Master Thesis

Implementing efficient geo queries for the SPARQL engine QLever

Jonathan Zeller

March 03, 2025

Albert-Ludwigs-Universität Freiburg im Breisgau
Technische Fakultät
Institut für Informatik

Bearbeitungszeitraum
02. 09. 2024 – 03. 03. 2025

Gutachter
Prof. Dr. Hannah Bast
Prof. Dr. Fabian Kuhn
Betreuer
Johannes Kalmbach

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Freiburg, 3.3.2025
Ort, Datum

J. Zeller
Unterschrift

Contents

Abstract

In this thesis the SPARQL engine QLever was extended by a SpatialJoin operation. The SpatialJoin operation receives two SPARQL subresults as input. Each input contains, among other things, geometries. Then, each pair of geometries is added to the result if and only if the distance between these two geometries is smaller than a user-defined limit. The SpatialJoin operation is implemented using R-trees to achieve fast runtimes. The algorithm has been evaluated on two datasets and compared to a baseline solution. In addition, a detailed theoretical analysis was performed, including a correctness proof of the algorithm.

Zusammenfassung

In dieser Masterarbeit wurde die SPARQL Suchmaschine QLever um eine SpatialJoin Operation erweitert. Die SpatialJoin Operation erhaelt zwei SPARQL-Teilergebnisse. Beide dieser Teilergebnisse erhalten unter anderem Geometrien. Von diesen Geometrien wird jedes Paar dem Ergebnis hinzugefuegt, genau dann wenn die Distanz zwischen den beiden Geometrien kleiner als ein benutzerdefiniertes Limit ist. Die SpatialJoin Operation wurde mithilfe von R-trees implementiert um schnelle Laufzeiten zu ermoeglichen. Der Algorithmus wurde anhand von zwei Datensaetzen evaluiert und mit einer Baseline verglichen. Zudem wurde eine ausfuehrliche theoretische Analyse durchgefuehrt, welche unter anderem die Korrektheit des Algorithmus beweist.

1. Introduction

This thesis starts by presenting some real-world examples of how geolocation queries can be used with the result of this thesis. After the motivation, it presents related work and gives an overview of this thesis.

1.1. Motivation

The use of geolocation information is ubiquitous. Most people use navigation services to find the fastest way to their destination. When they want to eat out, they look at an online map that shows the closest restaurants. And if they want to know about places of interest at their vacation destination, they can look them up beforehand. But geolocation services aren't just used by individuals; they're also used by businesses. Obvious examples include logistics companies that need to route their vehicles, and telecommunications companies that need to find the most densely populated areas so they can begin upgrading phone antennas there.

One way to use these geolocation services is to use the OpenStreetMap [1] dataset. The dataset contains a lot of information including geometries of the represented objects. In order to find geometries or objects of interest and to combine them with other constraints about the objects, a search engine like QLever [2] can be used. In the version prior to the start of this thesis, it was not possible to efficiently query the dataset for objects that were within a certain distance of each other. This thesis introduces an algorithm that efficiently computes all geometries that are at most a user-specified number of meters away from each other. Algorithm 1 shows how a restaurant query can be expressed in SPARQL[1]. In this example, the user is looking for restaurants that are close to a tram station and not too far away from the current position. Figure 1.1 shows what this query looks like when posted to the OpenStreetMap dataset [2] via the Qlever UI. The result of this query can be seen in Figure 1.2.

Another example might be a telecommunications company looking for the densest areas to start upgrading its antennas. By upgrading the antennas in the densest area first, the new antenna can serve the most customers first. This allows the

[1] for those unfamiliar with the SPARQL language, a very short introduction is given in section 2.2

[2] The OpenStreetMap dataset has been converted to a knowledge graph using the OSM2RDF tool. More about this in subsection 3.1.1

Algorithm 1 Sparql Query for restaurants in Freiburg

```
SELECT ?restaurant ?tramstation WHERE {
    ?restaurant <is-building> "Restaurant" .
    ?tramstation <public-transport> "Tram Station" .
    ?tramstation <max-Distance-in-meters:100> ?restaurant .
    ?restaurant <max-Distance-in-meters:3000> currentPosition .
}
```

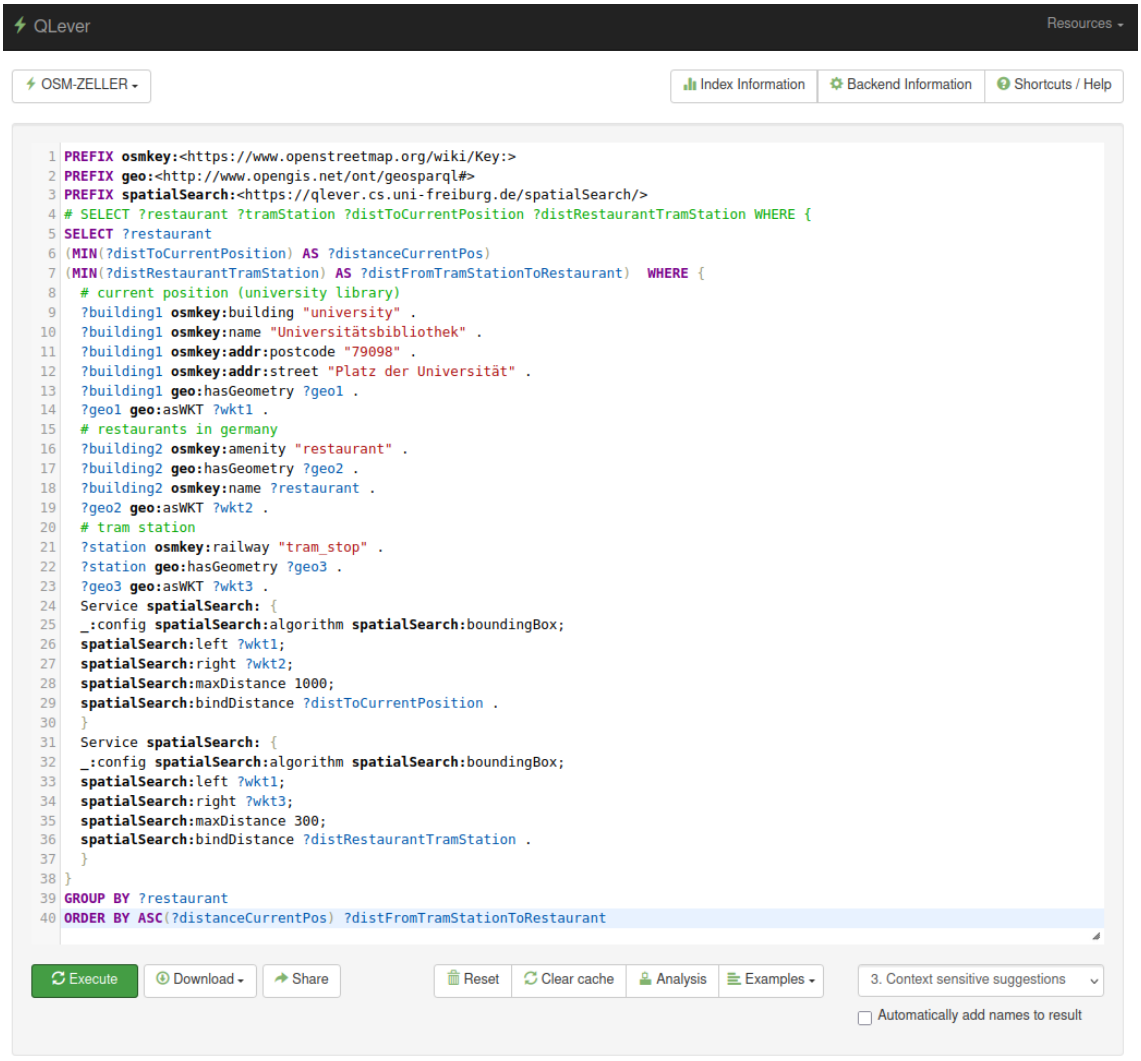


Figure 1.1.: This figure shows how a SPARQL query can be posted to QLever using the QLever UI. The query is about finding restaurants, which are near a tram station and not too far away from the current position looks.

Query results:

150 lines found | 2,480ms in total | 2,479ms for computation | 1ms for resolving and sending

Limited to 100 results; show all 150 results

	?restaurant	?distanceCurrentPos	?distFromTramStationToRestaurant
1	UNI Galerie	0.0157695	0.0973664
2	La Culinaria im Theater	0.0722287	0.0973664
3	Ristorante Pizzeria Bürgerstube	0.091399	0.0973664
4	Greencity	0.0965973	0.0973664
5	La Pepa	0.112243	0.0973664
6	Mensa Rempartstraße	0.143135	0.0973664
7	Olivia	0.148186	0.0973664
8	Muho	0.155988	0.0973664
9	Mai Wok	0.160974	0.0973664
10	M9	0.166726	0.0973664
11	Divan	0.172986	0.0973664
12	Schwarzer Kater	0.173218	0.0973664

Figure 1.2.: This figure shows the result of the query from Figure 1.1

telecommunications company to sell the faster phone service to more people compared to the situation where a less dense area is upgraded first. Algorithm 2 shows how to express this query in SPARQL. Using the GROUP BY statement, we can ensure that each antenna has only one entry in the result table[3]. Since all entries of the temporary table containing an antenna are aggregated into one row, the system needs to know how to aggregate the rows for the position and the houses. In the case of the antenna position, the keyword SAMPLE is used. This just uses an entry from one of the rows. This is fine because the position for the antenna is the same in all rows (since we are aggregating over the different antennas). So it doesn't matter from which row the position is taken. The aggregation for the houses is done with the COUNT keyword. This counts the number of rows that result in the number of houses that are within range of the antenna. In summary, the result table contains one row for each antenna, which contains the position of the antenna and the number of houses reached by the antenna.

[3]otherwise, each antenna would be shown with each house, as the cross-product would be built

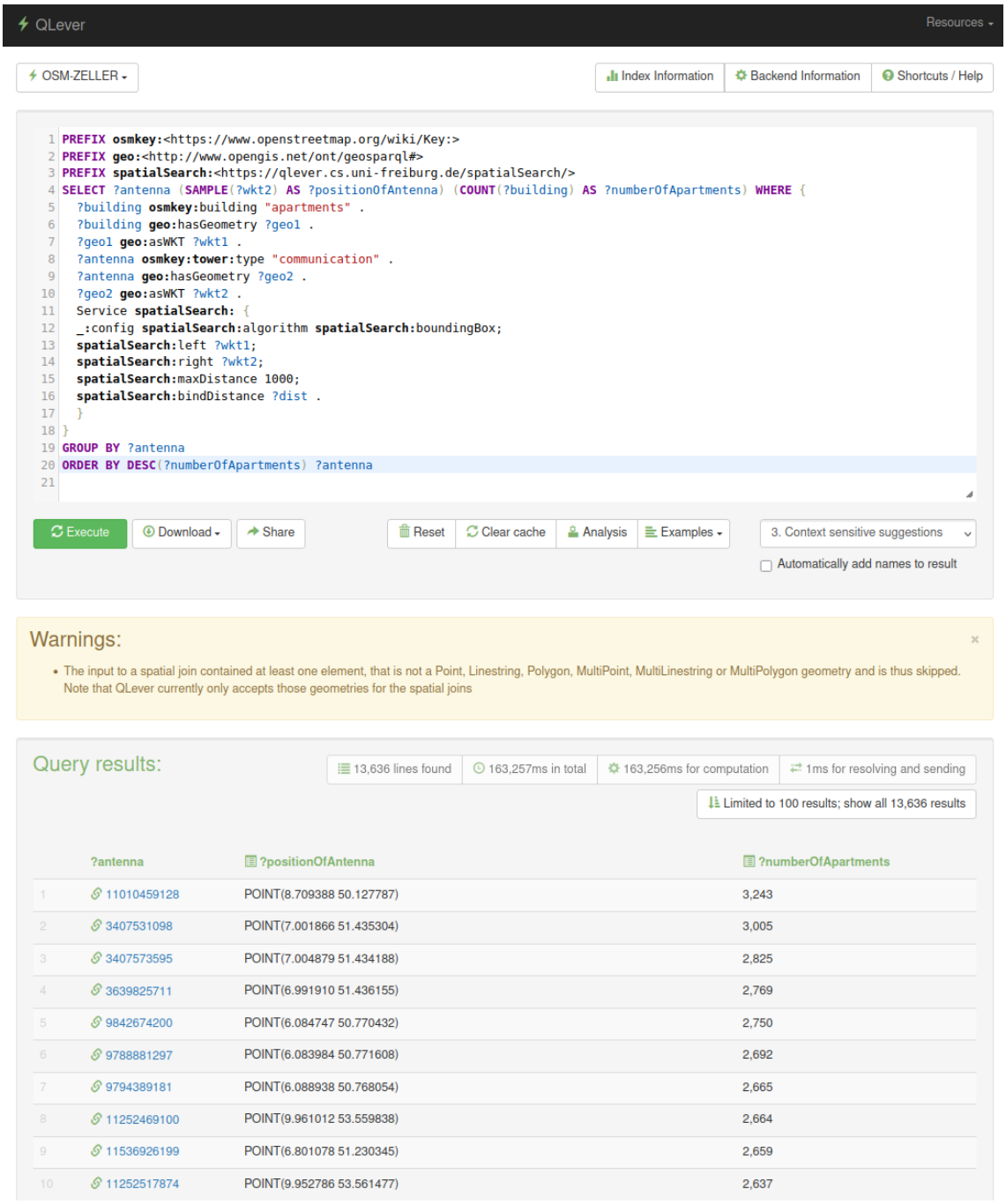


Figure 1.3.: This figure shows the query and the results of the query, which searches for antennas, and how many apartments each antenna reaches

Algorithm 2 SPARQL query to find the best antenna position to upgrade

```
SELECT ?antenna
        (SAMPLE(?position) AS ?positionOfAntenna)
        (COUNT(?house) AS ?numberOfHouses)
WHERE {
   ?antenna <is-a> "Antenna" .
   ?antenna <has-position> ?position .
   ?house <is-building> "House" .
   ?house <has-position> ?positionHouse
   ?position <maxDistanceInMeters:1000> ?positionHouse .
}
GROUP BY ?antenna
ORDER BY DESC(?numberOfHouses) ?antenna
```

1.2. Related work

The most important related work is the paper that introduces the search engine QLever [2]. QLever is a search engine by Prof. Bast et. al [2] for data structured as a knowledge graph. Since this thesis extends the capabilities of QLever, QLever is presented in detail in chapter 2.

Another very important paper is the OSM2RDF paper by Prof. Bast et.al [3]. It translates the OpenStreetMap data provided in XML format into the RDF triple format without any data loss. Since this tool is also essential for this thesis, it is presented in detail in subsection 3.1.1.

The implementation of the algorithm in this thesis uses R-trees, which were introduced in the R-trees paper by Guttman [4]. Since R-trees are a fundamental part of this thesis, they are presented in chapter 4.

Instead of using R-trees, the algorithm could have been implemented using another data structure. The S2 library, developed by Google [5], stores the data on a sphere and aims to be an alternative to planar geometry algorithms. Because the data is stored on a sphere rather than a plane, it avoids edge cases around the poles and the -180/180 degree longitude line. The S2 library stores the data in a hierarchical structure to allow fast queries. The S2 library is a very promising approach to the problem this thesis tries to solve. When I started this thesis, I started the development by implementing the infrastructure for SpatialJoin operations in general (no matter which algorithm). During this time, another student started his thesis on a problem related to SpatialJoin: Nearest Neighbor search. For this approach he uses the S2 library. Since the S2 library is already used for an algorithm, it was decided that my implementation should be done using planar R-trees. Since our thesis finish at a similar time, it was unfortunately not possible to compare both approaches, but this should definitely be done in the future.

Another important related work is the GeoSPARQL standard [6]. It extends SPARQL with geospatial query capabilities. It has some mandatory features, such as support for geometries and encoding them in well-known text representations. In addition to mandatory features, there are also optional features, such as the distance function, which calculates the distance between two geometries. Implementers of the standard can therefore decide whether or not to support the optional distance function. It's also optional whether they implement it in an efficient way, like using an R-tree, or whether they just use a simple filter that has to compute the distance for all pairs. Some examples of SPARQL servers that support the GeoSPARQL standard are Apache Jena GeoSPARQL [7] or GraphDB [8]. Due to time constraints, it was not possible to compare the performance of the presented algorithm with other algorithms implementing the GeoSPARQL standard.

1.3. Structure of this thesis

After this introduction, the thesis continues by introducing knowledge graphs, SPARQL, and QLever in chapter 2. The next chapter introduces OpenStreetMap, a dataset on which the algorithm will be used. After that, a basic introduction to R-trees is given, which is a fundamental building block of the presented algorithm. Then the coordinate system of the earth is explained, as well as a mapping from the spherical surface to a planar surface. As soon as these chapters have given the basics, chapter 6 presents the changes in the infrastructure of the search engine QLever to make the developed algorithm fit into it. Then the next chapter explains the presented algorithm in detail. chapter 8 analyzes the algorithm on a theoretical basis. There, the problem that the algorithm solves is specified in a formal way. In addition, the runtime and space complexity of the algorithm is analyzed. Finally, the chapter proves that the algorithm works correctly. After the theoretical analysis, the practical analysis is presented. Here, the algorithm is analyzed on a synthetic dataset as well as on the OpenStreetMap of Germany. The last chapter of this thesis gives a conclusion and an outlook.

2. The search engine QLever

QLever is a search engine by Prof. Bast et. al [2] for data structured as a knowledge graph. This chapter presents the current state of QLever, which is the basis of this thesis. The chapter starts by defining what a knowledge graph is and how SPARQL can be used to retrieve information from a knowledge graph. After that, the implementation of QLever is presented. It's shown how operations are implemented, how the index is built. How queries are planned and how to interact with the program via the server and the UI.

QLever has many more features, which are not relevant for this thesis. One example is autocompletion, which helps the user to type queries [9]. Since this is not relevant for this thesis, it will not be presented here. For more information about autocompletion, see [9]. QLever is also able to search a linked text corpus in addition to the knowledge graph. For more information about this, see [2].

2.1. Knowlegde Graphs

A knowledge graph is a graph consisting of nodes and directed edges. The nodes are objects and the edges make some statements about the relationship between two objects. An example can be seen in Figure 2.1. The same information from the graph can also be stored as text. Then the nodes become subjects and objects, and the edges become predicates. The same information of the graph from Figure 2.1 can be seen in Table 2.1. The information in the table is stored as triples. This makes it easy to parse and edit. Examples of large knowledge graphs are Wikidata [10] and Freebase [11]. For a detailed description and comparison of these and some other knowledge graphs, see [12].

Table 2.1.: This table reprensents the information of a knowledge graph. The same information can be represented as a graph, which can be seen in Figure 2.1

subject	predicate	object
<Uni Freiburg>	<founded-in>	21.09.1457 .
<Uni Freiburg>	<located-in>	<Freiburg im Breisgau> .
<Uni Freiburg>	<is-a>	"university" .
<Albrecht VI.>	<has-founded>	<Uni Freiburg> .
<Minster of Freiburg>	<located-in>	<Freiburg im Breisgau> .
<Minster of Freiburg>	<located-in>	<Altstadt of Freiburg> .
<Minster of Freiburg>	<is-a>	"building" .
<Minster of Freiburg>	<is-a>	"church" .
<Freiburg im Breisgau>	<inhabitants>	236,000 .
<Uni Freiburg>	<has-library>	<University Library Freiburg> .
<University Library Freiburg>	<located-in>	<Freiburg im Breisgau> .
<University Library Freiburg>	<is-a>	"building" .
<University Library Freiburg>	<is-a>	"library" .
<Altstadt of Freiburg>	<located-in>	<Freiburg im Breisgau> .

To retrieve information from a knowledge graph, a query language such as SPARQL [13] can be used. The following chapter explains the basics.

2.2. SPARQL

SPARQL is a language that can be used to retrieve information from a knowledge graph. This chapter explains the basics of SPARQL. Advanced concepts and the full capabilities of SPARQL can be found in [13].

As a first introductory example, we will search for everything that is located somewhere. This can be done using algorithm 3. After the SELECT keyword, we declare the variables that will appear in the result table. In this case, it's the variable ?something, which represents an object that will be located somewhere. This somewhere is stored in the variable ?location. Except for these variables, other intermediate variables can be declared later, but they won't be part of the result table unless they are written here. After all the variables of interest have been declared, the WHERE keyword is used to start a block of constraints that must be satisfied. In this case, the only constraint is that the variable node must have an outgoing edge labeled "located-in" that connects to another node (see Figure 2.1 for the input knowledge graph). If the knowledge graph is stored as triples of subject, predicate and object, we search for the subjects and objects of the rows that have the predicate "located-in" (see Table 2.1 for the input triples). The result of this SPARQL operation can be seen in the result table, Table 2.2.

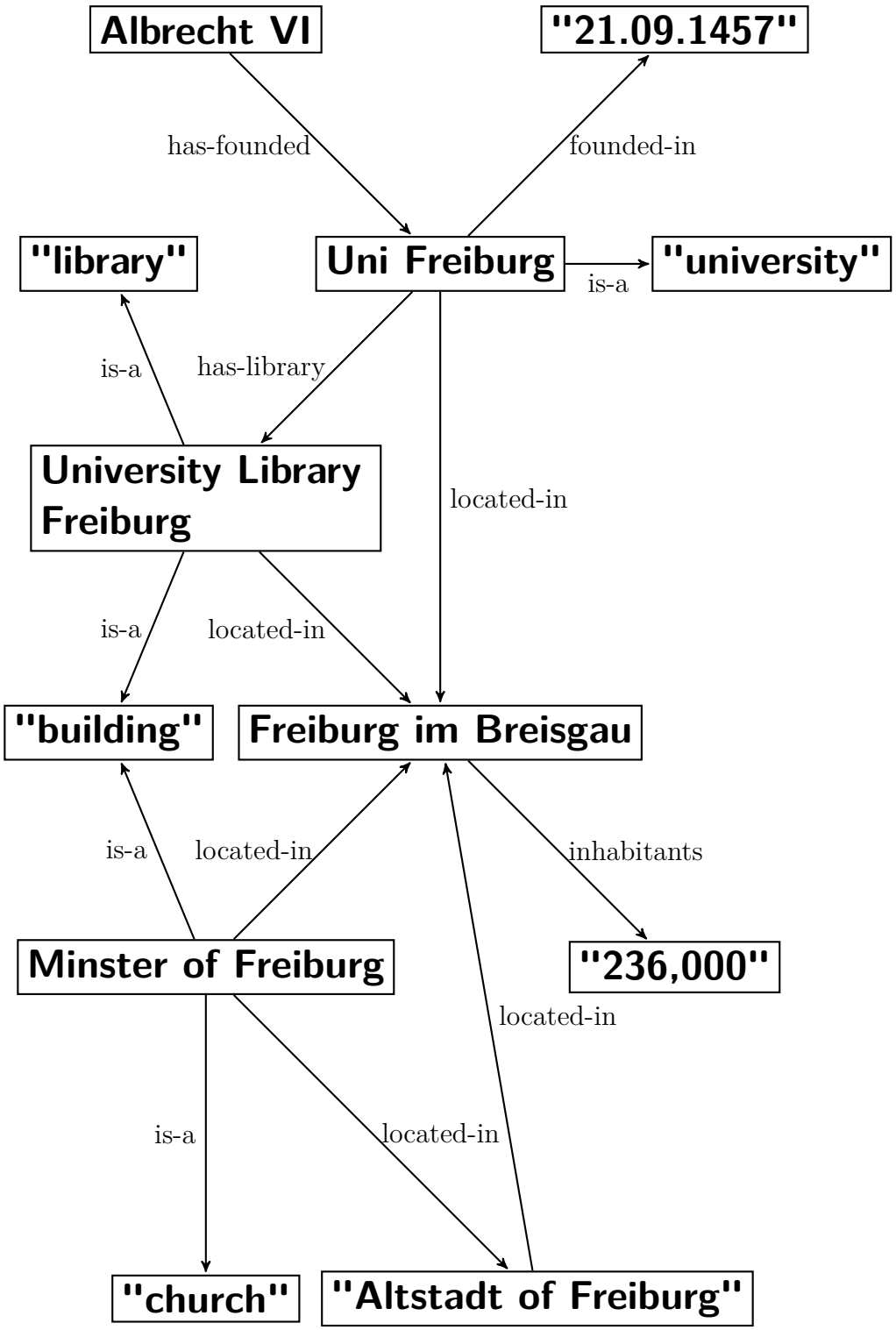


Figure 2.1.: This figure shows a knowledge graph. The nodes are objects and the relationship between nodes is represented by directed edges. The information contained in this knowledge graph can also be represented in text form, as shown in Table 2.1. Then the nodes are subjects or objects and the edges are predicates. Source: Self made

Algorithm 3 SPARQL query for everything, which is located somewhere

```
SELECT ?something ?location WHERE {
    ?something <located-in> ?location .
}
```

Table 2.2.: Resulting table from the SPARQL query of algorithm 3

?something	?location
Uni Freiburg	"Freiburg im Breisgau"
University Library Freiburg	"Freiburg im Breisgau"
Minster of Freiburg	"Freiburg im Breisgau"
Minster of Freiburg	"Altstadt of Freiburg"
Altstadt of Freiburg	"Freiburg im Breisgau"

Another simple SPARQL query might look like algorithm 4. Here we are looking for anything that has a generalizing "is-a" information. This information is stored in the variables ?something and ?itIsThis. The resulting table for this query can be seen in Table 2.3.

Algorithm 4 SPARQL query for everything, which has a generalizing predicate "is-a"

```
SELECT ?something ?itIsThis WHERE{
    ?something <is-a> ?itIsThis .
}
```

Table 2.3.: Resulting table from the SPARQL query of algorithm 4

?something	?itIsThis
Uni Freiburg	"university"
University Library Freiburg	"library"
University Library Freiburg	"building"
Minster of Freiburg	"church"
Minster of Freiburg	"building"

If we want to know about everything that's located somewhere and what its generalization is (or what they are in the case of multiple generalizations), we can combine the two queries from above. The combination can be seen in algorithm 5. Here we declare three variables to be in the resulting table and two constraints to be satisfied. The two constraints are exactly the same as in the two simple queries in algorithm 3 and 4. Therefore, the intermediate results of this query are also the two tables from

before (Table 2.2 and Table 2.3). Now we need to join these two tables to create the final result table. Since both tables have a common variable, in this case ?something, they must be matched. The matching starts with the first row of the intermediate result in Table 2.2. The variable ?something is "Uni Freiburg". Now it searches all rows in Table 2.3 for "Uni Freiburg" in the ?something column. It finds the one row that contains both "Uni Freiburg" and "university". So it joins those two rows, resulting in the row "Uni Freiburg", "Freiburg im Breisgau" (which comes from the first table) and "university" (which comes from the second table). The final row can be seen in Table 2.4. Next, it tries to match the next row from Table 2.2. The variable on which the two columns are joined is "University Library Freiburg". This variable occurs twice in the intermediate result of Table 2.3. Therefore, two rows are added to the resulting table. One row for the combination of "University Library Freiburg" and "Freiburg im Breisgau" with "library" and one row for the combination with "building". This can be seen in the second and third lines of Table 2.4. The next two rows contain "Minster of Freiburg". Although the variable is the same, each row is treated separately. The matching process is the same as before. Each of the two rows from Table 2.2 is joined with the two rows from Table 2.3 that also contain "Minster of Freiburg" in the ?something column, resulting in four rows in the result table. This can be seen in rows 4 to 7 of Table 2.4. The last element from Table 2.2 to be considered in the matching process is "Altstadt of Freiburg". Since this variable does not appear in the ?something column of Table 2.3, it is not part of the resulting table. Now the join process is complete and the resulting table, Table 2.4, can be returned as the output of the combined SPARQL query from algorithm 5.

Algorithm 5 SPARQL Query for everything, which is located somewhere and has a generalizing predicate "is-a"

```
SELECT ?something ?location ?itIsThis WHERE{
    ?something <located-in> ?location .
    ?something <is-a> ?itIsThis .
}
```

Table 2.4.: Resulting table from the SPARQL query of algorithm 5, which looks for something, which is located somewhere and has a generalization

?something	?location	?itIsThis
Uni Freiburg	"Freiburg im Breisgau"	"university"
University Library Freiburg	"Freiburg im Breisgau"	"library"
University Library Freiburg	"Freiburg im Breisgau"	"building"
Minster of Freiburg	"Freiburg im Breisgau"	"church"
Minster of Freiburg	"Freiburg im Breisgau"	"building"
Minster of Freiburg	"Altstadt of Freiburg"	"church"
Minster of Freiburg	"Altstadt of Freiburg"	"building"

At the end of this section, i want to give some final remarks about the join process.

- As we have seen in the join process, the resulting table, which has seven rows, is larger than both input tables, which have five rows each. This is due to the "cross-product effect". This effect can be seen when joining "University of Freiburg" and especially when joining "Minster of Freiburg". In each of these cases, a row from one table is joined with several rows from the other table. In the extreme case, the first join table contains n rows, all with the same variable. The second table contains m rows, all with the same variable, which also matches the variable from the other table. Then each of the n rows from the first table is matched with each of the m rows from the second table. This would result in a table of size m times n, which can be much larger than any of the input tables.
- In the explanation above, it was mentioned that for each variable of the first temporary table, the other table is fully searched. If both tables are sorted by the join column, a full search can be avoided, resulting in a faster and more efficient join process.
- The query can be interpreted as the table matching explained above. Another way to look at this query is in terms of graph pattern matching. In this case, the variables in the query impose constraints on the nodes and the edges between those nodes. The query can be thought of as a tiny subgraph (compared to the knowledge graph) that is searched in the knowledge graph. Figure 2.2 shows the subgraph for the query of everything that is somewhere and has an "is-a" generalization (see algorithm 5). The variables of this subgraph are replaced by the contents of the matching subgraph. In this case, we are looking for a node that has two outgoing edges, one labeled "located-in" and one labeled "is-a". If this pattern is found, the content of the node is assigned to the variable ?something. The content of the node connected to the node by the located-in edge is assigned to the variable ?location. And the content connected to the node by the "is-a" edge is assigned to the variable ?itIsThis.
- In the examples above, neither the nodes (subjects or objects) were constraints, nor the edges (predicates) were variables. However, both is possible. An example where the node (in this case the object) is a variable could be the query of everything that is "located-in" the "Altstadt of Freiburg". In this case, the node (object) is a constraint. An example where the predicate is a variable is the query of all stored relationships of the University of Freiburg ("Uni Freiburg" ?relationshipTo ?TheObject). The graph of this query would look like a node labeled "Uni Freiburg" (the constraint for the subject). The outgoing edge is variable, as is the node.
- Since a knowledgegraph can only connect two nodes, it seems that it can't store every kind of information. N-ary information, which makes a statement about n things simultaneously, can't be stored directly. To get around this limitation, a mediator node can be introduced. Then the subject is linked to

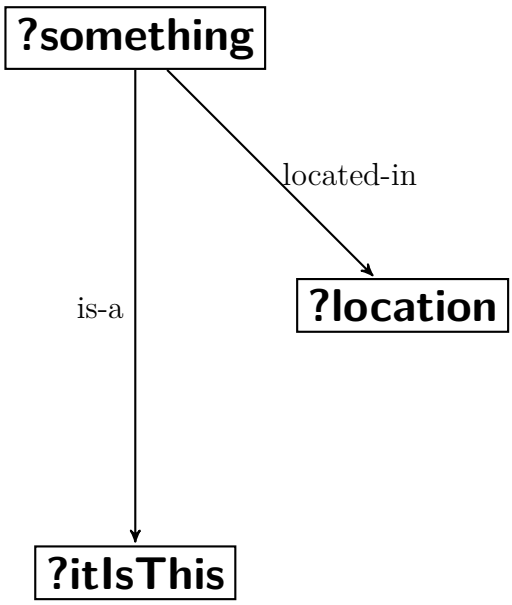


Figure 2.2.: This figure shows a representation of the query from algorithm 5. The structure of this graph will be searched in the complete knowledge graph of Figure 2.1. The variables in the nodes, will be replaced by the real content of the nodes from Figure 2.1. Source: Self made

the mediator node, and the mediator node is linked to the other $n-1$ nodes. This allows knowledge graphs to represent n-ary information and avoids the need for a hypergraph.

- As mentioned at the beginning of this chapter, this chapter has only provided a basic introduction to SPARQL. SPARQL is capable of many additional features, such as filtering the resulting table and merging multiple rows into one (for example, counting the number of rows that contain the same subject).

Now that knowledge graphs and SPARQL have been introduced, we can take a closer look at how QLever is implemented to compute Sparql queries and retrieve the information from the knowledge graph.

2.3. Operations

When computing the result of a SPARQL query, QLever translates the query into operations. The most basic operation is a SCAN operation. A SCAN operation is an operation that searches the entire knowledge graph for all matching triples. In the example algorithm 5 there are two SCAN operations. The first scans the entire knowledge graph for triples with a "<located-in>" predicate, and the second scans the entire knowledge graph for triples with an "<is-a>" predicate. The SCAN operation can also search for triples with three variables (this would return every

triple from the knowledge graph) or triples with only one variable. An example of a single variable would be if the ?location variable in the "<located-in>" line were replaced by "Freiburg im Breisgau". Then the SCAN would find all triples that have the predicate "<located-in>" and the object "Freiburg im Breisgau". Another important operation is the JOIN operation. It joins the result of two other operations, for example two SCANs. In the case of algorithm 5, the two scan operations would need to be joined. Since they share the common variable ?something, the JOIN operation must take this into account and only join a row from one SCAN operation with a row from the other SCAN operation if the variable ?something has the same value. If the SCAN operations didn't have a common variable, the cross-product of the subresults would have been built. If the SPARQL query would have more than two SCAN operations, then multiple JOIN operations are required. In the case of three SCANs, first two SCANs would be joined and then the result of this JOIN operation would be joined with the remaining SCAN operation. There are many more operations, such as grouping, ordering, sampling, and counting, that would be needed in algorithm 2. Since these operations are not really relevant for this thesis, they will not be explained here.

QLever has an abstract operation class that groups the common properties of all operations and forces others to be implemented by subclasses. The reason for this enforcement is that, among other things, the QueryPlanner, which will be described later, needs these functions to work properly. One of these functions is a size estimator, which estimates the size of the result. Another function estimates the multiplicity, i.e. how many times the same entry is contained in a certain row of the result table. In the example of the result table of Table 2.4, each column contains a certain entry multiple times. For example, "Freiburg im Breisgau" appears five times in the ?location column. Another function calculates whether the result is known to be empty. Another function calculates the variable to column map. This function returns a map containing key value pairs. The keys are the variables stored in the result table of this operation and the value is the column where the information of this variable is stored. The variable to column map for Table 2.4 would look like this:

- ?something: 0
- ?location: 1
- ?itIsThis: 2

The last method that must be implemented by any subclass of the operation class is the computeResult method. This method is called to compute the result.

2.4. IndexBuilder

To enable fast queries, QLever first computes an index of the knowledge graph. The triples from the knowledge graph are stored in the index in a way that allows fast re-

trieval of SCAN operations. The way they are stored is not in the usual triple format as in the input knowledge graph, but ordered by subjects, predicates and objects. An example of this is the PSO[1] permutation. There, each predicate has a pointer to a block in memory where all subjects and objects that occur in combination with that predicate are stored. If a predicate has many subject-object combinations, another optimization is added, where for each predicate-subject combination all objects are stored in a separate place. Storing the information this way has the advantage that the needed data is stored contiguously, which is cache-friendly and allows for very fast read times, since no redundant information has to be read (such as the predicate, which in this example is not variable, but was specified by the user). When the information is stored this way, it's called a PSO permutation, because the information is stored first by predicate, then by subject, and then by object. All the other five permutations, such as POS, SPO, SOP, OPS, and OSP, can also be created. Then any combination of variables in a scan can be read efficiently. Another memory optimization is the use of a vocabulary. Here the subjects, predicates and objects are translated into IDs, which require less memory. The IDs are calculated in such a way that, for example, the ID of each string is smaller than the ID of all strings that are behind the string in a lexicographic sort and vice versa. This optimization makes it possible to sort the output without using the larger string objects. Finally, before presenting the data to the user, the IDs are translated back into strings. Some information of the triples that is small enough to fit into the 60 data bits of the ID is stored directly in the ID, without translation. Examples of this are integers, which don't need more than 60 bits. Parallel to my development of the SpatialJoin, another student was working on a similar task with SpatialJoins. He introduced an optimization in the Index Builder. Until this optimization, my prototype of the SpatialJoin operation would load the IDs of the geometries from the subresults. Then it would translate them back to strings, parse the strings to get the geometry information, and then do the further computation with the geometry information. Reading from disk is very slow compared to reading from memory, and that's where the other student's optimization comes in. If the geometry is just a point, it would be stored in the ID. IDs like this are then called GeoPoint. Then the whole translation of the ID into the string, which would go over the disk, could be skipped. Since the ID only has room for 60 data bits, and each point consists of a latitude and a longitude, there is a small loss of precision. 30 bits per coordinate corresponds to a spatial resolution of about 4 cm, which should be accurate enough in almost all cases. For areas the optimization does not work, because 60 data bits are not enough. So areas still have to be read from disk.

