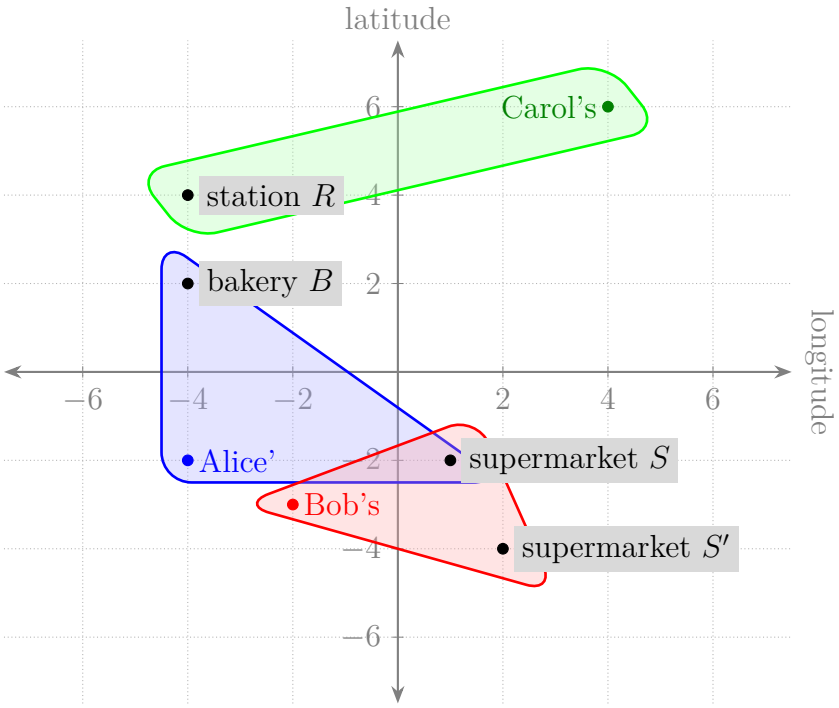


Figure 3.2: Coordinate system as a fictional map with all points from the left and right example tables, demonstrating the semantics of a nearest neighbors search with maximum number of results 2 and maximum distance 8.5

$\#_L$	Name_L	Point_L	$\#_R$	Type_R	Name_R	Point_R
1	Alice' House	(-4,-2)	2	Supermarket	S	(1,-2)
1	Alice' House	(-4,-2)	4	Bakery	B	(-4,2)
2	Bob's House	(-2, -3)	2	Supermarket	S	(1,-2)
2	Bob's House	(-2, -3)	3	Supermarket	S'	(2,-4)
3	Carol's House	(4, 6)	1	Station	R	(1,-5)

Table 3.3: Result table for the nearest neighbors search example. Columns from the left table as "column_L" and columns from the right table as "column_R"

Depending on whether or not the maximum number of results is limited, the properties of the join operation are different. If the user specifies only the maximum distance, all pairs of rows from the left and right tables are returned where the distance between the points in their join columns is at most the given maximum distance. If a maximum number of results is given, for every row from the left table, a pair with each of the rows from the right table with the closest points in their join column will be returned. At most the maximum number of such pairs is returned, no matter their distance, unless the distance is also limited. If no row from the right table meets the criteria, the row from the left table will appear nowhere in the result. This corresponds to the normal join semantics between tables, if no join partner is found.

The distinction made here – between a nearest neighbors search with or without a maximum number of results – is important. To visualize this, Figure 3.3 provides a very small, hypothetical map. We assume for our example, that the distance n is larger than the distance between any two points on the map. The maximum distance criteria is always fulfilled. It is easy to see, that a search using only a maximum distance is symmetrical: *all supermarkets and stations, whose distance is at most n meters* is equivalent to *all stations and supermarkets, whose distance is at most n meters*. In the example, both searches return one pair with station R and supermarket S and one pair with station R and supermarket S'. However using a maximum number of results, this is no longer the case: *for each supermarket the one closest station* will return supermarket S and station R as a pair and supermarket S' and station R as a second pair. But *for each station the one closest supermarket* will return only station R and supermarket S.

Also note that for the maximum distance, like with a normal equivalence join, we can decide if a pair of rows from the left and right table is part of the result by only looking at the pair in question. With a maximum number of results this can not be decided without checking if other rows have an even smaller distance.

The difference between a nearest neighbors search with or without a maximum number of results also applies to further restrictions to the join tables. It makes a difference, whether we search for the nearest neighbors first and apply restrictions afterwards or the other way round. Considering our example: if we search for the station R's nearest neighbor first, we get only the pair station R and bakery B as a result. Filtered for supermarkets, the result is empty. However filtering all points for supermarkets gets supermarket S and S'. Searching for the station

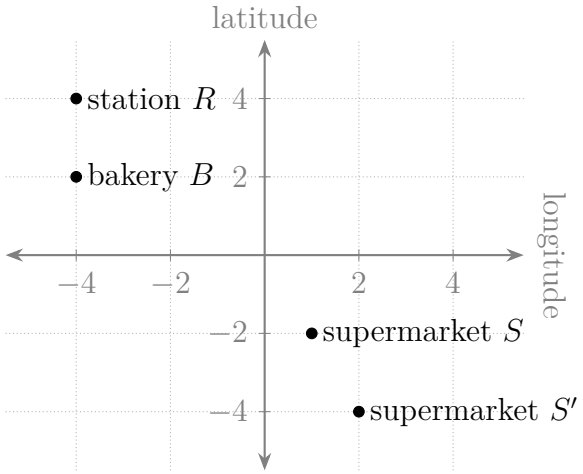


Figure 3.3: Fictional map used to illustrate the asymmetrical nature of nearest neighbors search with maximum number of results

R's nearest neighbor within the filtered points returns a non-empty result with supermarket S. The explained difference needs to be taken into consideration, while thinking about the implementation as well as possible optimizations.

In the worst case the number of result rows of a nearest neighbors search with maximum distance as a parameter is equivalent to the cartesian product. That is, for the left table A and the right table B with $|A|$ and $|B|$ rows respectively, the result could have up to $|A| * |B|$ rows. This is the case if and only if the pairwise distance between all points in the join columns of A and B is below or equal to the given maximum distance. Since this worst case scenario is presumably rare in practice, the implementation will divide the estimate for the result table size by a damping factor of 1 000 to improve query planning in the average case. In the scenario, where a maximum number of results k is given, the result size may only be up to $|A| * k$. In this case, no damping factor is used, because the worst case always occurs if no maximum distance is given and there are at least k rows in B.

In the following, we present a baseline algorithm as well as an index-based algorithm. The baseline algorithm using a nested loop, effectively a cartesian product, is implemented as a target for comparison and as a proof of concept. The efficient index-based algorithm is implemented using the index data structure for points from the s2geometry library. Both algorithms return the same results but their computation has different runtime characteristics. Additionally, we present the integration of the spatial search into the SPARQL syntax understood

by QLever.

Both algorithms currently operate on GeoPoint objects. To make the spatial search more useful we suggested to add centroids to the output of osm2rdf, which the project thankfully adopted. In newer versions of osm2rdf, centroids can now be accessed via `geo:hasCentroid/geo:asWKT`. For example, this allows performing spatial searches between buildings, which are often represented as polygons.

3.2.1 Nested-Loop Baseline Algorithm

As intended the baseline algorithm is kept fairly simple. In addition to the natural language explanation, a pseudocode version is given in Code 3.4.

The algorithm iterates over the left join table and for each row, iterates over the right join table. Thus the algorithm calculates the distance for every pairwise combination of rows from the left and right tables. Results are filtered according to the search parameters. If only a maximum distance is given, rows can be omitted or added to the result immediately. In the case of a search with the maximum number of results parameter k, for every row from the left table, a priority queue ordered largest-first is initialized. It holds the k closest points processed so far. If a new point is processed it is added to the priority queue. If the priority queue exceeds k in size, the first (largest) element is removed. Once all rows from the right table have been processed, the rows stored in the priority queue correspond precisely to the k rows with the closest points and can be added to the result.

In the case of the baseline algorithm it is not necessary to consider optimizations regarding which table is larger, because the full cartesian product is iterated in any case.

Using the tables A and B, the running time of the baseline algorithm is in $\mathcal{O}(|A| * |B|)$ due to the algorithm's nested loop. The priority queue being used has no impact on the asymptotic running time with regard to the size of the input tables. Since the size of the priority queue is capped at the constant $k + 1$ during any step of the processing, its operations are in $\mathcal{O}(1)$ regarding the input's size. The priority queue also has a constant running time characteristic in the real world because k is typically a very small constant, often even 1.

Baseline Algorithm
Input: left join table A with join column a, right join table B with join column b, maximum number of results or none, maximum distance or none
Output: result table

```
1: for all row_A ∈ A do
2:     Q ← new PriorityQueue(Sort=LargestFirst)
3:     for all row_B ∈ B do
4:         if row_A[a] or row_B[b] are not instance of GeoPoint then
5:             continue
6:         end if
7:         d ← dist(row_A[a], row_B[b])
8:         if d > maximum distance then
9:             continue
10:        end if
11:        if maximum number of results is not set then
12:            addResultRow(row_A, row_B, d)
13:        else
14:            Q.add(SortKey=d, Data=&row_B)
15:            if |Q| > maximum number of results then
16:                Q.removeFirstElement()
17:            end if
18:        end if
19:    end for
20:    if maximum number of results is set then
21:        for all q ∈ Q do
22:            addResultRow(row_A, *q.Data, q.SortKey)
23:        end for
24:    end if
25: end for
```

Code 3.4: Pseudocode for the baseline algorithm for nearest neighbors search

The current memory consumption excluding the result table is $\mathcal{O}(|A| + |B|)$, because the left and right tables are fully materialized. In the future, the left table could be iterated and would not need to be stored. Since the right table needs to be iterated multiple times and should not need to be recalculated, it will still need to be materialized.

3.2.2 Efficient Index-Based Algorithm

The efficient index-based algorithm makes use of the S2PointIndex. A pseudocode version is given in Code 3.5. The basic idea is as follows: for each query (*ad hoc*) a new S2PointIndex is constructed in memory. All points from the right table together with the index of their row as payload are inserted into the S2PointIndex. Then for each row from the left table, the index is queried for the nearest neighbors according to the search parameters. For each result, the row from the right table is fetched using its index and the pair is added to the result.

In case of a search with only a maximum distance, the tables can be switched for optimization if the right table is larger.

The point index cannot be precomputed because the user may specify arbitrary criteria to construct the join tables. An algorithm which searches for nearest neighbors in a precomputed index containing all points and omits the results until the current one matches the other query requirements would lead to a cartesian product of the left table with all points in the worst case.

Recalling the background on the s2geometry library (2.5), the S2PointIndex is a B-tree containing the S2CellIds of the points in the index as search keys. Insert as well as search operations in a B-tree have a running time of $\mathcal{O}(\log n)$ for a tree with n nodes [10]. The nearest neighbors algorithm therefore requires a running time of $\mathcal{O}(|B| * \log |B|)$ to build the index, because for every row the point needs to be inserted. Then $\mathcal{O}(|A| * \log |B|)$ is required to lookup the result from the index. The total running time for building and querying the index is $\mathcal{O}((|A| + |B|) * \log |B|)$.

The current memory consumption excluding the result table is $\mathcal{O}(|A| + |B|)$, because the left and right tables are fully materialized. Since the index needs to be built, only the left table could be iterated without being fully materialized. The memory consumption could thus be $\mathcal{O}(|B|)$ after a possible implementation of lazy evaluation for the spatial search in the future.

Index-Based Algorithm
Input: left join table A with join column a, right join table B with join column b, maximum number of results or none, maximum distance or none
Output: result table

```
1: if maximum number of results is not set and |B| > |A| then
2:     A, B = B, A
3: end if
4: I ← new S2PointIndex
5: for all row_B ∈ B do
6:     if row_B[b] is instance of GeoPoint then
7:         p ← ConvertToS2Point(row_B[b])
8:         I.insert(Point=p, Data=&row_B)
9:     end if
10: end for
11: Q ← new S2ClosestPointQuery(I, maximum distance, maximum results)
12: for all row_A ∈ A do
13:     if row_A[a] is instance of GeoPoint then
14:         p ← PointTarget(ConvertToS2Point(row_A[a]))
15:         for all r ∈ Q.FindClosestPoints(p) do
16:             d ← ToKilometers(r.Distance)
17:             addResultRow(row_A, *r.Data, d)
18:         end for
19:     end if
20: end for
```

Code 3.5: Pseudocode for the index-based algorithm for nearest neighbors search

3.2.3 Integration into SPARQL Syntax

In order to make the nearest neighbors spatial search available to users of the QLever SPARQL engine, it needs to be integrated into the SPARQL syntax. This is primarily achieved by leveraging the generally available federated querying syntax `SERVICE <IRI> { … }` with a special IRI. This IRI is never actually contacted via network and there is no API actually hosted at this address. The IRI only serves as an identifier to activate the spatial querying feature in QLever. The `SERVICE` subquery is used to provide the spatial search with the required configuration parameters. Additionally it includes a SPARQL group graph pattern `{ … }` defining the right join table. In case of a nearest neighbors search using only a maximum distance, since it is symmetrical, its right table may also be defined outside of the `SERVICE` subquery. In this case the group graph pattern must be omitted. The left join table is always defined outside of the `SERVICE` subquery and is automatically selected as a join table by the query planning implementation in QLever.

A complete example showcasing all configuration parameters is given in Code 3.6. Only the left and right variables indicating the join columns and at least one of `spatialSearch:numNearestNeighbors` and `spatialSearch:maxDistance`, given in meters, are mandatory.

The selected algorithm `spatialSearch:s2` refers to the index-based algorithm from 3.2.2. The baseline algorithm from 3.2.1 can be activated with `spatialSearch:baseline`. Additionally `spatialSearch:bindDistance` may be used to add a column with the given variable name containing the distance between the left and right points. Because the distance has to be computed during the search itself, this avoids redundantly recomputing the distance using a `BIND(geof:distance(?left, ?right) AS ?dist)` statement.

Unlike the theoretical example for a nearest neighbors search in Figure 3.2, the implementation in QLever will not automatically add all columns of the right join table to the result: by default it includes only the join column. This is done as an optimization to remove columns necessary only for obtaining the right join table from further query processing steps. Instead the user may explicitly select columns from the right join table to include in the result using the `spatialSearch:payload` configuration parameter. It can be repeated once for every variable to be included or can be set to `spatialSearch:all` to achieve the same behavior

as the theoretical example. Details on the query syntax can now be found on the QLever Wiki[1].

```
PREFIX osmkey: <https://www.openstreetmap.org/wiki/Key:>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX spatialSearch:
 ↪ <https://qlever.cs.uni-freiburg.de/spatialSearch/>

SELECT ?transport_name ?restaurant_name ?distance WHERE {
  ?transport  osmkey:public_transport "platform" ;
              osmkey:name ?transport_name ;
              geo:hasCentroid/geo:asWKT ?transport_geo .

  SERVICE spatialSearch: {
    _:config  spatialSearch:algorithm spatialSearch:s2 ;
              spatialSearch:left ?transport_geo ;
              spatialSearch:right ?restaurant_geo ;
              spatialSearch:numNearestNeighbors 1 ;
              spatialSearch:maxDistance 500 ;
              spatialSearch:bindDistance ?distance ;
              spatialSearch:payload ?restaurant_name .
    {
      ?restaurant osmkey:amenity "restaurant" ;
                  osmkey:name ?restaurant_name ;
                  geo:hasCentroid/geo:asWKT ?restaurant_geo .
    }
  }
}
```

Code 3.6: Example of a spatial search showcasing all supported configuration parameters: "Get all pairs of public transport stops and their nearest restaurant with names and distance, where the restaurant is at most 500 meters away."

During QLever's query parsing the special `SERVICE` needs to be detected. This is done simply by comparing the IRI of `SERVICE` operations against the special IRI for the internal feature. If the special IRI is detected, the `SERVICE` operation is no longer treated as such. Instead, another function is called for translating the parsed query into QLever's internal representation. The function builds a `struct SpatialQuery`, which holds the configuration parsed so far and `std::nullopt` for all other parameters. This struct inherits from a newly

[1] https://github.com/ad-freiburg/qlever/wiki/GeoSPARQL-support-in-QLever

introduced `struct MagicServiceQuery`, which generalizes a special `SERVICE` so it can be used for other features as well.

For every parsed triple contained directly inside the `SERVICE` a method is called to treat the triple as a configuration parameter. The method removes the special IRI if applicable. This allows defining configuration parameters in two forms: `spatialSearch:parameter` or `<parameter>`. The method also checks that the value in the object of the triple is allowed for the configuration parameter given by the predicate. Then the configuration is added to the SpatialQuery object.

If a group graph pattern is found, it is parsed using the regular implementation and then added as a child to the SpatialQuery object.

As soon as all query code inside the `SERVICE` has been parsed, the SpatialQuery object is converted to a `struct SpatialJoinConfiguration`. Unlike the SpatialQuery it must be a complete and valid configuration. Therefore all invariants are checked during the conversion and an exception is thrown if they are violated. The SpatialJoinConfiguration is required as an input for constructing an object of the `class SpatialJoin`, which implements the actual query processing operation.