[1]PSO stands for the order of predicate subject object

2.5. QueryPlanner

QLever uses a QueryPlanner that evaluates different ways to compute the given SPARQL query and tries to find the fastest way to execute the query. The QueryPlanner uses dynamic programming. The base case of dynamic programming is the SCAN operation. Therefore, the first row contains all SCAN operations of the query. To get the next row, the row with index i, all possible subresults of earlier rows are merged if the merged subresults would cover exactly i basic operations from the input SPARQL query. When two of the subresults are merged, a JOIN operation is created. The JOIN operation has as children the two subresults of the earlier rows that were just merged. During merging, the QueryPlanner is using the variable to column map to figure out, which subresults share a common variable. Only if two subresults share a common variable they get merged[2]. Then the QueryPlanner uses the information about the shared variable and which column the shared variable is in each of the subresults to pass this information to the newly created JOIN operation, as the JOIN operation needs this information to know which columns to match. This process continues until the last row is reached, where all basic operations are covered. To evaluate the different query plans, the QueryPlanner uses the getSizeEstimate, getCostEstimate and getMultiplicityEstimate functions of each operation in the plan. This information can be used to calculate a runtime estimate. The QueryPlanner then takes the cheapest plan and executes it.

During the query planning process, so-called ExecutionTrees are built. These ExecutionTrees contain a hierarchical order of operations and the order in which they must be executed. Before each operation can be executed, the subresults of the child operation (the operations one hierarchy level below) must be present. To get the final result, the computeResult function of the root node simply needs to be called. It calls it's children to compute their subresults, which call their children to compute their subresults, and so on in a recursive manner. The ExecutionTree is thus computed in a bottom-up fashion. Each operation calls the computeResult method of it's children to get their results and then performs its own operation. An example of an ExecutionTree, from chapter 1 with the antenna query from algorithm 2, can be seen in Figure 2.3. The figure shows the order of the operations, the size of the subresults after each operation, the time it took to compute the operation, and the variables contained in the subresult.

2.6. QLever Server

The server is the main program of QLever. It accepts queries via an http request, calculates the result of the query and sends it back via http. The QueryPlanner

[2]unless in the end the last operations doesn't contain a shared variable. In this case, the cross product of the subresults must be built

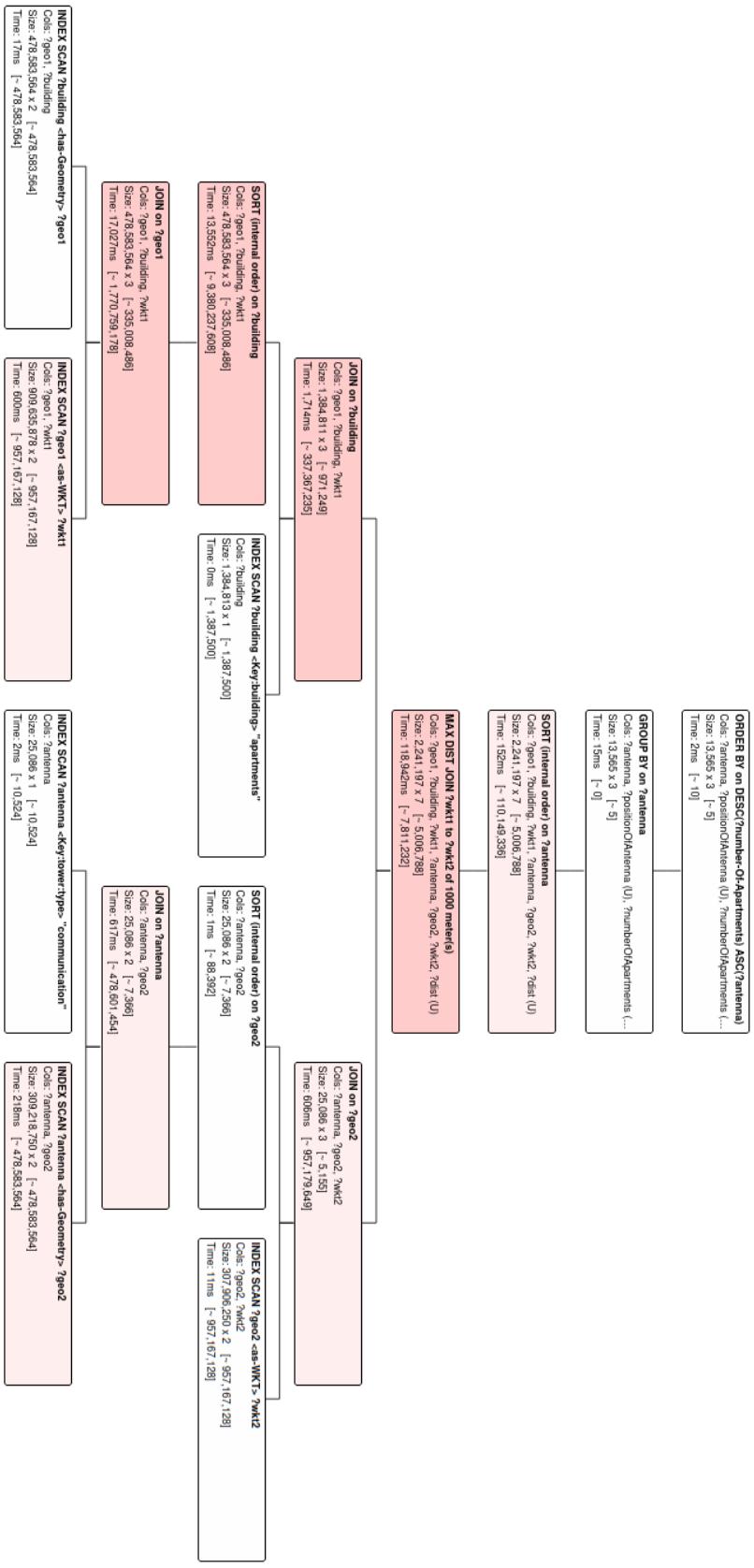


Figure 2.3.: This figure shows an ExecutionTree that has been constructed by the QueryPlanner and executed by the main program. The nodes of the ExecutionTree show the size of the (sub)results, the time taken to compute the (sub)results and the variables contained in the (sub)result tables.

and the operations are contained in the QLever server. As soon as a request is received, the server starts the QueryPlanner to find out the fastest execution order of the operations. Once the plan is built, it is executed and the result is sent back along with computation statistics. After each query is computed, the result is also cached (unless caching is disabled). This allows QLever to reuse the results of previous queries if they are part of another future query. In addition, the server also accepts partially typed SPARQL queries, where it computes context-sensitive auto-completion suggestions.

2.7. QLever UI

The QLever UI is a web-based user interface for sending queries to the QLever server, where the result is computed. Its appearance has already been seen in Figure 1.3. The main QLever program contains a server that accepts queries, calculates their result and sends it back to the QLever UI. The UI and the server share more data than just the result. They also exchange information about the characters the user enters in the query to provide context-sensitive auto-completion suggestions. After the result is computed, an analysis of the computation can also be seen in the UI. Such an analysis has already been shown in Figure 2.3.

3. OpenStreetMap

In this chapter, I will present the dataset that provides the information for this thesis, as well as the conversion of it to make it compatible with the SPARQL engine Qlever. Of course, the algorithm can also work with other datasets that meet the necessary requirements, but it was designed with the OpenStreetMap [1] in mind. For more information about the formal requirements that must be met for the algorithm to work on any dataset, see section 8.1. For an example of another dataset used with the engine, see section 9.1, where a synthetic dataset is created to evaluate and compare the performance of this algorithm. Since the main use case of this algorithm is to be integrated into QLever and to serve geo queries on the OpenStreetMap data, I will introduce the OpenStreetMap dataset next.

3.1. OpenStreetMap

OpenStreetMap is a free map of the world. It consists of roads, rivers, railroads, forests, buildings, etc. It is maintained on a voluntary and crowd-sourced basis by people from all over the world. It's goal is to provide a world map that is free for everyone to use. In addition, it offers the data not only in pre-calculated maps, but also in its raw format. This is especially useful for cases like this master thesis, where you want to edit the data or use it in its raw form in your own algorithms. Since the project is crowd sourced and volunteer driven, the amount of data per region can vary greatly. Some regions have very high quality data, where even trees and park benches are included in the dataset. In other regions, an entire village may be missing.

OpenStreetMap represents objects internally in three ways: nodes, ways, and relations [14]. Nodes are the most basic building block of geometric objects: points. They each have at least an ID and coordinates in latitude and longitude (using the WGS84 system [15]). In addition to these mandatory features, a node can have any number of key value pairs called tags, which store information like the name of the point, its elevation, and so on. The next internal object is a way. It consists of an ordered list of at least one point and can be as large as 20,000 points. All these points are of the node type mentioned above. The way can be used to represent line features such as a river or a road. If the first point in the list has the same coordinates as the last point, it is a closed way. This can be used as a boundary of a polygon, like a building, a country or a forest. As opposed to being the boundary

of an area, it could just be an object with a loop, like a roundabout. This ambiguity can be resolved by using tags. A tag could be "area=false" or "boundary=true". The final internal component is the relation. It consists of an ordered list of the other components: nodes, ways, and other relations. One possibility is a multipolygon with holes in it's area. Then one way can represent the outer boundary and other ways can represent the inner holes. Like the node and the way, the relation can have tags that further describe the object. Most OSM data is stored in XML format [16]. If the data is needed in a different format, a conversion is required. There are many different tools for this. One conversion tool is the osm2rdf tool by Prof. Bast et. al. [3], which will be explained in the next subsection.

3.1.1. OSM2RDF

Since the data is needed in RDF triples format, but is only provided in XML format, a conversion needs to be done. The OSM2RDF tool by Prof. Bast et. al. [3] provides such a conversion. Unlike other conversion tools, it preserves all data, especially all geometric information. In addition, the tool can add additional triples that cover geometric relations between objects. These relations are "intersects" and "contains". Using these additional triples, even SPARQL engines that can't normally handle geometric queries can handle them. However, the precomputation of these additional triples is very slow. For the whole planet of open street map, the precomputation takes 48 days, for the open street map data of Germany it takes 16 hours.

The OSM2RDF tool creates several predicates and prefixes during the conversion. The aforementioned OSM elements "nodes", "ways" and "relations" get the prefixes "osmnode:", "osmway:" and "osmrel:" and key value pairs get the prefix "osmkey:". The conversion adds a triple containing the geometry as WKT (well known text) and optionally the bounding box of the object using the predicate envelope.

4. R-trees

This chapter introduces the concept of R-trees. In this example, we try to find all points of a given set of points that are contained in a query rectangle. A naive approach would be to iterate over all points and check if a point is contained in the rectangle. Using the rtree, we can be much faster. The general idea behind rtrees is to improve query times by ordering the points in a hierarchical way. Using this ordering, many comparisons can be skipped. In the next chapter, the idea behind this concept will be explained in detail.

4.1. Working principle of R-trees

As an introductory example, let's consider the two-dimensional Cartesian plane. In addition, we have a set of points contained in this plane. An R-tree is a data structure of type tree. The root node represents the set of all points. The children of a node are one hierarchy level below the parent node. The union of all direct children results in the set of the parent node. The sets of the children are disjoint from each other. In this example, the sets are represented by rectangles whose sides are parallel to one of the axes of the coordinate system. An example can be seen on the left side of Figure 4.1. Here each point is represented by a blue dot. The nodes of the R-tree are the colored boxes that contain a subset of the nodes. The red node is the root node, which contains all points. The green nodes are the direct children of the red node, each containing half of the points. The yellowish nodes are the children of one of the green nodes, which contain half of the points of a green node (and therefore a quarter of the points of the red node). The tree representation of the R-tree can be seen on the right side of Figure 4.1.

Now that we have constructed the R-tree, we can use it. Suppose we want to know all the points contained in a rectangle that is aligned with the coordinate system. First, consider the following rectangle. Its lower left point is (-2.5, 1.5) and its upper right point is (-1.5, 2.5)[1]. When querying the R-tree with the rectangle, the procedure starts by comparing the query box with the root node. In this case, the result is that the query box is completely contained in the rood node (the query box is completely contained in the red box in Figure 4.1). Therefore, the comparison continues with the direct child nodes of the root node (the two green nodes in

[1]Since the rectangle is aligned with the axes of the coordinate system, we know that the upper left point is (-2.5, 2.5) and the lower right point is (-1.5, 1.5)

Figure 4.1). The upper green rectangle also completely contains the box, but the lower green rectangle has no intersection with the query rectangle. Therefore we know that all points contained in the lower green rectangle cannot be contained in the query rectangle. We can skip the calculation of whether the point is inside the query rectangle for all points in the lower green rectangle. Since the query rectangle is inside the upper green rectangle, the check continues with its direct children, the two yellowish rectangles. Again, the query rectangle is contained in exactly one of the children, the upper left yellowish rectangle. Therefore, we can exclude all points of the other rectangle. Since the upper left yellowish rectangle is a leaf node and has no children, each point contained in the node is now compared to the query rectangle to see if it is contained in it. Because of the R-tree, only 9 instead of 36 points had to be compared to the query rectangle. The overhead of checking the query rectangle against the rectangle of the nodes gets less and less as the set of points per node gets larger. In this example, checking if a box is contained in another box may not seem efficient, but once the number of points gets larger and a node represents more than just nine points, the improvements in query times are huge. Note that if a query rectangle is not completely contained within a node, but intersects two nodes, then the points of both rectangles must be checked against the query rectangle.

4.2. Usage of R-trees in this thesis

The goal of this thesis is to find all points that are within a certain user-defined distance from another point. To solve this problem, a box is computed that contains the entire area where all points are at most this distance away. This box will then be the query box mentioned above. If we are interested in areas that are at most a certain distance from another area or point, we can use the same technique again. We create a rectangle containing the area where all possible matches must be contained, and use this rectangle to query the R-tree. In this thesis the boost library, which provides an implementation of an R-tree [17], gets used.

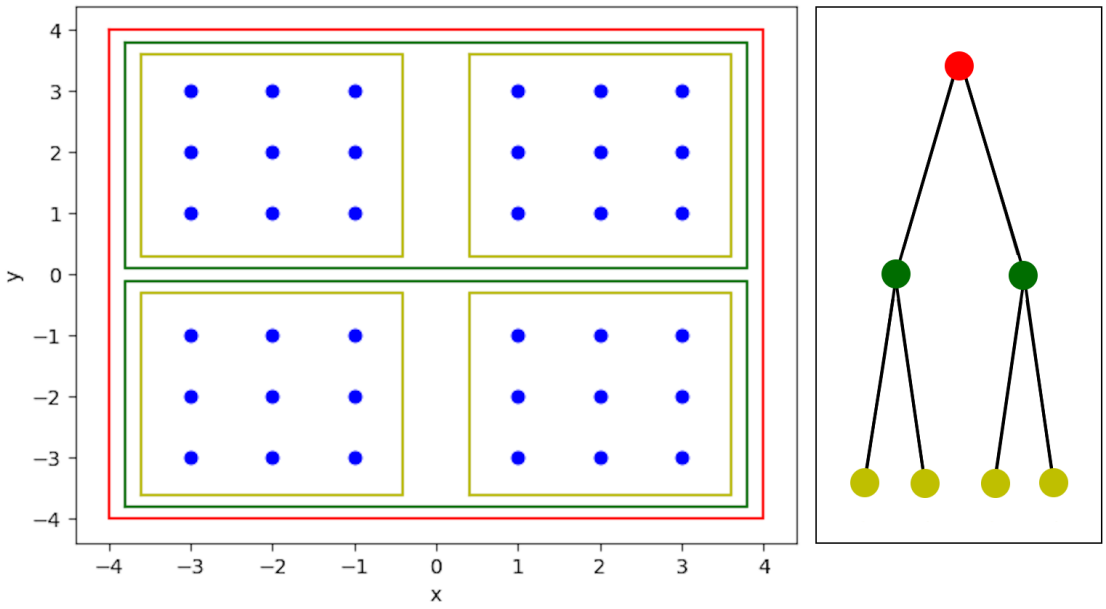


Figure 4.1.: This figure shows how the R-tree splits a sample dataset of blue points. On the left, the boxes show which points are contained in which nodes. On the right side you can see the tree structure of the R-tree. Note that the boxes on the left are usually as narrow as possible, but still contain all relevant points. Here, they are not as tight only for illustrational purposes. Source: Self made

5. The Coordinate System of the Earth

This chapter gives a very brief introduction to the coordinate system of the Earth. It's explained how the coordinate system works on the spherical surface of the Earth and how to map it to a Cartesian space.

5.1. Mappings and Projections

The Earth has the shape of an ellipsoid. This means that the distance from the Earth's surface to its center is not constant. Figure 5.1 shows an exaggerated illustration of this phenomenon. It shows that the distance from the center of the Earth to the equator is greater than the distance from the center to the pole. However, the difference between the major axis and the minor axis of the Earth is very small, the major axis is only 0.34 % larger than the minor axis. This difference is so small that i will assume that the Earth is a perfect sphere in this master's thesis. I will also assume that the surface of the Earth has a constant height (so mountains and valleys will be ignored). The reason for this is the same argument: The largest mountain above sea level is Mount Everest with a height of 8849 meters. This is only 0.14 % more than the radius of the Earth. These two approximations of the Earth make the math much easier and allow for faster calculation times, while still being very accurate for practical purposes.

Using these assumptions, we can identify any point on the Earth's surface using only two angles, called longitude and latitude. The first angle, called longitude, describes the position in a west/east direction. It ranges from -180 degrees to 180 degrees. By definition, negative numbers are east of the zero line of longitude and positive numbers are west of it. The second angle, called latitude, describes the position in the north/south direction and ranges from -90 to 90. Positive numbers are north of the equator and negative numbers are south of the equator. This can be seen in Figure 5.2.

Since R-trees work in Cartesian space and the surface of the sphere is not Cartesian, we need a mapping from the surface of the sphere to the Cartesian plane. A very popular mapping is the Mercator projection [20]. It is used, for example, by the Open Stree Map [1], which is the dataset I am using in this thesis. It works by projecting the surface of the Earth onto the surface of a cylinder of infinite height.

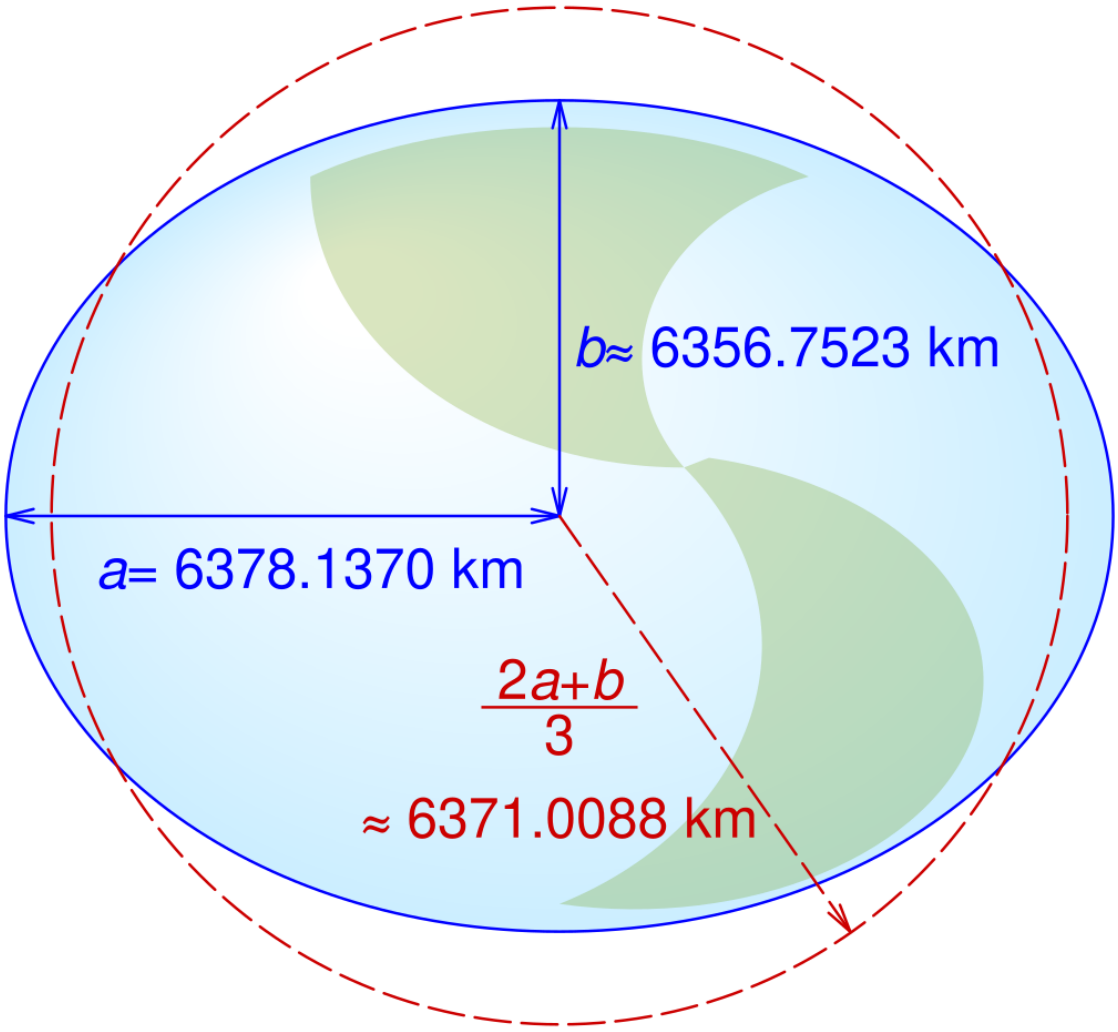


Figure 5.1.: This figure is an exaggerated illustration of the difference between the major and minor axis of the earth ellipsoid. Source: [18]

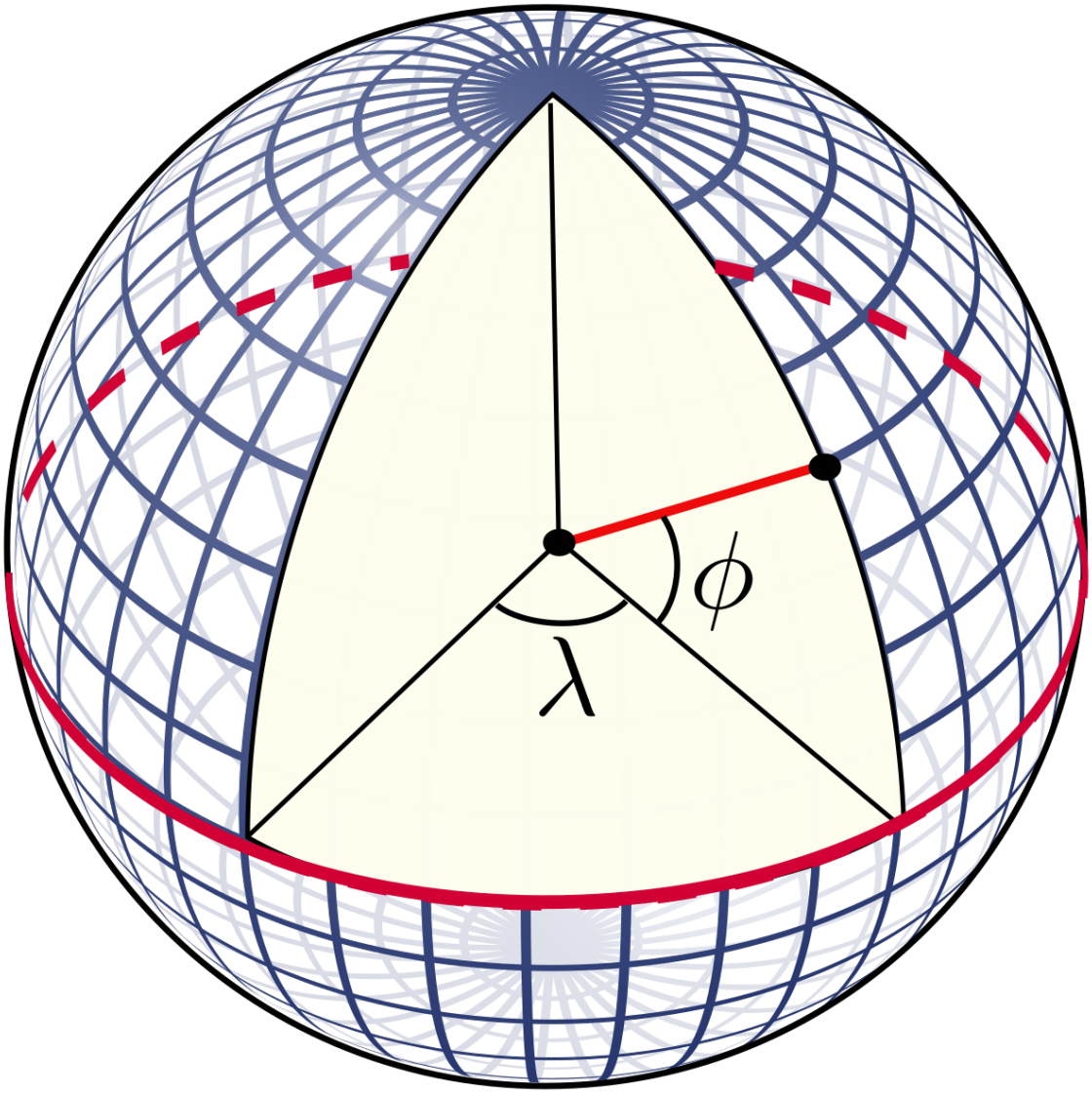


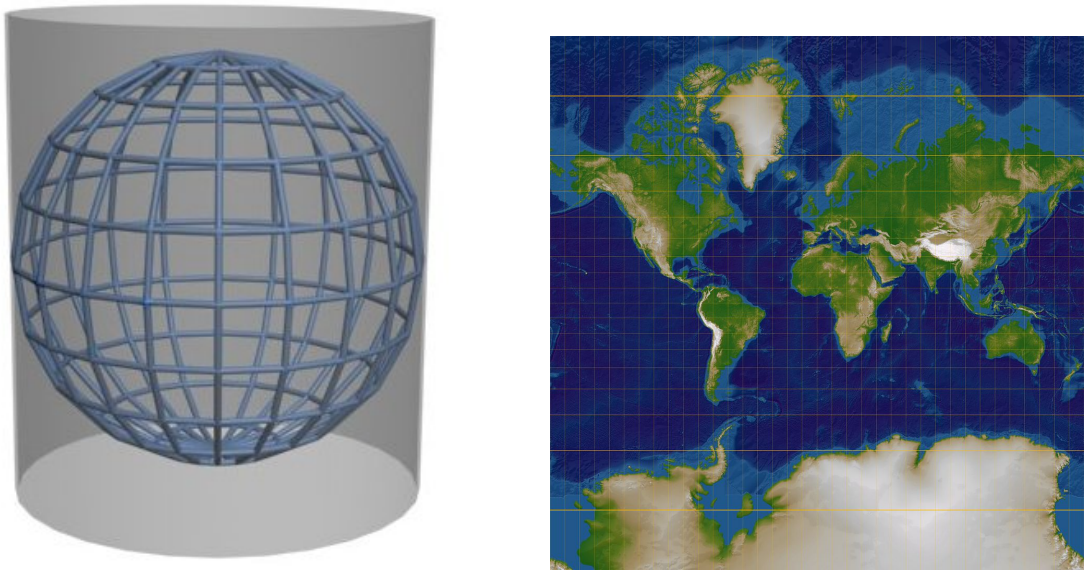
Figure 5.2.: This figure shows how the coordinate system of the earth is defined. λ is the angle of longitude and Φ is the angle of latitude. Source: [19]

Figure 5.3a shows what this looks like. The Earth would be inside the cylinder, and the equator and the surface of the cylinder would touch. To map a point from the Earth onto the cylinder, construct a line using the center of the Earth and the point you want to map. The intersection of this line with the surface of the cylinder is where that point is projected. After projecting each point onto the cylinder, the result would look like Figure 5.3b. The advantage of this mapping is that it preserves angles, which means that the length scale is roughly constant for small distances. The disadvantage is that distances, directions and areas are not preserved. The further away from the equator, the greater the distortion. The poles themselves can't be represented, because the line from the center of the earth through a pole doesn't intersect the surface of the cylinder (because of its infinite height). Just as an example of unequal distortion: In Figure 5.3b it appears that Greenland and Africa are the same size, even though Africa is 14 times larger. Because of these disadvantages, I chose a simpler mapping, basically the simplest mapping. It just takes the Earth's longitude and latitude coordinates and pretends they are Cartesian. To be formal, it maps longitude lines to horizontal lines and latitude lines to vertical lines, all of which have a constant distance from neighboring lines. This mapping is called equirectangular projection, and what it looks like for the Earth can be seen in Figure 5.4. Its advantage is that it's really easy to transform coordinates from one system to the other[1]. Its disadvantage is that the mapping does not preserve distance, area, or directions. Therefore, the disadvantages are the same as in the Mercator mapping, but because of the way easier and faster transformation I chose this mapping. In this mapping, the difference between two points that have the same latitude but are one degree apart in longitude can vary enormously. At the poles the difference is zero, and at the equator the distance between the lines is 111.2 km. Table 5.1 gives some examples of the distance between two points on the same latitude whose longitude lines are one degree apart. Because of the north-south symmetry of the latitude lines, it doesn't matter whether you are x° north or south. Also because of the symmetry, the exact longitude doesn't matter, only that they are one degree apart. Because of these huge differences, the algorithm has to compensate for them, which will be explained in chapter 7.

[1]Since nothing needs to be done

Table 5.1.: This table shows the difference between two points that have the same latitude and have a longitude difference of exactly one degree.

latitude coordinate	difference between the two points
0°	111.2 km
±15°	107.4 km
±30°	96.3 km
±45°	78.63 km
±60°	55.6 km
±75°	28.78 km
±90°	0



(a) Source: [21] **(b)** Source: [22]

Figure 5.3.: This figure shows the Mercator Projection. The construction is shown on the left. The Earth is inside a cylinder onto which the points are projected. The final result of this projection is shown on the right.

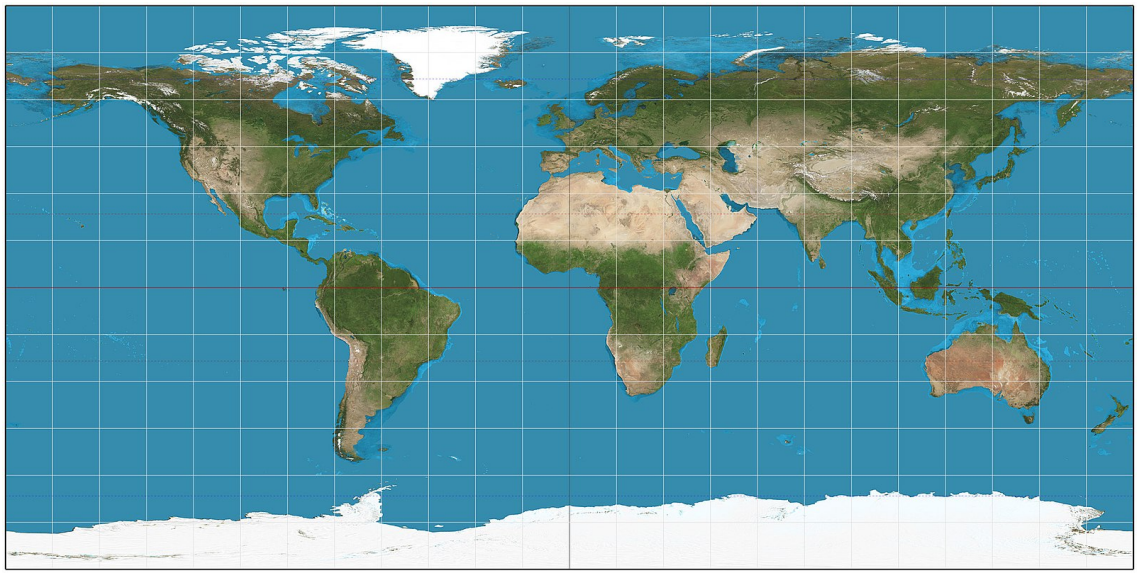


Figure 5.4.: This figure shows the equirectangular projection. The longitude lines are mapped to horizontal lines, and the latitude lines are mapped to vertical lines. Source: [23]

6. The SpatialJoin Infrastructure

This chapter contains the changes I made to QLever's infrastructure to make the SpatialJoin operation compatible with Qlever's current codebase. The basic properties of QLever have already been presented in section 2.3 and section 2.5. This chapter only discusses the changes that have been made. First, the changes in QueryPlanner are presented and then the SpatialJoin class is introduced. Since I did not make any changes in the IndexBuilder, these are the only changes that belong to the infrastructure. The SpatialJoin algorithm itself is not introduced here, but in the next chapter.

6.1. Changes in the QueryPlanner

For the SpatialJoin to be of any use, it must be used by the QueryPlanner. The QueryPlanner is the part of the program that takes the triples of the query and plans the strategy to get the output of the query in the fastest way. To achieve this goal, it uses dynamic programming, as explained in section 2.5.

This usual procedure is not compatible with the SpatialJoin class. The predicate "<max-dist-in-meters:1000>" (geometries that are at most 1000 meters apart) does not exist in the knowledge graph. Therefore, a SCAN, which would normally be the base case for each constraint in the SPARQL query, is not possible. Therefore, the triple must be filtered out. The special predicate "<max-distance-in-meters:xxx>" tells the QueryPlanner that this operation is not meant as a SCAN operation, but as a SpatialJoin operation, which is then created by the QueryPlanner. xxx is the number that gives information about the maximum distance that should be between the joined geometries. This parameter must be passed to the SpatialJoin class. After this is done, the triple containing "<max-dist-in-meters:xxx>" is removed from the query (so that no SCAN operation is performed). The SpatialJoin object we just created is added to the subplans. In its current form, the SpatialJoin doesn't have the input geometries yet, but it knows the names of the variables that contain the geometry information (since these variables are the subject and object of the triple with the special "<max-distance-in-meters:xxx>" predicate). When two subplans are joined, the SpatialJoin cannot rely on the default behavior. Instead of joining the SpatialJoin with the shared variable of a SCAN operation, the SCAN operation must be added as a child to the SpatialJoin. This gives the SpatialJoin the geometry input of that child. Once the SpatialJoin is merged with both variables (and

therefore has two children), it has all the information it needs to compute the result. From this point on, no further adjustments need to be made in the QueryPlanner. The result of the SpatialJoin operation can now be used with the standard behavior of the QueryPlanner, which is to create a JOIN operation when it is merged with another subplan, where the SpatialJoin and the other subplan are the children of the newly created JOIN operation. To ensure normal handling, the QueryPlanner calls the isConstructed method whenever one of the subplans is a SpatialJoin operation. Once both children have been added, the necessary information for the calculation is available and the construction of the SpatialJoin is complete. In this case, the method returns true and the special treatment mentioned above is skipped (otherwise the QueryPlanner would try to add a child to the SpatialJoin instead of performing a normal JOIN operation with the result of the SpatialJoin and another subresult of the query). More details about the isConstructed method and other methods of the SpatialJoin class that are also used during query planning are presented in the next section.

6.2. The SpatialJoin Class

In QLever, every operation inherits from the abstract Operation class. Because of this, there are many methods that need to be implemented. This ensures that the interaction with the different operations works correctly and that the operations have the same interface to the outside. The methods that need to be implemented and some more are described in this section. Note that I have excluded the NearestNeighbor SpatialJoin part, since this part was not written by me, but by another student. Therefore, the actual functions in the current repository may look slightly different, as they sometimes have a case distinction between the NearestNeighbor SpatialJoin and the MaxDist SpatialJoin.

- **selectAlgorithm:** This function accepts one of the available SpatialJoin algorithms. Currently these are the Baseline algorithm, the BoundingBox algorithm, and the S2 algorithm (which is used for the NearestNeighbors implementation). The S2 algorithm can't handle areas and the BoundingBox algorithm can't handle the NearestNeighbor search. This has to be considered when choosing the algorithm.
- **computeResult:** This function calls the SpatialJoin algorithm selected by the selectAlgorithm method to compute the result of the SpatialJoin operation, which is then returned.
- **resultSortedOn:** Some operations don't destroy an already existing sorting (or create a new one, like the SORT operation). Then the sorted column name(s) are returned. The baseline algorithm, which iterates over each row in the right subresult for each row in the left subresult and enters the pair of rows where $maxDist$ is greater than or equal to the actual distance, would preserve

the order of the left result table. However, since the Baseline algorithm exists only for evaluation purposes and the BoundingBox algorithm can't guarantee not to break a sorting, this function returns an empty list.