In order to differentiate between the symmetrical nearest neighbors search with a maximum distance and the asymmetrical nearest neighbors search with a maximum number of results and optionally a maximum distance, the *task* of the spatial join is defined as:

```
29 using SpatialJoinTask = std::variant<NearestNeighborsConfig,
   ↪  MaxDistanceConfig>;
```

Code 3.7: `SpatialJoin.h`: Type declaration of the spatial join task

The internal representation of the payload configuration parameters is a `class PayloadVariables`. It holds either a `std::vector<Variable>` or an instance of a `struct PayloadAllVariables : std::monostate`. The latter results in an empty one-byte object representing the selection of all variables.

The implementation of payload variables in the spatial join operation requires attention to detail. In order to conform with the SPARQL standard, undefined variables only result in a warning and are ignored. Requesting all variables and additionally stating explicit payload variables is allowed, but ignored. Variables

may be requested as payload multiple times but are included only once. The variable corresponding to the join column may or may not be selected, but is always included in the result.

The number of result columns is thus computed using a hash set of the given payload variables. If the join variable is not included one is added and if a distance variable was requested one more column is counted.

Up until now, operations removing columns from the result (like a `SELECT` subquery) simply declared the columns to be removed as invisible – they were no longer exported to the user in the final result. All of the columns had to be passed around during query processing anyway. This approach would have rendered the payload option useless, since its main intention is removing columns to save memory and computational resources. Therefore the spatial join operation truly removes the columns from the result.

In order to implement payload variables in the spatial join algorithms, a `std::vector<ColumnIndex> rightSelectedCols` is passed to the `SpatialJoinAlgorithms` object with the sanitized column indices to include in the result rows. The method `addResultTableEntry` then filters the columns of each row to be included in the result.

QLever stores intermediate tables as a sequence of ValueIds. Each row consists of exactly 64 bits per column. The individual rows are stored directly after each other. For this implementation the column names and indices are kept separately in a `VariableToColumnMap`. Like the name suggests, it maps column names to indices. Each operation implements its own `computeVariableToColumnMap` method. For the spatial join operation, this method needs to copy the variable to column map of the left table. For the following columns the method needs to filter the variable to column map of the right table according to the payload variables and close gaps between the indices to ensure consecutive column indices. Additionally the method needs to ensure that the join column is always included and, if requested, the distance variable is added as a last column.

A few details regarding query planning have to be considered when implementing the spatial join. In general the query planner builds different candidate query plans. A query plan is an order of applying operations to compute the query result and is mainly built from index scans for specific triples and join operations. The query planner calculates the result size and running time estimates for each of the operations and uses them to decide on an optimal query plan.

Since the spatial join can have left and right child tables which are computed using arbitrary subqueries, they have individual query plans. The query planner needs to combine the two query plans for the child tables in question only using a nearest neighbors search as a join operation. Any other join would destroy the intended semantics. Therefore, when building query plan candidates for any type of join, a spatial join needs to have priority. If operations to be combined into a query plan are tables of a spatial join, all other query plan candidates are omitted. However if the query plans for the child tables are already part of a spatial join's query plan, the spatial join query plan is treated atomically like any other operation. This is due to the fact that a regular join which uses the spatial join's result could be necessary.

To abbreviate the `SERVICE` syntax for spatial searches using only a maximum distance, a special predicate `<max-distance-in-meters:m>` is also supported. Due to the symmetrical properties of the maximum distance spatial search, it may be used along other triples that would otherwise invoke the computation of a cartesian product. Therefore this syntax can be used to easily migrate from a cartesian product with a `FILTER` statement. `FILTER(geof:distance(?left, ?right) <= 0.5)` simply becomes `?left <max-distance-in-meters:500> ?right`. An example is given in Code 3.8.

The special predicate syntax is implemented using the same configuration objects as the `SERVICE` syntax. During query planning every literal IRI contained in a predicate in the SPARQL query is processed. If it begins with the special predicate's prefix, it is parsed using a regular expression. The subject of the triple in question becomes the left variable, the object the right variable. The maximum distance parameter is set and just like a `SERVICE` query with the right join table defined outside of the `SERVICE`, the payload parameter is set to all implicitly. The remaining query processing is identical to the `SERVICE` syntax.

A special predicate syntax for nearest neighbors search with a maximum number of results is not favorable. This special predicate would have unstable semantics, because of the previously explained difference depending on the order of operations. With this syntax the order of operations would be determined by the query planner's result size and time estimates for the individual operations which depend on the input data. Therefore the exact same query using such a special predicate would have different semantics on different input data.

```
PREFIX osmkey: <https://www.openstreetmap.org/wiki/Key:>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
PREFIX spatialSearch:
 ↪ <https://qlever.cs.uni-freiburg.de/spatialSearch/>

SELECT ?transport_name ?restaurant_name ?distance WHERE {
  ?transport osmkey:public_transport "platform" ;
             osmkey:name ?transport_name ;
             geo:hasCentroid/geo:asWKT ?transport_geo .

  ?restaurant osmkey:amenity "restaurant" ;
              osmkey:name ?restaurant_name ;
              geo:hasCentroid/geo:asWKT ?restaurant_geo .

  ?transport <max-distance-in-meters:500> ?restaurant .

  BIND(geof:distance(?transport_geo, ?restaurant_geo)
       AS ?distance)
}
```

Code 3.8: Example of a spatial search using the abbreviated special predicate syntax: “Get all pairs of public transport stops and restaurants with names and distance, which are at most 500 meters away from each other.”

3.3 SPARQL Functions

In this section we present improvements and additions of useful SPARQL functions to the QLever engine.

3.3.1 Precision Improvement of Geographic Distance Function

We migrate the calculation of distances between points using the GeoSPARQL function `geof:distance(?point1, ?point2)` to use the distance computation provided by the s2geometry library. The previously used formula worked with a planar projection of an ellipsoidal approximation of earth published by the United States Federal Communications Commission [25]. The old formula is only sufficiently exact up to distances of 475 kilometers. In contrast, s2geometry has low distortion for the entire planet due to the approximation using a three-dimensional sphere instead of a two-dimensional projection.

The following example calculates the distance between Berlin and Tokyo. The new implementation returns 8 915.55 kilometers, the old implementation 10 282.2 kilometers. With most online sources stating values of about 8 900 to 9 000 kilometers, the old implementation has a distortion of more than 1 300 kilometers.

```
1 PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
2 PREFIX geo: <http://www.opengis.net/ont/geosparql#>
3 SELECT (geof:distance(
4             "POINT(13.411400 52.523403)"^^geo:wktLiteral,
5             "POINT(139.691711 35.689487)"^^geo:wktLiteral)
6         AS ?berlin_tokyo) WHERE {}
```

Code 3.9: GeoSPARQL query to compute the distance between Berlin and Tokyo

3.3.2 Support for Exponentiation Math Function

Many mathematical functions from the XPath standard [74] by the W3C were already implemented in QLever. However `math:pow(?base, ?exp)` was still missing. Raising a number to a power could only be accomplished by applying

`math:exp(?x)` together with `math:log(?x)` according to mathematical power laws:

$$a^x = e^{x \ln(a)}.$$

The new implementation directly makes use of the C++ standard library's `double std::pow(double base, double exp)` function. It produces more precise results compared to the previous method, which required the chaining of multiple numerical techniques with precision loss. An example to demonstrate the improved precision is given in Code 3.10. While the new implementation calculates 2^{50} correctly, the previous method misses the correct result by 152.

```
PREFIX math: <http://www.w3.org/2005/xpath-functions/math#>
SELECT  (math:exp(?exp * math:log(?base)) AS ?old)
        (math:pow(?base, ?exp) AS ?new)
WHERE {
  BIND(2 AS ?base)
  BIND(50 AS ?exp)
}
```

Code 3.10: Example usage of the new exponentiation function, where it returns a more precise result than the previous method

3.3.3 Support for Standard Deviation Aggregation Function

QLever, aiming for complete compliance with the SPARQL standard, has support for the standard aggregation functions for numerical values: `min(?numbers)`, `max(?numbers)`, `avg(?numbers)`, `sum(?numbers)` and `count(?numbers)`. It also supports the XPath function for the square root `math:sqrt(?number)`. The standard deviation can be calculated using these functions according to its formula:

$$\sigma(x) = \sqrt{\frac{\sum_{x_i \in x} (x_i - \overline{x})^2}{|x| - 1}}.$$

However the corresponding lengthy function call is inconvenient to the user, Code 3.11 shows an example. Additionally, if the expression of which the standard deviation is to be calculated requires further computation it would be calculated multiple times. Both can now be avoided using the new `stdev(?numbers)` aggregation function. For an example, see Code 3.12.

```
# Query for standard deviation previously
PREFIX math: <http://www.w3.org/2005/xpath-functions/math#>
SELECT  (math:sqrt(
            sum(
                (?var - avg(?var)) * (?var - avg(?var))
            ) / (count(?var) - 1)
        ) AS ?old)
WHERE {
  VALUES ?var { 1 2 3 4 5 6 7 8 9 }
}
```

Code 3.11: Example usage of the query code previously required for the computation of the standard deviation

```
# Query for standard deviation now
SELECT (stdev(?var) AS ?new) WHERE {
  VALUES ?var { 1 2 3 4 5 6 7 8 9 }
}
```

Code 3.12: Example usage of the new standard deviation aggregation function

Because the new function is an aggregation function, the implementation requires more steps. QLever parses its queries using the Antlr [56] library. The SPARQL syntax understood by QLever is given as a Antlr grammar. Since implementation details in QLever require that the parser can differentiate between aggregate and non-aggregate functions, the new `stdev(?numbers)` function needs to be added to the grammar.

In QLever an *aggregate expression* consists of an *aggregate operation*, which implements how the results are reduced and an optional *aggregate final operation*, which can modify the single reduced value from the aggregate operation. For example to implement `avg(?numbers)`, the aggregate operation is addition and the aggregate final operation divides by the number of rows.

To implement the standard deviation using this scheme, a helper non-aggregate *DeviationExpression* is defined. This is due to the standard deviation requiring two passes over the data: one to calculate the data's arithmetic mean and one to calculate the squared deviation using the arithmetic mean. In order to prevent having to calculate the child expression inside the standard deviation expression twice, the results of the child expression are buffered. During the buffering, the

data is summed to calculate the mean. Then the individual rows are replaced by the corresponding squared deviation.

With the helper DeviationExpression, we can define an aggregation expression which replaces its child expression with a DeviationExpression containing its former child and then calculates the sum as an aggregation operation. The aggregation final operation receives the sum of squares, divides by the degrees of freedom and finally applies the square root. The implementation has a linear running time and memory requirement in relation to the number of input rows.

Additionally, it is ensured that a single undefined value in the child expression makes the entire result undefined. As a convenience feature, an empty or one-element child expression has a standard deviation of 0 in this implementation.

3.4 Conversion of External Data Sets to RDF

In order to take advantage of QLever's new spatial search features, the users must be able to import the data they want to work with into QLever. The data must thus be converted to RDF. One of the biggest benefits of RDF and SPARQL is the ease with which different data sets can be combined and queried as one using a single user query. For this reason, along with formats for spatial data, also some non-spatial data formats and sources are considered here.

We introduce four new free and open-source tools[2], licensed under GPL, for the conversion of different data formats to RDF knowledge graphs in turtle format: csv2rdf, kml2rdf, gtfs2rdf and election2rdf. The programs are implemented in Python and are self-sufficient: they have no dependencies and only require modules from Python's standard library. Each of the programs can be used as a CLI or be imported as a library for further implementations. Detailed documentation on implementation details is not covered by this thesis but can be found in the *docstrings* and *readmes* provided in and with the source code.

In the following, the most important aspects of each of the programs are described.

[2]Source code and documentation available at https://ullinger.info/bachelor-thesis

3.4.1 Keyhole Markup Language (KML)

KML [50] is a data format for storing geographic information, usually named shapes like points, lines or polygons. It is based on the XML standard. We present kml2rdf, a program that is able to convert the most important information from a KML document to an RDF turtle file. Most importantly, the geometries from KML are converted to `geo:wktLiteral`.

In Code 3.13 an example KML file containing a point for the main building of the Technical Faculty of the University of Freiburg is given.

```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2">
  <Document>
    <Folder>
      <Placemark id="example_placemark_101">
        <name>Building 101</name>
        <description>
          Main Building of the Technical Faculty of Uni Freiburg
        </description>
        <Point>
          <coordinates>
            7.8350223,48.012647,0
          </coordinates>
        </Point>
      </Placemark>
    </Folder>
  </Document>
</kml>
```

Code 3.13: Example KML document containing a single point

As shown in the example, a `<Placemark>` element is used as a container for a geometry together with its associated metadata. The most important metadata fields are the Placemark's identifier in `<Placemark id="...">` as well as the `<name>` and `<description>`. Common geometries are `<Point>`, `<LineString>`, `<Polygon>` and `<MultiGeometry>`. A polygon always consists of an outer line using `<outerBoundaryIs>`. Optionally it can have holes using `<innerBoundaryIs>`. A multi geometry is a collection of multiple geometry elements.

The kml2rdf program parses its input file using `xml.etree.ElementTree` from the Python standard library. Since there are multiple possible XML

namespaces for the KML format – like `<kml xmlns="http://www.opengis.net/kml/2.2">` in this example – we extract and validate the namespace first. In the following steps the parsed KML file will be queried using XPath to extract all placemarks with their geometries and metadata.

The program models the entire KML file to be converted using a `class KMLDataset(Dataset)`. This class creates an object of the `class KMLXPathHelper`, which generates the appropriate XPath queries to execute on subtrees of the KML file. Using the XPath helper object, all `<Placemark>` elements are extracted and represented internally as objects of a `class KMLPlacemark`. The object holds the optional identifier, name and description. Furthermore it initiates the extraction of the geometries and stores them as an intermediate representation in an object model.

The intermediate object model for geometries is based on an abstract base class `class Geometry(ABC)`. Its subclasses

- `class Point(Geometry)`,
- `class LineString(Geometry)`,
- `class Polygon(Geometry)` and
- `class GeometryCollection(Geometry)`

implement the different geometry types stated previously. Each class supports the extraction from a parsed KML `<Placemark>` subtree as well as the export as a WKT literal. An advantage of this intermediate representation is that it could be extended easily to support further formats, like GPS Exchange Format (`.gpx` files), Geography Markup Language (`.gml` files) or Scalable Vector Graphics (`.svg` files).

In addition to `<LineString>` elements, line strings can also be read from geometries using an extension syntax' `<gx:Track>` element.

In WKT, homogeneous collections of multiple geometries, consisting only of elements of the same type, have different names: `MULTIPOINT`, `MULTILINESTRING`, `MULTIPOLYGON`. Otherwise collections are called `GEOMETRYCOLLECTION`. While KML does not have this differentiation and stores every collection as a `<MultiGeometry>` element, kml2rdf automatically detects homogeneous collections and selects the geometry type for its output WKT accordingly.

KML files can be stored as plain text XML (`.kml`) or can be stored as a compressed ZIP file (`.kmz`), which may contain arbitrary files including exactly

one `.kml` file. kml2rdf supports both formats by extracting compressed files on user request.

With the user-defined prefix `userprefix:` and data set name `my_places`, the triples for our example point from Code 3.13 are:

```
1 userprefix:1 a userprefix:my_places .
2 userprefix:1 rdfs:label "Building 101" .
3 userprefix:1 rdfs:comment "Main Building of the Technical Faculty
  ↪ of Uni Freiburg" .
4 userprefix:1 dct:identifier "example_placemark_101" .
5 userprefix:1 geo:hasGeometry userprefix:1_geo .
6 userprefix:1_geo geo:asWKT "POINT(7.8350223
  ↪ 48.012647)"^^geo:wktLiteral .
```

Code 3.14: Example kml2rdf output for the point from the example KML document in Code 3.13

To allow the precomputation of spatial relations, like containment or intersection, also between geometries from KML and from OSM, we suggested to the osm2rdf maintainers to allow reading further auxiliary geometries from a text file containing WKT literals. This was implemented and therefore we also implement the export of such auxiliary geometry files for osm2rdf in kml2rdf using an *aux geo callback*.

Because the entire XML is parsed as a tree and each entry is processed once, the time and space complexity of the program is $\mathcal{O}(n)$ for input size n.

3.4.2 Comma-Separated Values (CSV)

Presumably the most universal data exchange format for strings and numbers organized in tables is CSV [62]. In general the format simply is a plain text file containing rows as lines with the first row stating the column names. The columns in each row are separated by a comma. There are multiple variations of this format with different separator characters – for example TSV which uses a tabulator character. Occurrences of the separator character in the data are usually escaped by a quote character (`"`) and the quote character by two quote characters.

```
id,name,ref,geo
21769883,Freiburg (Breisgau) Hauptbahnhof,RF,lat=47.9977;lng=7.84129
27666887,Freiburg Messe/Universität,RFMU,lat=48.0129;lng=7.83277
2870211785,Freiburg-Zähringen,RFZ,lat=48.0245;lng=7.86389
2870227214,Freiburg-Herdern,RFHE,lat=48.0085;lng=7.85126
2870258631,Freiburg-Sankt Georgen,RFSG,lat=47.9755;lng=7.80292
4597125464,Freiburg-Littenweiler,RFLT,lat=47.9817;lng=7.89557
673462134,Freiburg-Wiehre,RFWI,lat=47.9825;lng=7.85471
7296418724,Freiburg-Landwasser,RFLW,lat=48.0282;lng=7.81166
8434675930,Freiburg Klinikum,RFK,lat=48.0059;lng=7.84237
```

Code 3.15: Example CSV data set containing all railway stops in Freiburg, Data Source: [54]

We introduce the csv2rdf program for conveniently transforming tables in CSV format to RDF turtle. The program implements a `class CSVDataset(Dataset)`, which uses the Python standard library `csv.DictReader`. Thanks to Python's iterable objects, the CSV file does not need to be loaded into memory as a whole. The input is processed line by line, allowing the processing of very large CSV files with limited memory. Because each row is processed once, but not stored, the time complexity is in $\mathcal{O}(n)$ for input size n, but the space complexity is in $\mathcal{O}(1)$.

By default, each row is treated as one entity. Each cell, a column value for the given row, will result in one output triple. The subject and predicate are prefixed with a prefix defined by the user. The result will then have the form `userprefix:row_entity userprefix:column_name "cell_value"`.

Since the CSV format stores no data types and encodes everything as strings, the RDF output would contain all literals as strings. To avoid this behavior, the common datatypes `xsd:integer`, `xsd:decimal`, `xsd:date`, `xsd:dateTime` and `geo:wktLiteral` are detected and added by csv2rdf using regular expressions.

The row entity is an IRI derived from an internal counter of the program by default. If the CSV data contains a unique column to be used as the row entity, the user may specify a *primary column*. In our example in Code 3.15, this could be the `ref` (or `id`) column. A triple stating a user configured `rdf:type` (shortcut `a`) for this entity is automatically emitted.

Not all column names from CSV are suitable as predicates. For example, IRIs should not contain whitespace characters, but CSV allows them. Furthermore the user may want to select predicates and their prefixes for certain columns.

These issues are addressed in csv2rdf by the user configuration option *column mapping*. It is a JSON object map from CSV column names to RDF predicates or `null`. If the value `null` is given, the column will be omitted. If a string value without a colon (`:`) is given, the generally configured user prefix is added, otherwise the column mapping's prefix takes priority. Consider `{"id": null, "name": "rdfs:label"}` in our example.

Not only the modification of column names is possible, but also the modification of cell values. For each of the predicates after applying the column mapping, the user may specify a list of arbitrary regular expression search and replace patterns using the *values mapping* configuration. The replacements will be applied to the object of the output triple. The configuration also takes a flag, whether the result after applying replacements should be treated as a literal or an IRI. In the example, this could be `{"geo": [["lat=(\\d+\\.\\d+);lng=(\\d+\\.\\d+)", "POINT(\\2 \\1)"]], "lit"]}`.

Additionally, when using csv2rdf as a library, an *extra triple callback* may be registered for each data set. This function will be called once for each row from the CSV file. The function's arguments contain the the current row as a Python dictionary `dict[str, str]`, as well as the row's subject. The program expects the extra triple callback to return an iterable object which produces additional triples for this row. The triples are forwarded to the output.

With the example data set and configuration, assuming the user-chosen prefix is `userprefix:`, the result for the first row is shown in Code 3.15:

```
1 userprefix:RF a userprefix:station .
2 userprefix:RF rdfs:label "Freiburg (Breisgau) Hauptbahnhof" .
3 userprefix:RF userprefix:ref "RF" .
4 userprefix:RF userprefix:geo "POINT(7.84129
  ↪ 47.9977)"^^geo:wktLiteral .
```

Code 3.16: Example csv2rdf output for a single row from the example CSV data set in Code 3.15, Data Source: [54]

Of course the csv2rdf program allows for the conversion of any spatial or non-spatial data in CSV format. Example use cases for spatial data (3.4.3) and for non-spatial data are discussed in the following (3.4.4).

3.4.3 General Transit Feed Specification (GTFS)

GTFS [32] is a standardized data model and format to store public transport information. The format includes for example stops with their locations and names, timetables when services are operated, geometries of the routes taken by vehicles and much more.

A GTFS Feed is transmitted as a ZIP archive file. It contains the tables of a relational database with a standardized schema stored as individual CSV files. Linked GTFS [52] is an additional standard for representing public transport information in an RDF knowledge graph.

We introduce gtfs2rdf, a program to translate GTFS to Linked GTFS. It supports a superset of all mandatory tables from the GTFS standard. The supported tables, as well as their primary keys (underlined) and foreign keys (dashed underlined) are shown in Figure 3.4. The arrows in the diagram represent the foreign key relations. A few attributes relevant for the further explanations are also included. Since the GTFS format is directly based on CSV, the program's implementation reuses the implementation of csv2rdf presented in 3.4.2. Unlike the no longer maintained Node.js reference implementation by the Open Transport Working Group, gtfs2rdf automatically adds GeoSPARQL triples to support `geo:hasGeometry/geo:asWKT`. At user request, the program can also convert the routes taken by vehicles to WKT `LINESTRING` literals. They are aggregated from the individual coordinates stored in the *shapes* table. The coordinates in this table are annotated by a *sequence* number as a sort key. gtfs2rdf sorts and combines the coordinates into a WKT literal.

The Linked GTFS standard does not directly correspond to the columns from the GTFS CSV files being translated into triples. Therefore gtfs2rdf's `class GTFSFeed` builds objects for each table from the `class CSVDataset` that make extensive use of the column mapping, values mapping and extra triple callback. The latter is responsible for adding the geometries of points from *stops* and *shapes*. If line strings were requested the extra triple callback is also used to collect the points in a dictionary. The dictionary holds for each shape, the point coordinates and sequence numbers. In the end, the points of each shape are sorted according to their sequence numbers. This is necessary as any points of any shape may occur in arbitrary order in the CSV file. This procedure requires all shapes to fit into memory, thus a space complexity of $\mathcal{O}(n)$ for input size n. Since sorting is

required the worst case time complexity is $\mathcal{O}(n * \log n)$. The worst case would apply if there is only one route to which all shape points belong. Otherwise, if line strings are not requested, gtfs2rdf can process all files line by line and does not require them to fit into memory. The space complexity in this case is $\mathcal{O}(1)$ and the time complexity is $\mathcal{O}(n)$.

Examples showcasing the useful querying possibilities of GTFS data in QLever are given in Code 3.17 and Code 3.18. The queries demonstrate a search for the names and distances of all places reachable from "Freiburg (Breisgau) Hbf" without change and a search for all stops with their average number of daily departures throughout the week. Two further examples are shown in Figure 3.5. The line geometries produced by gtfs2rdf in combination with QLever's fast map view feature, QLever Petrimaps[3] [7], allow easily viewing routes from query results.

Because GTFS is a standardized format, easily importing GTFS data into QLever also opens many further possibilities. Tools working with GTFS data can now be used with QLever. For example, the pfaedle[4] [5] program from the Chair for Algorithms and Data Structures of the University of Freiburg, which improves the quality of GTFS shapes. Another example is the pdf2gtfs[5] [36] tool introduced in a fellow student's thesis for the extraction of timetables from PDF files.

[3]GitHub repository: https://github.com/ad-freiburg/qlever-petrimaps
[4]GitHub repository: https://github.com/ad-freiburg/pfaedle
[5]GitHub repository: https://github.com/heijul/pdf2gtfs

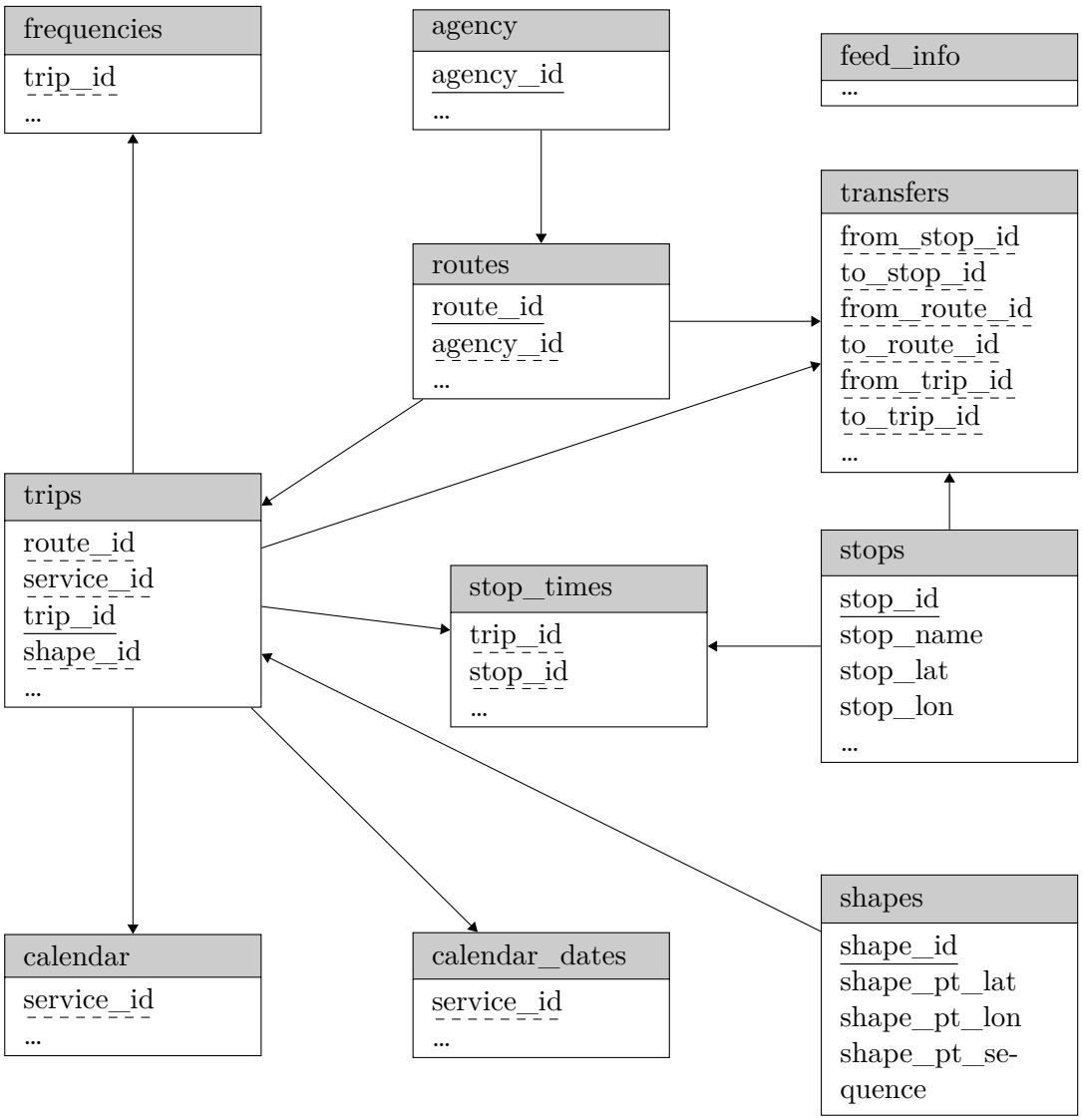


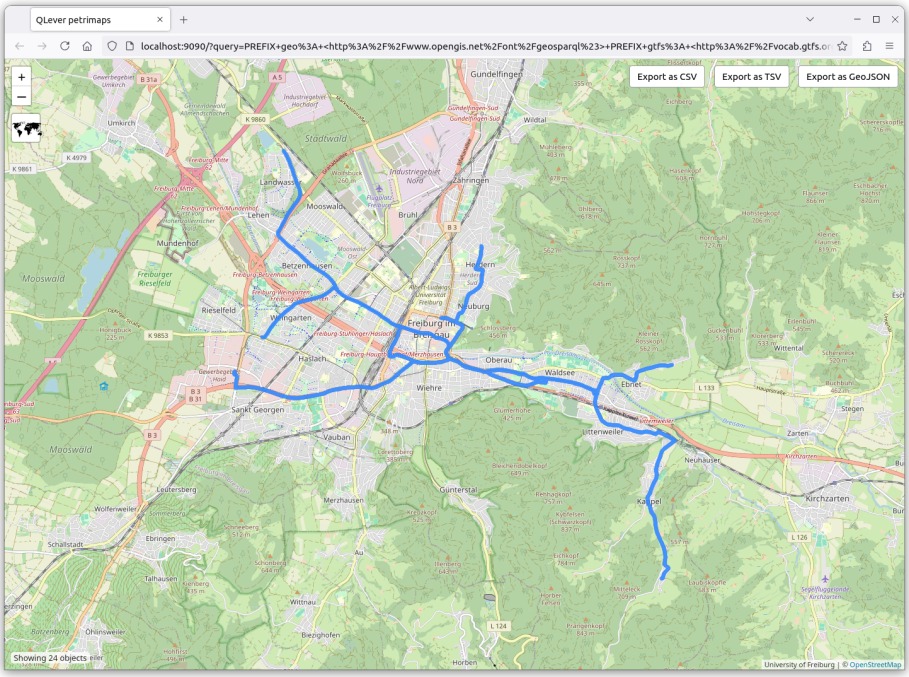
Figure 3.4: Diagram of GTFS tables supported by gtfs2rdf and their primary keys, foreign key relations (arrows) and attributes relevant for further explanations

```
PREFIX gtfs: <http://vocab.gtfs.org/terms#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>

SELECT ?stop_name (MIN(?dist) AS ?dist_freiburg_hbf) WHERE {
  ?stoptime1 a gtfs:StopTime ;
             gtfs:trip ?trip ;
             gtfs:stop ?stop1 ;
             gtfs:stopSequence ?seq1 .
  ?stop1 foaf:name "Freiburg Hauptbahnhof" ;
         geo:hasGeometry/geo:asWKT ?geo1 .
  ?stoptime2 a gtfs:StopTime ;
             gtfs:trip ?trip ;
             gtfs:stop ?stop2 ;
             gtfs:stopSequence ?seq2 .
  ?stop2 foaf:name ?stop_name ;
         geo:hasGeometry/geo:asWKT ?geo2 .
  FILTER(?seq1 < ?seq2)
  BIND(geof:distance(?geo1, ?geo2) AS ?dist)
}
GROUP BY ?stop_name
ORDER BY ?dist_freiburg_hbf
```

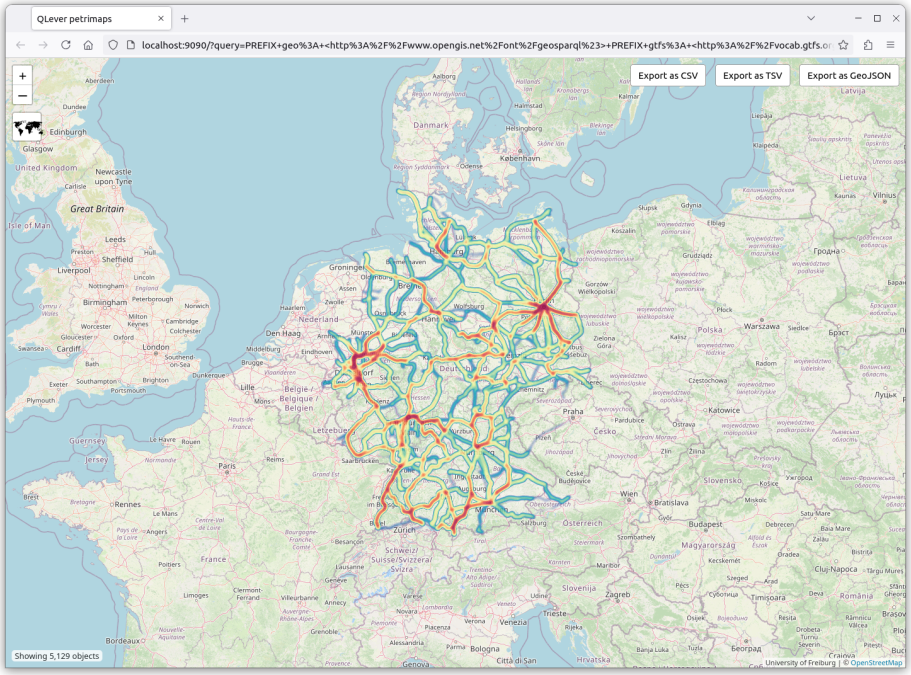
Code 3.17: SPARQL query for places reachable without change from “Freiburg Hauptbahnhof” in Linked GTFS

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX gtfs: <http://vocab.gtfs.org/terms#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT  ?stop ?stop_name ?stop_geo
        (COUNT(*) / 7 AS ?n_trips)
WHERE {
  ?stoptime a gtfs:StopTime ;
            gtfs:trip ?trip ;
            gtfs:stop ?stop .
  ?stop a gtfs:Stop ;
        foaf:name ?stop_name ;
        geo:hasGeometry/geo:asWKT ?stop_geo .
  ?trip a gtfs:Trip ;
        gtfs:service ?service .