- **sizeEstimate:** The sizeEstimate function tries to estimate the size of the result of the SpatialJoin operation. Since it is not possible to know in advance how many geometries are within $maxDist$ meters of each other, we have to assume the worst case, which is the cross product. To allow better query planning for the average case, the size estimate is scaled by a factor of 0.001. This doesn't change the asymptotic behavior, only the absolute size estimate. When the SpatialJoin operation is in the process of QueryPlanning, where it does not yet have its children, it cannot return the cross-product size because it does not know the size or size estimate of its children. Therefore, it returns a dummy value of 1 in this case.
- **costEstimate:** This function estimates the cost of computing the SpatialJoin. Let n be the number of rows of the left child and m the number of rows of the right child. Then it returns $(n \cdot log(m)) + (m \cdot log(m))+$ the cost estimate of the children. For a detailed discussion of the cost of the SpatialJoin operation, see section 8.2.
- **knownEmptyResult:** This function returns whether it is known in advance that the result will be empty. For the SpatialJoin operation, this is the case if at least one of the subresults has a known empty result.
- **getMultiplicity:** This function takes a column as a parameter and returns an estimate of the multiplicity of that column in the result table. This information is needed in QueryPlanning and in future JOIN operations that take the result of the SpatialJoin as an input. The multiplicity allows you to estimate how large the cross-product effect mentioned in section 2.2 may be. If the children are not yet available, no estimate can be made and a dummy multiplicity of 1 is returned. If both children are available, the multiplicity can be calculated. In general, the multiplicity of a column times the number of distinct entries in that column equals the number of rows in that column. When calculating a SpatialJoin, no new entries are added to the result, only existing entries from the left and right subresults are merged and copied into the result. Therefore, the number of distinct items does not change. If the size changes, it is simply due to items being copied multiple times in the result (if a geometry is less than $maxDist$ meters away from several other geometries, it is copied once with each of the other geometries into the result, increasing the multiplicity but not the distinctness). Therefore, the new multiplicity estimate can be calculated by taking the sizeEstimate of the SpatialJoin operation and dividing it by the distinctness of the column with that variable before the SpatialJoin operation. There is a special case, for the column containing the distance between the two geometries. This column is not included in any input of the subresults and is assumed to have a distinctness of 1. This is because it

is highly unlikely that the distance between two geometries will be the same as the distance between two other geometries, even if the difference occurs only after many decimal places.

- **CacheKeyImpl:** The CacheKeyImpl function returns a string that can be used to uniquely identify this operation. This is necessary because the result of an operation is cached and should only be reused if the future (sub)query is exactly the same. Therefore, all parameters that affect the result of the SpatialJoin operation must be included in this string along with their values. These parameters are: the CacheKey of the children, the join variables, the maximum distance that geometry objects are allowed to have to each other, the information whether the distance between the geometries should be part of the result table and in general all variables of the result column, as well as the information whether the midpoint approximation is used for areas or not. Note that the selected algorithm is not included in the CacheKey, since the algorithm does not affect the result, only the time it takes to compute the result. If the SpatialJoin does not have its children yet, it will just return a dummy string of "incomplete SpatialJoin class", since not all the necessary information is available to know whether the result can be looked up in the cache or needs to be computed again. Using this key, no result will ever be stored in the cache and therefore the QueryPlanner will never replace an incomplete SpatialJoin class with a result from the cache. Once both children are added, the CacheKeyImple returns the correct cache key, and if the result has already been computed, it can simply be looked up instead of being recomputed.
- **getDescriptor:** This function returns a short description of the SpatialJoin operation that is displayed in the analysis ExecutionTree of the QLever UI. It consists of the name of the operation, the two variable names of the geometry and the maximum distance the geometries are allowed to have. An example can be seen in Figure 2.3, where this function returns: "MAX DIST JOIN ?wkt1 to ?wkt2 from 1000 meter(s)".
- **addChild:** This function adds a child to the SpatialJoin operation. Because the children of the SpatialJoin algorithm are not always known when the SpatialJoin is constructed, they must be added later. The reason the children are not always known is that the QueryPlanner first constructs the base cases, which are all SCAN operations and the special case for the SpatialJoin operation, which is the SpatialJoin operation with no children. Because it then merges all possible combinations using dynamic programming, sometimes the children are just the SCAN for the geometry and sometimes the children have already been merged with other operations before being merged with the SpatialJoin. When merging another operation with the SpatialJoin, the QueryPlanner would normally construct a JOIN operation, but if one of the operations is a SpatialJoin that does not yet have both children, the QueryPlanner instead calls the addChild function to add the child to the SpatialJoin operation. The addChild method returns a new SpatialJoin operation that has

the child as an input. The reason for creating a new SpatialJoin operation is that the QueryPlanner tries all possible combinations of merging the current subplans. This results in the SpatialJoin being merged with many different subplans. Therefore, the SpatialJoin object before the addChild method must remain in existence so that it can continue to be merged with other subplans, and the new SpatialJoin must also remain in existence because it will also be merged with many different subplans. This results in a lot of sub-plans that all have a different order of operations. In the end, the QueryPlanner chooses the cheapest plan.

- **isConstructed:** This function returns true or false depending on whether the SpatialJoin operation is constructed or not. The SpatialJoin operation is constructed, when both of its children have been added. Then the function returns true. This function is used in the QueryPlanner to let the QueryPlanner know if it needs to handle the special case of adding a child to the SpatialJoin operation instead of the usual JOIN operation, or if the SpatialJoin is already constructed and can be handled like any other operation with the usual construction of the JOIN operation.
- **VariableToColumnMap:** Next, I will present the changes made to the VariableToColumnMap function. The VariableToColumnMap function returns which variables are part of the result table and in which column they are located. For the SpatialJoin operation, this function has a different return value depending on the state of the SpatialJoin. Either it informs the QueryPlanner about the variables that provide the geometric information for the SpatialJoin, so that the QueryPlanner can merge it with subplans that provide this information, or it gives the QueryPlanner information about the appearance of the result table after the result has been computed. The state of the SpatialJoin depends on the number of children that have already been added. First, the VariableToColumnMap returns the two variables that need to be added because they provide the necessary information to the SpatialJoin. With this information, the QueryPlanner knows that it can only merge other sub-plans with the SpatialJoin if they provide one of the two required geometric information. The QueryPlanner then joins the SpatialJoin with another sub-plan that contains one of the required geometry variables. The other subplan is then added as a child (instead of the usual creation of a JOIN operation). After the first child is added, the VariableToColumnMap function returns only the variable name of the missing child. This allows the QueryPlanner to merge it with another subplan that contains a column with the necessary geometry information. Once the SpatialJoin has both children, the VariableToColumnMap method can finally return what the result table will look like when the SpatialJoin operation is computed.
- **getResultWidth:** This function has a different output depending on the number of children already added by the QueryPlanner. If the QueryPlanner has not yet added any children, the size of the result width is two, because

the VariableToColumnMap needs to tell the QueryPlanner to merge it with these two variables. As soon as one variable is merged, the getResultWidth method returns one, because the VariableToColumn Map now only tells the QueryPlanner about the missing variable, so that the QueryPlanner joins the SpatialJoin with another plan that contains the missing variable. Once the SpatialJoin has both its children, it returns the result width that the SpatialJoin operation will have after it is calculated. Its size will be the result width of the left child plus the size of the right child. If the distance between each pair is to be added to the result, the result width is increased by one. Unlike the usual JOIN operation, the result width is not the combined size of the children minus one. Usually one is subtracted because of the common variable used to join the child results. In this case, the "join" variables are the geometry information, which can be different for each element of the pair, since a pair is added if the geometries are less than $maxDist$ meters apart.

Finally, I want to mention how the information from the query is parsed by the SpatialJoin operation. Here I will explain a slightly outdated prototype that was implemented by me, as the newer version was implemented by another student. In the prototype, the aforementioned special predicate <mas-dist-in-meters:xxx> indicated to the QueryPlanner that it needed to treat this triple differently. The complete triple was something like ?leftGeometry <max-dist-in-meters:xxx> ?rightGeometry. This triple would be passed to the Construtor of the SpatialJoin class.There it would be parsed to get the value of $maxDist$ and the names of the left and right variables. With this information, it could then tell the QueryPlanner via the variable-to-column map which subplans it could join with. A disadvantage of this prototype is that it would be very confusing to add additional parameters to the special predicate, such as whether the distance between the geometries should be added to the result column, which algorithm should be used, or whether the midpoint approximation should be enabled or not. Because of these disadvantages, another student created a new method called SpatialService. Its syntax has already been shown in Figure 1.1. Using this service and qlever's auto-completion suggestions, it is now easy to change all the parameters of the SpatialJoin and choose the algorithm to use. Before the SpatialService, these parameters could only be changed in the source code and the project had to be recompiled after changing a parameter.

7. The SpatialJoin Algorithms

The result of the SpatialJoin operation can be computed using several algorithms. The goal of each algorithm is to find the pairs of lines where one line comes from the left input, the other line comes from the right input, and both lines represent geometric objects that are at most $maxDist$ meters apart. $maxDist$ is a parameter and can be chosen by the user. During this thesis two algorithms were programmed, the baseline algorithm and the BoundingBox algorithm. The baseline algorithm is very simple, but not very fast. The bounding box algorithm is more complex, but also much faster. This chapter explains how both algorithms work. The proof that the algorithms work correctly can be found in section 8.4. The algorithms in this chapter include the running times in a comment. However, this chapter will ignore the runtimes, as they are discussed in section 8.2. Note that the algorithms are not implemented exactly as shown here. Some methods don't exist and are only introduced for clarity. For example, there is no function getMaxDist to get the maximum distance, since this is done when parsing the query and passed to the algorithm in the constructor, but to avoid using unintroduced class variables in the algorithms, I added this method. Another example would be combining multiple rows into a single function call that describes what the rows do, such as computeLeftLongitudeBound. Also, code that is just for numerical stability is left out of the explanation or included in the summary method calls (but mentioned in the proof chapter).

7.1. The Baseline Algorithm

The baseline algorithm implements the trivial solution, which simply checks each line of the left imput against each line of the right imput and keeps the line pair if the distance is less than $maxDist$. The exact algorithm is shown in algorithm 6[1]. First, the algorithm reads the parameter $maxDist$ and gets the subresults needed to compute the result of the SpatialJoin operation. For more information about retrieving subresults, see chapter 6. The algorithm then begins a nested for loop.

[1]Note that I have excluded some code that is only needed for the NearestNeighborSearch, as the algorithm was extended by another student. The excluded code is only needed if you want to have only the x nearest neighbors, but there are more candidates. Then you have to filter the candidates. This is done with a priority queue. Since this is not needed for the maximum distance query, which returns all results that are within the range of the maximum distance, this part will not be explained here

The outer for loop iterates over the rows of the left subresult, and the inner for loop iterates over the rows of the right subresult. Then the distance between the left geometry and the right geometry is computed using the computeDistance function (for details on this function, see section 7.3). If the computed distance is less than the maximum allowed distance, the lines are added to the result. After all pairs of lines have been checked, the result is returned.

Algorithm 6 Baseline Algorithm()

```
maxDist = getMaxDist()
resultLeft = computeLeftSubResult()
resultRight = computeRightSubResult()
result = createEmptyResultTable()
for rowLeft in resultLeft do                ▷ O(n · m · (g1 + g2)) or O(n · m · g1 · g2)
    for rowRight in resultRight do              ▷ O(m · (g1 + g2)) or O(m · g1 · g2)
        d = computeDistance(rowLeft, rowRight)      ▷ O((g1 + g2)) or O(g1 · g2)
        if d < maxDist then
            result.addRow(rowLeft, rowRight)
        end if
    end for
end for
return result
```

7.2. The BoundingBox Algorithm

This section deals with the more complex BoundingBox algorithm. First, I will introduce the general idea of the BoundingBox algorithm. Then I'll quickly show why the naive bounding box doesn't work and how to compute a more complex one.

7.2.1. General Idea

The general idea of the BoundingBox algorithm, shown in algorithm 7, is to apply an easy and fast to compute filter to reduce the number of candidates to be checked by a large margin. Only then do the expensive distance calculation for the already filtered set. To achieve the filtering, the already mentioned R-trees (see chapter 4) are used. We add all points or areas of the smaller subresult to the R-tree. Then we iterate over the larger subresult. Here we take the point or area and compute a box that is large enough so that all 4 sides of the box are at least $maxDist$ meters away from the point or area (on the spherical geometry). Therefore, all results must be inside this box. Then we query the R-tree with this box. The R-tree returns all points contained in the query box. In most cases the set of these points is much smaller than the complete set of points. Then we only need to do the expensive distance

calculation for this much smaller subset. The details of the distance calculation are discussed in section 7.3. If the distance is less than $maxDist$ and the line pair has not yet been added to the result, the line pair will be added to the result. Later in this chapter, we will explain how duplicates can occur if this step is not done. Let us return to the example from chapter 1. In algorithm 1 we entered the query for all restaurants that are located in Freiburg, close to a tram station and close to our current position. Let's assume that the SpatialJoin operation between the tram stations and the restaurants is computed before the SpatialJoin operation between the restaurant and our current position[2]. The input to the SpatialJoin operation is all restaurants and tram stations in the dataset and the maximum distance of 3000 meters. The OSM Germany dataset contains about 10,000 tram stations and 100,000 restaurants (see the size of the IndexScan operations in the analysis of the ExecutionTree in Figure A.35). Since there are fewer tram stations than restaurants, the algorithm builds the rtree for the tram station. Then it iterates over all restaurants. For each restaurant, it computes the box that is guaranteed to be larger than 3000 meters in all directions (on the spherical geometry). Then it queries the R-tree with this box. If we assume that only cities with more than 100,000 inhabitants have tram stations, each of these cities has the same number of tram stations and the query box is very inefficient and always contains the complete city of the restaurant, then each query of the R-tree would return either 0 entries (if the city is below 100,000 inhabitants) or 127, because there are 79 cities with more than 100,000 inhabitants [24] and 10,000 tram stations divided by 79 cities is 127. Therefore, we would only need to calculate either zero or 127 distances for each restaurant instead of 10,000 for each restaurant. This is a saving of 100 % or 98.73 %, depending on whether the city is smaller or larger than 100,000 inhabitants. This shows the huge potential of the BoundingBox algorithm compared to the baseline algorithm, which would have to compute the distance for each of the 1,000,000,000 pairs.

7.2.2. Limitations of the trivial query box

Now that we have seen the huge potential of the BoundingBox algorithm, let us get into the details. This subsection will explain why the naive query box doesn't work. Let's say we have a point M from which we want to get all points that are at most $maxDist$ meters away. First, $maxDist$ has to be converted to degrees using

$$maxDistDegrees = maxDist \cdot \frac{circumferenceEarth}{360}$$

[2] In practice, the QueryPlanner would do it the other way around, because the current position consists of only one triple. Therefore, this operation is faster and produces a smaller result. Since the result is then used by other operations, they will also be faster because their input will be smaller if this operation is done first.

Algorithm 7 BoundingBox Algorithm()

```
maxDist = getMaxDist()
resultLeft = computeLeftSubResult()
resultRight = computeRightSubResult()
result = createEmptyResultTable()
smallRes = getSmallerResult(resultLeft, resultRight)
largeRes = getLargerResult(resultLeft, resultRight)
rtree = buildRtree(smallerRes)                          ▷ O((n + m) · log(n + m))
for entry in largeRes do                  ▷ O(m · n · (g1 + g2)) or O(m · n · g1 · g2)
    alreadyAdded = []
    queryBox = getQueryBox(getGeometry(entry))
    candidates = rtree.query(queryBox)                                  ▷ O(m + n)
    for c in candidates do          ▷ O((m + n) · (g1 + g2)) or O((m + n) · g1 · g2)
        d = computeDistance(entry.geometry, c)       ▷ O((g1 + g2)) or O(g1 · g2)
        if d < maxDist AND (rowLeft, rowRight) NOT IN alreadyAdded then
            result.addRow(rowLeft, rowRight)
            alreadyAdded.add((rowLeft, rowRight))
        end if
    end for
end for
return result
```

Then the naive query box would just take the following bounds:

- upper latitude bound: lat(M) + maxDistDegrees
- lower latitude bound: lat(M) - maxDistDegrees
- left longitude bound: lon(M) - maxDistDegrees
- right longitude bound: lon(M) + maxDistDegrees

For the upper and lower latitude limits, this would work because the lines of longitude (where longitude is constant and latitude varies) are so-called great circles (a great circle can be constructed by intersecting the surface of the sphere with a Cartesian plane containing the center of the sphere). Shortest paths are always on great circles, more on this in section 8.4). Since lines of latitude (lines of constant latitude where the longitude varies) are generally not great circles, this naive approach fails. Therefore, the query box fails for all latitudes except the equator (because the equator is a great circle). Figure 7.1 shows an example where the longitude bound fails. The figure contains the point M as well as a longitude line with points A, B, C and D on it. Point B has the same latitude as M. Notice that the shortest path from M to B does not stay on the latitude of M and B, but leaves it. Therefore, you can reach point B from point M with a shorter distance than the naive query box would compute. Even if the naive query box would compute the shortest path from M to B correctly (so $maxDist$ would be 1.27 in this example),

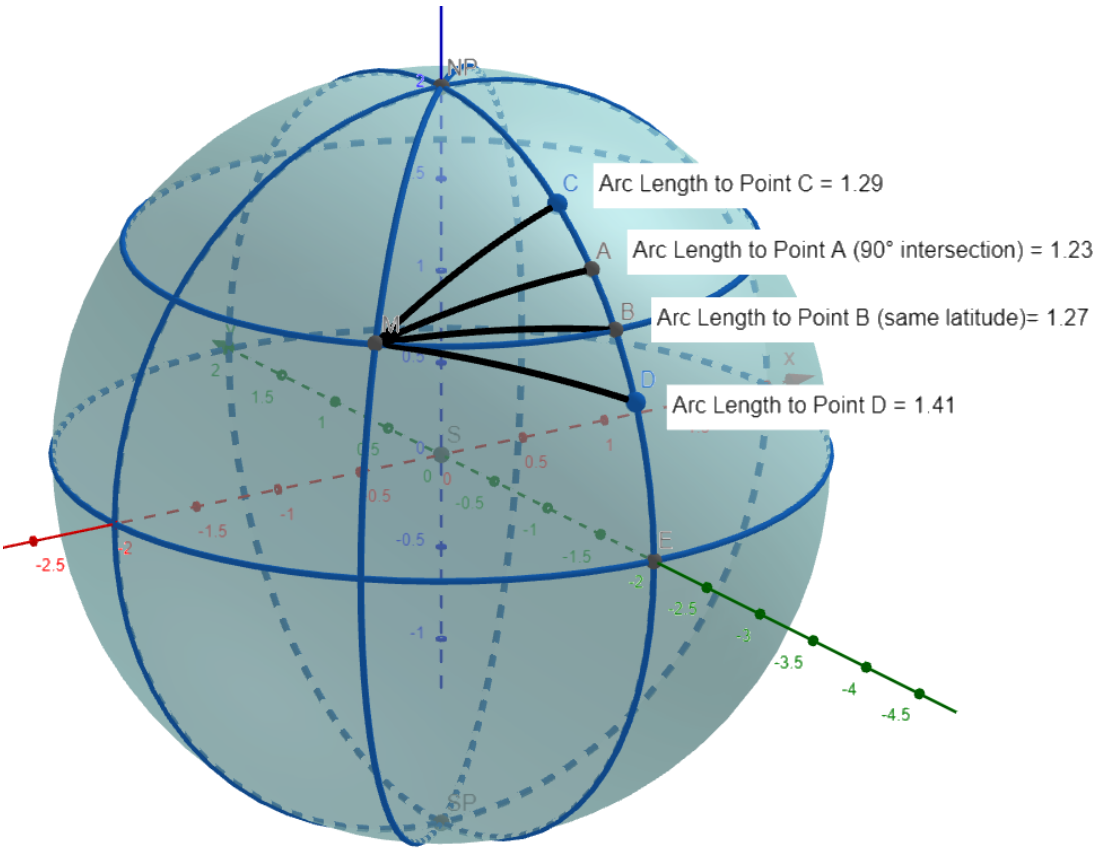


Figure 7.1.: This figure shows, different paths from M to the longitude line with four points on it. Note that the shortest path is not the path from M to B, which would have the same latitude as M. Source: Self made

the line of longitude with the four points on it can be reached with a shorter path. The path from M to A has a length of only 1.23. So you can reach a point outside the naive query box by going from M to A and then going a little further in that direction. Therefore, a different approach to computing the query box is needed, which is explained in the next subsection and proved in section 8.4.

7.2.3. Construction of the Query Box

Before I explain the concrete construction of the query box, I want to briefly explain the idea of how this approach can be extended to work not only for points, but also for areas. Suppose we have a geometry that is an area and we want to get all geometries (points or areas) that are closer than $maxDist$ meters away. Since the closest point from the area to other geometries is most likely not the same point for all other geometries, the construction of the query box must take this into account. First, the bounding box of the area is calculated (i.e., the smallest possible axis-

aligned box that completely encloses the area). Then the center of the bounding box is calculated. After that an upper bound is calculated for the distance from the center to any point in the bounding box. This upper bound is added to *maxDist*. Now we can compute the query box by pretending that the area is the center of the bounding box and *maxDist* is increased by the upper bound for the distance from the center to any point in the bounding box. The proof that this procedure works is contained in section 8.4.

Now that we have the general idea of how the query box handles areas, we can take a look at the implementation. Algorithm 8 shows the starting point of the query box calculation. If the geometry for which we want to get the query box is a point (represented as a GeoPoint), then we simply call another method called computeQueryBox, which gets the point geometry and zero as an argument. Zero is the upper bound from the center of the geometry to its bounding box. Since points don't need a bounding box, this distance is zero and *maxDist* doesn't need to be increased for this case. If the geometry is not a point, but an area, we apply the idea presented above. We calculate the bounding box, the center of the bounding box, and the upper bound for the distance from the center to any point inside the bounding box. Then we call the same method as in the point case, but we don't just give it the center of the area, but also the upper bound. Then the computeQueryBox function can take into account that we are dealing with an area by increasing *maxDist* by the upper bound.

Now that we have dealt with the areas by reducing them to their midpoints, we only need to compute the query box for the points (and have the option of adding the upper bound to *maxDist*). The calculation of the query box can be seen in algorithm 9. The first step is to increase *maxDist* by the bound to compensate for areas that are approximated by their center. For points, this does not change *maxDist* since this bound is just zero in this case. Then the function checks if the maximum distance is greater than half the circumference of the earth. If this is the case, then all points on the earth could be reached and the query box would have to cover the whole globe. In this case, we simply return the box containing all the points. A box is represented by its lower left point and its upper right point. If the value of *maxDist* is greater than 0.4 times the circumference of the earth, then we call an optimization for large query boxes, which will be explained later. If none of the above cases occurs, the default calculation starts. First, the upper and lower latitude bounds are calculated. If one of the poles is reached (so the upper bound is 90 or the lower bound is -90), we don't need to compute the longitude bounds, because any longitude can be reached (by first going to the pole that is in the query box, and then any longitude line can be reached by going an arbitrary small amount to the longitude line). Therefore, we return the query box containing all the longitude lines and the upper and lower latitude bounds just computed. If no pole can be reached, we have to calculate the longitude bounds as well. If the longitude bounds "cross" the -180 or 180 longitude line, we have an edge case. On the sphere we have only one query box, but on the equirectangular mapping we

can't represent the one box from the sphere with only one box. So we need to return two boxes. One box that contains everything on one side of the -180 or 180 degree longitude line, and one that contains everything on the other side. This can be seen in Figure 7.2. The red query box was split into two query boxes because the Cartesian space can't handle the "wrap around" of the Earth. If the query box does not cross the -180 or 180 line, we can just return the box with our 4 calculated bounds. So far, I have only mentioned that the bounds are calculated, but have not shown how this is done. The formulas will be explained and proven in section 8.4, and are therefore presented here without further explanation. Let u be the point from which we want to measure the distance to every other point. Let a be the distance that a point may have to u. Then all points to be checked are those in the following query box:

- **upper bound**: min{ 90, $\text{lat}(u) + a \cdot \frac{360}{\text{circumference}}$ }
- **lower bound**: max{ -90, $\text{lat}(u) - a \cdot \frac{360}{\text{circumference}}$ }
- **left and right bound**:
  Case 1(upper bound is 90 or lower bound is -90): all longitude lines need to be checked (left bound is -180, right bound is 180)
  Case 2: the left (use negative sign) and right (use positive sign) bounds are:

$$\text{lon}(u) \pm + \cos^{-1}\left(\frac{\cos(\frac{a}{r}) - \frac{\cos^2((90-|\text{lat}(u)|)\cdot\frac{2\pi}{360})}{\cos(\frac{a}{r})}}{\sin((90-|\text{lat}(u)|)\cdot\frac{2\pi}{360})\cdot\sin(\cos^{-1}(\frac{\cos((90-|\text{lat}(u)|)\cdot\frac{2\pi}{360})}{\cos(\frac{a}{r})}))}\right)\cdot\frac{360}{2\pi}$$

  Here r is the radius of the earth.

The correctness and the formulas themselves are proven and explained in Theorem 7. Also note that the longitudes from the formulas above may need to be subtracted or added by 360 to bring them into the range from -180 to 180 (see the note on normalization at the end of the proof).

Algorithm 8 getQueryBox(geometry)

```
if geometry.isGeoPoint() then
    return computeQueryBox(geometry.Point, 0)
else
    box = geometry.getBoundingBox()                    ▷ O(g)
    midpoint = calculateMidpointOfBox(box)
    bound = getUpperBoundMidpointToPointInBox(box)
    return computeQueryBox(midpoint, bound)
end if
```

If *maxDist* is large, but not large enough to reach every point on earth, an optimization can be done. The idea behind this optimization is as follows: When a pole is reached, all longitude lines must be checked. For *maxDist* values above

Algorithm 9 computeQueryBox(startPoint, boundingBoxBound)

```
maxDistQueryBox = getMaxDist() + boundingBoxBound
if maxDistQueryBox > 0.5 * circumference_earth then
    lowerLeftPoint = (-180, -90)
    upperRightPoint = (180, 90)
    return Box(lowerLeftPoint, upperRightPoint)
else if maxDistQueryBox > 0.25 * circumference_earth then
    return computeBoxForLargeMaxDistances(startPoint)
else
    upperLatBound = computeUpperLatBound(startPoint, maxDistQueryBox)
    lowerLatBound = computeLowerLatBound(startPoint, maxDistQueryBox)
    if isAnyPoleReached(upperLatBound, lowerLatBound) then
        lowerLeftPoint = (-180, lowerLatBound)
        upperLeftPoint = (180, upperLatBound)
        return Box(lowerLeftPoint, upperLeftPoint)
    else
        leftLonBound = computeLeftLonBound(startPoint, maxDistQueryBox)
        rightLonBound = computeRightLonBound(startPoint, maxDistQuery-
Box)
        if leftLonBound < -180 then
            lowerLeftPoint1 = (-180, lowerLatBound)
            upperRightPoint1 = (rightLonBound, upperLatBound)
            Box1 = Box(lowerLeftPoint1, upperRightPoint1)
            lowerLeftPoint2 = (leftLonBound + 360, lowerLatBound)
            upperRightPoint2 = (180, upperLatBound)
            Box2 = Box(lowerLeftPoint2, upperRightPoint2)
            return List(Box1, Box2)
        else if rightLonBound > 180 then
            lowerLeftPoint1 = (leftLonBound, lowerLatBound)
            upperRightPoint1 = (180, upperLatBound)
            Box1 = Box(lowerLeftPoint1, upperRightPoint1)
            lowerLeftPoint2 = (-180, lowerLatBound)
            upperRightPoint2 = (rightLonBound - 360, upperLatBound)
            Box2 = Box(lowerLeftPoint2, upperRightPoint2)
            return List(Box1, Box2)
        else
            lowerLeftPoint = (leftLonBound, lowerLatBound)
            upperRightPoint = (rightLonBound, upperLatBound)
            return Box(lowerLeftPoint, upperRightPoint)
        end if
    end if
end if
```

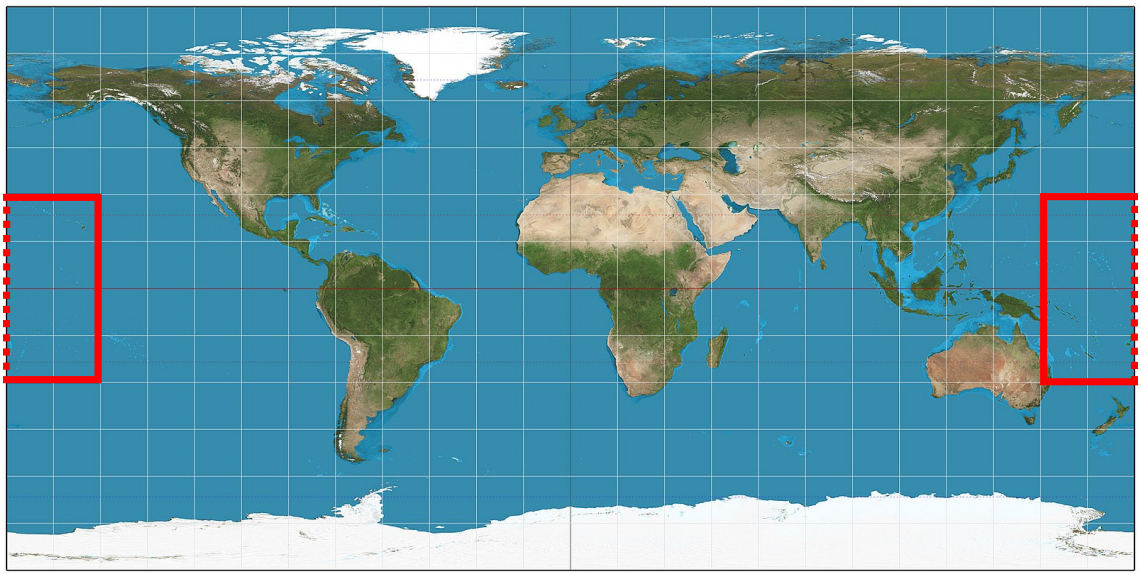


Figure 7.2.: This figure shows, how the case gets handled, when the query box crosses the -180 or 180 line: The query box gets split in two query boxes. Source: Adapted from [23]

0.25 times the circumference of the earth, a pole can always be reached. To avoid checking all longitudes all the time, this optimization was developed. The idea is to take the point that is antipodal[3] to the point for which the query box is computed. Then we compute an anti query box containing only points that are guaranteed to be further than *maxDist* meters apart. Next, we compute the distance for each geometry, except for the geometries in the anti query box. To avoid implementing a distinction between query box and anti query box, the anti query box is converted to a set of query boxes that cover everything except the anti query box. This can be seen in Figure 7.3. The anti query box is drawn in red and the converted query boxes are drawn and shaded in green. Earlier in this chapter it was mentioned that we need to check for duplicates. This can happen here (or in the case where the -180 or 180 longitude line is crossed by the query box) when an area is contained in multiple query boxes. For example, in Figure 7.3 the area could be contained in the top green box and the middle left green box.

Now that the general idea of large query box optimization is introduced, we can take a look at algorithm 10 that implements it. First the antipodal point is computed. Then the anti distance is calculated. The anti distance is the distance any point can be away from the antipodal point and still be further away than *maxDist* meters from the start point. The anti distance is then converted to degrees. Then the computeSingleBox function is called. This function basically does the same thing

[3] an antipodal point is exactly on the opposite side of the sphere. The most famous examples of antipodal points are the North and South Poles. To avoid always writing antipodal point, antipodal query box, and so on, this will be abbreviated as anti

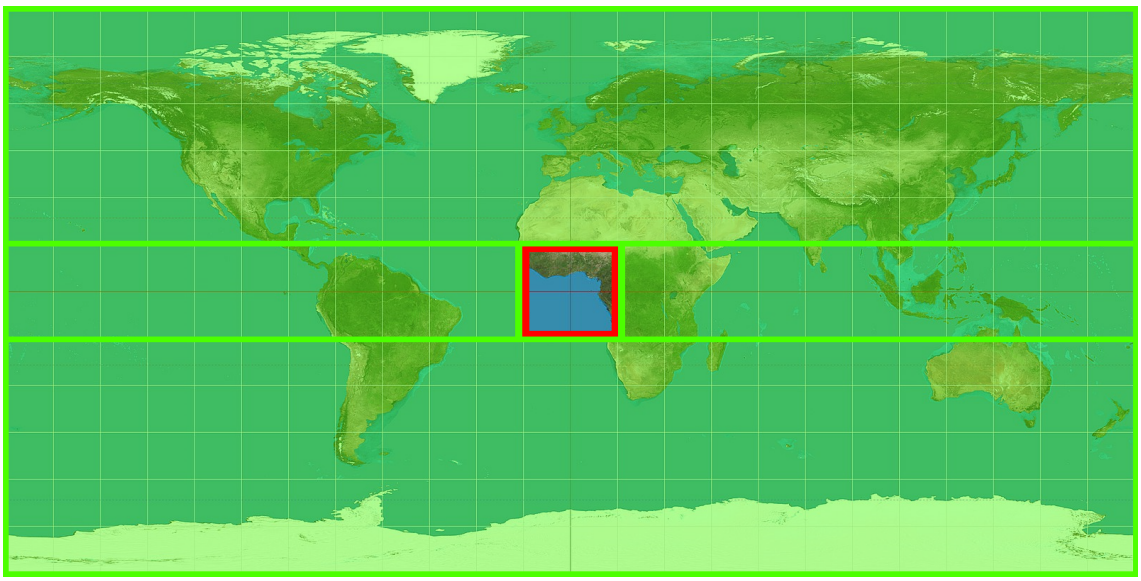


Figure 7.3.: This figure shows how the anti query box, drawn in red, gets converted to the four querie boxes, drawn and shaded in green. Source: Adapted from [23]

as computeQueryBox, with one difference. It makes sure that it stays small enough to not accidentally contain points that are *maxDist* meters away from the start point. This can be done by guaranteeing that each point in the anti bounding box is at most anti dist meters away from the anti point. If some points outside the anti query box are also less than antiDist meters away from the anti query box, it doesn't matter. This is the opposite of compute query box, which wants to make sure it's large enough to guarantee that all points that are less than *maxDist* meters away are contained in the query box. So the methods are basically the same, but one method generously rounds up, while the other generously rounds down. This is especially important when a pole is within reach of the antipoint. In this case, the rounding down stops at the pole and does not include all longitudes, as the query box would. This makes the anti query box smaller than it could be, but guarantees that all points inside are more than *maxDist* meters away from the start point. The correctness of this optimization is proved in section 8.4.

Algorithm 10 computeBoxForLargeMaxDistances(startPoint)

```
antiPoint = getAntipodalPoint(startPoint)
antiDist = circumference_earth / 2 - getMaxDist()
distDegrees = (360 / circumferenceEarth) * (antiDist / 2 )
antiBox = computeSingleBox(antiPoint, distDegrees)
return createBoxesAroundAntiBox(antiBox)
```

7.3. Distance computation

This section explains the calculation of the distance between two geometries, which is implemented in algorithm 11. First, the computeDistance function checks if the midpoint approximation is activated. If this is the case, both geometries are first converted to a point. If the geometry is already a point, the conversion function simply returns the point. Otherwise, it calculates the midpoint of the area and returns it. Then the calculateSphericalDistance function is called, which calculates the distance between the two points. Since the calculateSphericalDistance function already exists in QLever, it will not be discussed further. If the midpoint approximation is not activated, the algorithm calculates the closest points on the equirectangular mapping. It then uses these two closest points to compute the exact distance of those two points on the spherical geometry. Because the closest points are calculated on the equirectangular mapping (the mapping where nothing needs to be done because we just pretend the spherical coordinates are Cartesian coordinates), the two closest points may not actually be the two closest points. An example where this is not the case is when both areas are close to the -180 and 180 degree longitude line, but on different sides of it. Then the equirectangular map will map both geometries to different sides of the map. One will be completely on the left side and the other will be completely on the right side. The closest points on this mapping are the points that are actually farthest apart. Since the distance between the areas is then calculated using the spherical distance calculation, the calculated distance does not go around the earth to the other area, but the error is simply that the shortest path from the calculated points is taken. Since the points are on the wrong side of the geometry, the distance can be off by the sum of the diameters of both areas.