  # On which weekdays is this journey in service
  VALUES ?weekday {
    gtfs:monday gtfs:tuesday gtfs:wednesday gtfs:thursday
    gtfs:friday gtfs:saturday gtfs:sunday
  }
  ?calendar_rule  a gtfs:CalendarRule ;
                  gtfs:service ?service ;
                  ?weekday "true"^^xsd:boolean .
}
GROUP BY ?stop ?stop_name ?stop_geo
```

Code 3.18: SPARQL query to find the average number of daily trips per stop, together with each stop's location, from a Linked GTFS [52] dataset



(a) Routes of all busses and trams of VAG Freiburg with destination "Littenweiler". Data: [30, 54]



(b) Routes of all regional express trains in Germany. Data: [19, 54]

Figure 3.5: Map view using QLever Petrimaps with queries on different Linked GTFS data sets from gtfs2rdf augmented by line geometries

3.4.4 Election Data

The case study (4) discussing the impact of infrastructure on political polarization requires election data. The relevant data is published in multiple files in CSV format. Additionally, the geometries of the election districts are published in KML format. They are not part of the OSM data set and cannot be imported because the OSM project explicitly asks users to refrain from entering election districts[6].

Having implemented csv2rdf and kml2rdf, we can now combine these programs into a new election2rdf program. It reads a single configuration JSON file holding election metadata, shell commands to download the external data sets and the configuration for csv2rdf and kml2rdf. The program then outputs a single RDF turtle file containing the combined data from all input data sets plus metadata. In addition, `rdfs:member` triples are added to connect the data sets. The IRIs of entities produced by the conversion of child data sets are modified using an election identifier such that multiple election2rdf outputs can be combined into a single knowledge graph without conflicting entities. This way, data for many elections can form a single knowledge graph.

An example for the metadata triples emitted is shown in Code 3.19. More details on the included data are explained in the case study chapter (4).

```
@prefix election: <https://www.bundeswahlleiterin.de/#> .
@prefix geo: <http://www.opengis.net/ont/geosparql#> .
@prefix manifesto: <https://manifesto-project.wzb.eu/#> .
@prefix osmrel: <https://www.openstreetmap.org/relation/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix wd: <http://www.wikidata.org/entity/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
election:ew24_1 a election:election .
election:ew24_1 rdfs:label "Europawahl 2024" .
election:ew24_1 election:wikidata wd:Q112585123 .
election:ew24_1 election:osm osmrel:51477 .
election:ew24_1 election:countryname "Germany" .
election:ew24_1 election:date "2024/06/09"^^xsd:date .
election:ew24_1 election:year "2024"^^xsd:integer .
```

Code 3.19: Example of the first lines of output from election2rdf

[6] See the discussion on the OSM forum for more details: https://community.openstreetmap.org/t/shapes-mappen-wahlbezirke/81073/11

3.5 Generation of Large Spatial Queries

Using the new spatial search capabilities in QLever, we may want to run various spatial searches between different types of POIs like buildings, supermarkets, stations, etc. and get the results as a single output table. However, writing thousand-line queries can become error-prone and incomprehensible. This problem is addressed by the new free and open source compose_spatial program. It can combine *shards* of SPARQL queries into a combined query according to template queries and a manageably-sized user-provided configuration. A shard is a simple SPARQL graph pattern or subquery, which selects all POIs of a certain category, optionally along with further metadata. Code 3.20 shows an example.