Algorithm 11 computeDistance(geometry1, geometry2)

```
if midPointApproximation is activated then
    geo1 = convertToPoint(geometry1)                                  ▷ O(g1)
    geo2 = convertToPoint(geometry2)                                  ▷ O(g2)
    return calculateSphericalDistance(geo1, geo2)
else
    p1, p2 = calculateEuclideanClosestPoints(geometry1, geometry2)    ▷ O(g1 · g2)
    return calculateSphericalDistance(p1, p2)
end if
```

7.4. Code quality

The final section of this chapter describes the measures taken to ensure high code quality. First, the quality of the source code is discussed, and then the quality of the algorithm itself.

The most important step in achieving high quality code was code review. Before the code could be merged into the main repository, a pull request had to be created. This ensured that every line of code was seen by at least two people. The code reviewer sees the code from a different perspective than the programmer, and often finds improvements in style, organization, and optimization of the code. This is especially true if the reviewer has more experience with the software project and programming in general. To ensure good code quality before the code review, some additional measures were taken. First, static code analysis was performed using the Sonarqube tool. Static code analysis helped to find bugs in the code, such as unsafe pointer access, improve code style by enforcing consistent naming rules, and prevent overly complex, long, or nested functions. In addition, a format checker ensured that the look and feel of the code was consistent and easy to read, for example, by enforcing a maximum line length and ensuring that the parameters of a function were grouped in an easy-to-read manner.

The most important measure to ensure a high quality implementation of the algorithm is testing. Therefore, many tests have been written to ensure that the SpatialJoin infrastructure and the SpatialJoin algorithm work properly. The goal of the tests was to cover every possible case, especially the edge cases where bugs are most likely to occur. To avoid forgetting to test some lines of code, an automated test coverage tool checked which lines were covered or only partially covered by tests. In total, the SpatialJoin operation is covered by 1642 tests. To give some examples of such tests: A test is performed on a small dataset containing areas and points where the distance of the objects is known. Then the spatialJoin algorithm is called with different maximum distance values and the result is checked to see if exactly the pairs that meet the maximum distance constraint are contained in the result. Another example is the query box. To ensure that the query box works properly, several checks are performed. First, over each boundary of the query box. The distance to many points just a tiny bit outside the query box was calculated and asserted to be greater than the maximum distance allowed. To ensure correctness not only at the boundary of the query box, but also on a global scale, a grid test was performed with points all over the globe. If the point was not contained in the query box, it was asserted that the distance to the start point was greater than $maxDist$. To make sure that the query box works for all start points, including points on the -180 and 180 degree longitude lines and the North and South Poles, the above tests were done with query boxes with many different start points, including all edge cases.

8. Theoretical Analysis

In this section the theoretical analysis of the presented algorithm will be described. First, the problem is formally defined. Then the runtime complexity and the space complexity will be analyzed and in the last part of this chapter the correctness of the algorithm will be proved.

8.1. Formal problem definition

Let D be a data set of triples and E be the set of all entities in D. Then D contains data in the form of triples, where the triples consist of three entities:

$$\{(s,\ p,\ o) \mid s,\ p,\ o \in E\}$$

In order to query it, entities as well as variables can be used. Let all variables be contained in the set of variables V. A query is then a set of tuples, where each element of the tuple can be an entity or a variable:

$$\{(s,\ p,\ o) \mid s,\ p,\ o \in E \cup V\}$$

The SPARQL query from algorithm 5 is for example the following set:

$$\{(v_1,\ e_1,\ v_2), (v_1,\ e_2,\ v_3) \mid v_i \in V,\ e_k \in E,\ i \in \{1,\ 2,\ 3\},\ k \in \{1,\ 2\}\}$$

where e_1 is the entity "<located-in>" and e_2 is the entity "<is-a>".

The result of the SPARQL query is a mapping m from the set of variables to a tuple, containing elements from the set of entities. The tuple represents the column of that variable in the result table. It maps each variable from the query to a tuple of entities. The length of the tuple k is the number of rows in the result:

$$m : V \to (r_1,\ r_2,\ \ldots\ ,\ r_k) \mid r_i \in E,\ k,\ i \in \mathbb{N}$$

The concrete mapping of the query in algorithm 5, whose result table can be seen in Table 2.4, would look like this:

$v_1 \mapsto (r_1,\ r_2,\ r_2,\ r_3,\ r_3,\ r_3,\ r_3)$
$v_2 \mapsto (r_4,\ r_4,\ r_4,\ r_4,\ r_4,\ r_5,\ r_5)$
$v_3 \mapsto (r_6,\ r_7,\ r_8,\ r_9,\ r_8,\ r_9,\ r_8)$

where $r_i \in E$ are the following entries:

- r_1: Uni Freiburg
- r_2: University Library Freiburg
- r_3: Minster of Freiburg
- r_4: Freiburg im Breisgau
- r_5: Altstadt of Freiburg
- r_6: university
- r_7: library
- r_8: building
- r_9: church

With the basics above, we can now start to define the formal problem definition for the SpatialJoin algorithm. The input to the SpatialJoin algorithm is two (sub)queries, both of which must contain at least one tuple of the following type:

$$(b_1,\ \text{"<asWKT>"},\ b_2) \text{ with } b_i \in E \cup V,\ i \in \{1,\ 2\}$$

The variable b_2 should then be a WKT representation of a geometry (if it is not a WKT representation and therefore it is not possible to parse the geometry, the row is skipped and is not part of the result). Let l be the mapping of the left (sub)result and r the mapping of the right (sub)result. Let v_l be the variables of the left (sub)result and v_r be the variables of the right (sub)result. Let N_l be the set containing the row numbers of the left (sub)result and N_r be the set containing the row numbers of the right (sub)result. The result of the SpatialJoin algorithm is a mapping s with the following property:

$$s:\ \{v_g,\ v_h \mid v_g \in v_l,\ v_h \in v_r\} \rightarrow \{(l(v_g)_a,\ l(v_g)_b, ...) \text{ with } a,\ b \in N_l,\ (r(v_h)_x,\ r(v_h)_y, ...) \text{ with } x,\ y \in N_r\}$$

To better explain the mapping s, let's consider the variable w_1, which is contained in the domain of l and s. Every element in the tuple of $s(w_1)$ is also contained somewhere in the tuple of $l(w_1)$. It may be contained multiple times in $l(w_1)$ and $s(w_1)$, and it may be contained more often in $s(w_1)$ than in $l(w_1)$ (because of the cross-product effect). There may also be some elements in $l(w_1)$ that are not contained in $s(w_1)$, but there will be no element in $s(w_1)$ that is not contained in $l(w_1)$.

Whether an element of $l(w_1)$ is contained in $s(w_1)$ depends on whether the SpatialJoin condition is satisfied or not. Let e be the tuple containing the geometries from the left input and let f be the tuple containing the geometries from the right input. Let d be the distance function, which takes two geometries as input and maps them to the distance these two geometries have to each other on a perfect sphere, where the radius of this perfect sphere is equal to the radius of the earth. Let m be the maximum distance that any pair of geometries may have to each other in order to be included in the result. Then the condition for the SpatialJoin:

$$d(e_l, f_r) < m \text{ with } e_l \in e,\ f_r \in f$$

Lets consider the case, where $d(e_l, f_r) < m$ is true. Let a be the index of e_l in the tuple e and b be the index of f_r in the tuple f. Then we have the following equalities:

$s(v_g)_x = l(v_g)_a \quad \forall v_g \in v_l$
$s(v_h)_x = r(v_h)_b \quad \forall v_h \in v_r$

The two equations above summarize that the rows of the left and right (sub)result are joined in a common row x if their distance is less than m. This is the result of the SpatialJoin operation.

8.2. Runtime complexity analysis

The runtimes of the algorithms and their statements have been added to the algorithms in chapter 7. If a line does not contain a comment, then the runtime for the statement in that line is an element of O(1). Comments next to for loops indicate the runtime of the entire for loop, not just the line.

Before analyzing the runtimes, I want to introduce some variables that affect the runtime:

- Let n be the number of rows in the left input
- Let m be the number of rows in the right input
- Let g be the number of points contained in the geometry. If the geometry is a point, then g is one, but if the geometry is a polygon, then g can have any number of points.

ComputeDistance: The runtime of the computeDistance function, shown in algorithm 11, is $O(g_1 + g_2)$ when the midpoint approximation is enabled. The reason for this is that every point in the geometry must be considered for the conversion to a point. The calculation of the distance between the two points is done in constant time. However, if the midpoint approximation is not activated, the runtime is

$O(g_1 \cdot g_2)$, because for every point of one geometry, every point of the other geometry has to be considered.

BaselineAlgorithm: The runtime of the baseline algorithm, shown in algorithm 6, depends on whether the midpoint approximation is used. If the midpoint approximation is used, the runtime is an element of $O(n \cdot m \cdot (g_1 + g_2))$. This is because the algorithm iterates over each row of the left input. For each of these rows, it iterates over the right input, and for each of these rows, the distance computation requires a runtime of $O((g_1 + g_2))$. When the midpoint approximation is not used, only the runtime of the distance function changes. This changes the total runtime of the algorithm to $O(n \cdot m \cdot g_1 \cdot g_2)$.

ComputeBoxForLargeMaxDistances: This function only performs operations that take a constant amount of time.

ComputeQueryBox: This function only performs operations that take a constant amount of time.

GetQueryBox: The runtime of this function is an element of $O(g)$, because the bounding box has to be computed, which has to consider every point of the geometry. The calculation of the center and the bounding box consists of only a few basic operations with constant runtime, and the function computeQueryBox is also constant.

BoundingBox algorithm: For the runtime analysis of the BoundingBox algorithm shown in algorithm 7, I will start with the innermost if statement. Adding a row to the result table has a constant runtime (at least amortized) and adding an entry to a set is also constant. The computation of the if statement is constant, as is a comparison and a set lookup. Computing the distance between two entries has a runtime of $O((g_1 + g_2))$ or $O(g_1 \cdot g_2)$, depending on whether the midpoint approximation is used or not. This part of the BoundingBox algorithm is executed for each candidate of the R-tree. In the worst case (if $maxDist$ is large enough to reach all points in the R-tree), all elements of the R-tree (the smaller (sub)result) can be candidates. The smaller size of the sub(results) is in $O(m + n)$. So the runtime so far is $O((m + n) \cdot (g_1 + g_2))$ or $O((m + n) \cdot g_1 \cdot g_2)$. The first statement before the for loop is to query the R-tree, which takes at most $O(m + n)$ time (if all entries of the R-tree are contained in the query box. The runtime is usually $O(\log(m + n)))$. Adding $O(m + n)$ to the already analyzed runtime does not change anything, since the runtime is already greater than $O(n + m)$. Creating a query box has a constant runtime, as does creating an empty set. Therefore, the runtime of the already analyzed part is $O((m + n) \cdot (g_1 + g_2))$ or $O((m + n) \cdot g_1 \cdot g_2)$. All of the previously analyzed parts of the BoundingBox algorithm are done for each row of the larger input. The larger part of the input is an element of $O(m + n)$. Therefore, the total runtime for everything done so far is $O((m + n)^2 \cdot (g_1 + g_2))$ or $O((m + n)^2 \cdot g_1 \cdot g_2)$. In the part of $(m + n) \cdot (m + n)$, $(m + n)$ is meant once for the larger (sub)result and once for the smaller. Therefore, this part can be shortened to $m \cdot n$, so that the total runtime so far is an element of $O(m \cdot n \cdot (g_1 + g_2))$ or $O(m \cdot n \cdot g_1 \cdot g_2)$. The last

nonconstant statement is to build the R-tree, which takes $O((n+m) \cdot \log(n+m))$, where $(n+m)$ stands for the size of the smaller subresult. Since it stands for the smaller size, the runtime is also en element of $O(n \cdot m)$. Since the runtime for building the R-tree has to be added to the runtime of everything else, it doesn't change the total runtime, which is already larger. To sum up the runtime for the BoundingBox algorithm, the runtime is in $O(m \cdot n \cdot (g_1 + g_2))$ when the midpoint approximation is used, and in $O(m \cdot n \cdot g_1 \cdot g_2)$ when the midpoint approximation is not used. This is the same runtime as the baseline algorithm. However, the runtime is only the same if the *maxDist* value is large enough to cover the entire data set. The usual runtime, where querying the R-tree takes only O(log(n+m)), reduces the runtime of the bounding box algorithm, making it faster than the baseline algorithm. For a practical analysis that also measures the constant factors that are ignored by the O-notation, see chapter 9. Note also that the geometries g_1 and g_2 are usually not large, and for practical purposes they can be considered a constant factor. Since the QLever project wants to estimate the most likely runtime and not the absolute worst case, the geometries g_1 and g_2 are seen as constant and the query of the R-tree is estimated as $\log(m)$.

8.3. Space complexity analysis

computeDistance: The (temporary) space used during the execution of the method is an element of $O(g_1 + g_2)$, since each geometry has to be stored. This is true both when midpoint approximation is enabled and when it isn't.

Baseline algorithm: The space complexity for the baseline algorithm is an element of $O(n \cdot m \cdot (g_1 + g_2))$. This worst-case space consumption occurs if the value of *maxDist* is large enough so that the distance between each pair of geometries is less than *maxDist*. If that's the case, then each row of the left input is merged with each row of the right input and inserted into the result table, which takes the mentioned amount of space. In addition, it needs to store the geometries for both (sub)results, which adds the $(g_1 + g_2)$ part to the total space consumption.

ComputeBoxForLargeMaxDistances: This method uses a constant amount of space and does not need to store input size dependent information.

ComputeQueryBox: This method uses a constant amount of space and does not need to store input size dependent information.

GetQueryBox: This function has a space requirement of $O(g)$ because the complete geometry must be present when the bounding box is calculated.

BoundingBox algorithm: As with the baseline algorithm, the worst-case space consumption is an element of $O(m \cdot n \cdot (g_1 + g_2))$, since the algorithm must add $m \cdot n$ entries to the result table if all pairs of geometries have a distance to each other less than *maxDist*. In addition, it needs to store the geometries for both (sub)results, which adds the $(g_1 + g_2)$ part to the total space consumption.

8.4. Proofs of some properties

This section proves the correctness of the algorithm and the calculations it performs. It begins by introducing the notation used in this section. Then important properties of spherical geometry are introduced and the proofs of correctness are presented.

8.4.1. Notation and Conventions

This subsection covers the notations and conventions used in the following subsections for the theoretical part, especially for the proofs. They are presented in the following list:

- **Points**: Points on a sphere are given as lowercase letters, such as u, v, or w. The center of the sphere is called M.
- **Subtended angles**: A subtended angle is defined as follows: Let M be the center of the sphere and u and v be points on the surface of the sphere. Then the angle between the (Euclidean) lines $\overline{Mv}$ and $\overline{Mu}$ is called the subtended angle. The line from u to v on the surface of the sphere has length $subtended\_angle \cdot radius$. To distinguish subtended angles from other angles, subtended angles have a tilde over them, like $\widetilde{\alpha}$, $\widetilde{\beta}$, or $\widetilde{\gamma}$.
- **Lines on the surface of a sphere**: Unless otherwise noted, a line connecting two points on the surface of a sphere is the smaller fragment of a great circle (a definition of a great circle can be found in subsection 8.4.2) containing those two points. This is also the shortest possible path between these points. The line can be identified by a subtended angle or by the length of the circle fragment. If the length is given separately, rather than as a subtended angle, it will be a lowercase letter, such as a, b, or c. Note that a is the distance that corresponds to the subtended angle $\widetilde{\alpha}$, b corresponds to $\widetilde{\beta}$, and c corresponds to $\widetilde{\gamma}$. However, this definition is not unique. If the points are exactly on opposite sides of the sphere (for example, the north and south poles), then there are infinitely many shortest paths. In this case it will be specified which of them is meant.
- **Angles**: Angles are given in radians and written with a Greek letter such as α, β, or γ. Angles on the surface of the sphere are defined as follows: Let a and b be two lines on the surface of our sphere that start and intersect at point u. Let P be the plane containing point u and let P be tangent to the surface of the sphere. Project the first ϵ % of each of the lines a and b onto the plane P and let $\epsilon \to 0$ [1]. The angle is then measured on the projected lines using the usual rules of Euclidean geometry.

[1] Only project the first ϵ % because the surface of the sphere can be approximated by a plane if the region is very small

- **Coordinates**: Up to this point, longitude and latitude coordinates have been used as a range from -180 degree to 180 degree for the longitudes and -90 degree to 90 degree for the latitudes. In this chapter the longitudes will be in the range from 0 degree to 180 degree east or west. The latitudes will be in the range from 0 degree to 90 degree north or south

8.4.2. Spherical Geometry

This chapter introduces some concepts of spherical geometry that will be needed for some proofs.

Big Circle: Let M be the center of our sphere and let P be a plane containing M. Then the intersection of P with the surface of the sphere results in a big circle.

Antipodal point: Let p be a point on the sphere. The antipodal point is the point on the opposite side of the sphere. The antipodal point of p is also the point furthest away from p. The most famous pair of antipodal points is the North and South Poles.

Shortest Path: Let u and v be two points of our sphere with center M. Let B be the big circle containing u and v. u and v divide the big circle into two fragments. The smaller of the two fragments is the shortest path from u to v (and vice versa from v to u).

Euler Triangle: Let u, v, and w be points on the surface of a sphere. Let each point be connected to every other point so that they form a triangle. Let α, β and γ be the angles between two lines. If all these angles are less than π radians, the triangle is called an Euler triangle.

Triangle Inequality: Unlike in Euclidean space, the triangle inequality is not generally applicable to triangles on a sphere. However, if we restrict the triangles to Euler triangles, then the triangle inequality is applicable. This means: If a, b, and c are the lengths of the lines of an Euler triangle, then the sum of two lines is at least as large as the remaining line (so $a + b \geq c$, $a + c \geq b$, and $b + c \geq a$).

Spherical law of cosines and spherical Pythagoras: Let the three points u, v and w define a triangle. The lines a, b and c of the triangle are fractions of big circles, which could also be represented by their subtended angles $\widetilde{\alpha}$, $\widetilde{\beta}$ or $\widetilde{\gamma}$. The lines are angled to each other at the angles δ, ϵ and ζ. All of this can be seen in Figure 8.1. The first spherical law of cosines states, that

$$\cos(\widetilde{\alpha}) = \cos(\widetilde{\gamma}) \cdot \cos(\widetilde{\beta}) + \sin(\widetilde{\gamma}) \cdot \sin(\widetilde{\beta}) \cdot \cos(\delta)$$

If the angle δ is equal to $\frac{\pi}{2}$ then the formula simplyfies to the spherical Pythagorean analog (because $\cos(\delta) = \cos(\frac{\pi}{2}) = 0$):

$$\cos(\widetilde{\alpha}) = \cos(\widetilde{\gamma}) \cdot \cos(\widetilde{\beta})$$

Law of Haversines: Let the three points u, v and w define a triangle. The lines a, b and c of the triangle are fractions of big circles, which could also be represented by their subtended angles $\widetilde{\alpha}$, $\widetilde{\beta}$ or $\widetilde{\gamma}$. The lines are angled to each other at the angles δ, ϵ and ζ. All of this can be seen in Figure 8.1. The haversine is defined as

$$\text{hav}(\theta) = \sin^2(\frac{\theta}{2}) = \frac{1 - \cos(\theta)}{2}$$

Using the spherical law of cosines, the fact that $\cos(\theta) = 1 - 2 \cdot \text{hav}(\theta)$ and the addition identity $\cos(a - b) = \cos(a) \cdot \cos(b) + \sin(a) \cdot \sin(b)$ one can derive the law of haversines:

$$\text{hav}(\widetilde{\alpha}) = \text{hav}(\widetilde{\gamma} - \widetilde{\beta}) + \sin(\widetilde{\gamma}) \cdot \sin(\widetilde{\beta}) \cdot \text{hav}(\delta)$$

We will also need the inverse of the haversine, which is defined as follows:

$$\text{archav}(\theta) = 2 \cdot \arcsin(\sqrt{\theta}) = \arccos(1 - 2 \cdot \theta)$$

From a mathematical point of view, there's no difference between using the law of cosines or the law of haversines. But for small angles, the haversine function provides numerical stability, while the cosine function does not. For angles close to π, the haversine function can result in a complex number if a rounding error becomes too large; the law of cosines doesn't have this problem. Therefore, we will need both laws, depending on the angle.

8.4.3. Proofs

This subsection presents the proofs for the correctness of the algorithm used. Since some proofs share logic, this logic is split into separate proofs. This improves readability by making the proofs shorter and avoids repetition.

The first proof of this chapter shows, that the edge of a spherical triangle, which is opposite of the $\frac{\pi}{2}$ angle, is the longest, if the adjacent sides are at most $\frac{1}{4}$ of the circumference.

Theorem 1. *In a triangle formed by three points u, v, and w, where each side adjacent to v is at most $\frac{1}{4}$ of the circumference of the sphere and the angle at point v is $\frac{\pi}{2}$, the side opposite to v is greater than or equal to the other sides in the triangle. For a detailed construction of the points and sides, see Figure 8.2.*

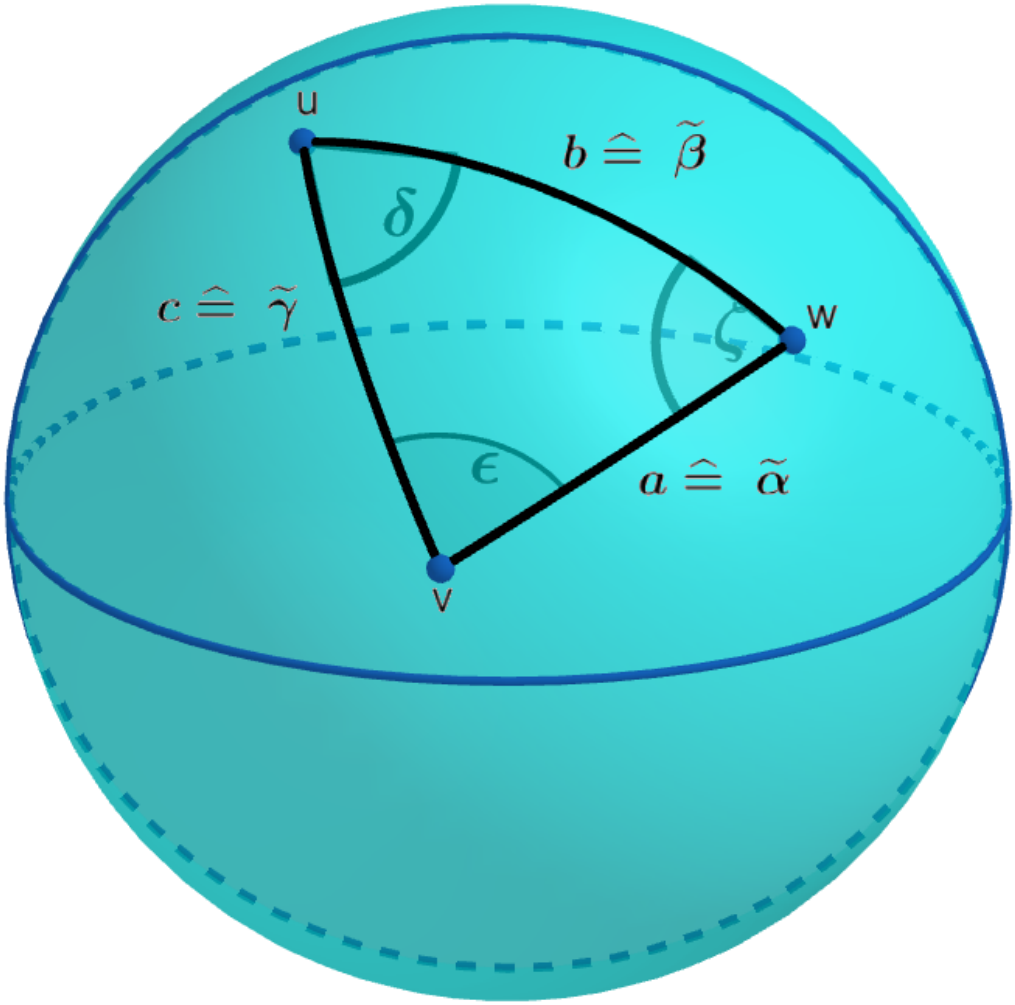


Figure 8.1.: This figure shows the setup for the Spherical Pythagorean Theorem, the Spherical Law of Cosines, and the Spherical Law of Haversines. Source: Self made

Proof. Since we have a $\frac{\pi}{2}$ angle, we can use the spherical analog of the Pythagorean theorem. Therefore we have

$$cos(\widetilde{\beta}) = cos(\widetilde{\alpha}) \cdot cos(\widetilde{\gamma}) \tag{8.1}$$

Since we required that each side adjacent to v be at most $\frac{1}{4}$ of the circumference, we know that each of these subtended angles is between 0 and $\frac{\pi}{2}$. This means that

$$0 \leq \widetilde{\alpha} \leq \frac{\pi}{2} \tag{8.2}$$

$$0 \leq \widetilde{\gamma} \leq \frac{\pi}{2} \tag{8.3}$$

The cosine function is strictly monotone decreasing from $cos(0) = 1$ to $cos(\frac{\pi}{2}) = 0$. Therefore, $cos(\widetilde{\alpha})$ must be between 0 and 1. Using this information and the equality in Equation 8.1, we know that $cos(\widetilde{\gamma})$ must be at least as large as $cos(\widetilde{\beta})$. Since the cosine function is strictly monotonically decreasing, we know that $\widetilde{\gamma}$ must be less than or equal to $\widetilde{\beta}$. The lengths of the sides can be easily calculated, given the subtended angle and the radius r:

$$c = \widetilde{\gamma} \cdot r \leq \widetilde{\beta} \cdot r = b \tag{8.4}$$

This equation shows that b is greater than or equal to c. Using Equation 8.3, Equation 8.1 and the same reasoning as before, we know that $\widetilde{\alpha}$ must be less than or equal to $\widetilde{\beta}$ and therefore $a \leq b$.

Since both a and c are less than or equal to b and b is the side opposite the $\frac{\pi}{2}$ angle, the claim that this side is at least as long as the other sides has been proved.

□

The next proof shows that the distance from a point not contained on a big circle to that big circle is unique, or that all points on the big circle have the same distance to that point.

Theorem 2. *The shortest path from a point u to a big circle C that doesn't contain u is unique or all points on C have the same distance to u.*

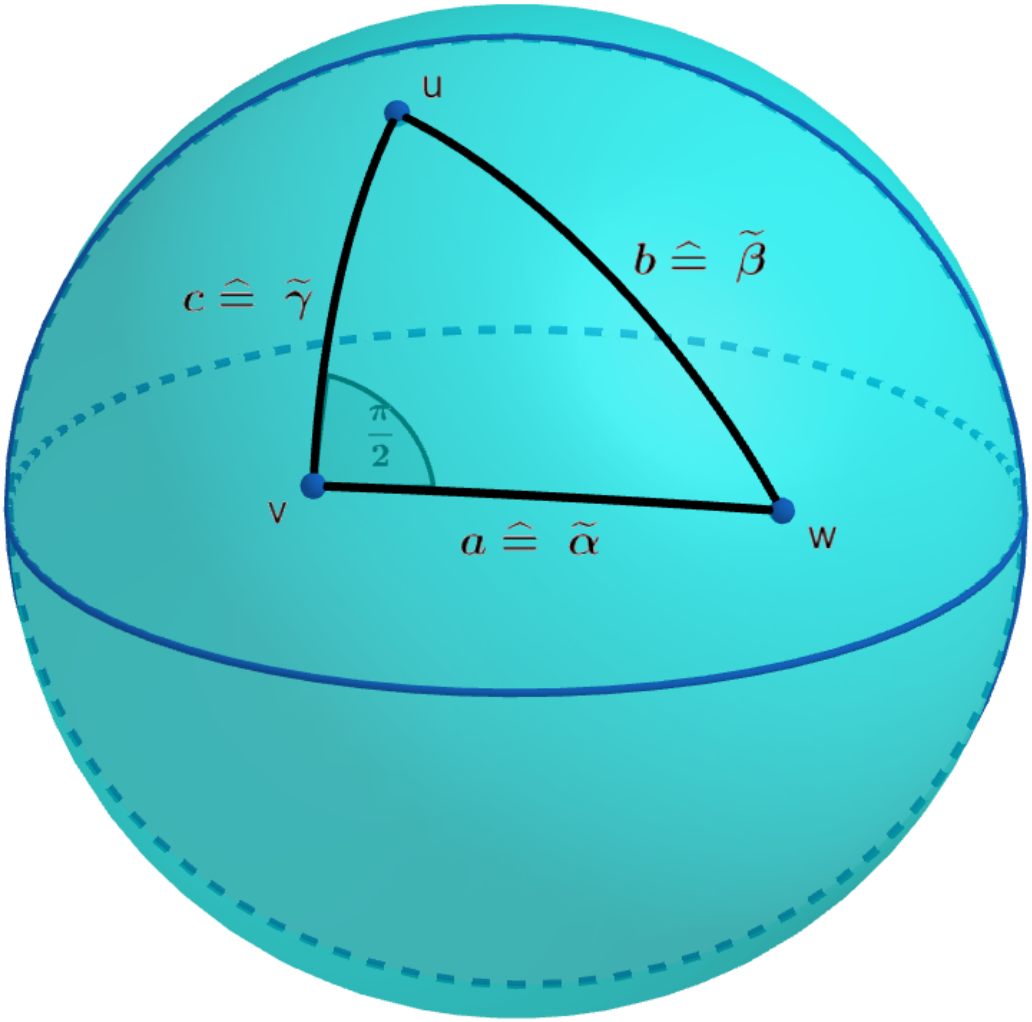


Figure 8.2.: Set up to prove that the side opposite the $\frac{\pi}{2}$ angle is the longest or at least equal to the other two sides. Source: Self made

Proof. Let u be a point on the sphere, which is, without loss of generality, the North Pole. Next, consider the plane P of a big circle C. It must contain the center M of the sphere. In addition, the plane contains at least two points of the equator, which are on opposite sides. W.l.o.g. let these points be on the 90th longitude south and east. Since these three points are on a line, the plane is still not uniquely defined. Let v be a final point to uniquely define the plane, which is w.l.o.g. on the 0th longitude. We only need to consider the cases where v is between the 0 degree latitude (equator) and the 90 degree latitude north, since all other cases are symmetrical to one of these cases.

Case 1: v is on the equator (v has 0 degree latitude)

The big circle formed by the intersection of P and the sphere is the equator. On the equator, all points have the same distance to the North Pole, because "just going straight up" is walking along a line of longitude, and lines of longitude are big circles and therefore shortest paths.

Case 2: v is above the equator (latitude of v is > 0 degree latitude north)

The plane P can also be defined by a point and two Euclidean lines. One point is M and the first Euclidean line e is the line connecting the two points on the equator, which are on the 90th meridian east and west (so all points on this line have the same latitude). The second Euclidean line f is the line from M to v. When the latitude of v is changed, the line e stays in P because it is orthogonal to the line $\overline{Mv}$. Since the latitude of v is greater than 0, P is tilted with respect to the equatorial plane. Since we restricted the latitude of v, the angle α of P to the equatorial plane is $0 < \alpha < \frac{\pi}{2}$. Let w be a point on C. As we decrease the longitude of w (going toward the longitude of 0, which is the longitude of v), the latitude increases (because of the angled plane). Since the decrease can be done from either the west or the east, and both sides are symmetric, the unique point with the lowest longitude, and therefore the highest latitude, is v. As in case 1, the shortest path from any point on C to u is "just going up". Since v has the highest latitude, it's distance to u is the smallest.

These two cases cover all possibilities, and in both cases the claim is valid. Therefore, the claim is proven. □

The next theorem proves, that the shortest path from a point to a big circle intersects the big circle at an angle of $\frac{\pi}{2}$.

Theorem 3. *The shortest path from a point u to a big circle C, intersects C at an angle of $\frac{\pi}{2}$.*

Proof. From Theorem 2 we know that we need to consider two cases. Let the setup be the same as in this proof (the point u is w.l.o.g. the North Pole, the plane P of the big circle C is w.l.o.g. defined by the following four points: the center of the sphere M, the two points on the equator at longitudes of 90 degrees east and west, and the point v on the 0th longitude line with a variable latitude).

Case 1: v is on the equator (v has a latitude value of zero)

The plane of the equator and P are the same plane. The coordinates of v are therefore 0 latitude and 0 longitude. The shortest path from v to u is the fragment of the line of longitude 0 from v to u. The tangent of this line at the equator has an angle of $\frac{\pi}{2}$ to the equatorial plane and therefore to C.

Case 2: v is above the equator (latitude of $v > 0\hat{A}^\circ$ north)

The tangent of C at point v is the Euclidean line e, which is parallel to the Euclidean line from the 90th meridian east to the 90th meridian west of the same latitude. Let Q be the plane containing the line of longitude 0 (since this is a non-Euclidean line, it uniquely defines the plane). This plane is orthogonal to the line e. All tangent vectors of the 0th longitude are contained in Q and therefore the tangent line of C at point v is orthogonal to the line from v to u.

These two cases cover all possibilities, and in both cases the claim is valid. Therefore, the claim is proven.

□

The next proof in this chapter shows, that the shortest distance between two latitudes is obtained, when the path between the latitudes intersects the start latitude and target latitude at an angle of $\frac{\pi}{2}$.

Theorem 4. *The shortest path between two latitudes is, where the line from the first to the second latitude intersects both latitudes at an angle of* $\frac{\pi}{2}$.

Proof. Since the distance from the upper latitude to the lower latitude is the same as the distance from the lower latitude to the upper latitude, we consider only the case from the lower latitude to the upper latitude. [2]

First, we define the point u to be the point of our lower latitude, which has an arbitrary longitude. W.l.o.g., let this longitude be zero. Point v has the same longitude as u, but a higher latitude. Next, we define a great circle. The first point on the big circle is point v, the second point is the center of the sphere and the third point (which must not be on the line of v and the center of the sphere [3]) is on the equator and has a longitude of 90. It doesn't matter in which direction, both points on the equator with longitude 90 are contained in the great circle, because the great circle is symmetric about the line from v to the center of the sphere. This west-east symmetry ensures that the angle of intersection of the great circle at point v with the line from u to v must be $\frac{\pi}{2}$. All this can be seen in Figure 8.3, where blue is the lower latitude and orange is the upper latitude. The big circle is drawn in red. Now consider any point w which has the same latitude as v but a different longitude.

[2] Lower latitude means a smaller latitude number and therefore being closer to the equator (no matter if north or south of the equator)

[3] otherwise the plane of the big circle would not be uniquely defined

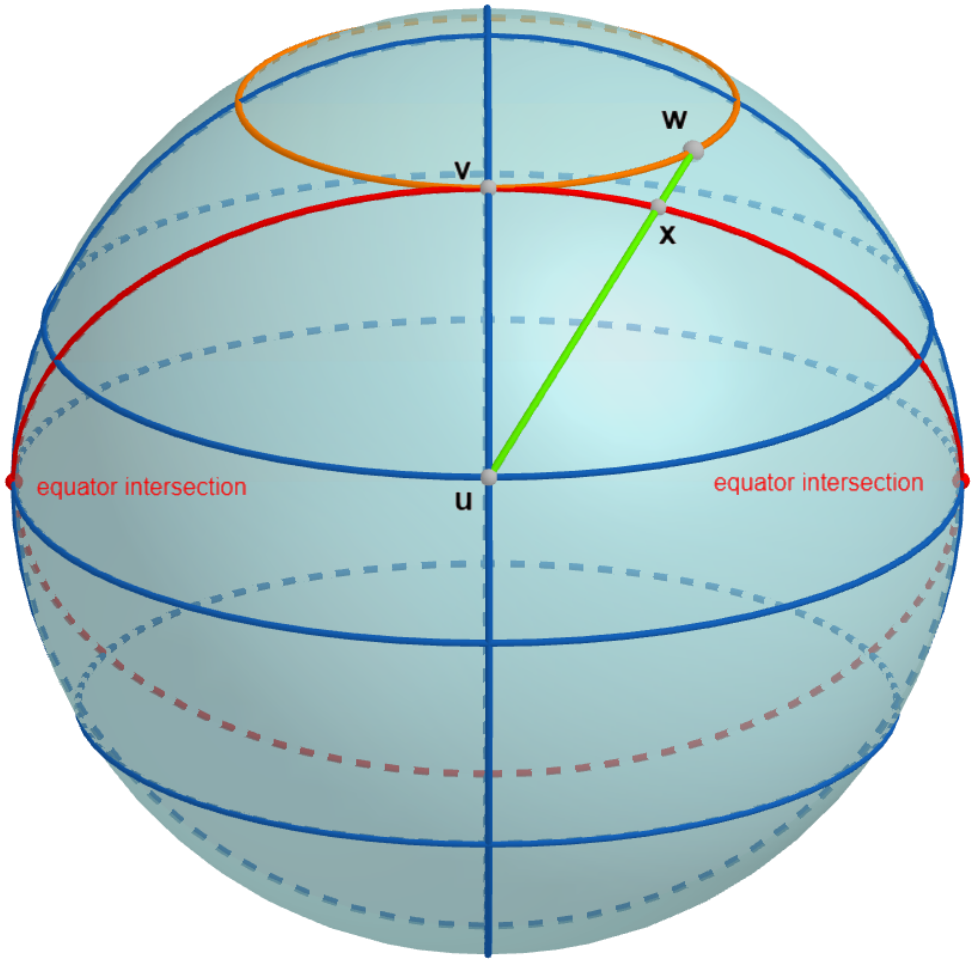


Figure 8.3.: This figure shows the setup for the proof of the shortest distance between two latitudes. Source: Self made

Consider the line from u to w, shown in green. This line is divided into two parts by the big circle. Let x be the point where the big circle intersects this line. From Theorem 1 we know that the line from u to x is at least as long as the line from u to v (since this is the line opposite the angle of $\frac{\pi}{2}$ in the triangle formed by u, v, and x). Since the line fragment from x to w is greater than zero, the line from u to w must be longer than the line from u to v. Therefore, the shortest path between two latitudes is where the line between the latitudes intersects both latitudes at an angle of $\frac{\pi}{2}$, which was claimed.