```
1 # All restaurants with names
2 ?restaurant osmkey:amenity "restaurant" ;
3             osmkey:name ?restaurant_name .
```

Code 3.20: Example SPARQL shard for selecting restaurants with their names

The configuration must be given as a file system directory or as a ZIP file, which contains the main configuration given as a JSON file as well as the template and shard files. The program loads or extracts the required files in-memory and parses the configuration JSON using an internal object representation using the following classes:

- `class QueryConfig`,
- `class Template`,
- `class ReplaceRule`,
- `class SpatialSearch`,
- `class RightShard`,
- `class PayloadVariables`,
- `class SpatialSearchConfig`,
- `class GroupTemplate` and
- `class ProvidedValues`.

The generation of a query starting from the user configuration consists of multiple steps. The query as a whole is sourced from a template. The code for the spatial searches is inserted into the template using a replace pattern. Afterwards, the user-supplied regular expression replace rules are applied as long as there are

still occurrences of the search patterns left. These replace rules can be recursively nested up to a recursion limit of currently 100.

The query code for the spatial searches is obtained from each of the objects for the possibly many spatial searches given. The objects hold spatial search configuration, which defines the algorithm and search parameters. Additionally, the spatial search contains a list of left and right query shards. They are combined as a cartesian product: exemplarily assuming we have configured *buildings, stations* as left and *supermarkets, restaurants* as right, the resulting query will perform the spatial search operations *buildings – supermarkets*, *buildings – restaurants*, *stations – supermarkets* and *stations – restaurants*.

The spatial search operations are grouped according to their left shard and optionally the groups can be split to form smaller queries. These groups of spatial search operations are then inserted into a group template, which is inserted as a subquery into the main template.

Right shards for the spatial searches have the option to include payload variables, as introduced in 3.2.3. Furthermore the entire spatial search has the option to declare provided values: a `VALUES ?variable { … }` statement that will be included automatically inside the group graph patterns in each of the spatial search `SERVICE` subqueries to avoid redundancy in the query shards. Of course, the `VALUES` statement is not included, if `?variable` is not used in the respective query shard.

The program automatically adds `geo:hasCentroid`/`geo:asWKT` triples to select the points to be used for search. The `spatialSearch:bindDistance` configuration parameter and `BIND(COUNT(*) AS ?count_variable)` statements for right shards are also added where necessary.

Information on the program's usage and implementation details are provided along with the source code[7].

[7] https://ullinger.info/bachelor-thesis

3.5.1 Interactive Graphical User Interface

While writing a 50-line configuration is much more user-friendly than manually writing a thousand-line SPARQL query, the usability can be improved even further. This is done by implementing a GUI. The user interface is realized as a lightweight 30 kilobyte standalone web app which does not depend on any frameworks and is included with the compose_spatial program. A screenshot is given in Figure 3.6.

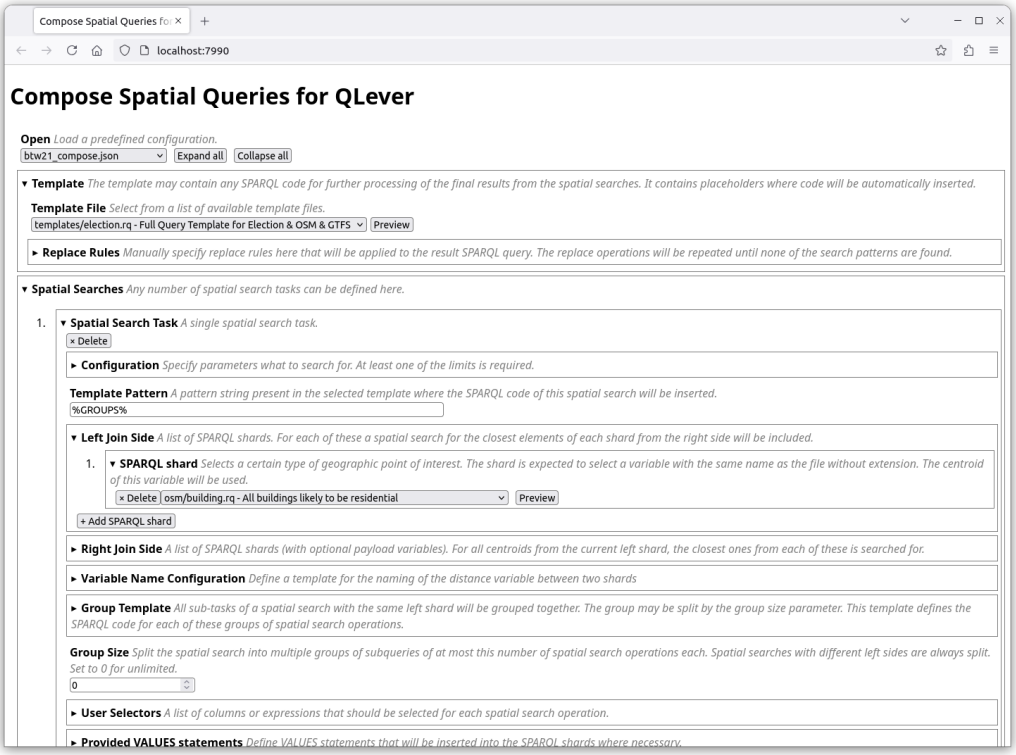


Figure 3.6: Interactive GUI with tree-like structure for the compose_spatial program

The user interface can be used by starting the compose_spatial program with an appropriate option and configuration. The configuration directory includes a JSON file for configuring the web app and the SPARQL templates, shards and JSON configuration files as described previously. After creating the required shards and templates once or reusing the provided ones, users can easily construct many different queries.

The backend of the web app included with the compose_spatial program, consists of a `class ComposeSpatialHTTPServer(http.server.HTTPServer)` along with a customized request handler `class ComposeSpatialHTTPRequestHandl-`

`er(http.server.BaseHTTPRequestHandler)`. Since SPARQL shards and templates are usually only a few kilobytes in size, the implementation caches all files required for composing queries using the given configurations in memory. This is helpful for speed and security, since no escaping the web server directory is possible due to the server only answering queries from its in-memory cache of allowed files.

In addition to the cached configuration and SPARQL files, the server also provides the web interface with required internal information as JSON objects, partially dynamically computed based on input.

The core idea of the web app is that it is fully agnostic to the concrete structure of the compose_spatial configuration. A change to the configuration would not require a change in the web app's code. The tree-like structure of the web app directly represents the structure of the configuration file and no structure is hard-coded. The web app retrieves a `structure.json` file from the server which has the server's expected structure but contains dummy values. By traversing this template and using the types of the values in it, the web app recursively builds the appropriate user interface objects. During this process, the paths to each configuration value from the root of the configuration object are constructed. Using these paths, the web app retrieves additional information not present in the `structure.json` file: default values for configuration parameters from `defaults.json`, lists of options to chose from, for example for the selection of shards, from `selects.json` and explanations of configuration options to display from `descriptions.json`. Descriptions of shards are provided as part of `selects.json` and are obtained dynamically from the individual shard files: if the first line of a shard contains a comment, it is used as a description.

For convenient use, the web app supports previewing SPARQL shards and templates. The user interface adapts to the user's preference for a light or dark color scheme.

When a user requests to convert their configuration to a SPARQL query, the web app traverses the tree representation of its user interface (Document Object Model) and recursively assembles a configuration JSON object directly from it. The JSON object is sent to the server, which returns the SPARQL query. Both files, the configuration JSON and the SPARQL query are displayed to the user with an option to save them locally.

4 Case Study: On Infrastructure and Political Polarization

In this chapter the presented approach is applied to a real-world use case. Based on the current research question from political science, whether the availability of public infrastructure influences political polarization, the implemented workflow is tested. This case study makes use of every single implemented functionality. It requires the conversion of ten input data sets to RDF: six CSV data sets, one KML data set, two GTFS data sets and one OSM data set using auxiliary geometries from KML. For each of two German elections, 13 nearest neighbors searches using a maximum number of results are performed on OSM and one nearest neighbors search using a maximum distance with payload is run on a large GTFS data set. For each election, the entire data set is generated without intermediate steps for the user by querying QLever using a single complex query. The query is constructed using the compose_spatial program and uses both of the implemented new SPARQL functions. Finally, we also present the real-world results for the two elections.

4.1 Background

The indicator polarization "belongs to the standard repertoire for the analysis of party systems" [68] and was introduced in the 1970s [63, 66]. It "is based on the ideological distance between the individual parties of a party system" [68].

The formula [68] used to quantify polarization measures the divide of voters along a one-dimensional right-left scale of ideological position. Polarization is calculated as a variance of the parties' election results weighted by position: let A_i be the share of votes of party i and P_i be the right-left score of the party i, we compute the party system average ("the middle") as

$$\text{PASYSAV} = \sum_{i=1}^{n} (P_i * A_i).$$

Using the party system average, polarization is defined as

$$\text{POLARIZATION} = \sum_{i=1}^{n} (P_i * (A_i - \text{PASYSAV})^2).$$

There are many hypotheses about the reasons for increasing polarization. One of them, based on the urban – rural line of conflict from the theory of "cleavages" [45], is the marginalization or modernization loser hypothesis. In essence it states that where the basic supply of public infrastructure is insufficient, people are radicalized from feeling left behind. There are various studies on related topics [46, 27, 64, 20] but none of them are based on analyzing large geographic data sets to verify the hypothesis. Such an approach is now demonstrated using the software implemented in this thesis. The approach operationalizes the availability of public infrastructure by measuring the distance of residential housing to various places of basic services, like supermarkets and hospitals. Additionally, to measure the access to public transport, the average number of daily journeys reachable within 300 meters from the respective residential building is also considered.

As control variables a few more important indicators are used. A common hypothesis suggests that older people have different political views, therefore the share of citizens over 60 years of age is included. Furthermore, the fraction of foreigners in the district and the average income are used as indicators. Specifically for Germany, there is another important aspect to be considered: The citizens of the states of the former German Democratic Republic have significantly different political views due to historical circumstances and their consequences today. Of course the population density is also taken into consideration in the analysis.

4.2 Data Set Generation using QLever

Since the RDF data model is very flexible it can easily combine multiple datasets. This is taken advantage of here.

We obtain the official election results [14, 12], socioeconomic structural data for each of the election districts [16, 13] as well as the geometries of election districts [15] from the German Federal Returning Officer (Bundeswahlleiterin). These data sets are downloaded and converted to RDF using the election2rdf program based on the csv2rdf and kml2rdf implementations.

The map dataset used to obtain infrastructure information is an OSM database export for Germany [54], which is converted and merged with the election districts using osm2rdf. The spatial searches are performed on the centroids precomputed by osm2rdf.

Furthermore information on public transport is extracted from the data [18, 19] provided by "Durchgängige elektronische Fahrgastinformation e.V." (DELFI). These data sets contain the entire country's timetables in GTFS format, licensed under CC-BY. Germany as an EU member state must provide the timetables freely in accordance with an EU regulation [23]. DELFI forms the *national access point* to comply with this regulation. The GTFS data sets are converted to RDF using the gtfs2rdf program.

The formula to be used to determine polarization requires a political right-left score for each party. A renowned source of data on political positioning of parties is the Manifesto Project Dataset [44]. Unfortunately the Manifesto Project Dataset's non-free license is incompatible with the ODBL used by OSM. Therefore we approximate the right-left score using the parliamentary seating arrangement [21, 24] as shown in Table 4.1.

We perform a plausibility check using the data for the German federal election 2021. The Pearson correlation coefficient between the right-left score from the Manifesto Project and the seating-arrangement-based alternative yields $r = 0.9692$. Thus both are very strongly correlated and the alternative can be used. For the European parliament election 2024 a plausibility check is not possible, because the Manifesto Project has not yet released an updated dataset.

The right-left scores are saved in CSV format and are also included in the input to the election2rdf program. The output turtle files of election2rdf for both elections are indexed and form one QLever instance. A second instance is set up using the combination of both GTFS data sets from DELFI for 2021 and 2024. The third instance is built upon the osm2rdf output. Using the compose_spatial program, one SPARQL query for each election is generated. It can access the data from all three QLever instances using SPARQL's federated querying syntax `SERVICE <IRI> { … }`. Along with the spatial search, also invoked using a `SERVICE`, the query uses the `math:pow(?base, ?exp)` function to compute the polarization formula. The `avg(?numbers)` function and the new `stdev(?numbers)` function are used as aggregates for the distances from the spatial search. The query uses the spatial relations between OSM data and the election districts from KML to

Party	RILE
LINKE	1
SPD	2
90/Greens	3
SSW	3
FDP	4
CDU/CSU	5
AfD	6

(a) German federal election (Bundestagswahl) 2021 [21]. Both christian-democratic parties (CDU and CSU) had individual candidacies but a shared election program, thus they are considered together.

Party	RILE
LINKE	1
BSW	1.5
Die PARTEI	2
SPD	2
90/Greens	3
Volt	3
Tierschutzpartei	3
FDP	4
FREIE WÄHLER	4
PdF	4
CDU/CSU	5
FAMILIE	5
ÖDP	5
AfD	8

(b) European parliament election (Europawahl) 2024 in Germany [24]. The value for parties whose Members of Parliament are non-attached to a parliamentary group were manually estimated.

Table 4.1: Right-left score (RILE) based on parliamentary seating for the successfully elected parties

group the buildings for aggregation.

The complete query for each election is evaluated by QLever and returns the complete data set for analysis as a single CSV or TSV file within approximately ten minutes. One possible scenario for the further use of the query result is demonstrated in 4.3: The query result can be loaded directly into the *R* software for statistical computations [59, 28].

4.3 Real-World Results

In general, it is important to note that polarization is a very complex phenomenon influenced by many factors. Thus polarization, of course, cannot be explained only by the distance to a certain type of basic service. Following from this, correlations are not very strong. Unsurprisingly, the density of public basic services is correlated with the population density (Table 4.2). However, the correlation between

supermarkets or the number of public transport trips and polarization is higher than between population density and polarization. Additionally, the exemplary three indicators (average distance to the nearest hospital and supermarket, average number of reachable daily public transport trips) show significance in some of the linear regression models given by Table 4.3. All of the control variables prove to be relevant indicators, especially the differentiation for the former states of the German Democratic Republic.