□

The next proof gives an upper bound, for the shortest parth from any point to a longitude line.

Theorem 5. *The shortest path s from a given point u to any line of longitude is upper bounded by the line from u to the nearest pole.*

Proof. Since the lines of longitude are fragments of a large circle containing the poles, we know that the shortest path a from u to the pole is the path that stays on the line of longitude. From the pole, we only need to go an ϵ amount in the direction of the target longitude. The triangle inequality gives us an upper bound on the path length:

$$a + \epsilon \geq s$$

For $\epsilon \to 0$, the upper bound converges to

$$\lim_{\epsilon \to 0} a + \epsilon = a \tag{8.5}$$

Therefore the distance to each longitude is upper bounded by a, which is the distance to the corresponding pole. □

As we just showed: All longitudes can be reached from a given point u by simply going "straight up or down" to the corresponding pole. Let the distance of this upper bound be a. The next proof considers the case, where we are interested in the most distant longitude, that can be reached within a distance of less than a.

Theorem 6. *The most distant line of longitude L that can be reached from point u at a distance less than the distance to the nearest pole intersects the line from u to L at an angle of $\frac{\pi}{2}$ at a unique point.*

Proof. Let a be the distance from u to the closer pole and $b < a$ be the distance, which is the maximum line length allowed. Since the distance to the nearer pole is at most $\frac{1}{4}$ of the circumference (equality happens if u is on the equator), we know that b must be less than that. Let L be the farthest longitude that can be reached from u at a distance of b. Let v be the point where the line from u intersects L. We know that v must be the closest point of the big circle containing L (otherwise we could go to L by a shorter path, which is especially smaller than b. So we could take that shorter path and reach an even more distant line of longitude from there, which would be a contradiction to our definition of L). From Theorem 2 we know that v is the unique shortest point (since b is smaller than $\frac{1}{4}$ of the circumference of the sphere, which would be the distance if all points on L had the same shortest distance). From Theorem 3 we know that the path from u to L intersects L at an angle of $\frac{\pi}{2}$. □

In subsection 7.2.3, there were a lot of formulas for calculating the query box. The next proof, will prove the correctness of these formulas and the query box in general.

Theorem 7. *The query box, as explained in subsection 7.2.3, always contains all points, that are at most $a \in \mathbb{R}^+$ meters away.*

Proof. Let u be our starting point, from which all paths start. Then three cases can occur: both poles can be reached with a distance of at most a meters, only one pole or zero poles.

Case 1 (both poles are reachable): If both poles can be reached, then the upper latitude limit is 90, because $\min\{90, \text{lat}(u) + a \cdot \frac{\text{circumference}}{360}\}$ evaluates to 90. This is because $\text{lat}(u) + a \cdot \frac{\text{circumference}}{360}$ evaluates to a higher number than 90, since the second part just translates the distance of a in degrees that a covers on a big circle. This is added to the current latitude of u. The result must be greater than or equal to 90, since this is the case when both poles can be reached. The reason why the lower bound is -90 is analogous. According to the construction of the query box, all longitudes are checked (because of Theorem 5, the distance to all longitudes is upper bounded by the distance to the pole. If the pole can be reached, a point on any longitude line can be reached). Therefore, we have just created a query box that contains all points. Therefore, no point can be missed.

Case 2 (only one pole can be reached): First we consider the case where the pole is the North Pole. As in case 1, the upper latitude bound is 90 (see the calculation and reasoning above). The longitude bounds are -180 to 180, since the upper latitude bound is 90. This is necessary, because of Theorem 5. Since all longitudes are included, a point can only be missed if the latitude is too high or too low. Since the upper bound for latitude is 90, a point above u can't be missed. Therefore, we only need to prove that points with a latitude lower than the query box can't be reached from u. Let l be the lower bound $\text{lat}(u) - a \cdot \frac{\text{circumference}}{360}$. Let v be the point with a latitude of l that has the same longitude as u. The distance from u to v is a (the formula just converted the distance of a from meters to degrees of latitude along a large circle). According to Theorem 4, the shortest path to any point on l is the path on the same longitude (so from u to v) (since the longitude line intersects both latitude lines at an angle of $\frac{\pi}{2}$). Therefore, any point on the same latitude as l or on a latitude less than l has a longer path than the path from u to v. Since the path from u to v has a length of a, all other paths are longer than a, and therefore the query box contains all points that are reachable from u.

Case 3 (no pole can be reached): The arguments for the upper and lower bounds of the query box are analogous to the lower bound in case 2. To prove the longitude bound, we need to construct a triangle. W.l.o.g. let u be on the northern hemisphere. The case where u is on the southern hemisphere is analogous. The setup of this proof can be seen in Figure 8.4. u is the first point of the triangle and the North Pole, labeled w, is the second point. v is the point on the longitude farthest away from u, but still within a distance of a. From Theorem 6 we know that the angle at v must be $\frac{\pi}{2}$. The distance from u to v is a, which can be converted to a subtended angle using

$$\widetilde{\alpha} = \frac{a}{r} \tag{8.6}$$

where r is the radius of the earth. The subtended angle for the line from u to w (the north pole) is

$$\widetilde{\gamma} = (90 - \text{lat}(u)) \cdot \frac{2\pi}{360} \tag{8.7}$$

This formula simply calculates the difference between the latitudes of u and w and converts it to radians. Now that we have two of the three subtended angles and a right angle at v, we can use the Spherical Pythagorean Theorem to calculate the third subtended angle $\widetilde{\beta}$:

$$\cos(\widetilde{\gamma}) = \cos(\widetilde{\alpha}) \cdot \cos(\widetilde{\beta})$$

$$\widetilde{\beta} = \cos^{-1}(\frac{\cos(\widetilde{\gamma})}{\cos(\widetilde{\alpha})}) \tag{8.8}$$

Note that we could already calculate the latitude of v, since the latitude of v is just $\widetilde{\beta}$ from the North Pole on any longitude line. For the proof of the longitude bound, we don't need to know the specific latitude of v, so we don't explicitly calculate it here.

Now that we have all three subtended angles, we can use the law of cosines [4] to obtain the angle δ:

$$\cos(\widetilde{\alpha}) = \cos(\widetilde{\gamma}) \cdot \cos(\widetilde{\beta}) + \sin(\widetilde{\gamma}) \cdot \sin(\widetilde{\beta}) \cdot \cos(\delta)$$

$$\cos(\delta) = \frac{\cos(\widetilde{\alpha}) - \cos(\widetilde{\gamma}) \cdot \cos(\widetilde{\beta})}{\sin(\widetilde{\gamma}) \cdot \sin(\widetilde{\beta})}$$

$$\delta = \cos^{-1}(\frac{\cos(\widetilde{\alpha}) - \cos(\widetilde{\gamma}) \cdot \cos(\widetilde{\beta})}{\sin(\widetilde{\gamma}) \cdot \sin(\widetilde{\beta})}) \tag{8.9}$$

δ is the longitude difference of u and v, which we are interested in. Therefore we just need to convert it to degrees and add it to u (or subtract it from u if we want to know the left bound):

$$\text{lon}(v) = \text{lon}(u) \pm \delta \cdot \frac{360}{2\pi} \tag{8.10}$$

[4] In practice, we would use the law of haversine for small distances (or small subtended angles representing the maximum distance) because of numerical stability. For the proof, it is sufficient to use only the law of cosines

If we plug Equation 8.9, Equation 8.8, Equation 8.7 and Equation 8.6 into Equation 8.10 and use the north-south symmetry[5], then we get the following formula:

$$\text{lon}(u) \pm +\cos^{-1}\left(\frac{\cos\left(\frac{a}{r}\right) - \frac{\cos^2((90-|\text{lat}(u)|)\cdot\frac{2\pi}{360})}{\cos(\frac{a}{r})}}{\sin((90-|\text{lat}(u)|)\cdot\frac{2\pi}{360})\cdot\sin(\cos^{-1}(\frac{\cos((90-|\text{lat}(u)|)\cdot\frac{2\pi}{360})}{\cos(\frac{a}{r})}))}\right)\cdot\frac{360}{2\pi}$$

This is the formula from chapter subsection 7.2.3. Because of Theorem 6, there is no longitude further away than a meters. Therefore the query box is also valid in this case.

Note that at this point a coordinate normalization may be necessary to have the coordinates in their usual form. For example, this formula might calculate the left boundary to be at -190. In this case, all we need to do is add 360, which results in 170. If the result is greater than 180, we would need to subtract 360. This doesn't change the resulting point, because adding or subtracting 360 from the longitude value just means going around the Earth once at a constant latitude and coming back to the same point.

These three cases cover all possible cases, and in all cases the query box is valid. Therefore, the statement is proven.

□

The next proof shows an optimization that is not necessary for correctness or completeness, but can improve the performance of the algorithm. The optimization is useful when the maximum distance is large (i.e. it is used as soon as the maximum distance is larger than $\frac{1}{4}$ of the circumference). In this case, a pole is always within reach of maxDist, so the longitude bounds would always cover all longitude lines. To improve this, we can compute an anti-query box for the antipodal point, which is guaranteed to contain only points further away than the maximum allowed distance. Then we check everything except this anti-query box. A more detailed explanation can be found in subsection 7.2.3. To prove the correctness of the anti-query box, we first need to prove a helper theorem about the maximum distance from the center of the query box to any point inside the query box.

Theorem 8. *Let p be the center of a symmetric query box[6] that does not contain a pole [7]. Let u be the point on the left side of the query box that has the same latitude as p, and let v be the point on the top side of the query box that has the*

[5]The symmetry of the northern and southern hemispheres can be used by taking the absolute value of the latitude.

[6]Symmetric means that the left and right sides have the same distance to the center, and the top and bottom sides have the same distance to the center.

[7]The proof would still work in this case, but the argument would be more complicated. So the algorithm just stops at the pole.

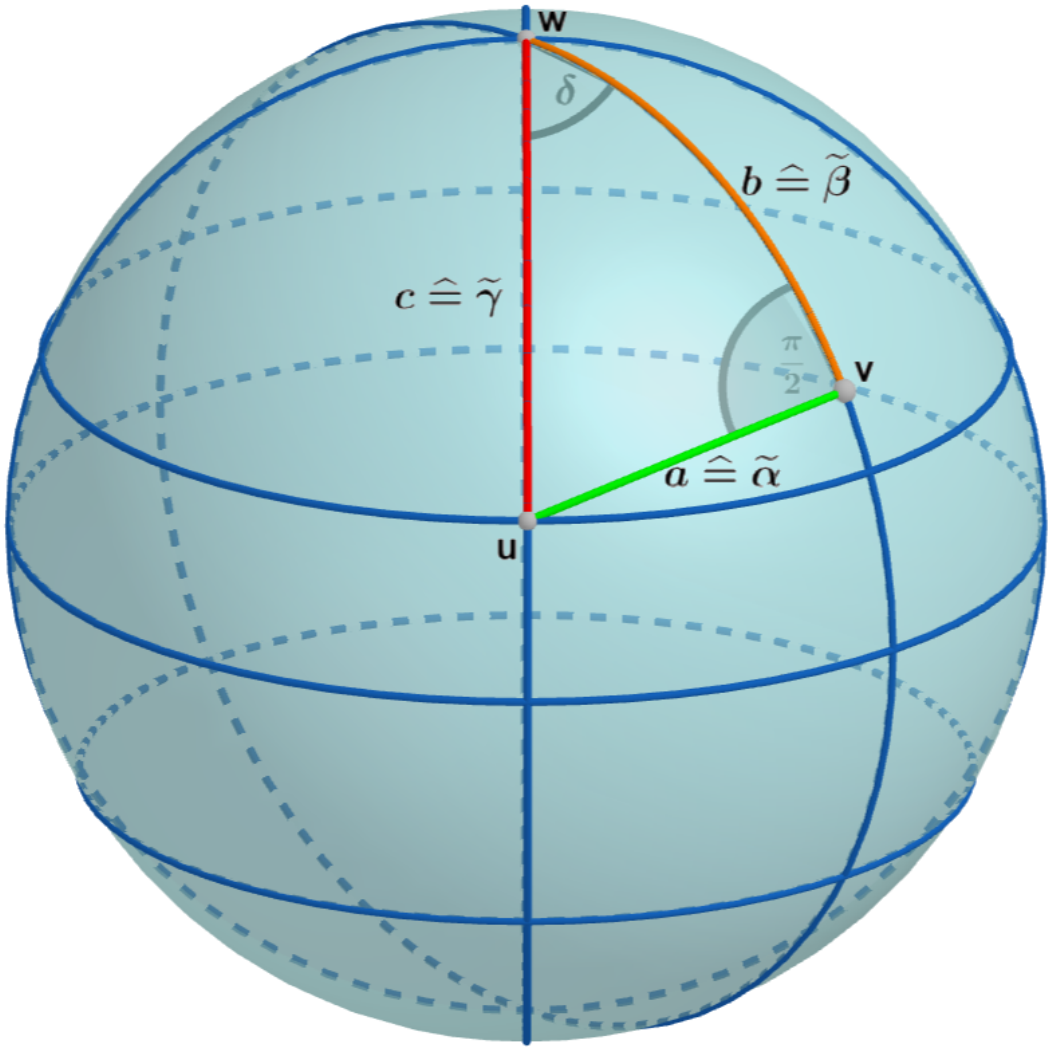


Figure 8.4.: Setup for the proof of the longitude bounds of the query box. Source: Self made

same longitude as p. Let d be the distance from p to u and e be the distance from p to v. Let d and e be less than $\frac{1}{4}$ of the circumference of the Earth. Then the distance from the center p to any point inside the query box is upper bounded by the sum $d + e$.

Proof. Let w be any point in the bounding box. Due to the symmetry of the bounding box, we can assume that w is in the upper left quadrant. Let x be the point that has the same latitude as p and the same longitude as w. Then construct a triangle with the points p, w, and x. Let a be the distance from p to x, b be the distance from x to w, and c be the distance from p to w. All this can be seen in the figure Figure 8.5. Note that the distance e (from p to v) mentioned in the proof statement is not shown, and instead the distance from u to v' is shown, where v' is the point that has the same latitude as v and the same longitude as u. The distance from p to v is the same as the distance from u to v'. In both cases, the distance is e, because the difference in latitude is the same, and p and v have the same longitude, and u and v' have the same longitude[8]. Since both d and e are less than $\frac{1}{4}$ of the circumference, the triangle inequality can be applied. Using the triangle inequality, the following statement follows:

$$a + b \geq c$$

Let d be the distance from p to u and e be the distance from p to v. The latitude of x and u is the same, so $d \geq a$. The longitude of x and w is the same, so $e \geq b$. This gives the following inequality:

$$c \leq a + b \leq d + e$$

This inequality proves the statement. □

Note that the proof requires that u be the point on the same latitude as p, not the point where the distance from p to the left is smallest. If the smallest distance from p to the left were above u, the triangle inequality could not be applied to points below p.

Now that we have proved the helper theorem, we can continue to prove the correctness of the anti query box.

Theorem 9. *Let u be the point from which we want to get all points that are at most m meters away, and let m be less than half the circumference of the Earth. Let v be the antipodal point of u. Then create a symmetric anti query box around*

[8] When the longitude of two points is the same, the shortest path between the two points is to stay on the longitude, because a longitude is a subset of a great circle

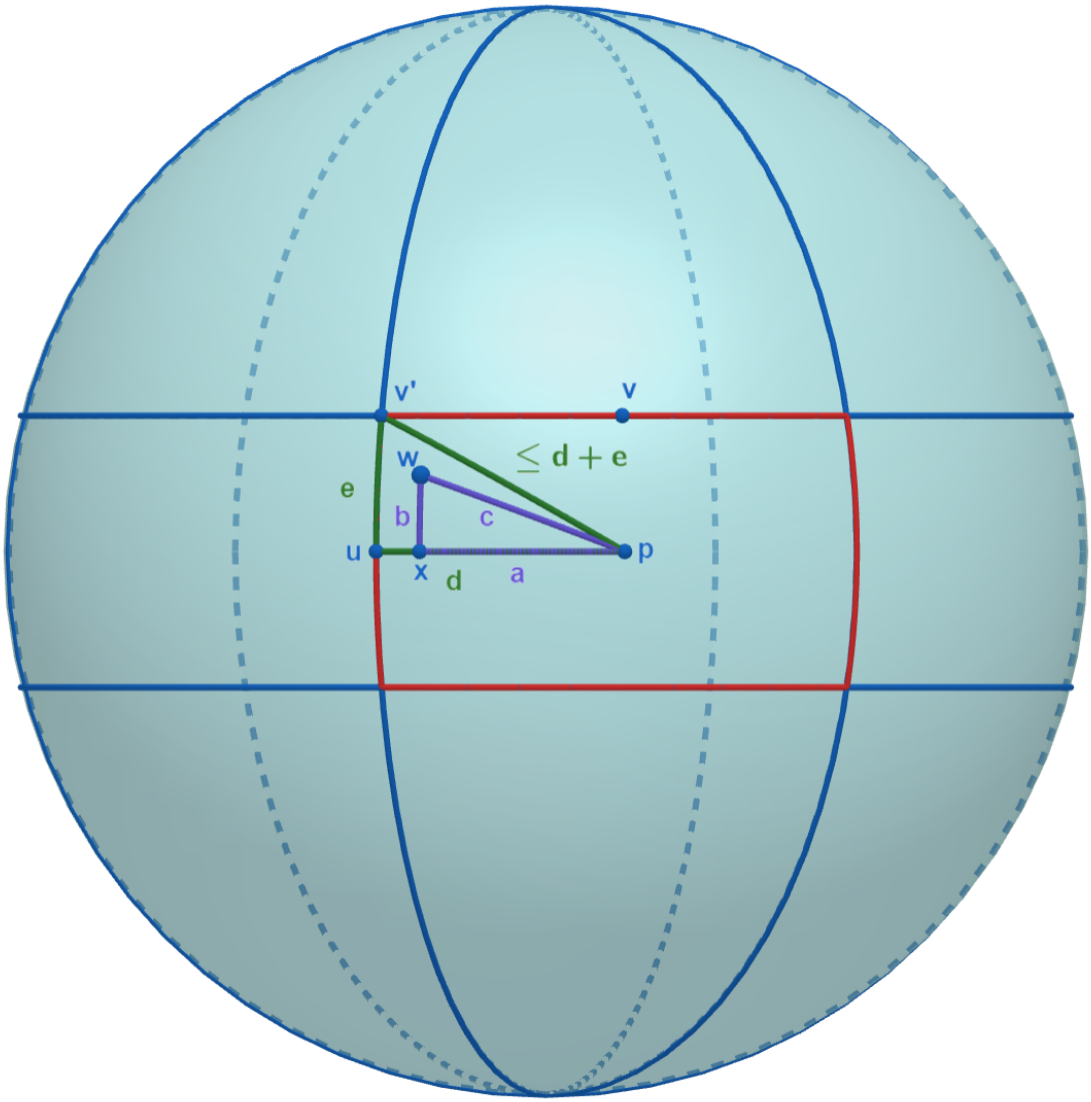


Figure 8.5.: Setup for the proof of the maximum distance from the center point of the bounding box to any point in the inside of the bounding box. Source: Self made

v. Let a be the distance from v to the left side of the anti query box and b be the distance from v to the top of the anti query box[9]. Let $c = a + b$ and let d be half the circumference of the Earth. If $c < d - m$, then every point contained in the anti query box has a distance to u that is greater than m.

Proof. Let w be any point in the anti query box. Then there exists a shortest path from u to v that contains w[10]. Going along this shortest path, starting at u, the intersection of the anti query box is reached. Let x be the intersection point and e be the distance from u to x. If w is on the edge of the anti query box, then $w = x$. Otherwise you have to go a little longer on the shortest path to w for a distance greater than zero. So the distance from x to w is ≥ 0. This means that if the shortest path from u to x is already greater than m, then the path from u to w is definitely greater than m. From Theorem 8 we know that the distance from x to v is less than or equal to c. Let this distance be f. Since d is the distance between u and v (since they are antipodal), and x splits the path, we know that $e + f = d$. From the proof statement we know that $c < d - m$, which is equivalent to $c + m < d$. So we have $c + m < e + f$. Since $f \leq c$, we have $m < e$. Since e is the distance from u to the anti query box (to x) and this distance is already greater than m, the distance to w is at least as large as e and therefore greater than m. So the distance from u to w is greater than m. This proves the statement that every point in the anti query box has a distance to u that is greater than m. □

Next, I will prove the correctness of the area part of the algorithm and that the query box contains all possible candidates and that no candidate is missed. Let me briefly recapitulate the construction of the query box. First, the bounding box of the given area is computed. Next, the center u of this bounding box is calculated. Then we use Theorem 8 to calculate the upper distance from the midpoint u to any point inside the bounding box. Let b be this distance. Then we construct the query box in the same way as the query box for points using point u, but we increase the maximum allowed distance a by b. So the new maximum distance a point can be away from u is $a + b$.

Theorem 10. *If a point v is at most $a \in \mathbb{R}^+$ meters away from an area, then it is at most $a + b$ meters away from the center of the area.*

Proof. Consider the shortest path from u to v. Let x be the point where this shortest path intersects the bounding box of the area. This can be seen in Figure 8.6. With Theorem 8 the distance from u to x is at most b meters. Because of the requirement that v is at most a meters away from the area, and the entire area is contained in the bounding box, the distance from x to v can be at most a (if the area intersects

[9] because of the symmetry, the distance to the left side is the same as to the right side, and the distance to the top is the same as to the bottom

[10] There are an infinite number of shortest paths from u to v, since they are antipodal. One of these paths contains w

the bounding box at x, otherwise the distance from x to v is less than a). Therefore, the total distance from u to v can be at most $a + b$ meters. □

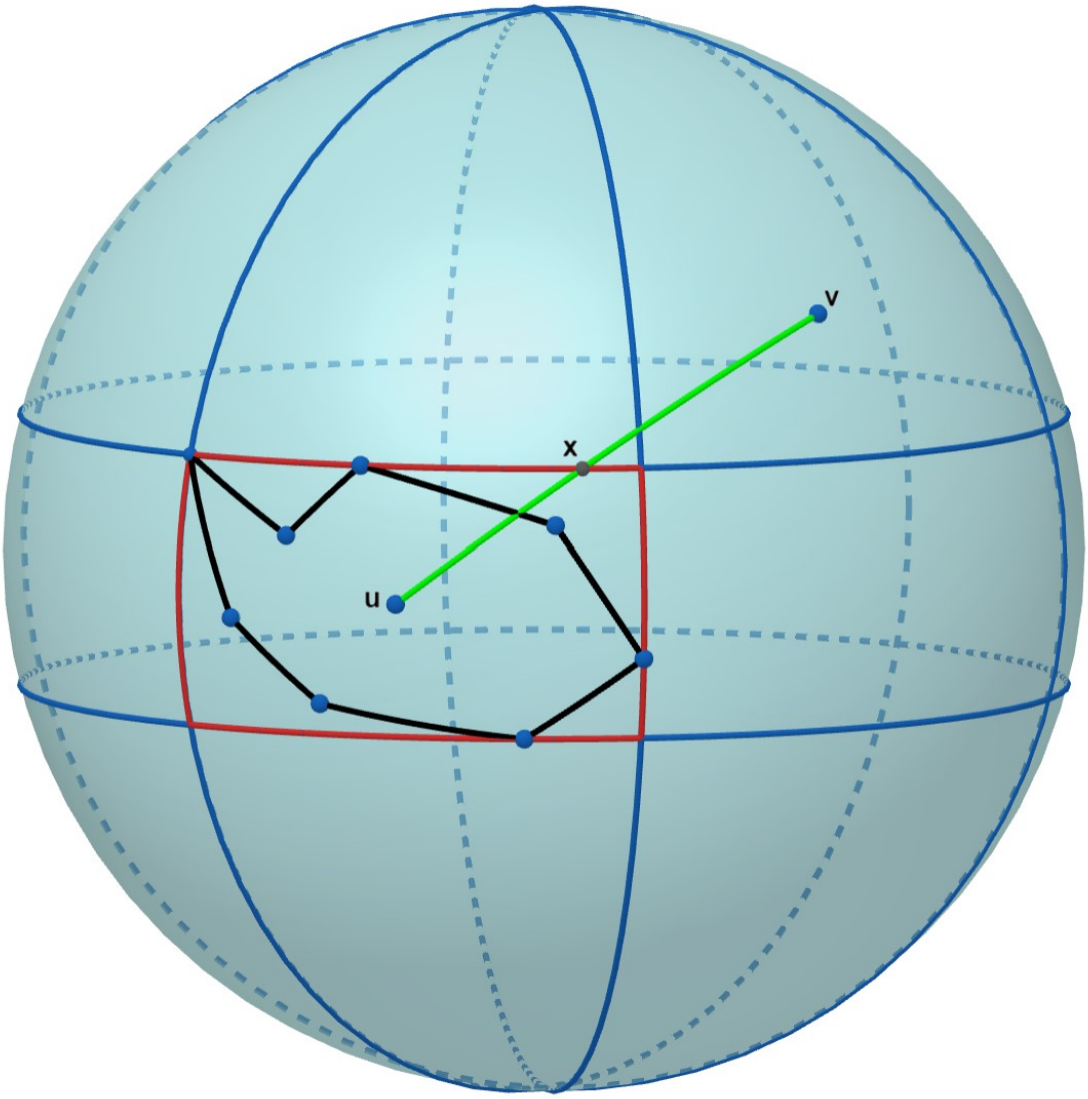


Figure 8.6.: Setup for the proof of the area part of the algorithm, where the maximum allowed distance gets increased to compensate for the size of the area. Source: Self made

9. Practical Analysis

In this chapter, the performance of the presented algorithm is evaluated and compared. The evaluation is performed on two datasets. The first dataset is a synthetic test dataset developed for the purpose of this evaluation. The second dataset is the OpenStreetMap dataset of Germany. After introducing the synthetic dataset and the general testing procedure, the results of the different evaluations are presented. The first experiment evaluates whether the R-tree should be built for the larger or smaller input. After that, the efficiency of the query box was evaluated. Then, the different functions of the code were analyzed for their impact on the runtime, and finally, the performance of the algorithm was evaluated on the OSM Germany dataset.

9.1. Synthetic Dataset

In order to get very good evaluation results, a synthetic dataset has been developed. The reason why most of the tests were not performed on a real dataset like OpenStreetMap is that this could lead to wrong conclusions. Let's take as an example the analysis of the influence of the maximum distance on the query. Suppose an object is located in the center of Freiburg. There are many objects around it, but as soon as the maximum distance becomes large enough to reach objects outside of Freiburg, the density of objects is likely to decrease by a large amount because fields, meadows, etc. do not have as many geo-objects in the OSM data. Therefore, a false conclusion might be that the maximum distance does not have a big impact on the runtime. In reality, it may have a large effect that is counteracted by the lower density of geo-objects outside of Freiburg. Therefore, a synthetic dataset with a uniform density of geo objects was created. The creation of the dataset was done with algorithm 12 and 13. For each longitude and latitude, four points and one area are added to the dataset. Each point and area is given a unique name and the data set will contain at least two triples of them. The first triple specifies whether it is a point or an area (to allow filtering the data and to have queries where only points or only areas are evaluated). The second triple adds the WKT string(well known text, that represents the geo object). For some points and areas these are the only triples. However, if the longitude and/or latitude is divisible by a number between two and five, additional triples are added. These triples add the information that this point or area has a longitude or latitude that is divisible by that number. These triples are added to allow queries where the inputs have different sizes. If an input

geometry must also satisfy that its latitude or longitude is divisible by one or more numbers, the input size differs and decreases the more numbers it must be divisible by. Using this algorithm, a dataset with the following statistics was created:

- Number of triples: 1,396,512
- Number of subjects: 322,202
- Number of predicates: 6
- Number of objects: 322,209

The following sections, all use this dataset, unless otherwise noted.

Algorithm 12 Algorithm to create the test dataset. Note that for simple explanations, edge cases like handling of the poles and the -180 and 180 longitude edge cases are not contained in this description of the algorithm.

```
for latitude from -90 to 90 do
    for longitude from -180 to 180 do
        addPoint(longitude, latitude)
        addPoint(longitude + 0.5, latitude)
        addPoint(longitude, latitude + 0.5)
        addPoint(longitude + 0.5, latitude + 0.5)
        addArea(longitude, latitude)
    end for
end for
```

Algorithm 13 Algorithm for adding a point to the synthetic dataset. The addArea method is analogous to this addPoint method, but creates a polygon consisting of 9 points instead of a single point as here

```
name = getName(longitude, latitude)
addTriple(name, '<isPoint>', '<true>')
wkt = getWKTForPoint(longitude, latitude)
addTriple(name, '<asWKT>', wkt)
for index from 2 to 5 do
    if latitude is divisible by index then
        addTriple(name, '<lat-is-div-by>', index)
    end if
    if longitude is divisible by index then
        addTriple(name, '<lon-is-div-by>', index)
    end if
end for
```

9.2. General test setup

The Evaluation procedure can be summarized using the following list:

1. Create/Get the knowledge graph
2. Create the Index
3. For each experiment:
    a) Make adaptions in the code for the current experiment
    b) Compile the code
    c) Run the qlever server
    d) Query the server with the test queries

The first step is to get the knowledge graph. For most tests, this would be the synthetic knowledge graph (see section 9.1). For some tests, this would be the OSM dataset for Germany. After the knowledge graph is available, the IndexBuilderMain program from qlever (see section 2.4) is used to create the index. Once the index is created, the part all experiments have in common is done. The next steps had to be done for each experimental setup. In most cases, some code was added to log runtime data, more on this in the following sections of the experiments. After that, the changed code has to be recompiled. Since the changes in the SpatialJoin algorithm do not affect the index building process, the index does not need to be rebuilt. Once the code is compiled, we can start the server. At this point, we need to change many of the default settings to ensure reproducible testing:

- Cache size is set to 0 B
- max size of a single cache entry is set to 0 B
- Set the maximum number of entries in the cache to 0
- Number of queries computed in parallel is set to 1
- Limit memory usage to 100 GB

The most important change is to disable any kind of caching. If multiple queries are run consecutively with only a slight change in a parameter, the following queries should not be able to use partial or even complete results from previous experiments. This would make comparing results useless. Another important setting is to disable parallel queries. If the server were to process multiple queries at the same time, they could interfere with each other. An example would be if both queries tried to access the same file at the same time and one of them has to wait.

The final step is to send test queries to the server and log the data from each test query. For most queries, a Python script has been written that sends queries of a similar format to the server, where each query has a unique combination of parameters. The varied parameters include the maximum distance allowed, the size

of the left subresult, the size of the right sub-result, the algorithm used, whether to query only points, only areas, or both, and a few more. A typical query might look like algorithm 14. The first line selects the variables we want to query. The variable ?dist is added internally by the SpatialJoin and contains the exact distance between ?a and ?b. The first two lines after the select clause initialize the variable ?a. If only points should be queried, ?isPointA gets replaced by <true>. If only areas are to be returned, it is replaced by <false>. The next line binds the WKT representation of the geometry of ?a to the variable ?wktA. The next two lines do the same for the variable ?b. When both variables are initialized, the constraints are added. There are nine possible constraint levels for each child. As the level increases, all the constraints of the previous levels must be satisfied, as well as the constraint of the current level. It starts with a constraint level of zero, which is no constraint. The first real constraint is that the longitude must be divisible by two. The next constraint level adds that the latitude must be divisible by two. After that, a third constraint is added, which limits the longitude to values that are also divisible by three. Then the latitude must also be divisible by three. This continues until the latitude and longitude values must be divisible by two, three, four, and five. In this case, the variable ?a has constraint level zero, which means no constraints, and the variable ?b has constraint level three. So it has three constraints. Finally, the SpatialJoin triple or SpatialJoin service is added. For the sake of brevity, and because I didn't implement it myself, I'll just mention the SpatialJoin service and refer to chapter 6 for more information.

Algorithm 14 This shows an example SPARQL query generated by the script. The algorithm for creating queries like this needs to know the maximum distance, whether ?a and ?b should be points, areas, or both, the constraint level for ?a and ?b, and the algorithm to use to compute the result.

```
SELECT ?a ?b ?dist WHERE  {
    ?a <isPoint> ?isPointA .
    ?a <asWKT> ?wktA .
    ?b <isPoint> ?isPointB .
    ?b <asWKT> ?wktB .
    ?b <lon-is-div-by> <two> .
    ?b <lat-is-div-by> <two> .
    ?b <lon-is-div-by> <three> .
    addSpatialJoin('?wktA', '?wktB', maxDist, algorithm)
}
```

9.3. Build R-tree analysis

This section analyzes whether it is more efficient to build the rtree for the smaller subresult and then query it using the elements from the larger subresult, or if it

should be done the other way around. The test queries for this analysis are built and sent by a query script, which is shown in algorithm 15. The first parameter varied is the maximum distance, which has values from 1 m to 10,000,000 m. The other two varied parameters are the size of the left child and the size of the right child. The query is built as explained in section 9.2. Since the algorithm builds the R-tree for the larger or smaller subresult, no matter if the smaller subresult is the left subresult and the larger subresult is the right subresult or vice versa, we can use this symmetry and query the server only if the right subresult has a smaller or equal number of constraints. For this experiment, three time intervals were logged. The first interval is the time from the start of the spatialJoin operation to the end of the spatialJoin operation. The second interval is the time it took the program to build the R-tree, and the third interval is the time it took to query the R-tree using the other subresult.

Algorithm 15 This algorithm shows, how the queries for this analysis got constructed and send to the server.

```
for maxDist in [1, 10, 100, 1000, 10000, 100000, 1000000, 10000000] do
    for restrictionLeft from 0 to 8 do
        for restrictionRight from 0 to 8 do
            if restrictionRight <= restrictionLeft then          ▷ use symmetry
                query = createQuery(maxDist, restrictionLeft, restrictionRight)
                sendQuery(query)                    ▷ logging is done by the server
            end if
        end for
    end for
end for
```

The results of this experiment can be seen in Figure 9.1. In the appendix, the partial results of the R-tree build time ratios are shown in Figure A.1 and the query time ratios are shown in Figure A.2. [1]. Figure 9.1 shows the ratio of the time for the complete BoundingBox algorithm ($\frac{\text{total time for building the smaller rtree}}{\text{total time for building the larger rtree}}$). A ratio less than one means that it was faster to build and query the smaller R-tree, a ratio greater than one means that it was faster to build and query the larger R-tree. A number of -1, colored in red, indicates that more than 100 GB of memory would have been required, which is why the calculation could not be performed. The y-axis shows the different experiments, sorted by the size ratio of the children ($\frac{\text{size smaller child}}{\text{size larger child}}$). The x-axis shows the maximum distance allowed between two geometries. The figure clearly shows that it's much faster to build the smaller R-tree when the children have a large size difference (i.e. small size ratios of the children, shown in the lower half of the figure). However, there are very few exceptions where it is faster to build the larger R-tree. Note that the cases where both children have the same size (i.e.