While an extensive further analysis and discussion of the data at hand is beyond the scope of this thesis, a few interesting evaluations are given on the next pages (Table 4.2, Table 4.3 and Figure 4.1). It seems clear that polarization is a complex topic, but the data indicate at least some influence of the availability of public infrastructure on polarization for the two elections.

More importantly, the case study demonstrates how all of the implemented software can work together to build a real-world data set using the new efficient spatial search in QLever. The task, which previously required manually dealing with ten different datasets, is significantly simplified. Unlike before, all data set construction can now be done in a single step using only one SPARQL query.

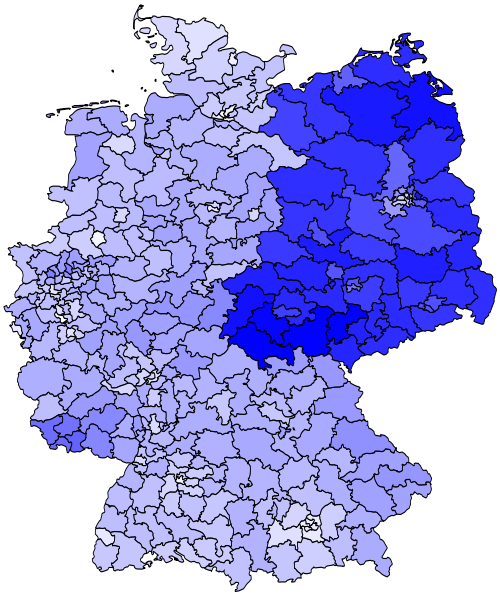
The implemented software is not limited to this use case, but can be applied to countless questions. Another use case could be, for example, the measurement of poorly served neighborhoods for cities' public transport planning.

	Avg. dist. transport	Avg. dist. supermarket	Avg. dist. butcher	Avg. dist. motorway	Avg. dist. hospital	Avg. dist. university	Avg. dist. fuel	Avg. dist. gastronomy	Avg. dist. kindergarten	Avg. dist. hairdresser	Avg. dist. school	Avg. dist. bakery	Avg. dist. pharmacy	Publ. Transp. Trips	Population density	Polarization
Avg. dist. transport	1	0.66	0.6	0.54	0.61	0.65	0.62	0.64	0.65	0.66	0.64	0.64	0.65	-0.48	-0.39	0.4
Avg. dist. supermarket	0.66	1	0.75	0.66	0.87	0.84	0.94	0.92	0.91	0.94	0.96	0.94	0.97	-0.7	-0.63	0.49
Avg. dist. butcher	0.6	0.75	1	0.52	0.72	0.67	0.71	0.85	0.76	0.77	0.78	0.82	0.76	-0.5	-0.45	0.35
Avg. dist. motorway	0.54	0.66	0.52	1	0.65	0.62	0.66	0.66	0.67	0.68	0.64	0.65	0.68	-0.5	-0.43	0.28
Avg. dist. hospital	0.61	0.87	0.72	0.65	1	0.78	0.81	0.83	0.79	0.84	0.83	0.82	0.87	-0.69	-0.66	0.32
Avg. dist. university	0.65	0.84	0.67	0.62	0.78	1	0.81	0.8	0.82	0.85	0.82	0.84	0.84	-0.65	-0.58	0.39
Avg. dist. fuel	0.62	0.94	0.71	0.66	0.81	0.81	1	0.88	0.89	0.91	0.94	0.92	0.95	-0.64	-0.57	0.57
Avg. dist. gastronomy	0.64	0.92	0.85	0.66	0.83	0.8	0.88	1	0.92	0.91	0.95	0.95	0.92	-0.61	-0.54	0.5
Avg. dist. kindergarten	0.65	0.91	0.76	0.67	0.79	0.82	0.89	0.92	1	0.93	0.94	0.93	0.93	-0.64	-0.55	0.52
Avg. dist. hairdresser	0.66	0.94	0.77	0.68	0.84	0.85	0.91	0.91	0.93	1	0.93	0.94	0.95	-0.65	-0.58	0.43
Avg. dist. school	0.64	0.96	0.78	0.64	0.83	0.82	0.94	0.95	0.94	0.93	1	0.96	0.96	-0.63	-0.56	0.54
Avg. dist. bakery	0.64	0.94	0.82	0.65	0.82	0.84	0.92	0.95	0.93	0.94	0.96	1	0.95	-0.61	-0.54	0.51
Avg. dist. pharmacy	0.65	0.97	0.76	0.68	0.87	0.84	0.95	0.92	0.93	0.95	0.96	0.95	1	-0.68	-0.6	0.48
Publ. Transp. Trips	-0.48	-0.7	-0.5	-0.5	-0.69	-0.65	-0.64	-0.61	-0.64	-0.65	-0.63	-0.61	-0.68	1	0.83	-0.39
Population density	-0.39	-0.63	-0.45	-0.43	-0.66	-0.58	-0.57	-0.54	-0.55	-0.58	-0.56	-0.54	-0.6	0.83	1	-0.24
Polarization	0.4	0.49	0.35	0.28	0.32	0.39	0.57	0.5	0.52	0.43	0.54	0.51	0.48	-0.39	-0.24	1

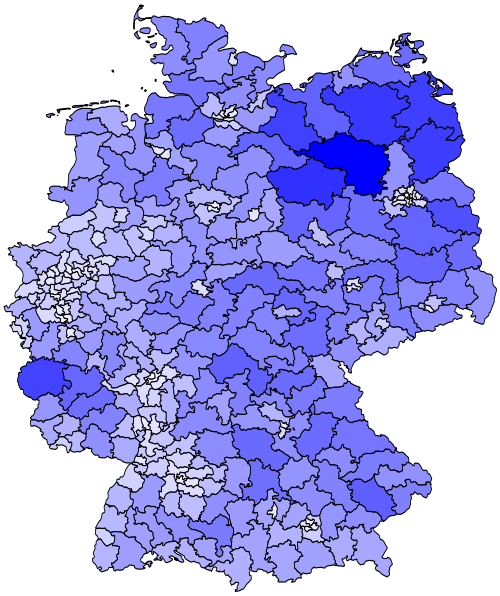
Table 4.2: Correlation matrix between all distance variables, the average number of public transport trips, the population density and the polarization for German Federal Election 2021

Variable	Federal Election 2021			European Election 2024		
	Germany	Only East	Only West	Germany	Only East	Only West
Average distance to nearest hospital in km	-0.02221*	-0.0587*	-0.02204*	0.05752**	0.01813	0.06137**
	(0.01002)	(0.02897)	(0.011)	(0.01762)	(0.04037)	(0.01924)
Average distance to nearest supermarket in km	0.10637**	0.17599*	0.09072*	-0.13864*	0.12558	-0.27713***
	(0.03635)	(0.07551)	(0.04342)	(0.06188)	(0.11278)	(0.07263)
Average reachable daily public transport trips	-0.00038**	-0.00068	-0.00044***	-0.00021	-0.00145·	-0.00035
	(0.00011)	(0.00054)	(0.00013)	(0.00027)	(0.00082)	(0.00029)
Population over 60 years of age in percent	0.02976***	0.03861***	0.02177***	0.13257***	0.11791***	0.13078***
	(0.00442)	(0.00871)	(0.00566)	(0.00864)	(0.01423)	(0.01054)
Location in East German state	0.63613***			2.08804***		
	(0.03618)			(0.07193)		
Foreign population in percent	0.01228***	-0.00548	0.01053**	0.05705***	0.05314*	0.05135***
	(0.00334)	(0.0173)	(0.0036)	(0.00665)	(0.02194)	(0.00731)
Average yearly income in Euro	-0.00003***	-0.00008*	-0.00003***	-0.00011***	-0.00003	-0.00012***
	(0)	(0.00003)	(0)	(0.00001)	(0.00004)	(0.00001)
Constant	1.85612***	3.23759***	2.14877***	1.65035***	2.38433*	2.06633***
	(0.18294)	(0.797)	(0.22723)	(0.40814)	(1.16198)	(0.47683)
Adjusted R^2	0.8368	0.61408	0.35474	0.89622	0.64165	0.51436
F Statistic	219.28598	13.72947	23.81492	493.23283	23.08363	58.1938
Number of districts N	299	49	250	400	75	325

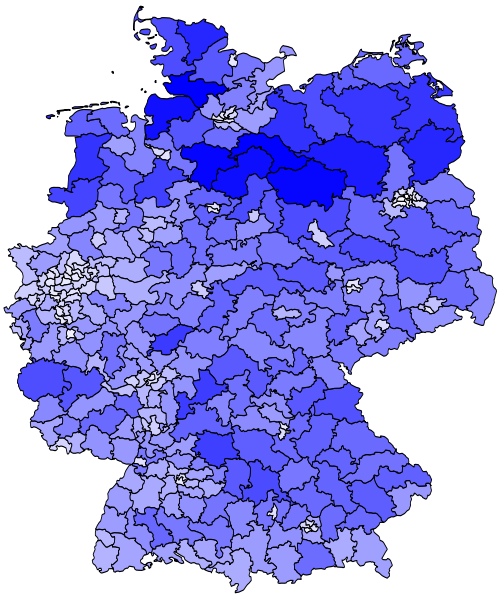
Table 4.3: Multivariate linear regression models to explain political polarization in two German elections using infrastructural and socioeconomic indicators, given as *Coefficient (Standard error)*, probability of confidence: *** >99,9%; ** >99%; * >95%; · >90%



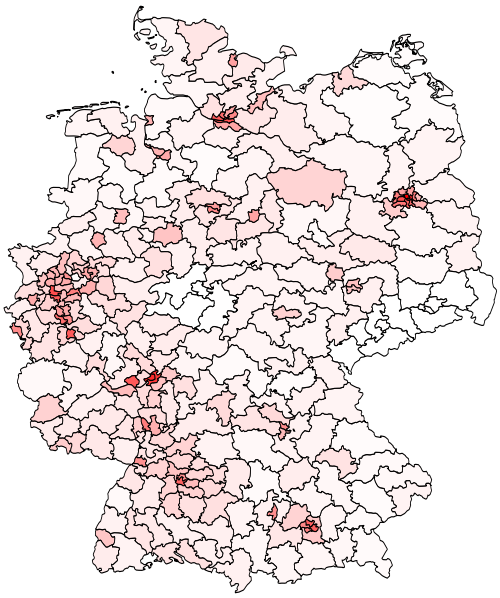
(a) Polarization (min: ☐, max: ■)



(b) Average distance to the nearest supermarket (less is better, min: ☐, max: ■)



(c) Average distance to the nearest hospital (less is better, min: ☐, max: ■)



(d) Average number of daily public transport trips reachable within 300 meters (more is better, min: ☐, max: ■)

Figure 4.1: Map of Germany divided into the election districts for the 2021 Federal Election. Colored to visualize different variables

5 Evaluation

In this chapter we present how the implemented code can be evaluated as well as the results of this evaluation for various data sets. In addition to the benchmarks and comparisons in this chapter, all implemented software's correctness is extensively tested by unit tests with high code coverage ($> 95\%$).

5.1 Experimental Setup

All running times that will be discussed in the following refer to benchmarks run on a powerful, recent consumer computer. Its specifications are given by Table 5.1.

Operating System	Ubuntu Server 24.04.1 LTS
Processor	AMD Ryzen 9 7950X (32 threads, 4.5 GHz)
Memory	128 GB DDR5
Disk Storage	8TB on WD BLACK SN850X NVMe SSDs

Table 5.1: Technical specifications of the machine used for evaluation

The software used for evaluation is run in a containerized manner using the *podman* system. We implement a large purpose-built program *reproduction.py* in the Python programming language to automatically perform all parts of the evaluation from compiling the software, downloading data sets to running the benchmarks. The program is split into 58 individual steps, which each have dependencies on other steps. Since the full evaluation takes approximately 80–85 hours to complete, individual evaluation parts can be reproduced by selecting specific steps. The program checks that the steps' dependencies are met. Details on how to run the reproduction can be found in the appendix (chapter 9).

In the evaluation we compare QLever with PostgreSQL and its geospatial extension PostGIS. For both programs we also compare different types of queries: baseline approaches and index-based approaches. Where appropriate we also compare QLever with a previous version of itself. Since we care about fast

response times not in theory but for the user, all benchmarks are measured as end-to-end times for the respective query engine to produce the result. The only exception to this is the exclusion of data download times since internet connection speed varies and is not under the influence of the query engines' implementations.

PostgreSQL [58] is a widely used relational database management system queried using SQL, which has been in development for over 35 years. It supports a vast variety of spatial functions through the PostGIS extension [61], which is built on the libGEOS library [31]. OSM data can be imported into a PostgreSQL database using the osm2pgsql tool [39]. The workflow using PostgreSQL, PostGIS and osm2pgsql is chosen as a competitor method for comparison against QLever because it is the most widely used solution. The OSM project itself uses this workflow for map rendering and search features in its web application [40].

PostgreSQL takes advantage of a generalized search tree (GiST) [37, 42] which is able to index multi-dimensional data. The index stores the two dimensional coordinates of points in the case of our PostGIS evaluation. This is a different approach to the one-dimensional S2CellIds (see 2.5) used by s2geometry and therefore QLever.

The evaluation queries explained in the following are run on various subsets selected from OSM Germany data [54] and multiple different GTFS data sets [18, 19, 30, 26]. The case study additionally uses official election data [16, 14, 21, 15, 12, 13, 24].

First, we analyze the time taken to import the respective data and build the required indices. For fairness to QLever, which indexes everything, PostgreSQL is instructed to build indices on all columns. Additionally, just like osm2rdf, we precompute and index centroids of all geometries in PostgreSQL. Furthermore we employ the spatialjoin[1] [8] program by the Chair of Algorithms and Data Structures of the University of Freiburg to also precompute all spatial relations for OSM Germany for PostgreSQL. The program is based on the same code base used by osm2rdf.

Second, we compare the running times for QLever with and without the implementation of efficient representation of geographic points (3.1) using a minimalistic query.

Third, we run a nearest neighbors spatial search between differently sized

[1] GitHub repository: https://github.com/ad-freiburg/spatialjoin

types of POIs from OSM. In total 150 benchmarks on 42 different pairs of input tables are evaluated. For the pairs of tables, where the cartesian product does not exceed 250 million rows, we compare

- the cartesian product aggregated to minimum distance in PostgreSQL using a `CROSS JOIN`,
- the index-based nearest neighbors search in PostgreSQL using a precomputed GiST index for all centroids and `CROSS JOIN LATERAL` [57, 60],
- the index-based nearest neighbors search in PostgreSQL using a temporary table with the required subset of points and an ad hoc GiST index, queried with `CROSS JOIN LATERAL`,
- the baseline algorithm in QLever using the nearest neighbors search `SERVICE` syntax (3.2.1),
- the ad hoc s2geometry index-based algorithm in QLever using the nearest neighbors search `SERVICE` syntax (3.2.2).

For larger tables, the cartesian product in PostgreSQL and the baseline algorithm in QLever are not computed.

Fourth, we compare QLever's baseline and index-based nearest neighbors search on different GTFS data sets. A search using a maximum distance is applied to find public transport stops that are within 100 meters of each other.

Fifth, we compare the running times of an ad hoc GiST index in PostgreSQL with the ad hoc s2geometry index algorithm in QLever for a very large spatial search computing the average distance between each building in Germany and the nearest POI for 13 different types of POIs. This corresponds to the OSM part of the case study query.

5.2 Results

In the following we present and discuss the results for the aforementioned evaluation benchmarks. Running times in tables are indicated in the format [hours:][minutes:]seconds. If the running time is below one minute, two decimal places for fractions of a second are also indicated.

Aside from the data set generation and import, benchmarks were usually run 10 times. The times are indicated as the *arithmetic mean* $\pm$ *standard deviation* of the running time after 10 iterations. Results over 10 minutes are marked with

a star (*) and were only run once. If cells are marked with a green background, the average time indicated is the minimum for the respective benchmark. If cells are marked with a red background, it highlights a running time over one minute.

5.2.1 Data Generation and Import

Table 5.2 displays the running times and resulting size of the data and index for the import of OSM Germany. It compares osm2pgsql and PostgreSQL with and without the precomputation of spatial relations using the spatialjoin program with QLever and osm2rdf. It can be seen that PostgreSQL is much faster, but after many queries the import time is amortized. Also the restriction must be made that the osm2pgsql configuration only imports the 64 most important OSM keys, but osm2rdf imports every tag from OSM. The remaining factors were kept as similar as possible. All benchmarks were run on the exact same PBF file.

	PostgreSQL		**QLever**
	osm2pgsql	osm2pgsql + spatialjoin	osm2rdf
Convert PBF	8:24	42:58	2:11:54
Build Index	27:26	1:16:07	9:52:14
Disk Size	70 GiB	375 GiB	344 GiB

Table 5.2: Running times and disk usage for data set preparation and index build on OSM Germany

The remaining evaluation data sets were generated using the new programs introduced in 3.4. Their running times and output file sizes are depicted in Table 5.3. The table also shows the running time and size for a QLever index build on the concatenation of all triples of the election and GTFS data sets respectively. The observations correspond roughly to a linear runtime behavior regarding the input size and are overall acceptable. The successful inclusion of the Finnish railway traffic as a GTFS feed demonstrates the universality of the implemented software as well as the advantages of standardized formats.

Election			GTFS		
election2rdf	Time	Disk	**gtfs2rdf**	Time	Disk
EW 2024	1.37	909 KiB	DELFI 2024	43:18	1.1 GiB
BTW 2021	1.69	4.1 MiB[2]	DELFI 2021	35:15	877 MiB
			Fintraffic 2024	1:54	61 MiB
			VAG FR 2024	33.59	17 MiB
QLever Index	1.75	13 MiB	**QLever Index**	14:25	11 GiB

Table 5.3: Running times and disk usage for data set preparation and index build on election and GTFS data sets. Additional GTFS data sets for Freiburg (VAG FR) [30] and Finland (Fintraffic) [26] are included for later use in the evaluation. BTW refers to Bundestagswahl (German Federal Election), EW refers to Europawahl (European Parliament Election in Germany).

5.2.2 Distance Measurement on Points

We use the minimal query in Code 5.1 to reduce the noise of other parts of the query while measuring the speed improvement of the new efficient representation of geographic points. This way the effect of folding points into ValueIds is isolated as much as possible. In Table 5.4, the improvement shows a more than three times faster query evaluation. The result of the "distance to Berlin" query on our combined GTFS data set is 422.305 kilometers for the new implementation and 423.554 kilometers for the old implementation. The difference is most likely influenced more by the implemented improvement of distance precision (3.3.1) than by the precision loss due to the ValueId representation of points.

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (AVG(?dist) AS ?avg_dist) WHERE {
  ?osm_entity geo:hasCentroid/geo:asWKT ?geometry .
  BIND(geof:distance(
    ?geometry, "POINT(13.369661 52.524945)"^^geo:wktLiteral
  ) AS ?dist)
  FILTER(BOUND(?dist))
}
```

Code 5.1: QLever: Average distance in kilometers of any centroid in the data set to Berlin

[2]Includes auxiliary geometries for osm2rdf.

System	Time
QLever without `GeoPoint`	8.70 ± 0.09
QLever with `GeoPoint`	2.85 ± 0.02
Speed-up	305.37%

Table 5.4: Evaluation results for efficient point representation for "Distance to Berlin" (Code 5.1) on the combined index of all GTFS data sets (Table 5.3)

5.2.3 Nearest Neighbors Spatial Search

In this part of the evaluation we compare the running times of a nearest neighbors search with a maximum of one result. With the goal of testing the nearest neighbors search on various inputs with different characteristics, we choose seven OSM tags with differing frequency in OSM Germany. The tags and the number of entities tagged are shown in Table 5.5.

Key		Value	Entities
`leisure`	=	`sauna`	1 354
`railway`	=	`station`	4 315
`tourism`	=	`viewpoint`	29 556
`shop`	=	`supermarket`	33 815
`amenity`	=	`restaurant`	101 840
`amenity`	=	`bench`	703 479
`building`	=	*all values*	37 536 278

Table 5.5: The OSM tags used to get differently sized join tables. For each tag the number of tagged entities in OSM Germany is given.

We consider a cartesian product of these OSM tags with themselves to be used for the left and right join tables. We remove only the pairs where both tags are equal because the one nearest neighbor in a self-join is always the left point itself. In each benchmark, the two tags are used directly without further restrictions to generate a left or right join table respectively. The queries used are given for the example of railway stations as a left table and supermarkets as a right table. All other queries are carried out analogously, only the tags are changed. The QLever query is given in Code 5.2, the PostgreSQL query with a cartesian product in Code 5.3, with a full precomputed GiST index in Code 5.4 and using an ad hoc GiST index in Code 5.5.

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX osmkey: <https://www.openstreetmap.org/wiki/Key:>
PREFIX spatialSearch:
 ↪ <https://qlever.cs.uni-freiburg.de/spatialSearch/>

SELECT (AVG(?min_dist) AS ?avg_min_dist) WHERE {
  ?left osmkey:railway "station" ;
        geo:hasCentroid/geo:asWKT ?left_geometry .
  SERVICE spatialSearch: {
    _:config spatialSearch:algorithm spatialSearch:s2 ;
             spatialSearch:numNearestNeighbors 1 ;
             spatialSearch:left ?left_geometry ;
             spatialSearch:right ?right_geometry ;
             spatialSearch:bindDistance ?min_dist .
    {
      ?right osmkey:shop "supermarket" ;
             geo:hasCentroid/geo:asWKT ?right_geometry .
    }
  }
}
```

Code 5.2: Example evaluation query on QLever: Nearest neighbor join between all stations and supermarkets. This query can be run with different algorithms: `s2` and `baseline`.

```
SELECT AVG(min_dist) AS avg_min_dist FROM (
  SELECT MIN(
      ST_Distance(ST_Transform(left_.centroid, 4326)::geography,
      ST_Transform(right_.centroid, 4326)::geography)
    ) / 1000 AS min_dist
  FROM (
    SELECT left_.osm_id, left_.centroid
    FROM osm_centroids AS left_
    WHERE left_.railway = 'station'
  ) AS left_ CROSS JOIN (
    SELECT right_.centroid
    FROM osm_centroids AS right_
    WHERE right_.shop = 'supermarket'
  ) AS right_
  GROUP BY left_.osm_id
);
```

Code 5.3: Example evaluation query on PostgreSQL: Cartesian product between all stations and supermarkets aggregated to the minimum per station

```sql
SELECT AVG(left_.centroid::geography
           <-> right_.centroid::geography) / 1000 AS avg_min_dist
FROM (
  SELECT osm_id, centroid
  FROM osm_centroids
  WHERE railway = 'station'
) AS left_ CROSS JOIN LATERAL (
  SELECT right_.centroid,
         right_.centroid <-> left_.centroid AS dist
  FROM osm_centroids AS right_
  WHERE right_.shop = 'supermarket'
  ORDER BY dist
  LIMIT 1
) AS right_;
```

Code 5.4: Example evaluation query on PostgreSQL: Nearest neighbor join between all stations and supermarkets on a full GiST index using the official syntax [60]

```sql
-- Create ad-hoc table and compute index on the selected subset
CREATE TEMPORARY TABLE right_cache (centroid geometry);
INSERT INTO right_cache
  SELECT centroid FROM osm_centroids WHERE shop = 'supermarket';
CREATE INDEX idx_right_cache ON right_cache
  USING GIST (centroid);
-- Query the new index
SELECT AVG(left_.centroid::geography
           <-> right_.centroid::geography) / 1000 AS avg_min_dist
FROM (
  SELECT osm_id, centroid
  FROM osm_centroids
  WHERE railway = 'station'
) AS left_ CROSS JOIN LATERAL (
  SELECT right_.centroid,
         right_.centroid <-> left_.centroid AS dist
  FROM right_cache AS right_
  ORDER BY dist
  LIMIT 1
) AS right_;
```

Code 5.5: Example evaluation query on PostgreSQL: Nearest neighbor join between all stations and supermarkets using an ad hoc GiST index on a temporary table

Where the cartesian product is at most 250 million rows in size it was computed. The results for this benchmark between five strategies are given in Table 5.6. The remaining benchmarks computed only with the three index-based strategies can be found in Table 5.7. Both tables are ordered by ascending size of the left join table, then the right join table.

We can observe a few interesting results here. Between the two cartesian product strategies, the one in PostgreSQL and our baseline algorithm in QLever, our algorithm is much faster in all cases, 991% on average.

The remaining benchmark results require further background information to understand. We can see that in a few cases, mostly when the right join table is *all buildings*, the full GiST index is fastest but otherwise it is very slow. When looking at the query plan for this strategy using SQL's `EXPLAIN` statement (Code 5.6) the reason becomes clear. What PostgreSQL does when it is queried using the full GiST index is search for nearest neighbors of any kind first and filter the results afterwards until enough nearest neighbors have been found. Since buildings are very frequent unlike other tags in all parts of Germany, a search for all nearest neighbors will quickly return a building. Because the index build is not counted towards the query time, this strategy is very fast in these cases. However for other queries it is often unusably slow because the search for nearest neighbors of any kind has a similar behavior to a cartesian product with all points if the points of the right join table are rare in the full index.

At the same time the described scenario of a very frequent tag for the right join table and a relatively small left join table is the ad hoc strategies' weak point. The ad hoc index for all of the many rows in the right table has to be built once no matter how often it is queried. This explains the comparably slow running times of the ad hoc index strategies in these cases.

Additionally, we can observe that QLever, while still only taking 1–2 seconds, for the smaller queries is slower than an ad hoc index in PostgreSQL. However, QLever does not spend the time for the nearest neighbors search but for building the left and right join tables. This can be seen for the station – supermarket example in QLever's query analysis tree depicted in Figure 5.2. In this example building the join tables takes 1727 ms, but the nearest neighbors search takes only 9 ms. QLever's disadvantage in this case is that it has to work with very large indices containing all triples. Due to principle, QLever has to select the required centroids with a join from all centroids. The table in PostgreSQL on the other

hand does not contain all geometries, only centroids. Furthermore PostgreSQL has individual indices for each OSM key due to them being stored in individual columns. These indices are thus much smaller and faster to query. Because of its data organization in a table, PostgreSQL can immediately retrieve the required centroids from the table row after applying the search condition.

Another observation of interest is the disproportionate decline in query speed of the ad hoc GiST index in PostgreSQL for a large right join table with all buildings. When looking at the query plan for the ad hoc GiST index (Code 5.7) this characteristic is not immediately clear. The query plan suggests an analogous query approach with QLever. However, the suboptimal performance can be partially explained by different design decisions of the query engines. PostgreSQL prioritizes wide hardware compatibility and being able to run on low-resource machines. Therefore it will not store the temporary table in memory once it exceeds a threshold but write it to disk. By default this threshold is configured at 8 MiB. The larger right join tables exceed this limit and are written to disk. Querying the table then requires many random access read operations on the hard disk. The read operations on a disk are of course much slower than in random access memory intended for this purpose.

In the scenario where the left join table contains all buildings and is thus very large, the query on QLever's ad hoc s2geometry index is on average 3442% faster than on PostgreSQL's ad hoc GiST index. This applies to cases with small as well as large right join tables used for the ad hoc indices. Therefore this performance difference can be attributed to the time for querying the index 37.5 million times instead of the time for writing a temporary table to disk.

Overall, our index-based algorithm in QLever has the most stable running times. QLever is reasonably fast or fastest in all benchmarks as can be seen in Figure 5.1. QLever performs especially well for queries involving very large join tables. Unlike PostgreSQL, there are no outliers in any of the running time benchmarks for our index-based algorithm. Additionally the SPARQL query (Code 5.2) is much less complex and easier to understand and write than the equivalent SQL query (Code 5.5).

Left	Right	Rows Left×Right	PostgreSQL			QLever	
			Cartesian Product	Complete GiST Index	Ad hoc GiST Index	Baseline Algorithm	Ad hoc S2 Index
sauna	station	5 842 510	7.89 ± 0.08	22.16 ± 1.01	0.18 ± 0.03	1.06 ± 0.01	0.55 ± 0.13
sauna	viewpoint	40 018 824	53.85 ± 0.51	11.93 ± 0.15	0.28 ± 0.02	5.40 ± 0.06	1.62 ± 0.03
sauna	supermarket	45 785 510	1:01 ± 0.22	3.53 ± 0.03	0.34 ± 0.03	5.93 ± 0.05	1.57 ± 0.02
sauna	restaurant	137 891 360	2:34 ± 1.78	1.37 ± 0.02	0.52 ± 0.02	15.38 ± 0.14	2.35 ± 0.04
station	sauna	5 842 510	7.64 ± 0.03	3.02 ± 0.03	0.22 ± 0.02	1.06 ± 0.01	0.52 ± 0.02
station	viewpoint	127 534 140	2:54 ± 1.00	46.01 ± 0.72	0.37 ± 0.02	13.96 ± 0.14	1.85 ± 0.06
station	supermarket	145 911 725	2:36 ± 1.27	9.19 ± 0.05	0.43 ± 0.02	15.59 ± 0.12	1.78 ± 0.03
viewpoint	sauna	40 018 824	52.51 ± 0.44	17.71 ± 0.10	0.82 ± 0.02	5.39 ± 0.06	1.64 ± 0.05
viewpoint	station	127 534 140	2:49 ± 1.18	9:04 ± 16.66	0.72 ± 0.02	13.94 ± 0.14	1.86 ± 0.04
supermarket	sauna	45 785 510	59.37 ± 0.40	20.05 ± 0.50	0.98 ± 0.02	5.93 ± 0.04	1.61 ± 0.05
supermarket	station	145 911 725	2:37 ± 1.19	9:56 ± 31.23	0.85 ± 0.03	15.60 ± 0.11	1.78 ± 0.02
restaurant	sauna	137 891 360	2:28 ± 1.02	57.86 ± 0.57	2.62 ± 0.03	15.40 ± 0.16	2.37 ± 0.05

Table 5.6: Evaluation results for nearest neighbor searches on OSM Germany with sizes where a cartesian product is feasible

Left	Right	PostgreSQL		QLever
		Complete GiST Index	Ad hoc GiST Index	Ad hoc S2 Index
sauna	bench	1.14 ± 0.03	1.88 ± 0.02	2.53 ± 0.03
sauna	building	0.27 ± 0.02	46.35 ± 0.56	14.82 ± 0.11
station	restaurant	4.53 ± 0.02	0.63 ± 0.03	2.54 ± 0.04
station	bench	2.67 ± 0.02	2.03 ± 0.02	2.73 ± 0.04
station	building	0.67 ± 0.02	46.37 ± 0.55	14.96 ± 0.11
viewpoint	supermarket	2:52 ± 1:10	1.07 ± 0.02	2.91 ± 0.06
viewpoint	restaurant	31.96 ± 0.82	1.48 ± 0.02	3.70 ± 0.05
viewpoint	bench	7.30 ± 0.19	3.22 ± 0.04	3.88 ± 0.05
viewpoint	building	4.78 ± 0.12	47.68 ± 0.50	16.10 ± 0.13
supermarket	viewpoint	6:50 ± 2.14	1.26 ± 0.03	2.88 ± 0.03
supermarket	restaurant	29.16 ± 0.13	1.71 ± 0.03	3.57 ± 0.04
supermarket	bench	24.42 ± 0.52	3.64 ± 0.05	3.81 ± 0.05
supermarket	building	3.56 ± 0.05	47.78 ± 0.50	15.95 ± 0.14
restaurant	station	27:38*	2.26 ± 0.02	2.56 ± 0.05
restaurant	viewpoint	16:33*	3.16 ± 0.04	3.68 ± 0.06
restaurant	supermarket	3:47 ± 11.62	3.09 ± 0.03	3.64 ± 0.05
restaurant	bench	53.78 ± 0.94	7.11 ± 0.04	4.72 ± 0.08
restaurant	building	10.96 ± 0.18	50.31 ± 0.57	16.84 ± 0.14
bench	sauna	6:44 ± 8.27	15.09 ± 0.19	2.82 ± 0.08
bench	station	3:00:11*	12.38 ± 0.15	2.96 ± 0.04
bench	viewpoint	1:14:53*	18.17 ± 0.20	4.16 ± 0.08
bench	supermarket	29:25*	17.41 ± 0.19	4.15 ± 0.05
bench	restaurant	10:03*	23.68 ± 0.27	4.95 ± 0.09
bench	building	1:29 ± 0.59	1:21 ± 0.88	18.71 ± 0.15
building	sauna	6:03:44*	12:42*	28.07 ± 0.30
building	station	> 24:00:00	10:36*	24.82 ± 0.11
building	viewpoint	> 24:00:00	16:36*	31.26 ± 0.15
building	supermarket	21:32:31*	14:54*	27.97 ± 0.24
building	restaurant	9:41:47*	20:31*	28.99 ± 0.30
building	bench	5:22:03*	29:01*	36.73 ± 0.28

Table 5.7: Evaluation results for nearest neighbor searches on OSM Germany with sizes where a cartesian product is not feasible

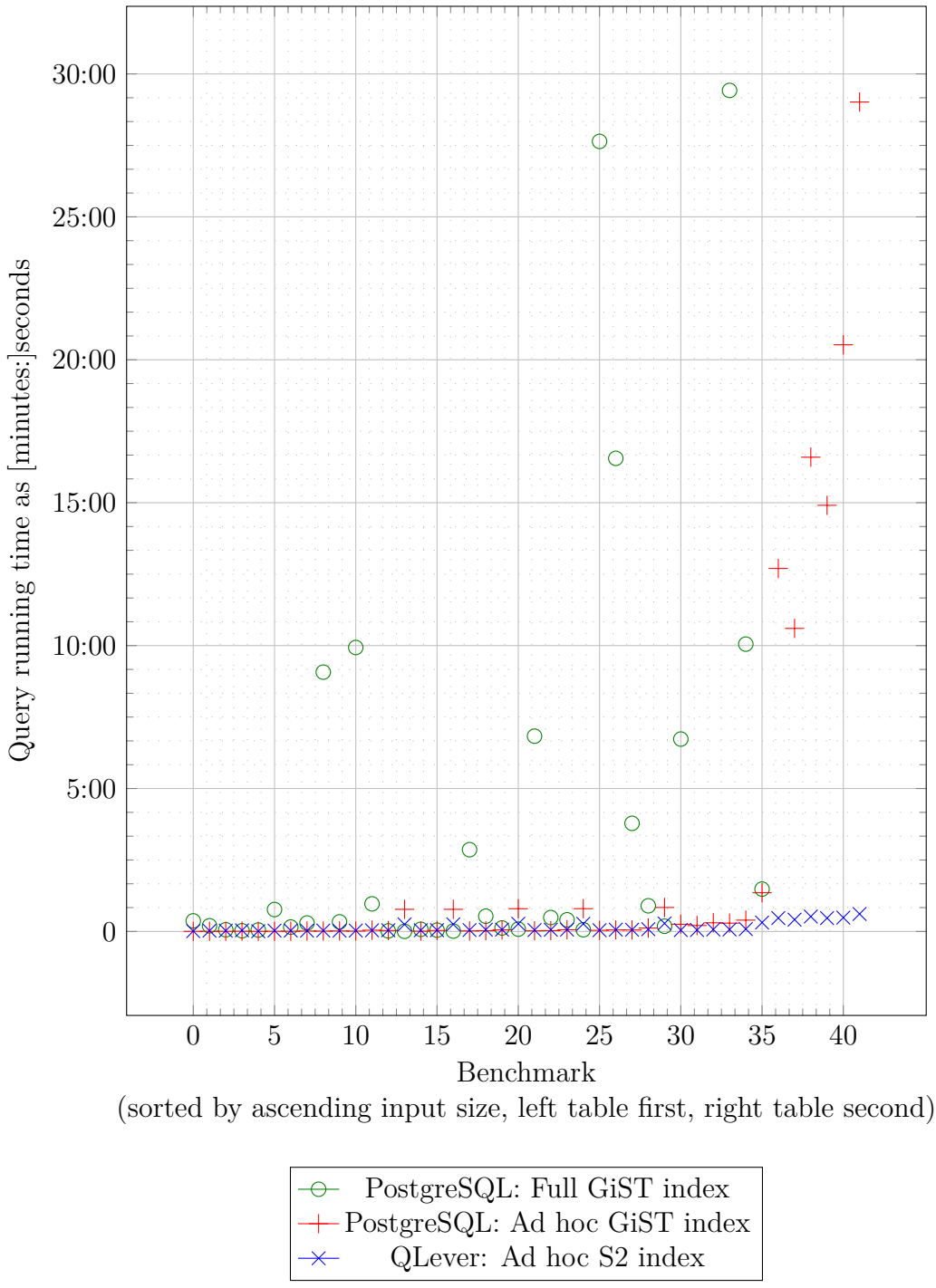


Figure 5.1: Plot of the running times for the nearest neighbors benchmark using the index-based strategies. For PostgreSQL's full GiST index, times over 30 minutes are not depicted.