[1] Since it's kind of obvious that building the smaller R-tree is faster than building the larger R-tree, and that querying more entries takes longer than querying fewer entries, these figures are only in the appendix.

size ratios of one, shown in the upper quarter) do not contribute to the decision whether it's more efficient to build the smaller or the larger R-tree, since both have the same size. The reason for including these cases is to get a feel for the seemingly random fluctuations. Technically, these cases should take exactly the same amount of time whether the smaller or larger R-tree is built (since they have the same size). If the numbers fluctuate a bit, it shows that other factors contributed to it. In basically all of these cases, it was faster to build the larger R-tree. Since this can't be explained by the code, other factors like server load were probably the cause. Since I did all the measurements with building the R-tree for the smaller child first, and then all the measurements with building the larger child, it seems that the second time slot had a lower server load or some other favorable conditions. To draw a conclusion, we have to ignore the experiments with a child size ratio of one and just keep in mind that there is a slight variance in the time ratios. In general, there are very few cases where it is faster to build the larger R-tree, but many cases where it is much more efficient to build the smaller R-tree. And of those few cases where it's faster to build the larger R-tree, the time saving is at most 30 %, while the many cases where it's faster to build the smaller rtree, the time saving is sometimes more than 50 %. Therefore, it is very clear that the preferred option should be to build the smaller R-tree. Because of this, the next evaluations have all been done with the setting to build the smaller R-tree.

9.4. Efficiency of the query box

This section analyzes how efficient the query box is. To measure the efficiency of the query box, we look at how many points are in the result table and how many points were in the query box. The closer the ratio $\frac{\text{size of result table}}{\text{geometries in query box}}$ is to one, the more efficient the query box is. The analysis of the efficiency on the runtime is measured by the ratio $\frac{\text{geometries in query box}}{\text{geometries in R-tree}}$. The smaller this ratio, the greater the time savings, since only this fraction needs to be analyzed by the algorithm in detail. The setup for this experiment is as follows: One of the subresults consists of a single point from which the measurements are taken. This point is varied by different experiments. Since the density of points along a latitude line is constant, it doesn't make sense to vary the longitude of this point. Therefore, only the latitude of the point is varied. The other child is set up as described in section 9.2, with the restriction levels varying from zero to eight. The next parameter to be varied is the maximum distance, which has the values 1 m, 10 m, 100 m, ..., 10000000 m. Some representative results are shown in Figure 9.2, Figure 9.3 and Figure 9.4. All other results can be seen in section A.2. Next, we analyze both ratios.

First, let us take a closer look at the ratio $\frac{\text{size result table}}{\text{geometries in query box}}$ shown on the right side of the figures. For a latitude of zero degree, the ratio is at least 50 %, but for some distances it's also 100 % or close to 100 %. The heatmap for the latitude of 50 degree shows an efficiency of 100 % in almost half of the cases. In about a quarter

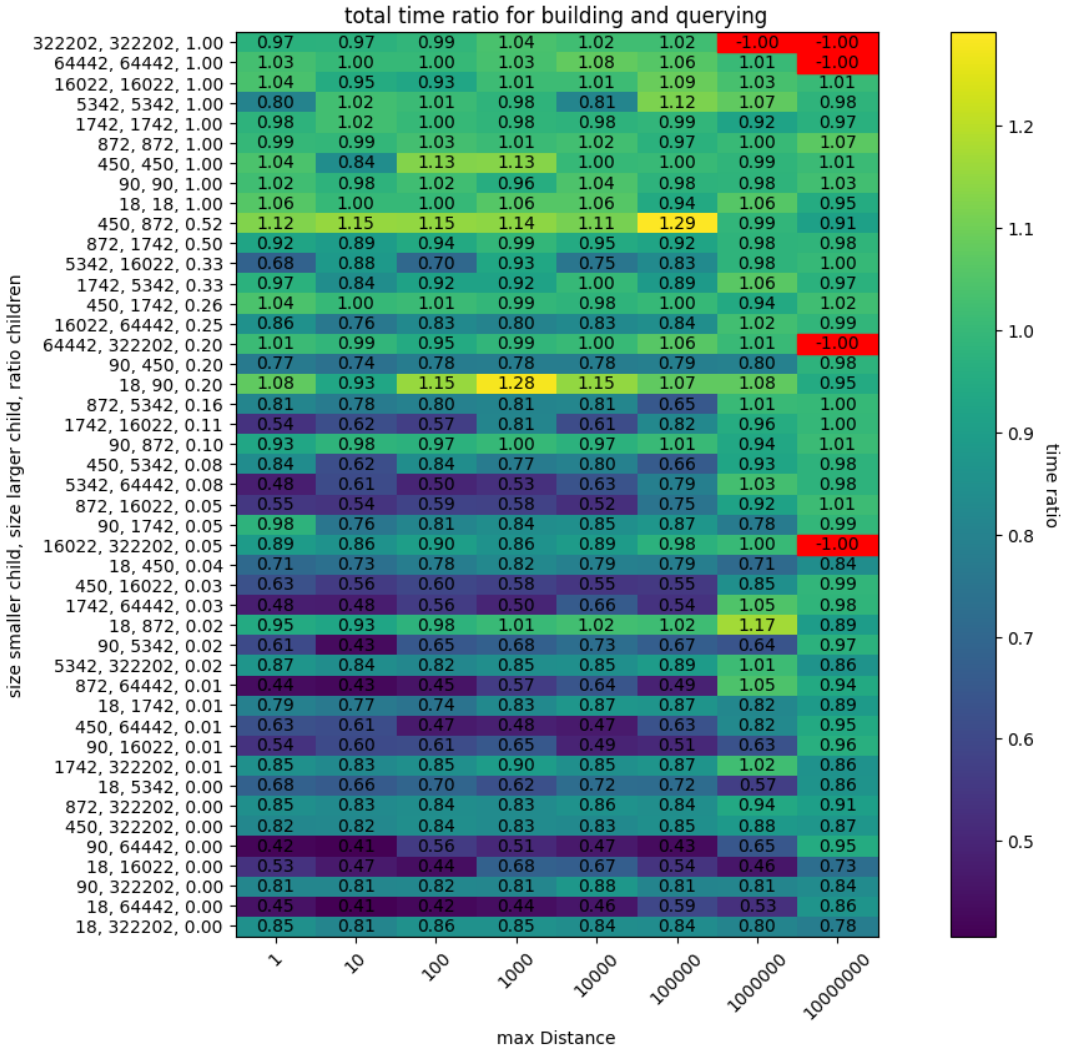


Figure 9.1.: This figure shows the ratio of the total time of the spatialJoin operation ($\frac{\text{total time when building the smaller rtree}}{\text{total time when building the larger rtree}}$). A ratio smaller than one means, that it was faster to build and query the smaller R-tree, a ratio bigger than one means, that it was faster to build and query the larger R-tree. A number of -1, colored in red, indicates, that more than 100 GB of memory would have been required, which is why the calculation could not be done

of the cases the efficiency is 50 % and in the other quarter it is 70 % to 80 %. The heatmap for the latitude of 90 degrees has an efficiency of 100 % in almost all cases. Very few cases have an efficiency of 90 % and only one case has an efficiency of 70 %. The reason that the ratios get higher as the latitude increases is because of the pole handling. When a pole is reached, the query box has to consider all longitudes. The closer the query point is to the pole, the fewer latitudes will be reached and the smaller the query box will be. The reason why smaller maximum distances that don't reach the pole in combination with smaller number of geometries in the R-tree have percentages of either 100 % or 50 % is that the result table is very small. If there are only a few geometries in the R-tree and they have a uniform density on the earth, then there is hardly any geometry close to the query point. Then it often happens that the result table has one entry and either the query box also has one entry, resulting in 100 % efficiency, or it has two entries (because the second one was barely in reach), resulting in 50 % efficiency. Because of this effect, there can be no values in between. Therefore, the most meaningful part of the query is the upper right quadrant. Here we usually have ratios from 60 % to 90 %, most of the time above 80 %.

Second, we take a closer look at the ratio $\frac{\text{geometries in query box}}{\text{geometries in the R-tree}}$ shown on the left side of the figures. This ratio is especially important for the runtime of the algorithm. The lower it is, the fewer cases have to be checked. Varying the latitude has little effect on this ratio. The ratio is basically always below 10 % in all cases, which means that at most only a tenth of the comparisons have to be done and 90 % of the comparisons can be skipped. This effect is even greater if both subresults are large. Then for all points of the subresult that wasn't used to build the R-tree, 90 % of comparisons can be saved. Looking at the heatmaps, there is only one case where not 90 % of the comparisons can be saved. This is the case when the maximum distance is 10000000. Then the query box contains more than 50 % of the points. However, looking at the ratio $\frac{\text{size result table}}{\text{geometries in query box}}$ for these cases, one can see that basically all points in the query box are also contained in the result (except for very low latitudes, where an unefficient edge case of an edge case occurs (the pole is barely reached with a large max distance value). If one of these conditions would not be met, then basically all points of the query box would be included in the result). Therefore, it is not a weakness of the algorithm to have a high ratio for the max distance of 10,000,000, because those points need to be checked, since basically all of them are contained in the result. In fact, it would be very bad if the ratio for this max distance value were low, because that would mean that geometries would have been ignored, even though they should have been included in the result.

In summary, the query box is a very efficient way to filter the geometries. In almost all cases, at least 90 % of the comparisons could be saved, and of the 10 % that were compared, most end up in the final result.

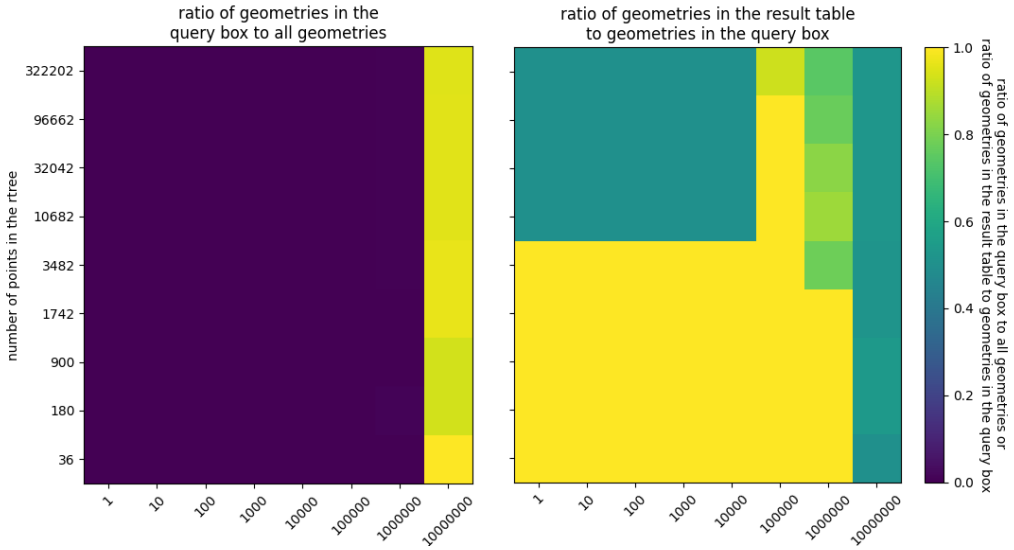


Figure 9.2.: This figure shows the analysis of the query box efficiency for a latitude of 10. The left side shows the ratio $\frac{\text{geometries in query box}}{\text{geometries in R-tree}}$ and the right side shows the ratio $\frac{\text{size result table}}{\text{geometries in query box}}$.

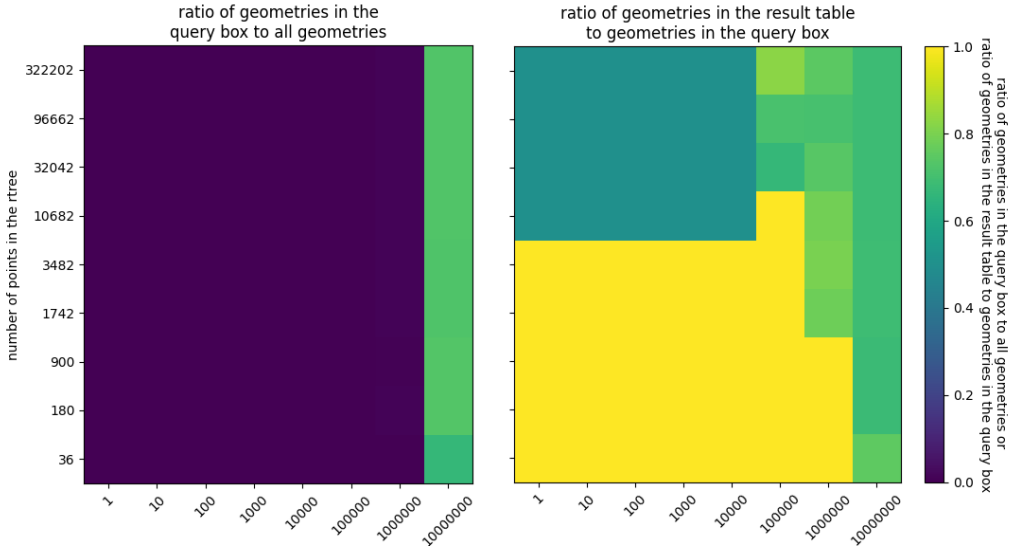


Figure 9.3.: This figure shows the analysis of the query box efficiency for a latitude of 50. The left side shows the ratio $\frac{\text{geometries in query box}}{\text{geometries in R-tree}}$ and the right side shows the ratio $\frac{\text{size result table}}{\text{geometries in query box}}$.

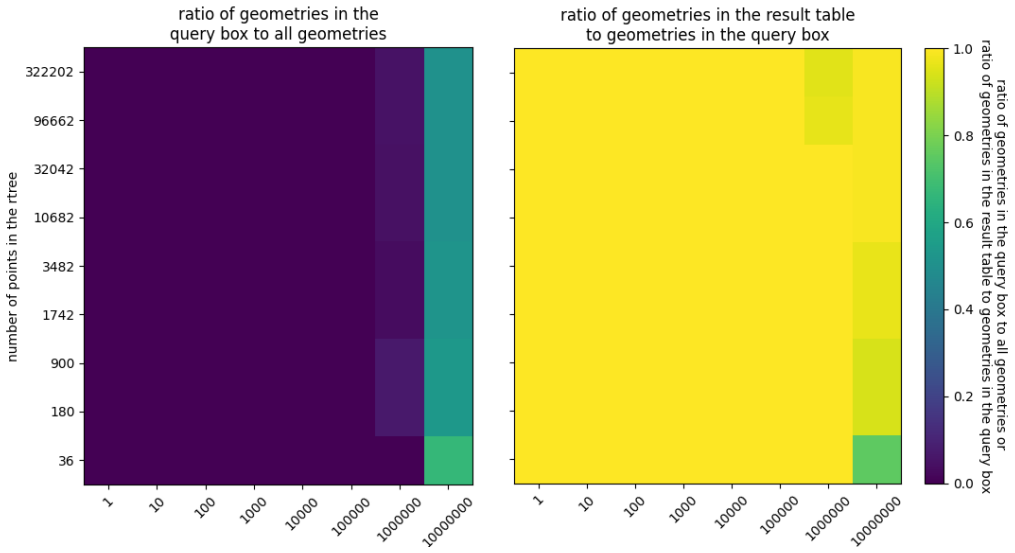


Figure 9.4.: This figure shows the analysis of the query box efficiency for a latitude of 90. The left side shows the ratio $\frac{\text{geometries in query box}}{\text{geometries in R-tree}}$ and the right side shows the ratio $\frac{\text{size result table}}{\text{geometries in query box}}$.

9.5. Analysis on the OpenStreetMap data of Germany

This section analyzes the algorithms on the OpenStreetMap dataset of Germany. This dataset is much larger than the synthetic dataset introduced above. It has the following statistics:

- Number of triples: 30,903,390,824
- Numer of subjects: 1,455,708,653
- Number of predicates: 30,032
- Number of objects: 3,073,801,549

9.5.1. Pairs of university building

In this analysis, the query was to find all pairs of university buildings in Germany that are at most a certain distance apart. This distance was given the values 0.01 km, 0.1 km, 1 km, 10 km, 100 km and 1000 km. Each tested distance was queried with the bounding box algorithm with and without midpoint approximation and with the baseline algorithm. An example query with a maximum distance of 1 km and using the bounding box algorithm can be seen in Figure 9.5. The results of all

queries can be seen in Table 9.1[2] and are visualized in Figure 9.6. The heatmaps in Figure 9.6 all have three rows, showing the size, the total computation time, and the time taken just for the spatialJoin operation. The rows show the distance the pairs of university buildings were allowed to be from each other. Since the sizes and times have different orders of magnitude, a logarithmic scale was chosen. Looking at the times of the spatialJoin operation of the BoundingBox algorithm without midpoint approximation, it can be seen that with each increase in the size of the distance, the time for the spatialJoin operation also increases significantly, but not by exactly one order of magnitude. When using the BoundingBox algorithm with midpoint approximation, this increase is much less than an order of magnitude and the times are much smaller than the times of the BoundingBox algorithm without midpoint approximation. From this we can already conclude that the exact distance calculation is expensive, more about this later. If we compare the times for the SpatialJoin operation of the BoundingBox algorithm with the Baseline algorithm, we can clearly see that the BoundingBox algorithm is much faster than the Baseline algorithm. The times of the Baseline algorithm don't depend on the maximum distance, so the outperformance of the BoundingBox algorithm increases the smaller the maximum distance becomes. We can also see that for the baseline algorithm, basically the entire time of the query computation was spent on the spatialJoin operation. This isn't the case when looking at the BoundingBox operation. Comparing the total time for the query and the time for the spatialJoin operation, we can see that there is a huge difference in most cases, especially when the midpoint approximation is used. Looking more closely at the total time for the BoundingBox algorithm with midpoint approximation, we can see that the time does not seem to depend on the maximum distance. This indicates that another part of the query is the bottleneck and not the spatialJoin operation. To analyze the query, we can take a closer look at the ExecutionTree and analyze the timings of each operation. An example ExecutionTree is shown in Figure 9.7. In this illustration of the ExecutionTree, operations that took a long time are highlighted in red. Here we can see that the two operations that took the most time were a join operation with 18 seconds and a sort operation with 15 seconds. When comparing this to the 0.285 seconds of the spatialJoin operation, it can be seen why the total time of the query does not seem to depend on the maximum distance. One last note: I don't know why the QueryPlanner builds the ExecutionTree the way it does. As you can see in the query in Figure 9.5, the query for building1, geo1, and wkt1 is the same as for building2, geo2, and wkt2. Even though it is the same, the QueryPlanner planned them differently. The left child of the SpatialJoin operation of the ExecutionTree in Figure 9.7 shows a very efficient ordering of the operations to get the first subresult of the spatialJoin operation. Why the QueryPlanner did not plan the other subresult equally efficiently should be evaluated. Finally, I want to take a look at the result size of the spatialJoin operation. The size of the result query increases

[2]The appendix contains the ExecutionTree with timing information for all queries in subsubsection A.3.1.1

as the maximum distance parameter increases. With each increase in the maximum distance, the size of the result table also increased by about an order of magnitude. That's to be expected. But if you take a closer look at Table 9.1 and compare the size of the result for the BoundingBox algorithm without midpoint approximation with the size of the result for the BoundingBox algorithm with midpoint approximation, we can notice a slight difference in most values. For a maximum distance of 100 km the difference is only 0.09 %, but for a maximum distance of 0.01 km the result size of the spatialJoin without midpoint approximation is more than twice as large as the one with midpoint approximation. The reason for this is that when the maximum distance and the size of the geometry of the object, in this case the university building, are of the same order of magnitude, the approximation is not good. If we approximate two buildings that are 5 by 5 meters with their center, then the walls of the building can be at most 5 meters apart, otherwise their centers are more than 10 meters apart, which would exclude them from the result. If the approximation isn't used, the walls of the two buildings can be up to 10 meters apart and still be in the result. This also explains why the difference for the maximum distance for 100 km is so small. There are hardly any pairs of university buildings that are less than 100 km apart when measured wall to wall, but more than 100 km apart when measured center to center. In this case, the size of the university building is small enough compared to the up to 100 km distance between the buildings that this approximation is very good. Another example where the midpoint approximation would fail badly is if you take the river Rhine. It runs from the south of Germany to the North Sea in the Netherlands. If the river were approximated by the center of its bounding box, the center would probably be close to the center of the western border of Germany. When calculating the distance from, say, the university library of freiburg to the Rhine, the midpoint approximation would give a very poor estimate and should not be used. These two examples show that the midpoint approximation can be good or bad depending on the geometry and its size compared to the maximum distance. Since computing the true distance is much more expensive than computing the distance using the midpoint approximation, one should consider whether the current query requires the true distance or whether the midpoint approximation is good enough.

9.5.2. Comparison to the OSM2RDF paper

In this section I want to compare the performance of the SpatialJoin algorithm with the OSM2RDF tool [3]. Since the OSM2RDF tool and the SpatialJoin operation do not implement the same concept, but only share some common features, this comparison is not meant to imply that we should replace one with the other. The "contained in" and "intersects" predicates that the OSM2RDF tool adds to the dataset cannot be computed by the SpatialJoin operation, and the OSM2RDF tool does not allow you to specify a maximum distance between objects. The OSM2RDF paper analyzed it's performance based on four queries. These were

```
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 ?building1 ?building2 ?dist WHERE {
  ?building1 osmkey:building "university" .
  ?building1 geo:hasGeometry ?geo1 .
  ?geo1 geo:asWKT ?wkt1 .
  ?building2 osmkey:building "university" .
  ?building2 geo:hasGeometry ?geo2 .
  ?geo2 geo:asWKT ?wkt2 .
  Service spatialSearch: {
  _:config spatialSearch:algorithm spatialSearch:boundingBox;
  spatialSearch:left ?wkt1;
  spatialSearch:right ?wkt2;
  spatialSearch:maxDistance 1000;
  spatialSearch:bindDistance ?dist .
  }
}
```

Execute | Download | Share | Format | Clear cache | Analysis | Examples | 3. Context sensitive suggestions

Automatically add names to result

Figure 9.5.: Example SPARQL query to get pairs of university buildings, which are at most 1 km apart from each other.

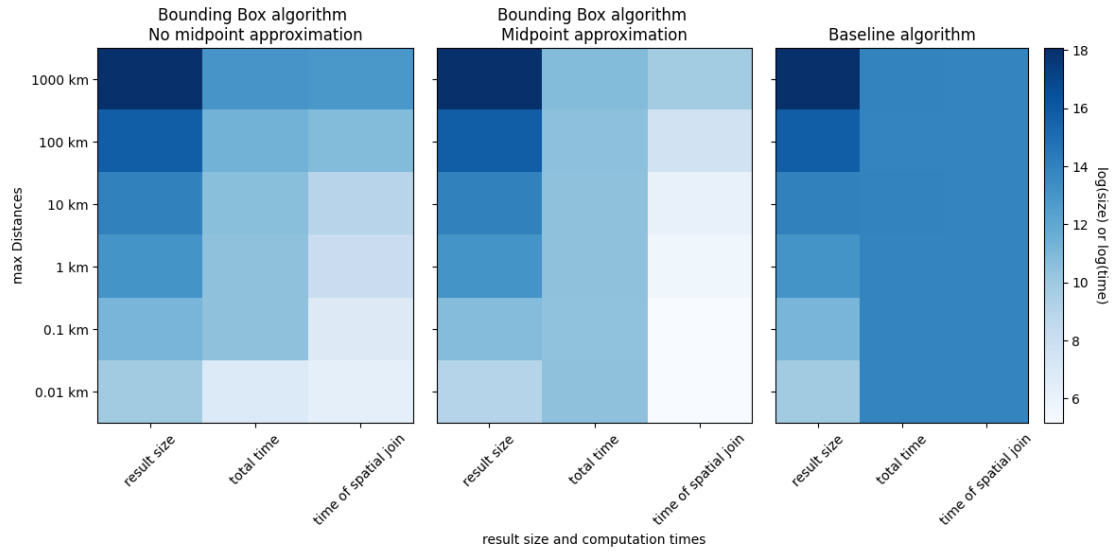


Figure 9.6.: Visualization of the times and result sizes for the university pair query for different maximum distances computed with different algorithms. Note the logarithmic scaling.

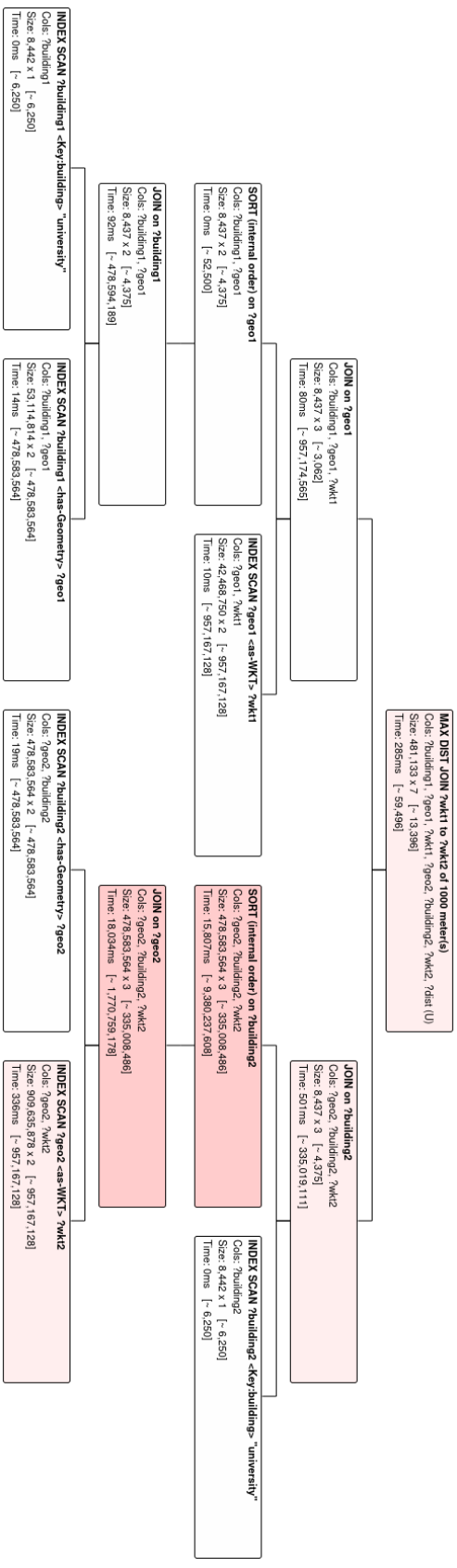


Figure 9.7.: This figure shows an example ExecutionTree for the BoundingBox algorithm with no midpoint approximation and a maximum distance of 1 km

Table 9.1.: Results of the university pairs queries on the OpenStreetMap germany dataset. In the experiment column, BoundingBox or baseline refers to the used algorithm and "no midpoint" or "midpoint" gives the information, whether the midpoint approximation was used or not.

experiment	dist	size input	size result	total time in ms	SpatialJoin time in ms
BoundingBox, no midpoint	1000 km	8437 x 8437	70,812,225	438,670	403,025
BoundingBox, midpoint	1000 km	8437 x 8437	70,812,255	51,975	18,012
baseline, no midpoint	1000 km	8437 x 8437	70,812,255	1,089,868	1,055,074
BoundingBox, no midpoint	100 km	8437 x 8437	7,277,251	85,733	50,539
BoundingBox, midpoint	100 km	8437 x 8437	7,270,635	36,816	2,122
baseline, no midpoint	100 km	8437 x 8437	7,277,251	1,074,835	1,041,136
BoundingBox, no midpoint	10 km	8437 x 8437	1,208,191	41,545	7,701
BoundingBox, midpoint	10 km	8437 x 8437	1,205,023	34,972	471
baseline, no midpoint	10 km	8437 x 8437	1,208,191	1,077,733	1,044,641
BoundingBox, no midpoint	1 km	8437 x 8437	493,357	36,202	3,365
BoundingBox, midpoint	1 km	8437 x 8437	481,133	35,179	285
baseline, no midpoint	1 km	8437 x 8437	493,357	1,059,261	1,026,718
BoundingBox, no midpoint	0.1 km	8437 x 8437	72,317	35,657	958
BoundingBox, midpoint	0.1 km	8437 x 8437	48,357	34,135	177
baseline, no midpoint	0.1 km	8437 x 8437	72,317	1,071,177	1,038,522
BoundingBox, no midpoint	0.01 km	8437 x 8437	18,055	998	573
BoundingBox, midpoint	0.01 km	8437 x 8437	8,731	35,760	174
baseline, no midpoint	0.01 km	8437 x 8437	18,055	1,040,716	1,040,279

1. all university buildings
2. all university buildings in the bounding box of germany
3. all university buildings in the bounding box of Freiburg
4. all university buildings in Freiburg

The first query does not contain any distance information, so a SpatialJoin operation would not be part of it. Therefore, it can't be compared. The second query could be simulated (by picking a university building in the center of Germany and then making a query where you want to have every university building that is at most "radius of Germany" meters away from it. However, this would require the OSM data of the planet, otherwise the comparison would be unfair. Since the machine of the evaluation has only 128 GB RAM and other people are working on this machine in parallel, it was not possible to use the OSM data of the whole planet. Therefore, this query can't be compared either. The third query can be compared using the simulation approach mentioned above. We choose the university library in Freiburg and want to get all university buildings that are at most 25 km away from the university library. The fourth query can be simulated in the same way as the third query. Table 9.2 shows the results of this comparison[3]. Looking at the times you can see that all operations are quite fast, but the fastest is query four, which uses the pre-computed triples from the OSM2RDF tool. Query three, the bounding box algorithm without midpoint approximation, and the baseline algorithm all have similar total times. And the BoundingBox algorithm with midpoint approximation has the slowest time. This is a surprising result, since the midpoint approximation is supposed to make the algorithm faster at the expense of accuracy. Looking at the previous evaluation result, this is also clearly the case, this query is the exception. A possible reason could be that for very small sizes (as one child consists of only one entry: the university library of freiburg) the construction of the approximation takes longer than the computation of the actual distance. Another explanation could simply be random noise in the form of server load, interrupts, or operating system scheduling. To summarize this section, the best query times for contained in relations can be achieved using the pre-calculated triples. If the data set does not change often, or if you need to use contained in triples frequently, the pre-calculation time is definitely worth it, especially since it produces accurate results and does not simulate the contained in Freiburg relation over a building in Freiburg and a distance. If the dataset changes a lot and you do not need exact contained in predicates, then the SpatialJoin is a good alternative, since it does not require any additional pre-computation time and is also very fast - just not as fast as pre-computed triples. As mentioned earlier, this should not be seen as a replacement, since the data from the OSM2RDF tool is needed to get the geometry information into the knowledge graph. But one can choose to skip the step of pre-computing

[3]The ExecutionTree for each SpatialJoin operation can be seen in the appendix in subsubsection A.3.1.7. The times for the OSM2RDF queries were taken from the OSM2RDF paper [3]]

Table 9.2.: Comparison of the query of university buildings in the bounding box of Germany from the OSM2RDF paper with the simulated version of this query using the SpatialJoinAlgorithm

algorithm	total time	spatialJoin time	pre computation time
OSM2RDF Query 3	391 ms	-	16 hours
OSM2RDF Query 4	134 ms	-	16 hours
BoundingBox with midpoint approximation	646 ms	390 ms	-
BoundingBox no midpoint approximation	339 ms	106 ms	-
Baseline no midpoint approximation	350 ms	130 ms	-

the predicates if an approximation of the contained in predicate and the intersecting predicate[4] is good enough. For the OpenStreetMap dataset of the whole planet this could save 48 days of time, for the OpenStreetMap dataset of Germany this would save 16 hours of time.

9.5.3. Other queries

In this section, I just want to quickly show the computation times of the sample queries from the introduction chapter.

The results for the restaurant query from algorithm 1 and the antenna query from algorithm 2 are shown in Table 9.3[5]. The restaurant query is fast no matter which algorithm is chosen. As in the previous section, the time for the analysis with the midpoint approximation is slightly longer than the time without the approximation. This indicates that the approximation overhead does not pay off for very small input sizes. For the antenna query, the midpoint approximation does pay off, but only slightly. The runtime of the baseline algorithm for the antenna query was very bad. It timed out after 45 minutes. A look at the log showed that the algorithm was just at a progress of 16.5 %. If it had continued at that rate, the baseline algorithm would have taken 4.5 hours to compute the result. Since the BoundingBox algorithm can compute the result in less than 3 minutes, the Baseline algorithm is no match here.

[4]Via a distance of zero and no midpoint approximation

[5]The ExecutionTrees for each query are included in the appendix in subsection A.3.2 and subsection A.3.3

Table 9.3.: This table shows the query times for the examples from the introduction

query	time of BoundingBox algorithm with midpoint approximation	time of BoundingBox algorithm without midpoint approximation	time of baseline algorithm
restaurant query	2,780 ms	2,479 ms	4,662 ms
antenna query	153,481 ms	163,256 ms	$>$ 45 minutes

9.6. Runtime analysis of the code

The final analysis of the evaluation is the runtime analysis of the code. For this analysis, the synthetic dataset is used again. The queries were again sent by the query script as explained in section 9.2 and algorithm 15. The code of the SpatialJoin class was adapted by adding the statistical measurements. Each method that was measured had a current time measurement as the first statement in its method body. Then the last statement of the method (which could be the end of the method or the line before each return statement) was also a time measurement. The time difference was the time taken by the method (including the time taken by other functions called by that method). Since a method can be called very often, for example over 300,000 times, if it is called for each element of the input, the time difference of each function call was added to a class variable and logged once at the end. So we get the time spent in each logged method during the whole query. If the function takes a large amount of time, it may be because it is rarely called but is very expensive, or because it is called very often. When the time measurement was added, the code sometimes had to be changed. If the return statement called an expensive method, that time would not be logged. To avoid this and to also log the expensive call, the expensive call was assigned to a dummy variable. Then the time was measured and only then the dummy variable was returned. As mentioned in algorithm 15, the analysis was done for all size combinations of the children and many max distance values. In addition, the analysis was done once with the midpoint approximation enabled and once with the true distance computed. The memory limit was 100 GB, but this was not enough to compute each query. The queries with the following parameters had an out-of-memory exception (each entry had the exception regardless of whether midpoint approximation was enabled or not):

- maxDist: 1,000,000 size children: 322202 and 322202
- maxDist: 10,000,000 size children: 322202 and 322202
- maxDist: 10,000,000 size children: 322202 and 96662
- maxDist: 10,000,000 size children: 96662 and 96662
- maxDist: 10,000,000 size children: 322202 and 32042
- maxDist: 10,000,000 size children: 96662 and 32042

- maxDist: 10,000,000 size children: 322202 and 10682

The out-of-memory exception occurs when the max distances are very large and the size of the children is very large. Of the 720 queries tested, only the 14 queries mentioned above had an out of memory error (each one above once with midpoint approximation and once without).

A total of seven functions were measured. The reason why only these seven functions were measured is that the other functions are only called by one of these basic functions. Therefore, the time of the other functions is indirectly measured by measuring only these seven functions. Unfortunately, simply adding the time spent in each method does not give the total time of the algorithm. The reason is that some times are counted twice. For example, when constructing the R-tree, the method getRtreeEntry is called. Therefore, the time of getRtreeEntry for the entries of the smaller child is counted twice. The same happens with the computeDistance and computeDistanceArea functions. The time spent in the computeDistanceArea function is counted twice.

Since the runtime of the algorithm and the runtime of the different functions can differ by many orders of magnitude, depending on the size of the children, a logarithmic transformation was chosen for the visualization. This means that the time t spent in each function was the input to the function $\log_e(t)$. In order to easily compare the runtime of the algorithm with different inputs, each run of the algorithm is displayed in one line as a cumulative bar plot. The only disadvantage of this visualization is that you can't read the total time of the algorithm on the x-axis, because summing logarithms is like multiplying the original values. In the runtime of the code, however, the original values are added, not multiplied. Therefore, the absolute value of the x-axis is meaningless in this case, and only the absolute size of each color is meaningful (when this absolute value is put into the exponential function, the runtime of that part of the code is output). Changing the x-axis to a logarithmic scale and then displaying the cumulative values without transforming each with the logarithm would not be helpful in comparing the runtimes of the functions. The reason for this is that the x-axis is then non-linear, and the runtime of the first functions compared to the last functions would appear to be much greater than the difference really is. Using the procedures with the linear x-axis and the logarithmic times, the runtimes of the functions can be compared to each other and to other runs of the algorithm, which is the main reason for this evaluation. The transformed and untransformed versions of the data can be seen in the appendix in section A.4.

Figure 9.8 shows the results of this analysis for the query with a maximum distance of 100,000 meters between the geometries. It shows the logarithm of the time taken by each method, depending on the size of the input children and whether the midpoint approximation was used or not. If a color is not shown, it means that the time spent on that method was less than two microseconds[6]. Figure 9.9 shows the

[6]If the time was one microsecond, then $\log_e(1) = 0$ and therefore no time is shown. If the method

timing analysis with a linear x-axis.

In the following list, the impact of each method on the runtime is quickly analyzed.

- **getRtreeEntry:** This function is very expensive because it has to read the data from disk. This process has to be done for each geometry of the left entry and the right entry. Therefore, the larger the size of the children, the more time is spent in this function. The maximum distance value has no effect on this function, nor does the midpoint approximation. Compared to the other functions, this one is quite expensive. For small child sizes, this is often the only time that can be seen in the linear plot.
- **computeQueryBox:** This function is run for each of the geometries in the larger input. Since the runtime of a single run of this function is constant, the total time spent in this function depends only on the size of the larger child. Compared to the other functions, this one is really cheap.
- **computeDist:** This function is called for each pair in the query box. In the worst case, this can be the product of the sizes of the children. Depending on the maximum distance value, the query box will contain more or less points. Since the time of the computeDistArea function is also included, the runtime also depends on the midpoint approximation. If the midpoint approximation is disabled, the expensive computeDistArea function is added to the time of this function. Therefore, this function is faster when the size of the children and the maximum distance is small and the midpoint approximation is disabled. Compared to the other functions, it has a small impact when the midpoint approximation is enabled and a large impact when it is disabled.
- **computeDistArea:** This function will only be called if the midpoint approximation is disabled. Like the computeDist function, it is called for each pair of geometries in the query box, but with the restriction that one of them must be an area. The runtime of this function depends on the number of elements in the query box, which depends on the maximum distance. In addition, it depends on the size of the children, since it is called for each potential pair. Compared to the other functions, this function is very expensive.
- **addResultTableEntry:** This function is called for each pair of geometries included in the output. Therefore, it depends on the maximum allowed distance, but also on the size of the children. When there are few results in the table, the time spent on this function is so small that it doesn't even show up in the log-transformed times. But if the size of the children is large and the maximum allowed distance is also large, then this function is rather expensive.

is fast enough (or was not called, such as the computeDistArea function when the midpoint approximation was enabled) for the start time and end time to be the same time (with an accuracy of one microsecond), then log(0) would throw an error. When this happened, a zero was added by the exception handling

- **buildRtree:** This function depends on the size of the smaller child, as each geometry of the smaller child is inserted into the R-tree. It does not depend on the maximum allowed distance or the midpoint approximation. As the values have to be read from the disk, this function can take longer, if the smaller child is large, but compared to other functions, like the computeDistArea function, it is still quite cheap.
- **queryRtree:** This function gets called for each entry of the larger child. Depending on the maximum allowed distance and the size of the R-tree (the size of the smaller child), it needs more or less time, as more or less values from the R-tree need to be returned. Compared to the other functions it is cheap, when the size of the children and the maximum distance is small, when they are large its time is only rather cheap.

Comparing the runtime of queries with midpoint approximation to queries without midpoint approximation shows that the midpoint approximation is very efficient. The results for all queries can be seen in the appendix in section A.4. There they are presented in logarithmic and normal plots.

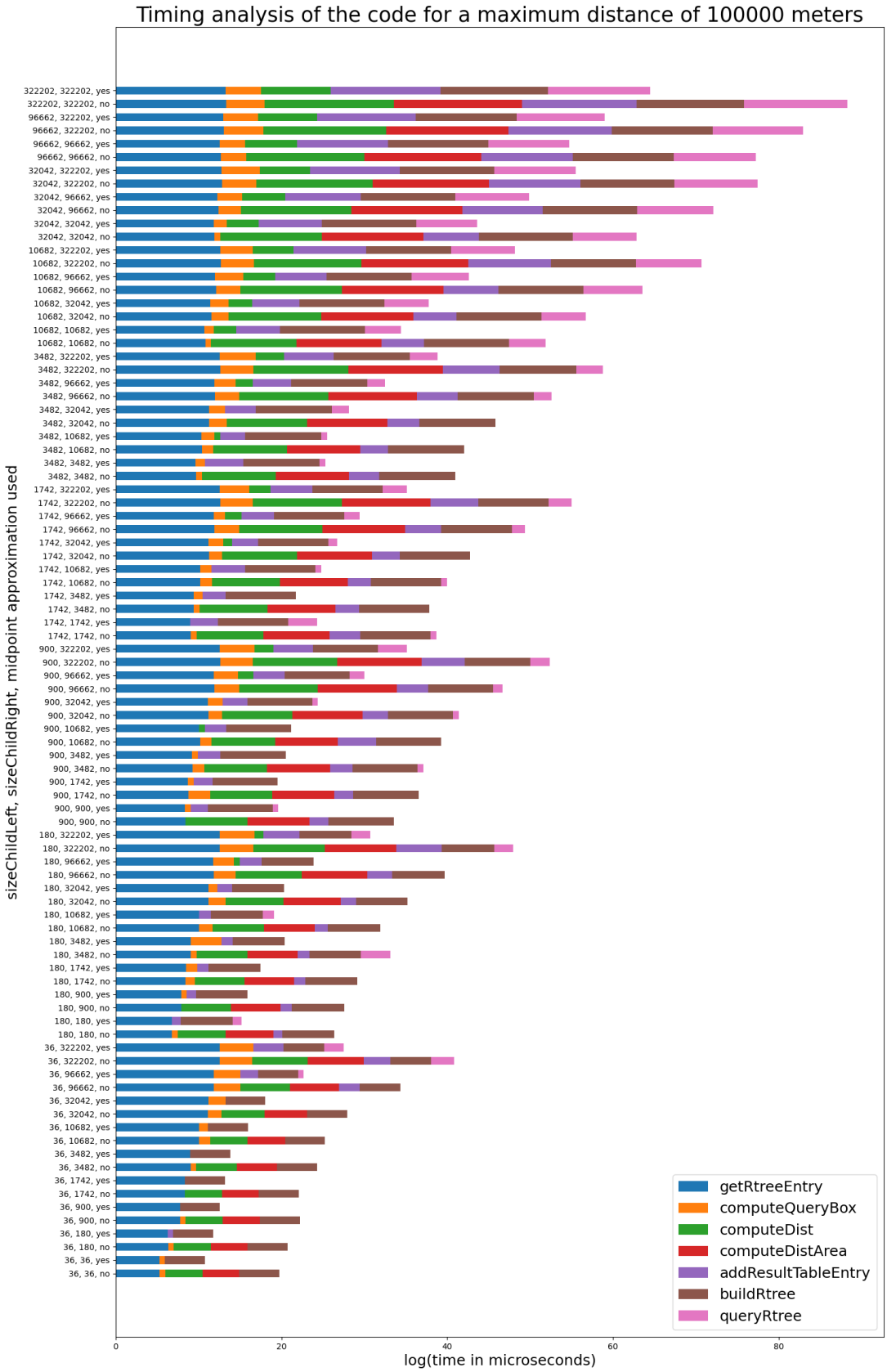


Figure 9.8.: This figure shows the times spent in different methods for the query with a maximum distance of 100000 meter, depending on the size of the children and if the midpoint approximation is used or not. The x axis is logarithmic

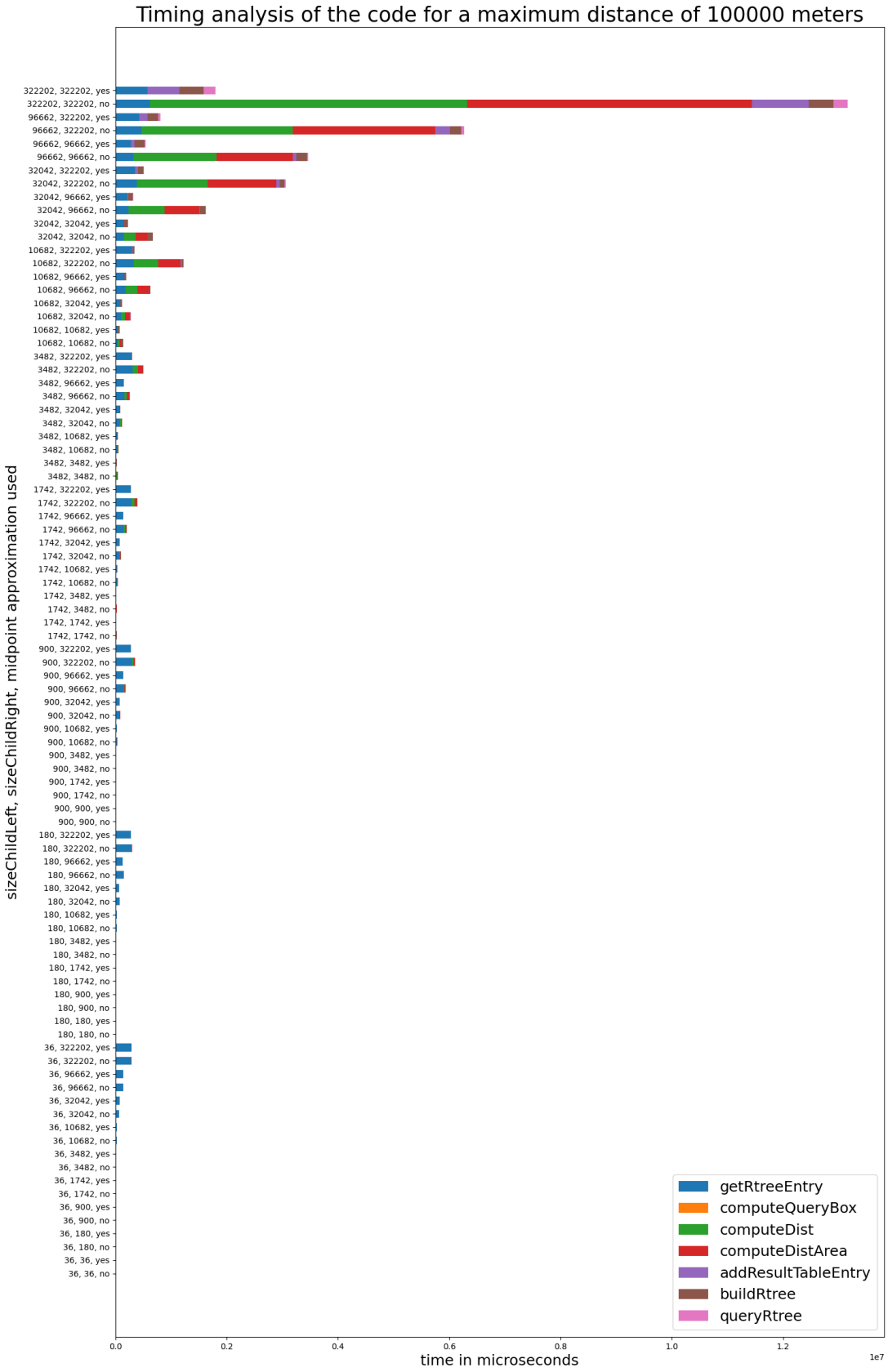


Figure 9.9.: This figure shows the times spent in different methods for the query with a maximum distance of 100000 meter, depending on the size of the children and if the midpoint approximation is used or not.

10. Conclusion and Outlook

The last section of this thesis will briefly conclude the master thesis and give an outlook on improvements that could be made to make the algorithm even better than it already is.

10.1. Conclusion

During this thesis, I developed an efficient algorithm to compute the SpatialJoin operation, which computes all pairs of geometries that are at most a user-defined distance apart from each other. The algorithm was successfully integrated into the search engine QLever and the QueryPlanner of QLever was adapted to correctly consider the special case of the SpatialJoin algorithm. Both the algorithm itself and the infrastructure changes were developed under the highest coding quality standards. The code itself has been analyzed by a reviewer, a static code analysis, and a format checker. The changes in the infrastructure and the algorithm itself are confirmed to work properly by more than 1600 tests, which check the correct functioning of the normal use cases, as well as the edge cases. The algorithm has been thoroughly analyzed and proven correct on a theoretical level. Last but not least, the algorithm has undergone a practical evaluation on a synthetic dataset developed for the purpose of testing and analyzing this algorithm. The algorithm was also tested on a real dataset, the OpenStreetMap of Germany. For both tests, the BoundingBox algorithm was compared to the baseline algorithm. The BoundingBox algorithm outperformed the baseline algorithm in all cases, especially as the input size increased.

10.2. Outlook

The algorithm can still be improved to give more accurate results or to give the same results faster. The following list gives some ideas that can be implemented in the future:

- In its current form, the algorithm only considers the latitude and longitude of the geometries. The third dimension, height, is still missing. The algorithm could be improved to include height data so that the distance between geometries on skyscrapers or cliffs and objects on the ground is correctly calculated.

- The second part of the algorithm, after building the R-tree, can be parallelized to reduce the runtime significantly. The R-tree is queried with each row of the second input child. The computation of the query box, the querying of the R-tree with the query box, and the computation of the distance are independent for each row. Therefore, it could easily be parallelized.
- The query box could be split into two boxes to exclude even more points that don't need to be checked. Suppose we have a point on the northern hemisphere for which we want to create a query box. In the current implementation, the farthest longitude that can be reached is always above the point. Therefore, if we were to split the current query box into two parts, we could shrink the bottom part because fewer longitudes could be reached below the point.
- To avoid loading all geometry data from disk, which is a really slow process, we could store a bounding box in the IDs to have them always in main memory. This would make building the R-tree much faster, since no data needs to be loaded from disk, and querying the R-tree with the other input would also be faster, since those geometries would not need to be loaded as well. Due to the limited number of 60 data bits the precision would not be perfect. The spatial resolution would be about $\frac{circumferenceEarth}{2^{15}} = 1.2$ km per coordinate. This is good enough to exclude a lot of results. It will not exclude as many results as the current method, but checking a few more results is probably faster than reading all the geometries from disk. Another option would be to store the bounding box in two IDs. Then each ID would contain a GeoPoint. One GeoPoint would represent the lower left corner of the bounding box and the other the upper right corner. Since GeoPoints are already implemented, this extension should be quite easy.
- The last option I want to present in this outlook is the option to extend this algorithm to ellipsoids of revolution like the Earth. Ellipsoids of revolution are three dimensional, but can be generated by a two dimensional ellipse rotated around an axis. Therefore an ellipsoid of revolution has only two axes (see Figure 5.1). To extend the presented algorithm to ellipsoids of revolution, one simply scales the larger axis until it has the size of the smaller axis. After this scaling process, we have constructed a perfect sphere. This scaling process does not increase the size between any points on the ellipsoid, many points now have a smaller distance compared to the unscaled ellipsoid. If we now run the presented algorithm on this smaller sphere, the filtering process of the algorithm (the query box constructed with $maxDist$) will return at least as many results as before (since the scaled version is smaller, the same distance can reach more points (or at least the same number of points on parts of the sphere not affected by the scaling)). So we reach all points and some more (so the filtering is a bit less effective), but no points are missed. After filtering, the distance is calculated for each pair individually. Here we have to calculate the distance on the unscaled ellipsoid using an updated formula that applies to ellipsoids. With these small adjustments, the algorithm can be easily adapted

to ellipsoids. Since the formulas for ellipsoids are more complicated and take longer to compute, the algorithm will be slower. Here you have to decide which trade-off between accuracy and speed you want to make. For most cases the assumption that the earth is a perfect sphere is good enough and will give good results.

A. Appendix

A.1. Build R-tree analysis

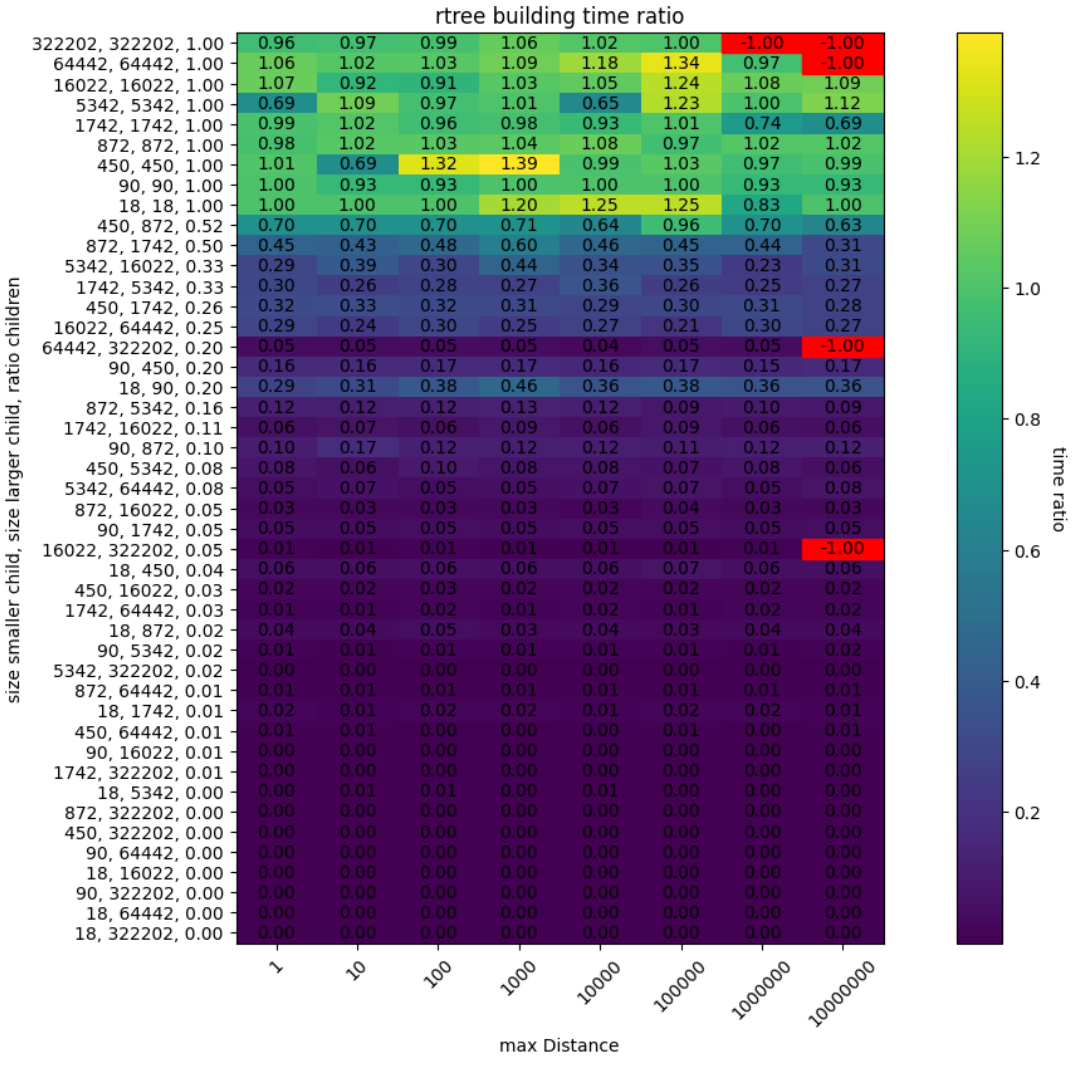


Figure A.1.: This figure shows the ratio of the time needed for building the rtree $(\frac{\text{time building the smaller rtree}}{\text{time building the larger rtree}})$.

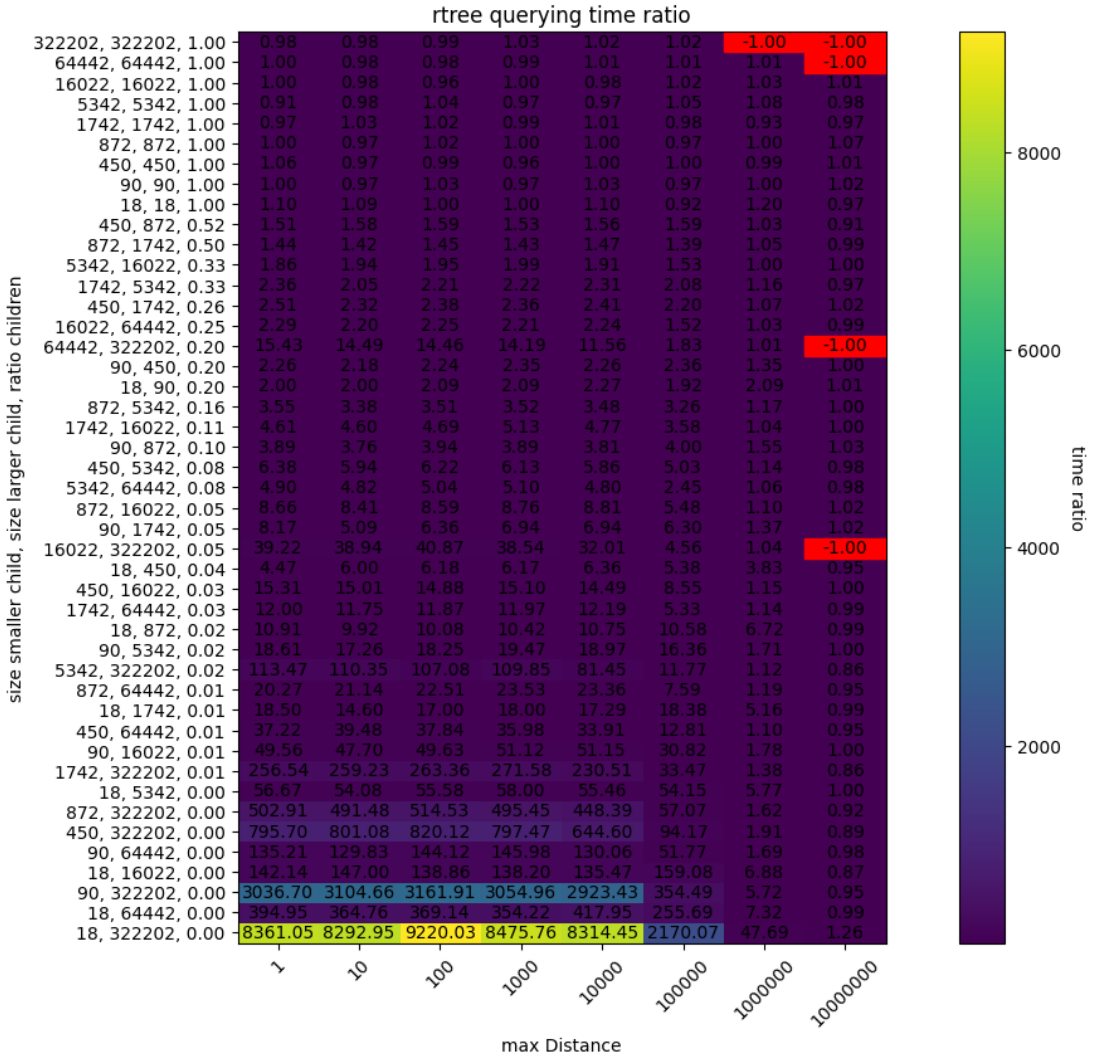


Figure A.2.: This figure shows the ratio of the time needed for querying $(\frac{\text{time querying the smaller rtree}}{\text{time querying the larger rtree}})$.

A.2. Query box efficiency

This section shows the results of the analysis for the query box efficiency for the latitudes, which were not shown in the main chapter about the query box efficiency.

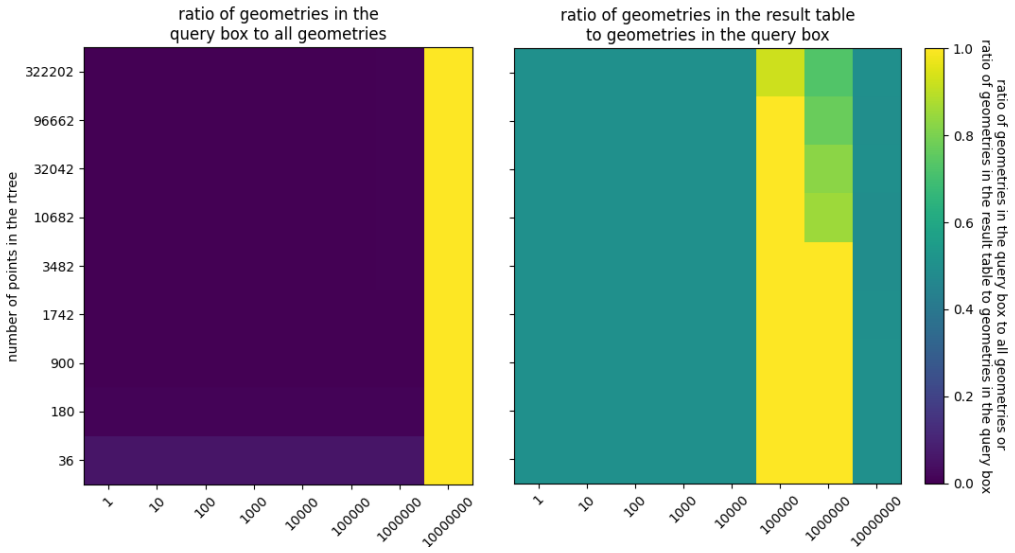


Figure A.3.: This figure shows the analysis of the query box efficiency for a latitude of 0. The left side shows the ratio $\frac{\text{geometries in query box}}{\text{geometries in rtree}}$ and the right side shows the ratio $\frac{\text{size result table}}{\text{geometries in query box}}$.

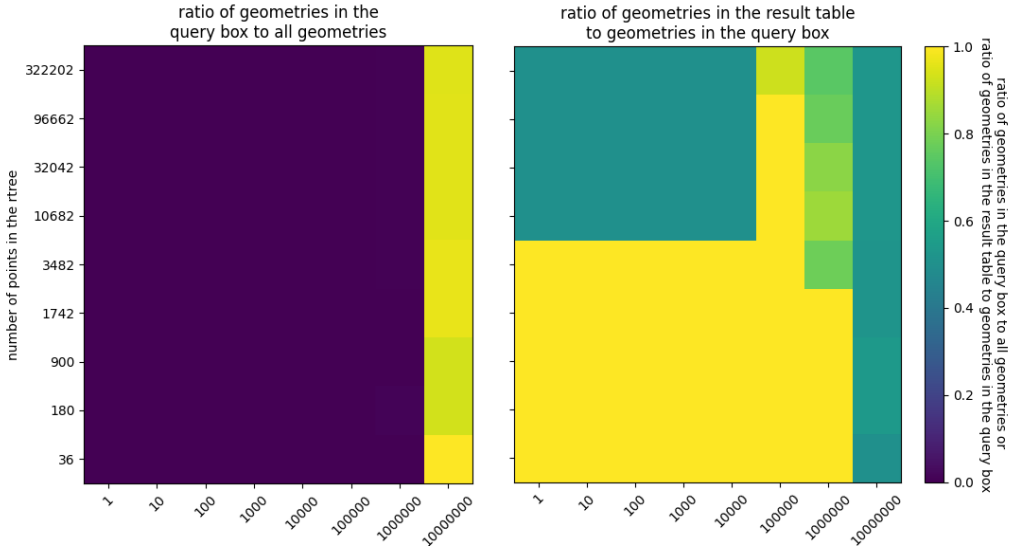


Figure A.4.: This figure shows the analysis of the query box efficiency for a latitude of 10. The left side shows the ratio $\frac{\text{geometries in query box}}{\text{geometries in rtree}}$ and the right side shows the ratio $\frac{\text{size result table}}{\text{geometries in query box}}$.

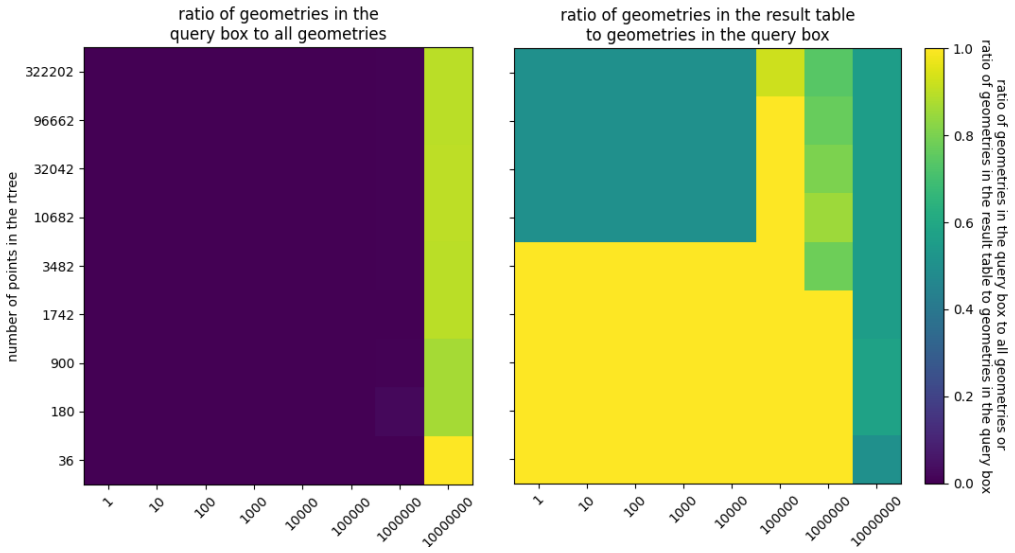


Figure A.5.: This figure shows the analysis of the query box efficiency for a latitude of 20. The left side shows the ratio $\frac{\text{geometries in query box}}{\text{geometries in rtree}}$ and the right side shows the ratio $\frac{\text{size result table}}{\text{geometries in query box}}$.

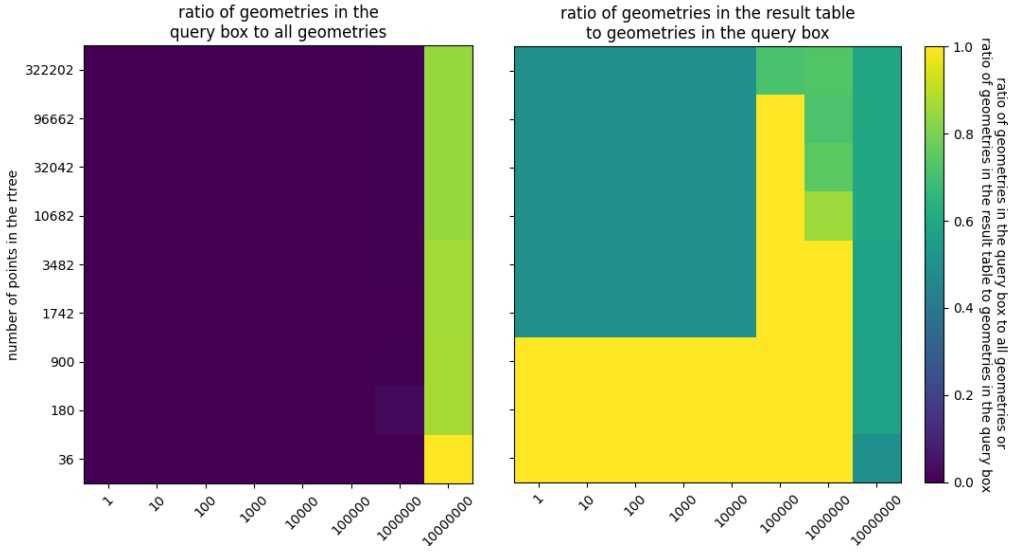


Figure A.6.: This figure shows the analysis of the query box efficiency for a latitude of 30. The left side shows the ratio $\frac{\text{geometries in query box}}{\text{geometries in rtree}}$ and the right side shows the ratio $\frac{\text{size result table}}{\text{geometries in query box}}$.

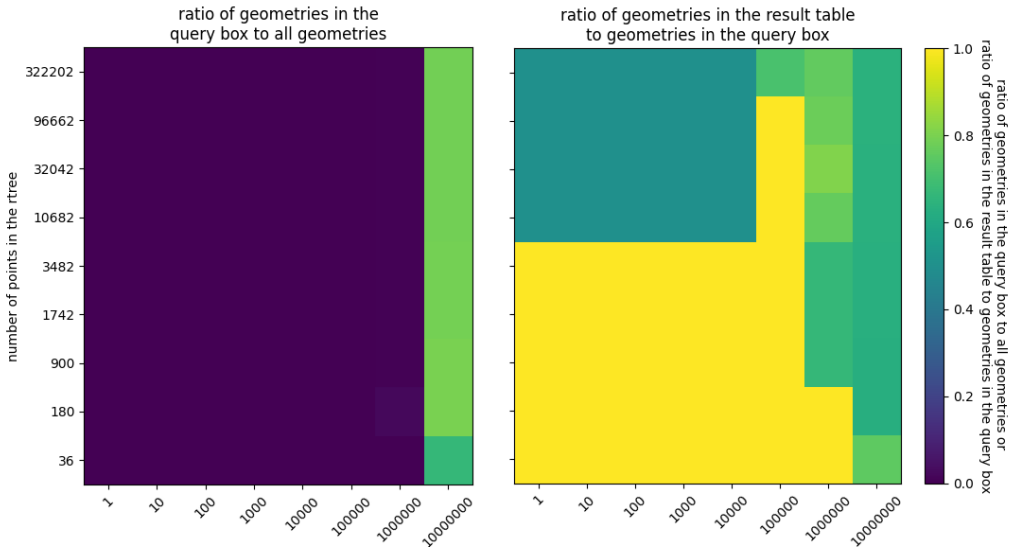


Figure A.7.: This figure shows the analysis of the query box efficiency for a latitude of 40. The left side shows the ratio $\frac{\text{geometries in query box}}{\text{geometries in rtree}}$ and the right side shows the ratio $\frac{\text{size result table}}{\text{geometries in query box}}$.

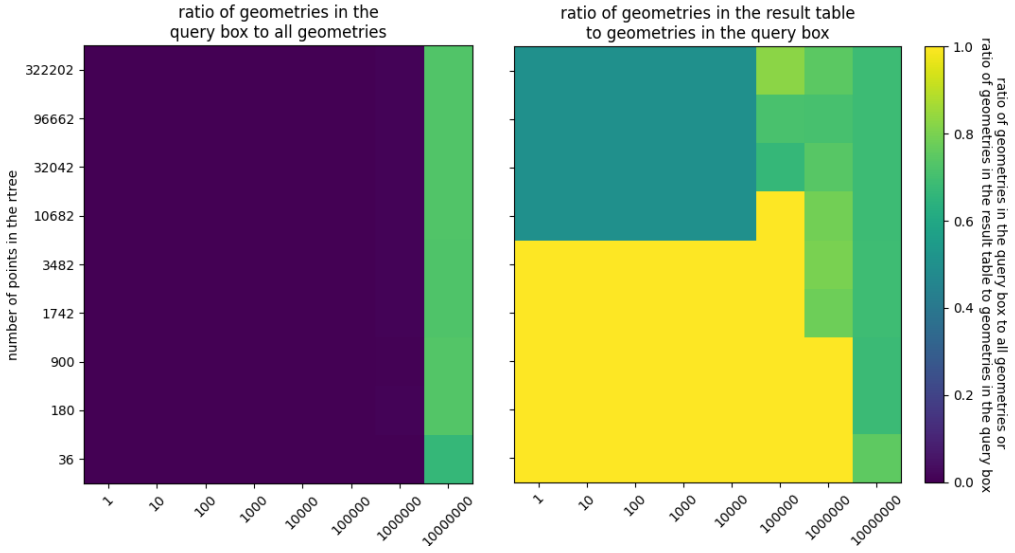


Figure A.8.: This figure shows the analysis of the query box efficiency for a latitude of 50. The left side shows the ratio $\frac{\text{geometries in query box}}{\text{geometries in rtree}}$ and the right side shows the ratio $\frac{\text{size result table}}{\text{geometries in query box}}$.

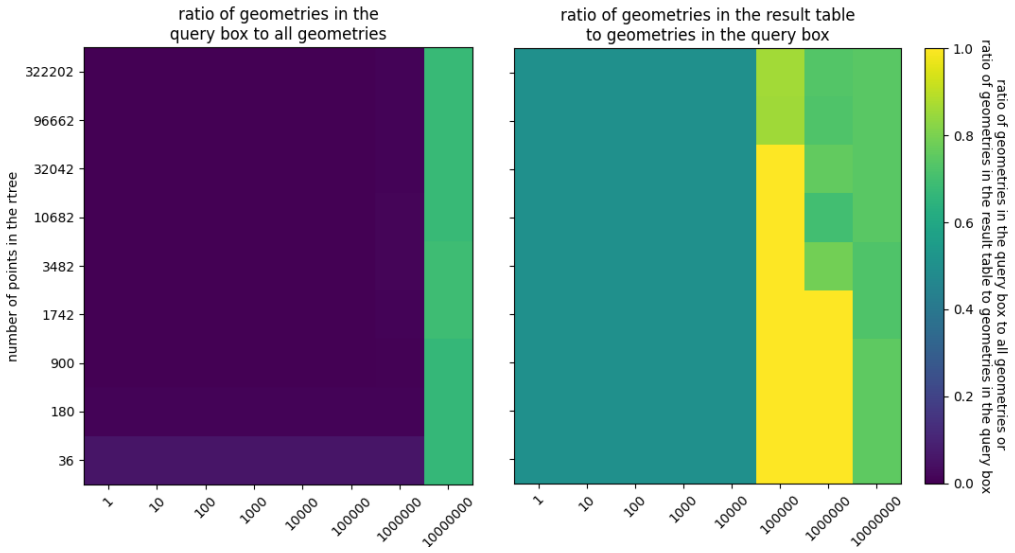


Figure A.9.: This figure shows the analysis of the query box efficiency for a latitude of 60. The left side shows the ratio $\frac{\text{geometries in query box}}{\text{geometries in rtree}}$ and the right side shows the ratio $\frac{\text{size result table}}{\text{geometries in query box}}$.