```
QUERY PLAN
Aggregate  (cost=45523221.09..45523221.11 rows=1 width=8)
  ->  Nested Loop  (cost=99.93..45520996.81 rows=8638 width=64)
        ->  Bitmap Heap Scan on osm_centroids
        ↪  (cost=99.51..32079.29 rows=8638 width=32)
              Recheck Cond: (railway = 'station'::text)
              ->  Bitmap Index Scan on idx_railway
              ↪  (cost=0.00..97.35 rows=8638 width=0)
                    Index Cond: (railway = 'station'::text)
        ->  Limit  (cost=0.42..5266.12 rows=1 width=40)
              ->  Index Scan using idx_centroids on osm_centroids
              ↪  right_  (cost=0.42..202618775.40 rows=38479
              ↪  width=40)
                    Order By: (centroid <->
                    ↪  osm_centroids.centroid)
                    Filter: (shop = 'supermarket'::text)
JIT:
  Functions: 13
"  Options: Inlining true, Optimization true, Expressions true,
↪  Deforming true"
```

Code 5.6: PostgreSQL `EXPLAIN` output (query plan) for the nearest neighbor join between all stations and supermarkets on a full GiST index from Code 5.4

```
QUERY PLAN
Aggregate  (cost=42854.23..42854.24 rows=1 width=8)
  ->  Nested Loop  (cost=99.79..40629.94 rows=8638 width=64)
        ->  Bitmap Heap Scan on osm_centroids
        ↪  (cost=99.51..32079.29 rows=8638 width=32)
              Recheck Cond: (railway = 'station'::text)
              ->  Bitmap Index Scan on idx_railway
              ↪  (cost=0.00..97.35 rows=8638 width=0)
                    Index Cond: (railway = 'station'::text)
        ->  Limit  (cost=0.28..0.97 rows=1 width=40)
              ->  Index Scan using idx_right_cache on right_cache
              ↪  right_  (cost=0.28..23483.31 rows=33916
              ↪  width=40)
                    Order By: (centroid <->
                    ↪  osm_centroids.centroid)
```

Code 5.7: PostgreSQL `EXPLAIN` output (query plan) for the nearest neighbor join between all stations and supermarkets on an ad hoc GiST index from Code 5.5

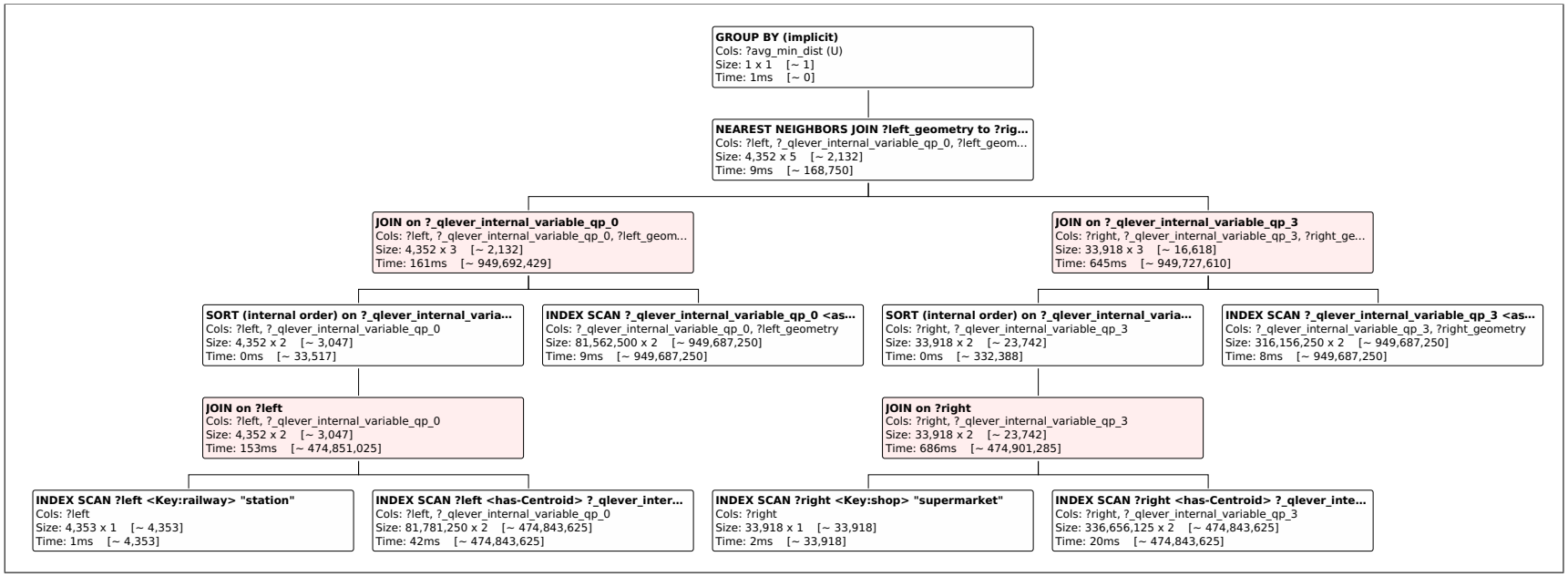


Figure 5.2: QLever: Query plan and running time analysis from QLever's GUI for a nearest neighbor join between all stations and supermarkets on an ad hoc s2geometry index

5.2.4 Maximum Distance Spatial Search

In this benchmark we perform a nearest neighbors search restricted only by a maximum distance. For four different GTFS feeds, we search for the average number of stops that are within 100 meters of any given stop. Such a maximum distance query could for example be relevant to routing applications which want to use departures from all nearby stops for transfers. This is practically relevant because each platform of a stop is usually stored as a separate stop entity in GTFS.

The GTFS feeds used for the comparison have different characteristics. We use the network of VAG Freiburg [30], a small GTFS feed with points that are geographically very close together. The maximum pairwise distance between points in the VAG Freiburg feed is 26 kilometers. As second feed is obtained from Finland's traffic authority [26]. It contains all railway traffic and stations. Due to the very low population density in Finland, the feed is small but the geographic stretch is very large. The maximum pairwise distance between points is 838 kilometers. Additionally, we use two very large GTFS feeds that contain all public transport in Germany from DELFI [18, 19].

The number of stops as well as the running times for the described query are shown in Table 5.8. The baseline algorithm is not feasible for the large GTFS feeds as it would require the comparison of 228 billion and 287 billion combinations of rows respectively.

GTFS Feed	Data Source	Number of Stops	Baseline Algorithm	Ad hoc S2 Index
VAG FR 2024	[30]	1 002	17.89 ± 0.12	0.06 ± 0.01
Fintraffic 2024	[26]	1 429	1.70 ± 0.03	0.14 ± 0.02
DELFI 2021	[18]	477 602	—	0.65 ± 0.01
DELFI 2024	[19]	535 873	—	0.73 ± 0.03

Table 5.8: Evaluation results for a nearest neighbors search between stops from GTFS data sets with a maximum distance of 100 meters

We can observe that the s2geometry index-based algorithm is very fast below one second in all cases, regardless of whether the stops are geographically close or distributed. The bad performance of the baseline algorithm for VAG Freiburg is due to suboptimal query planner decisions during the construction of the join tables.

5.2.5 Large Combined Spatial Search

In this final benchmark, we evaluate the queries that motivated this thesis. Table 5.9 shows the running times of the OSM subquery for the case study (4) without election districts. It asks for the average and standard deviation of the distance of each likely residential building in OSM Germany to its closest public transport stop, supermarket, bakery, butcher, gas station, gastronomic offering, hairdresser, hospital, kindergarten, motorway ramp, pharmacy, school and university. Buildings that are likely to be residential are selected using common values of the *building* key, excluding those that lay within *landuse* areas like landfills that prevent residential housing. This requires accessing the appropriate rows within the 4.4 billion spatial relations on OSM Germany (QLever: 4 seconds, PostgreSQL: 2 minutes 10 seconds). The SPARQL query is produced using the compose_spatial program (3.5). It makes use of the nearest neighbors spatial search using the `SERVICE` syntax (2.5) along with the `stdev(?numbers)` aggregation function (3.3.3). For PostgreSQL, its built-in `STDDEV_SAMP(numbers)` function is used.

System	**Strategy**	**Time**
PostgreSQL	Ad hoc GiST Index	2:48:36*
QLever	Ad hoc S2 Index	6:33 ± 12.83
Speed-up		2573%

Table 5.9: Running times for the OSM part of the case study: Distance between residential buildings and 13 types of POIs on OSM Germany. Comparison of QLever with PostgreSQL.

For this very large and complex query, QLever outperforms PostgreSQL by approximately 26 times. As expected, except for minor differences due to floating point operations and distance calculation the query results are the same for both systems. In addition to this purely spatial query, the running times for the full approximately 1 200 line case study queries are given in Table 5.10.

Election	**Year**	**Time**
German Federal Election	2021	10:58*
European Election in Germany	2024	10:32*

Table 5.10: Running times for the complete, combined case study SPARQL query described in 4.2 using QLever

6 Conclusion

In this thesis we presented a complete workflow for performing efficient spatial searches using the QLever SPARQL engine. Solutions for efficiently representing geographic points and a fast nearest neighbors search using a clean SPARQL syntax integration were implemented. Additional programs for importing data and constructing spatial queries more easily were proposed. In a case study on the influence of public infrastructure on political polarization the implemented software passed a practical test. The evaluation showed reliable and fast query times of the implemented spatial search in QLever, especially for very large data sets.

6.1 Future Work

The approaches presented in this thesis could be developed even further in multiple directions:

- To improve the memory requirements of the presented nearest neighbors search, QLever's spatial join operation should be extended to support lazy result evaluation such that the left join table can be processed chunk-wise without being fully materialized in memory.
- While operating only on centroids is very helpful in terms of performance, the logical next step is allowing support for the WKT geometries line strings, polygons and collections. In order to implement this, more advanced WKT parsing and helpers to convert the WKT geometries to s2geometry objects would be required. Furthermore the S2PointIndex cannot be used for non-point geometries. The use of a more general index data structure is therefore also required if non-point geometries are to be supported.
- It could be explored how non-point geometries could be represented more efficiently, for example by embedding parts of the S2CellId containing the geometry into the ValueId.
- Another very relevant task is the improvement of support for GeoSPARQL in QLever. This includes functions for converting literals, dynamically

calculating the length of lines and the area of polygons, computing centroids and spatial relations ad hoc, performing union or intersection on geometries as aggregations and more.

- The query planner in QLever could be extended to detect optimizable queries containing a filtered cartesian product and rewrite the query plan automatically to make use of the spatial search feature.
- Further statistical aggregation functions could be implemented in QLever, for example median, arbitrary quantiles or correlation.
- Using the intermediate geometry representation from kml2rdf, converters for other geometry file types could be implemented.
- The gtfs2rdf program could be extended to support more non-mandatory tables like translations, levels or pathways as an extension to the Linked GTFS standard.

7 Acknowledgments

First and foremost, I would like to thank Johannes Kalmbach for his time and expertise. He provided valuable guidance and answered countless questions. I also want to sincerely thank Hannah Bast for her very good ideas and constructive feedback on various topics as well as for agreeing to supervise my thesis. Additionally, I would like to express gratitude to Axel Lehmann for his implementation of centroids in osm2rdf and to Patrick Brosi for his work to support auxiliary geometries in osm2rdf. Both of these features were of great use to this thesis. Last but not least, I want to thank my family and friends for their support.

8 Bibliography

[1] Greg Albiston, Haozhe Chen, and Taha Osman. *Apache Jena: GeoSPARQL*. https://web.archive.org/web/20241230124009/https://jena.apache.org/documentation/geosparql/index.html. 2024.

[2] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. "Foundations of modern query languages for graph databases". In: *ACM Computing Surveys (CSUR)* 50.5 (2017), pp. 1–40.

[3] Apache Software Foundation. *Apache License 2.0*. https://web.archive.org/web/20250108024732/https://www.apache.org/licenses/LICENSE-2.0. 2004.

[4] Oleg Bartunov, Teodor Sigaev, and Andrew Gierth. *PostgreSQL 16 Documentation: Additional Supplied Modules and Extensions: `hstore` key/-value datatype*. https://web.archive.org/web/20241217211743/https://www.postgresql.org/docs/16/hstore.html. 2023.

[5] Hannah Bast and Patrick Brosi. "Sparse map-matching in public transit networks with turn restrictions". In: *Proceedings of the 26th ACM International Conference on Advances in Geographic Information Systems*. 2018, pp. 480–483.

[6] Hannah Bast, Patrick Brosi, Johannes Kalmbach, and Axel Lehmann. "An efficient RDF converter and SPARQL endpoint for the complete OpenStreetMap data". In: *Proceedings of the 29th ACM International Conference on Advances in Geographic Information Systems*. 2021, pp. 536–539.

[7] Hannah Bast, Patrick Brosi, Johannes Kalmbach, and Axel Lehmann. "Efficient Interactive Visualization of Large Geospatial Query Results". In: *Proceedings of the 31st ACM International Conference on Advances in Geographic Information Systems*. 2023, pp. 1–4.

[8] Hannah Bast, Patrick Brosi, Johannes Kalmbach, and Axel Lehmann. "Efficient Spatial Joins for Large Sets of Geometric Objects". In: *Proceedings of the 32nd ACM International Conference on Advances in Geographic Information Systems*. 2024. Submitted.

[9] Hannah Bast and Björn Buchhold. "QLever: A query engine for efficient SPARQL + text search". In: *Proceedings of the 2017 ACM Conference on Information and Knowledge Management*. 2017, pp. 647–656.

[10] Rudolf Bayer and Edward McCreight. "Organization and maintenance of large ordered indices". In: *Proceedings of the 1970 ACM SIGFIDET Workshop on Data Description, Access and Control*. 1970, pp. 107–141.

[11] Tim Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259, https://web.archive.org/web/20241204204821/https://www.rfc-editor.org/rfc/rfc8259.txt. 2017.

[12] Bundeswahlleiterin. *Europawahl 2024. Ergebnisse nach Kreisen und kreisfreien Städten*. Statistisches Bundesamt, Wiesbaden. Datenlizenz Deutschland - Namensnennung - Version 2.0. https://web.archive.org/web/20241207021943/https://bundeswahlleiterin.de/dam/jcr/32edebff-544d-489d-ae16-0d40faac180e/ew24_kerg2.csv. 2024.

[13] Bundeswahlleiterin. *Strukturdaten für die kreisfreien Städte und Landkreise zur Europawahl am 09.06.2024*. Statistisches Bundesamt, Wiesbaden. Datenlizenz Deutschland - Namensnennung - Version 2.0. https://web.archive.org/web/20241226172941/https://bundeswahlleiterin.de/dam/jcr/c8d845ab-1d69-4c16-a47a-9ed600641217/ew24_strukturdaten.csv. 2024.

[14] Bundeswahlleiterin. *Wahlergebnisse Bundestagswahl 2021 nach Wahlkreisen*. Statistisches Bundesamt, Wiesbaden. Datenlizenz Deutschland - Namensnennung - Version 2.0. https://web.archive.org/web/20241119164247/https://www.bundeswahlleiterin.de/dam/jcr/860495c9-83fb-4068-8a99-c1c985ffffd2/w-btw21_kerg2.csv. 2024.

[15] Bundeswahlleiterin. *Wahlkreiskarte für die Wahl zum 20. Deutschen Bundestag: Geographie der Wahlkreise*. Statistisches Bundesamt, Wiesbaden. Grundlage der Geoinformationen Geobasis-DE, BKG. https://web.archive.org/web/20250108092008/https://www.bundeswahlleiterin.de/dam/jcr/87bf809f-d778-493d-8de8-77ebee2af9c6/btw21_geometrie_wahlkreise_kml.zip. 2020.

[16] Bundeswahlleiterin. *Wahlkreiskarte für die Wahl zum 20. Deutschen Bundestag: Strukturdaten der Wahlkreise*. Statistisches Bundesamt, Wiesbaden. Grundlage der Geoinformationen Geobasis-DE, BKG. https://web.arch

ive.org/web/20241213004101/https://www.bundeswahlleiterin.de/dam/jcr/b1d3fc4f-17eb-455f-a01c-a0bf32135c5d/btw21_strukturdaten.csv. 2020.

[17] Creative Commons. *Attribution 4.0 International*. https://web.archive.org/web/20250108010019/https://creativecommons.org/licenses/by/4.0/. 2013.

[18] DELFI e.V. *Deutschlandweite Sollfahrplandaten (GTFS)*. Version 2021-09-24. Creative Commons Attribution 4.0 (CC-BY). https://web.archive.org/web/20250108091434/https://archiv.opendata-oepnv.de/DELFI/Soll-Fahrplandaten%20(GTFS)/2021/20210924_fahrplaene_gesamtdeutschland_gtfs.zip. 2021.

[19] DELFI e.V. *Deutschlandweite Sollfahrplandaten (GTFS)*. Version 2024-06-03. Creative Commons Attribution 4.0 (CC-BY). https://www.opendata-oepnv.de/fileadmin/datasets/delfi/20240603_fahrplaene_gesamtdeutschland_gtfs.zip (User registration required for download). 2024.

[20] Larissa Deppisch. "Die AfD und das 'Dornröschenschloss' — über die (Be-) Deutung von Peripherisierung für den Rechtspopulismuszuspruch". In: *Sozial- und Kulturgeographie* 48 (2022), pp. 103–121.

[21] Deutscher Bundestag. *Sitzverteilung des 20. Deutschen Bundestages*. https://web.archive.org/web/20241230215850/https://www.bundestag.de/parlament/plenum/sitzverteilung_20wp. 2024.

[22] Martin J. Dürst and Michel Suignard. *Internationalized Resource Identifiers (IRIs)*. RFC 3987, https://web.archive.org/web/20241207092323/https://www.rfc-editor.org/rfc/rfc3987.txt. 2005.

[23] European Commission. *Commission Delegated Regulation (EU) 2017/1926 of 31 May 2017 supplementing Directive 2010/40/EU of the European Parliament and of the Council with regard to the provision of EU-wide multimodal travel information services*. Official Journal of the European Union. L 272/1. https://web.archive.org/web/20241214203803/https://eur-lex.europa.eu/eli/reg_del/2017/1926/oj/. 2017.

[24] European Parliament. *Chamber seating plans*. https://web.archive.org/web/20250101034135/https://www.europarl.europa.eu/sedcms/pubfile/HEMICYCLE/PLAN_STR.pdf and https://web.archive.org/web/

20250101034134/https://www.europarl.europa.eu/sedcms/pubfile/HEMICYCLE/PLAN_BRU.pdf. 2024.

[25] Federal Communications Commission. *Reference points and distance computations*. Code of Federal Regulations (Annual Edition). Title 47: Telecommunication. 73 (208). https://web.archive.org/web/20241222101048/https://www.govinfo.gov/content/pkg/CFR-2016-title47-vol4/pdf/CFR-2016-title47-vol4-sec73-208.pdf. 2016.

[26] Fintraffic. *Digitraffic: Timetables, delays, locations and composition of trains operating in Finland — GTFS for railway traffic*. Version 2024-11-27. Creative Commons Attribution 4.0 (CC-BY). https://web.archive.org/web/20241127154525/https://rata.digitraffic.fi/api/v1/trains/gtfs-all.zip. 2024.

[27] Maximilian Förtner, Bernd Belina, and Matthias Naumann. "The revenge of the village? The geography of right-wing populist electoral success, anti-politics, and austerity in Germany". In: *Environment and Planning C: Politics and Space* 39.3 (2021), pp. 574–596.

[28] John Fox and Sanford Weisberg. *An R Companion to Applied Regression*. Version 3.1. Website: https://cran.r-project.org/web/packages/car/. 2019.

[29] Free Software Foundation. *GNU General Public License Version 3*. https://web.archive.org/web/20241031200145/http://www.gnu.org/licenses/gpl-3.0.html. 2007.

[30] Freiburger Verkehrs AG. *Fahrplandaten im GTFS-Format, Datensatz der VAG Freiburg*. Datenlizenz Deutschland - Namensnennung - Version 2.0. Website: https://www.vag-freiburg.de/service-infos/downloads/gtfs-daten. Data: https://web.archive.org/web/20241108074231/https://www.vag-freiburg.de/fileadmin/gtfs/VAGFR.zip. 2024.

[31] GEOS contributors. *GEOS computational geometry library*. Website: https://libgeos.org/. 2024.

[32] Google. *General Transit Feed Specification (GTFS)*. https://web.archive.org/web/20241223095056/https://gtfs.org/documentation/schedule/reference/. 2024.

[33] Google. *S2 Cell Hierarchy*. https://web.archive.org/web/20241202095947/https://s2geometry.io/devguide/s2cell_hierarchy. 2024.

[34] Google. *S2 Cell Statistics*. https://web.archive.org/web/20241210203618/https://s2geometry.io/resources/s2cell_statistics. 2024.

[35] Google. *s2geometry library*. Website: https://s2geometry.io. 2024.

[36] Julius Heinzinger. *pdf2gtfs: Timetable Extraction from PDF Files*. Bachelor of Science Thesis. Chair for Algorithms and Data Structures, University of Freiburg. 2023.

[37] Joseph Hellerstein, Jeffrey Naughton, and Avi Pfeffer. "Generalized Search Trees for Database Systems". In: *Proceedings of the 21st International Conference on Very Large Data Bases*. 1995, pp. 562–573.

[38] Sarah Hoffmann. *osm2pgsql version 2 Manual*. https://web.archive.org/web/20241202004009/https://osm2pgsql.org/doc/manual.html. 2024.

[39] Sarah Hoffmann. *osm2pgsql: an open source tool for importing OpenStreetMap data into a PostgreSQL/PostGIS database*. Version 1.8.0. Website: https://osm2pgsql.org. 2023.

[40] Sarah Hoffmann. *Who uses osm2pgsql?* https://web.archive.org/web/20241119155258/https://osm2pgsql.org/about/users/. 2024.

[41] International Organization for Standardization. *Information technology — Database languages SQL. ISO/IEC 9075*. https://web.archive.org/web/20241224102539/https://www.iso.org/standard/76583.html. 2023.

[42] Marcel Kornacker. *Access methods for next-generation database systems*. PhD Thesis. University of California, Berkeley. 2000.

[43] Kostis Kyzirakos, Dimitrianos Savva, Ioannis Vlachopoulos, Alexandros Vasileiou, Nikolaos Karalis, Manolis Koubarakis, and Stefan Manegold. "GeoTriples: Transforming geospatial data into RDF graphs using R2RML and RML mappings". In: *Journal of Web Semantics* 52.53 (2018), pp. 16–32.

[44] Pola Lehmann, Simon Franzmann, Tobias Burst, Sven Regel, Felicia Riethmüller, Andrea Volkens, Bernhard Weßels, and Lisa Zehnter. *The Manifesto Data Collection. Manifesto Project (MRG, CMP, MARPOR)*. Version 2024a. Berlin: Wissenschaftszentrum Berlin für Sozialforschung (WZB), Göttingen: Institut für Demokratieforschung (IfDem). https://web.archive.org/web/20250108091730/https://manifesto-project.wzb.eu/down/data/2024a/datasets/MPDataset_MPDS2024a.csv. 2024.

[45] Seymour Lipset and Stein Rokkan. “Cleavage structures, party systems, and voter alignments: an introduction”. In: *Party systems and voter alignments: Cross-national perspectives*. Ed. by Seymour Lipset and Stein Rokkan. New York Free Press, 1967, pp. 1–64.

[46] Clayton Nall. “The political consequences of spatial policies: How interstate highways facilitated geographic polarization”. In: *The Journal of Politics* 77.2 (2015), pp. 394–406.

[47] Thomas Neumann and Gerhard Weikum. “RDF-3X: a RISC-style engine for RDF”. In: *Proceedings of the VLDB Endowment*. 2008, pp. 647–659.

[48] Open Data Commons. *Open Database License 1.0*. https://web.archive.org/web/20250105094031/https://opendatacommons.org/licenses/odbl/1-0/. 2009.

[49] Open Geospatial Consortium. *GeoSPARQL — A Geographic Query Language for RDF Data*. https://www.ogc.org/publications/standard/geosparql/. https://web.archive.org/web/20250102013515/https://docs.ogc.org/is/22-047r1/22-047r1.html. 2024.

[50] Open Geospatial Consortium. *KML — Keyhole Markup Language*. https://www.ogc.org/publications/standard/kml/. https://web.archive.org/web/20250102100426/https://portal.ogc.org/files/?artifact_id=23689. 2024.

[51] Open Geospatial Consortium. *OpenGIS Implementation Specification for Geographic information — Simple Feature Access — Part 1: Common Architecture*. https://www.ogc.org/publications/standard/sfa/. https://web.archive.org/web/20241216014324/https://portal.ogc.org/files/?artifact_id=25355. 2011.

[52] Open Transport Working Group. *Linked GTFS Specification*. https://web.archive.org/web/20250108090634/https://github.com/OpenTransport/linked-gtfs/blob/master/spec.md. 2015.

[53] OpenStreetMap Contributors. *Elements*. https://web.archive.org/web/20241103202210/https://wiki.openstreetmap.org/wiki/Elements. 2024.

[54] OpenStreetMap Contributors. *OpenStreetMap PBF export for Germany*. Version 2024-11-27 21:21:30. Open Database License 1.0. https://download.geofabrik.de/europe/germany-latest.osm.pbf. 2024.

[55] OpenStreetMap Contributors. *PBF Format*. https://web.archive.org/web/20241209122441/https://wiki.openstreetmap.org/wiki/PBF_Format. 2024.

[56] Terence Parr. *ANTLR. Another Tool for Language Recognition*. Version 4.11.1. Website: https://www.antlr.org/. 2024.

[57] PostgreSQL Global Development Group. *PostgreSQL 16 Documentation: Queries: Table Expressions: The `FROM` clause: `LATERAL` Subqueries*. https://web.archive.org/web/20241123035453/https://www.postgresql.org/docs/16/queries-table-expressions.html#QUERIES-LATERAL. 2023.

[58] PostgreSQL Global Development Group. *PostgreSQL Relational Database*. Version 16.2. Website: https://www.postgresql.org. 2023.

[59] R Core Team. *R: A Language and Environment for Statistical Computing*. Version 4.2.2. Website https://www.R-project.org/. The R Foundation for Statistical Computing. 2022.

[60] Refractions Research. *PostGIS: Nearest Neighbor Join*. https://web.archive.org/web/20241113043738/https://www.postgis.net/workshops/postgis-intro/knn.html#nearest-neighbor-join. 2023.

[61] Refractions Research. *PostGIS: PostgreSQL Geographic Information System: spatial and geographic objects for PostgreSQL*. Version 3.3.2. Website: https://postgis.net. 2023.

[62] Yakov Shafranovich. *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. RFC 4180, https://web.archive.org/web/20241126230413/https://www.rfc-editor.org/rfc/rfc4180.txt. 2005.

[63] Lee Sigelman and Syng Nam Yough. "Left-Right Polarization In National Party Systems: A Cross-National Analysis". In: *Comparative Political Studies* 11.3 (1978), pp. 355–379.

[64] Annette Spellerberg, Denis Huschka, and Roland Habich. "Angleichung und Polarisierung: Entwicklung der Lebensqualität in ländlichen Kreisen". In: *Soziale Ungleichheit, kulturelle Unterschiede*. Ed. by Karl-Siegbert Rehberg. Campus Verlag Frankfurt a.M., 2006, pp. 839–861.

[65] SYSTAP LLC. *Geospatial Support in Blazegraph*. https://web.archive.org/web/20241230123951/https://github.com/blazegraph/database/wiki/GeoSpatial. 2020.

[66] Michael Taylor and Valentine Herman. "Party systems and government stability". In: *American Political Science Review* 65.1 (1971), pp. 28–37.

[67] Konstantinos Theocharidis, John Liagouris, Nikos Mamoulis, Panagiotis Bouros, and Manolis Terrovitis. "SRX: efficient management of spatial RDF data". In: *The VLDB Journal* 28 (2019), pp. 703–733.

[68] Uwe Wagschal. "Polarisierung der Parteiensysteme in Zeiten des Populismus". In: *The European Social Model under Pressure: Liber Amicorum in Honour of Klaus Armingeon*. Ed. by Romana Careja, Patrick Emmenegger, and Nathalie Giger. Springer, 2020, pp. 365–382.

[69] David R. Williams. *NASA Goddard Space Flight Center's Earth Fact Sheet*. https://web.archive.org/web/20250102224318/https://nssdc.gsfc.nasa.gov/planetary/factsheet/earthfact.html. 2024.

[70] World Wide Web Consortium. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. https://web.archive.org/web/20241224055222/https://www.w3.org/TR/2008/REC-xml-20081126/. 2008.

[71] World Wide Web Consortium. *RDF 1.1 Concepts and Abstract Syntax*. https://web.archive.org/web/20250105025544/https://www.w3.org/TR/rdf-concepts/. 2014.

[72] World Wide Web Consortium. *Resource Description Framework (RDF) Model and Syntax Specification*. https://web.archive.org/web/20241231073425/https://www.w3.org/TR/1999/REC-rdf-syntax-19990222/. 1999.

[73] World Wide Web Consortium. *SPARQL 1.1 Query Language*. https://web.archive.org/web/20250102012146/https://www.w3.org/TR/sparql11-query/. 2013.

[74] World Wide Web Consortium. *XPath and XQuery Functions and Operators*. https://web.archive.org/web/20241204060738/https://www.w3.org/TR/xpath-functions-31/. 2017.

9 Appendix

9.1 Software and Documentation

The new software for the conversion of external data sets to RDF (3.4) and the generation of large spatial queries (3.5) can be run as shown in Code 9.1.

```
curl -O -J -L https://ullinger.info/bachelor-thesis/main.tar.gz
tar xzf spatial-search-main.tar.gz && cd spatial-search-main
podman build -t spatial .
podman run --rm -it -v ./output:/output:rw -p 7990:7990 spatial
# In the container: use 'make help' for more information
```

Code 9.1: Commands to download and run the presented software

9.2 Reproduction of the Results

The reproduction of all evaluation and case study results can be performed as shown in Code 9.2 after cloning the code. The program requires a recent GNU/Linux operating system with basic utilities, `python3` (version $\geq$ 3.8), `podman` and `qlever-control` installed. The output directory should have 1 TiB of free disk space, podman's data directory should have 500 GiB of free disk space and the system should have 100 GiB of available memory. On the machine stated in Table 5.1, the full reproduction takes approximately 85 hours.

```
# View options:
python3 reproduction.py --help
# List reproduction steps:
python3 reproduction.py --list-steps
# Run full reproduction:
python3 reproduction.py --output ./output
```

Code 9.2: Commands for reproduction of the results