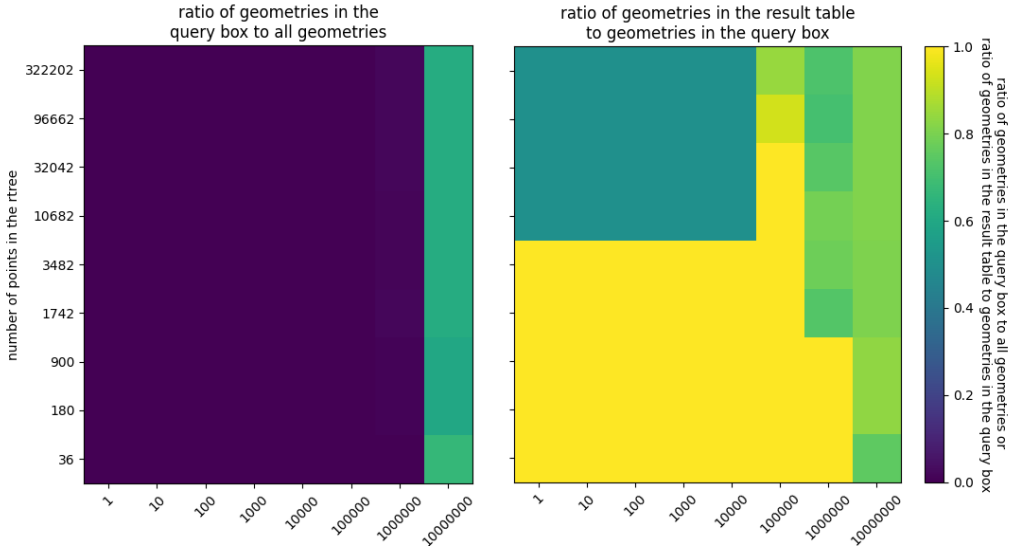


Figure A.10.: This figure shows the analysis of the query box efficiency for a latitude of 70. The left side shows the ratio $\frac{\text{geometries in query box}}{\text{geometries in rtree}}$ and the right side shows the ratio $\frac{\text{size result table}}{\text{geometries in query box}}$.

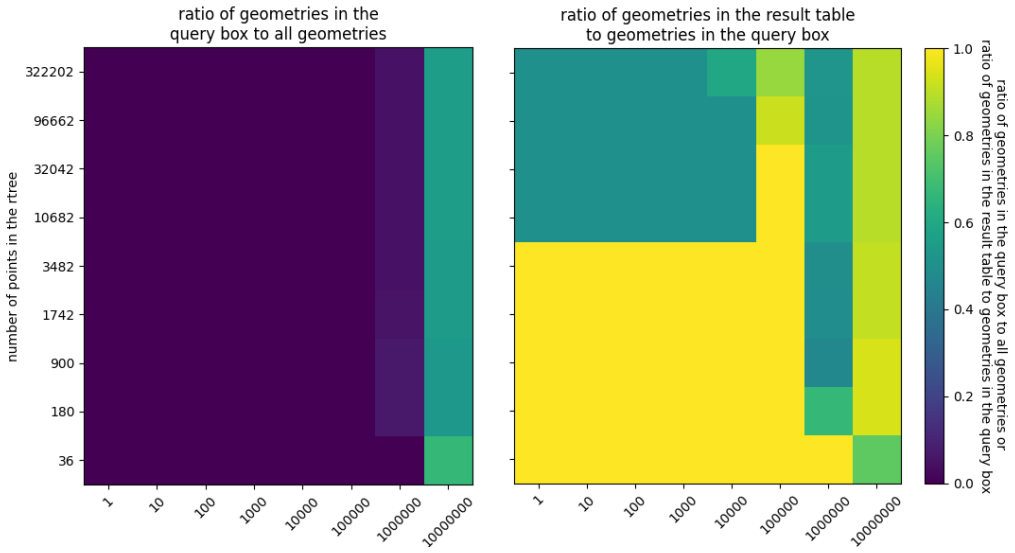


Figure A.11.: This figure shows the analysis of the query box efficiency for a latitude of 80. The left side shows the ratio $\frac{\text{geometries in query box}}{\text{geometries in rtree}}$ and the right side shows the ratio $\frac{\text{size result table}}{\text{geometries in query box}}$.

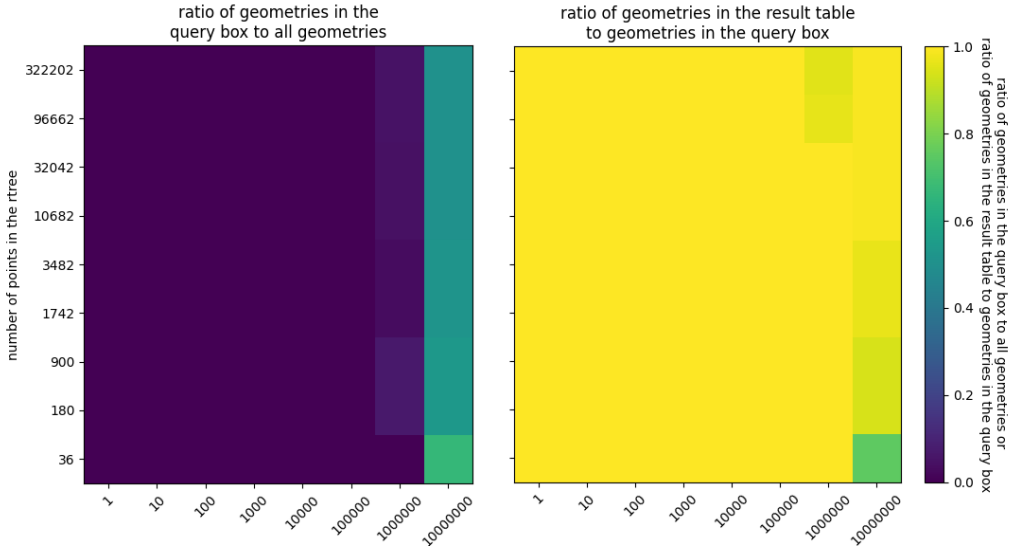


Figure A.12.: This figure shows the analysis of the query box efficiency for a latitude of 90. The left side shows the ratio $\frac{\text{geometries in query box}}{\text{geometries in rtree}}$ and the right side shows the ratio $\frac{\text{size result table}}{\text{geometries in query box}}$.

A.3. Complete data from the OpenStreetMap Germany evaluation

This section of the appendix contains all of the data from the evaluation on the osm germany knowledgegraph.

A.3.1. University queries

A.3.1.1. University buildings at most 1000 km apart from each other

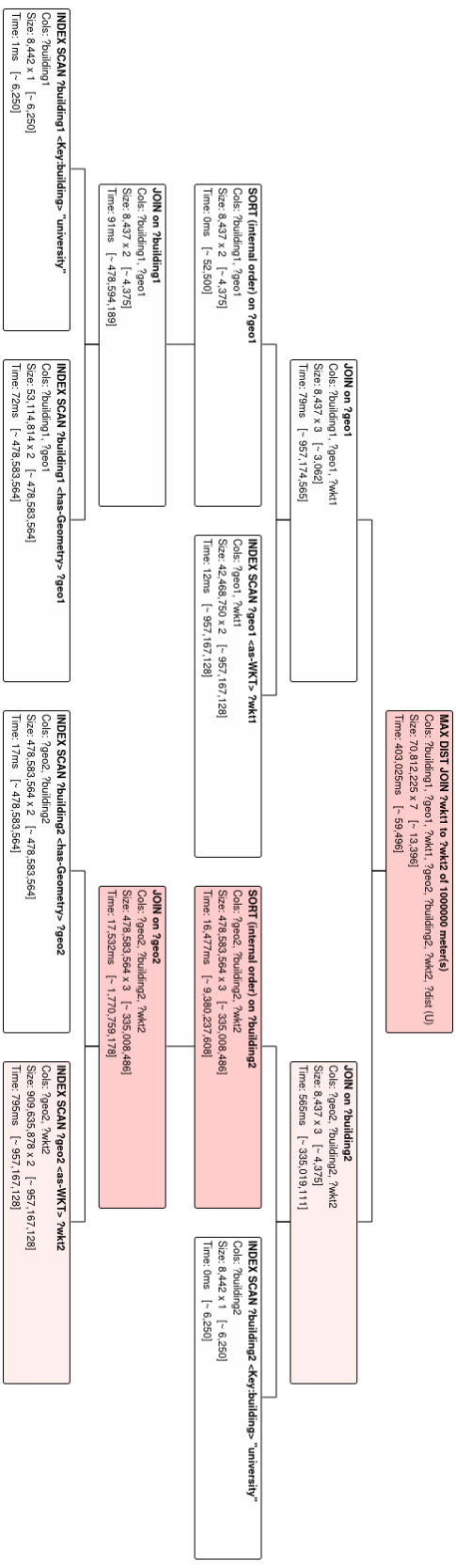


Figure A.13.: Analysis of the runtimes for the search for pairs of university buildings, which are at most 1000 km meters apart from each other. The result has been calculated using the boundingBox algorithm with no midpoint approximation

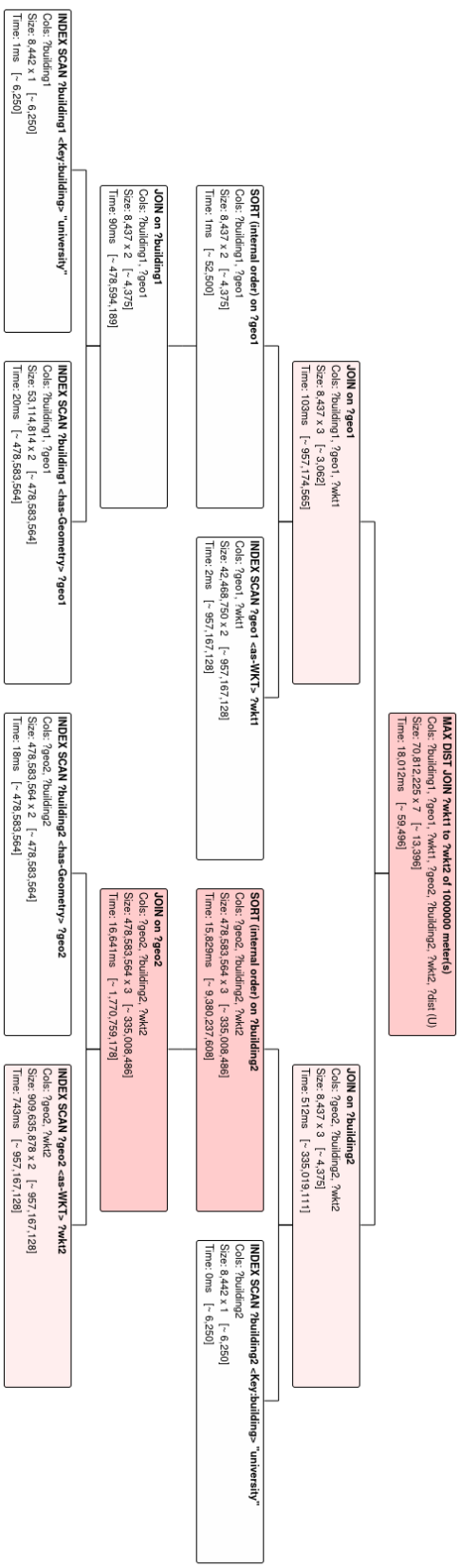


Figure A.14.: Analysis of the runtimes for the search for pairs of university buildings, which are at most 1000 km meters apart from each other. The result has been calculated using the boundingBox algorithm with the midpoint approximation.

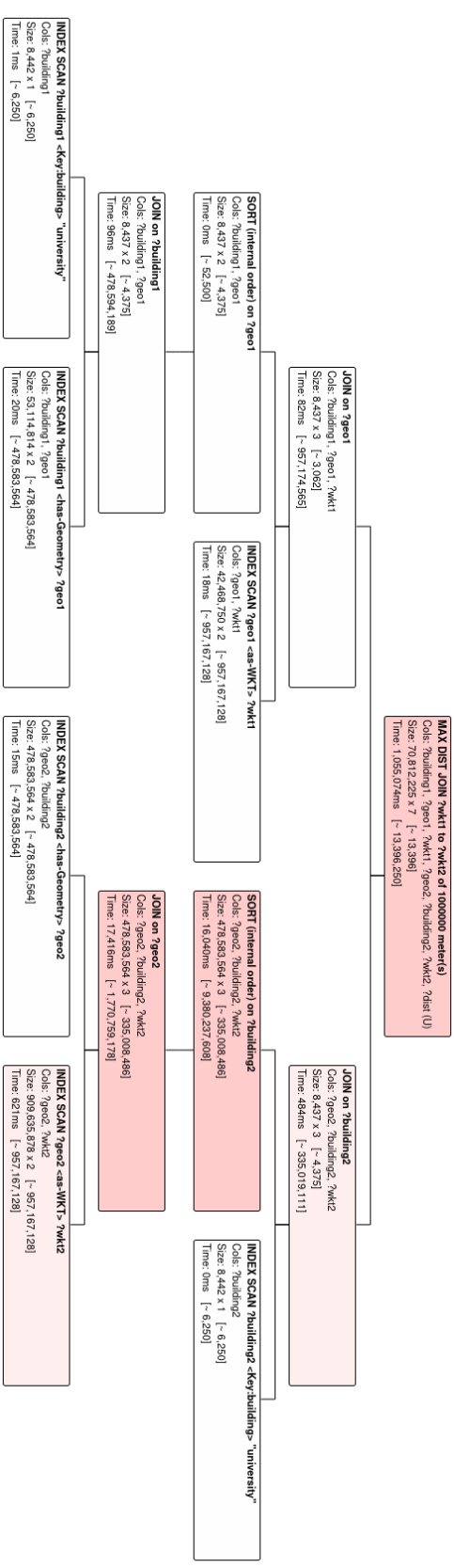


Figure A.15.: Analysis of the runtimes for the search for pairs of university buildings, which are at most 1000 km meters apart from each other. The result has been calculated using the baseline algorithm with no midpoint approximation

A.3.1.2. University buildings at most 100 km apart from each other

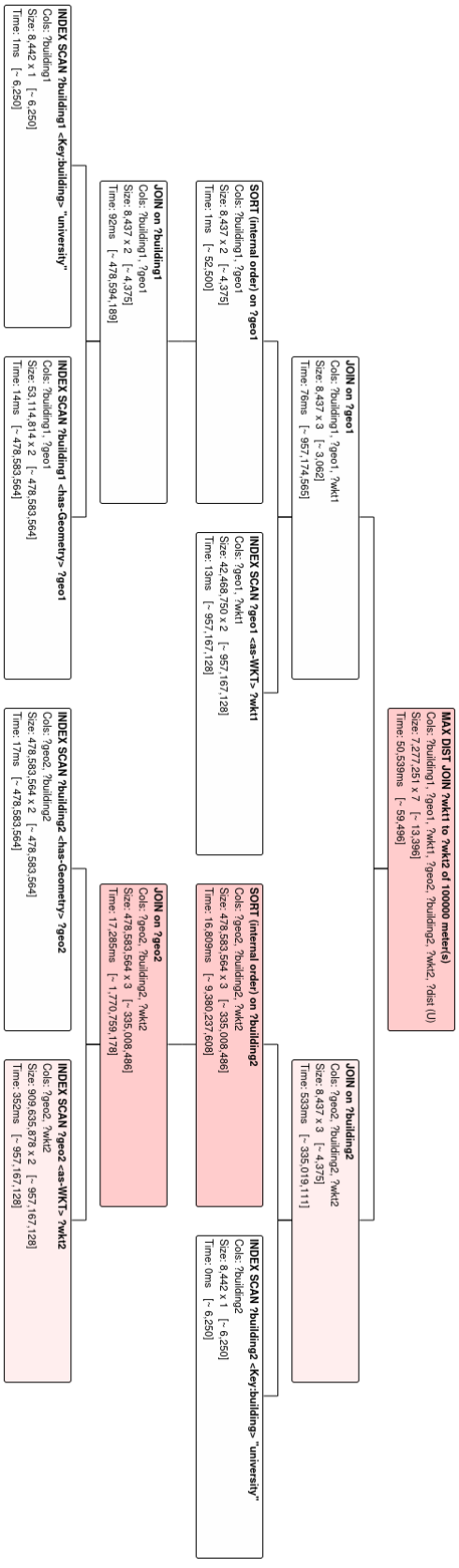


Figure A.16.: Analysis of the runtimes for the search for pairs of university buildings, which are at most 100 km meters apart from each other. The result has been calculated using the boundingBox algorithm with no midpoint approximation

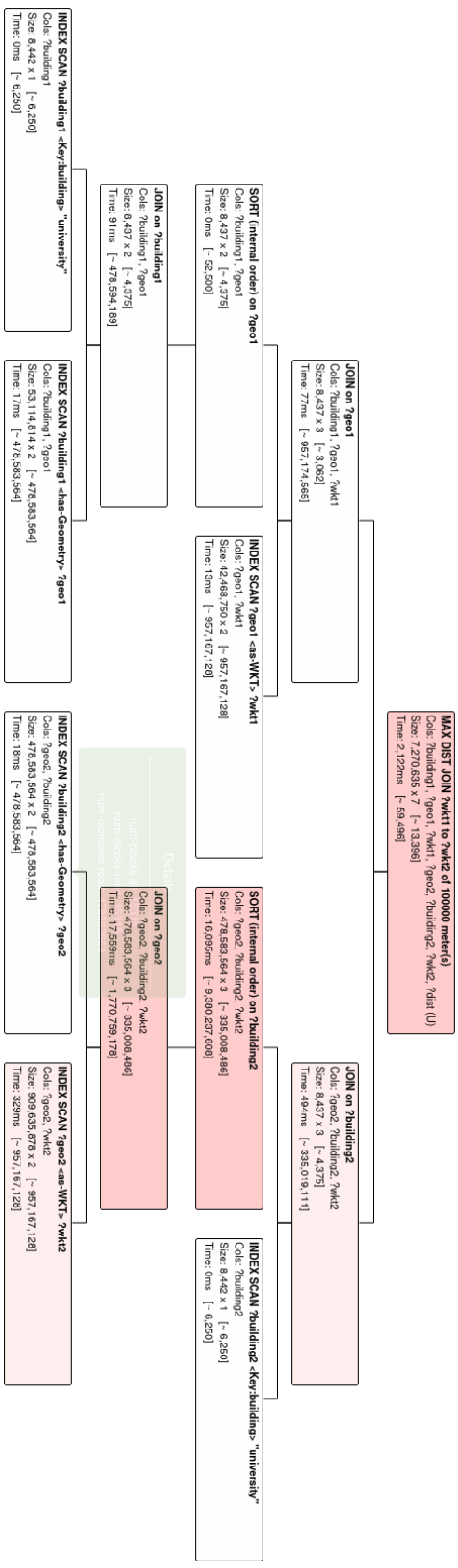


Figure A.17.: Analysis of the runtimes for the search for pairs of university buildings, which are at most 100 km meters apart from each other. The result has been calculated using the boundingBox algorithm with the midpoint approximation

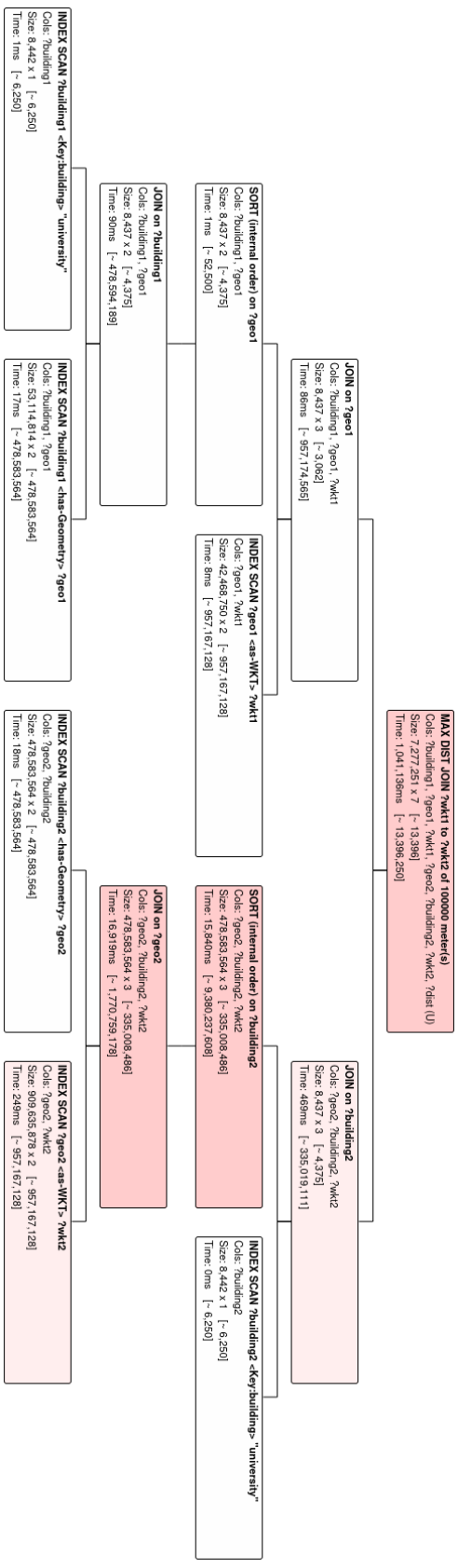


Figure A.18.: Analysis of the runtimes for the search for pairs of university buildings, which are at most 100 km meters apart from each other. The result has been calculated using the baseline algorithm with no midpoint approximation

A.3.1.3. University buildings at most 10 km apart from each other

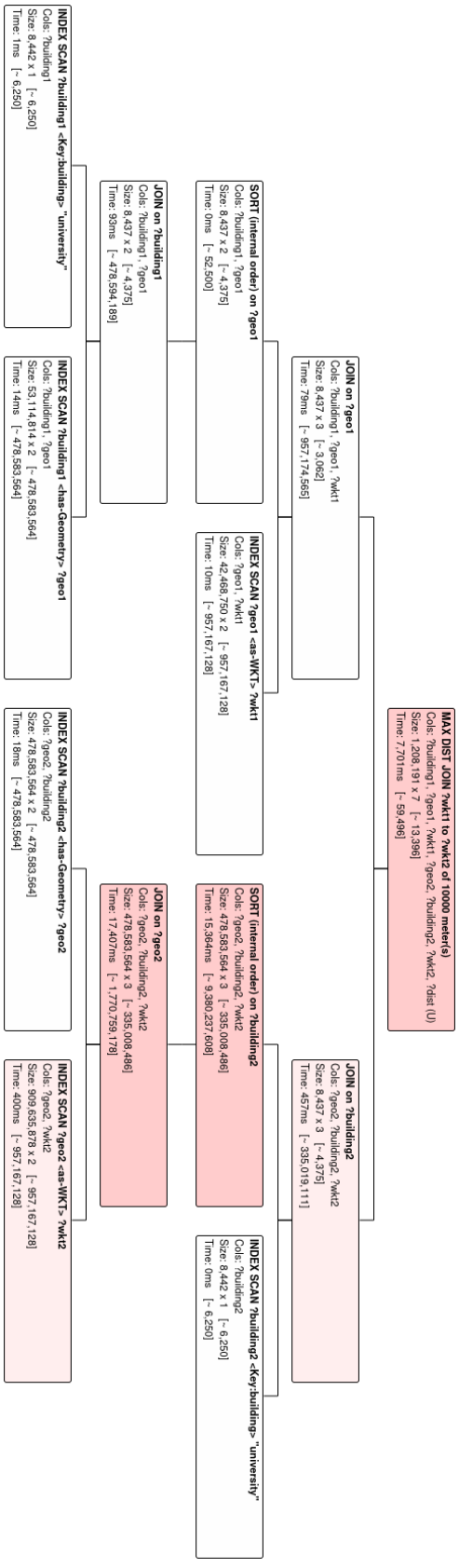


Figure A.19.: Analysis of the runtimes for the search for pairs of university buildings, which are at most 10 km meters apart from each other. The result has been calculated using the boundingBox algorithm with no midpoint approximation

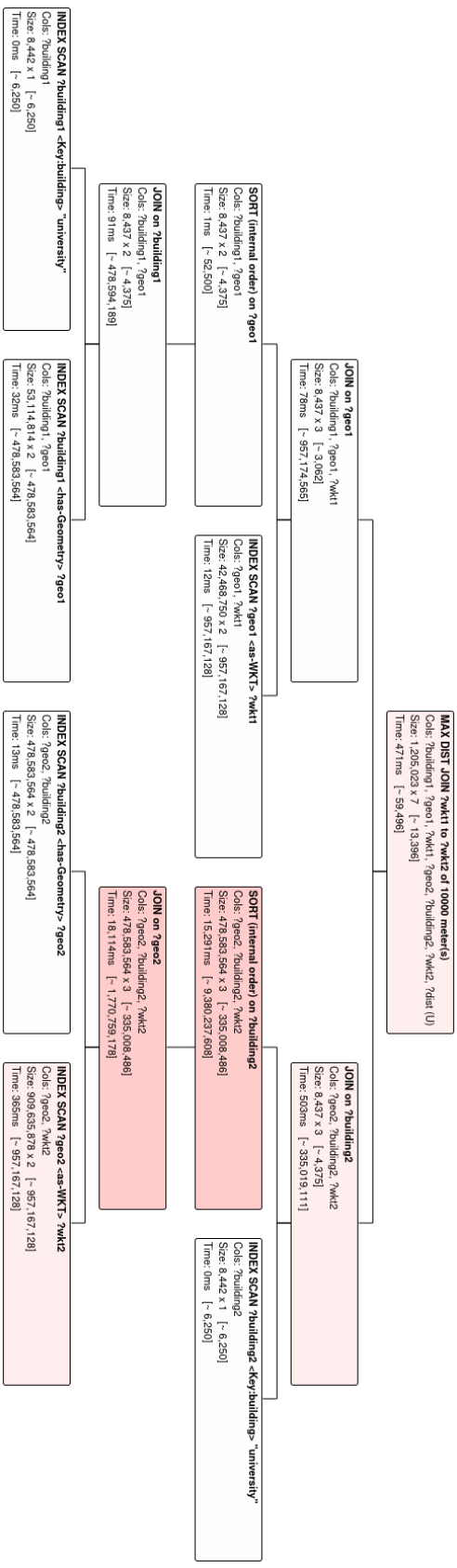


Figure A.20.: Analysis of the runtimes for the search for pairs of university buildings, which are at most 10 km meters apart from each other. The result has been calculated using the boundingBox algorithm with the midpoint approximation

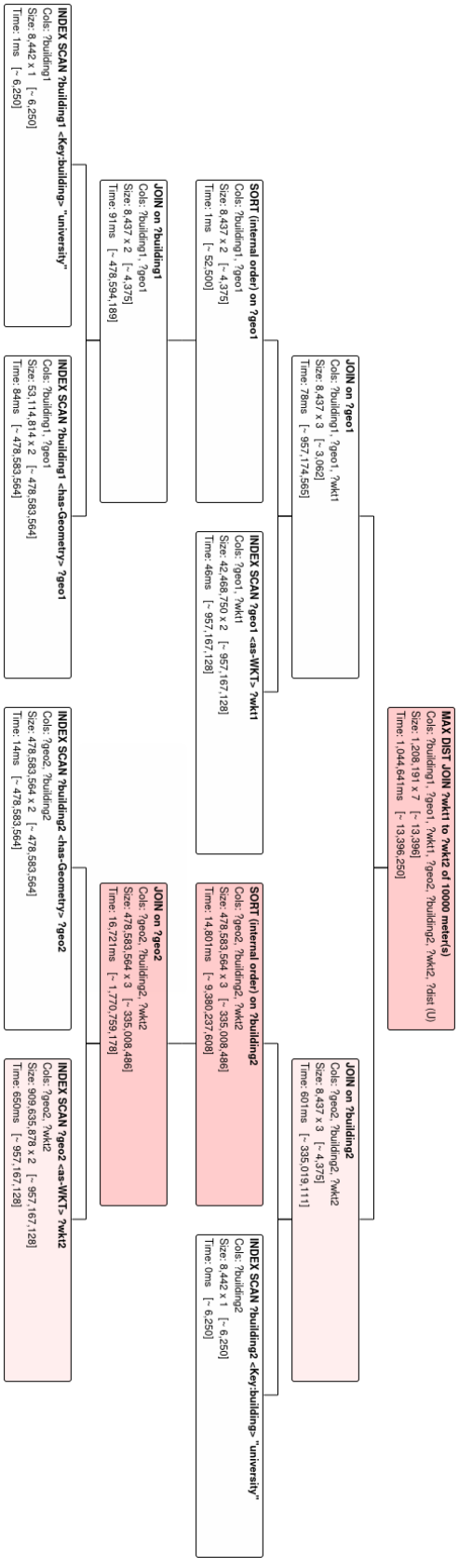


Figure A.21.: Analysis of the runtimes for the search for pairs of university buildings, which are at most 10 km meters apart from each other. The result has been calculated using the baseline algorithm with no midpoint approximation

A.3.1.4. University buildings at most 1 km apart from each other

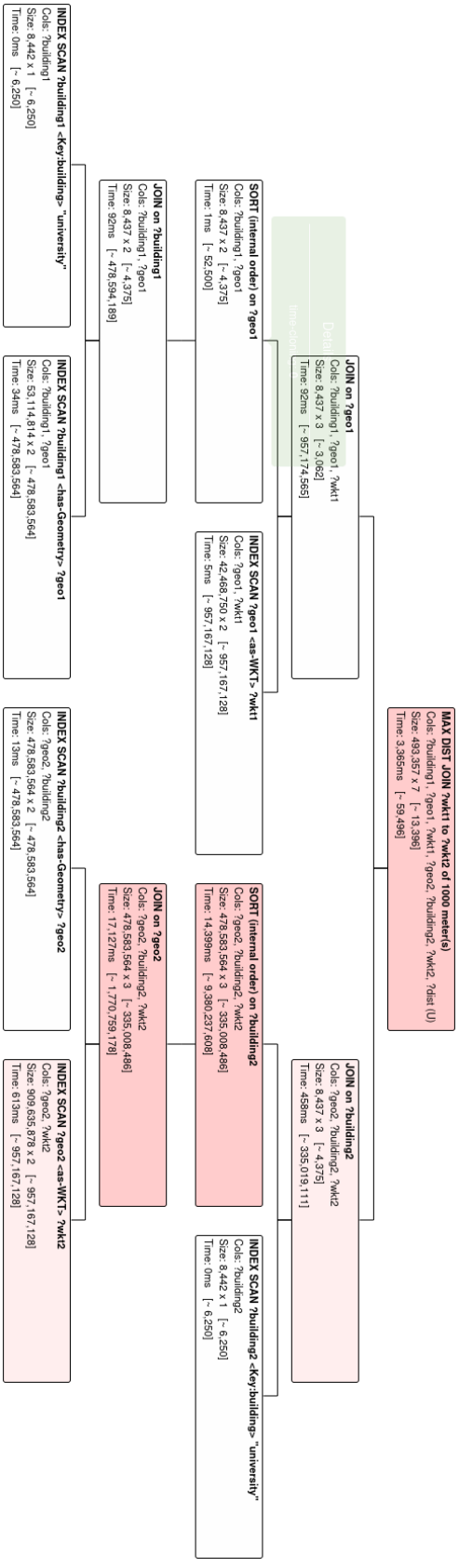


Figure A.22.: Analysis of the runtimes for the search for pairs of university buildings, which are at most 1km meters apart from each other. The result has been calculated using the boundingBox algorithm with no midpoint approximation

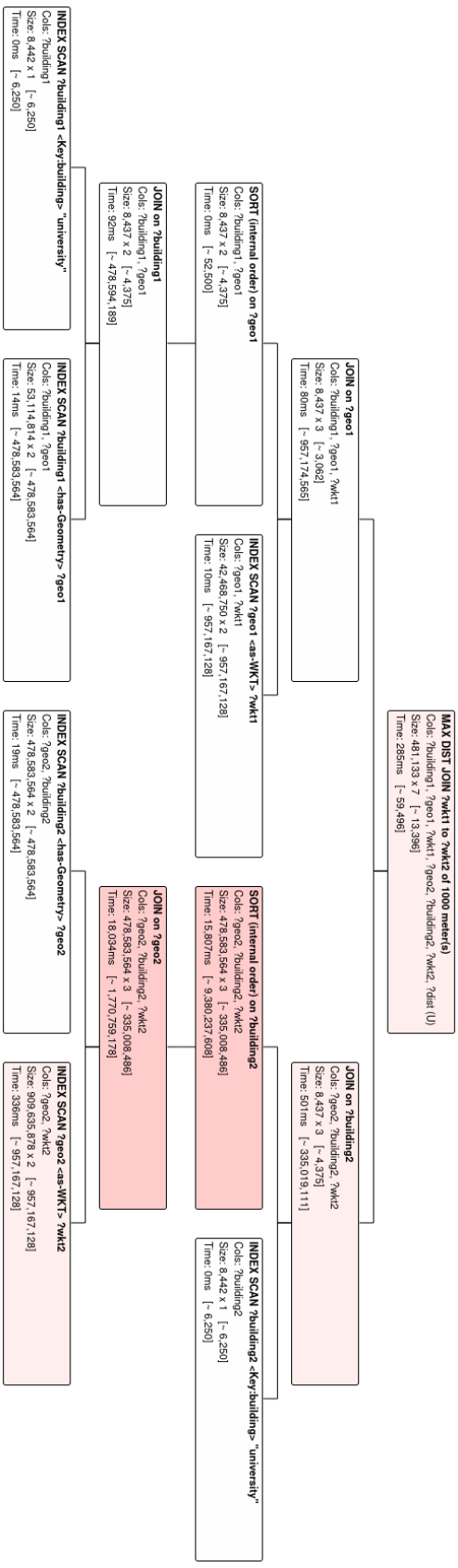


Figure A.23.: Analysis of the runtimes for the search for pairs of university buildings, which are at most 1 km meters apart from each other. The result has been calculated using the boundingBox algorithm with the midpoint approximation

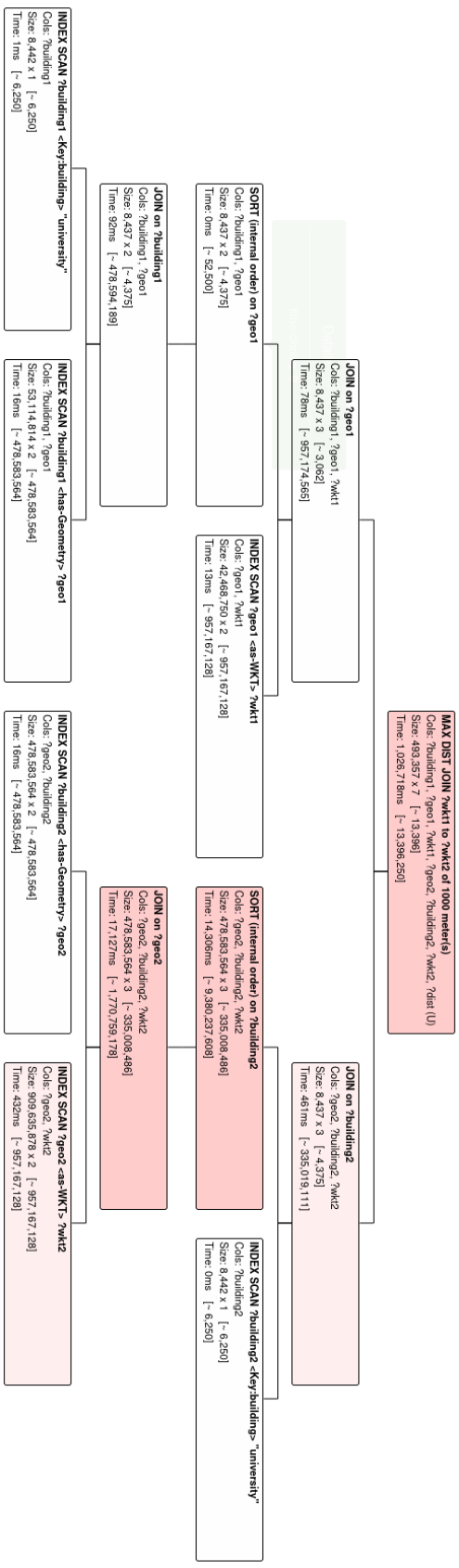


Figure A.24.: Analysis of the runtimes for the search for pairs of university buildings, which are at most 1 km meters apart from each other. The result has been calculated using the baseline algorithm with no midpoint approximation

A.3.1.5. University buildings at most 0.1 km apart from each other

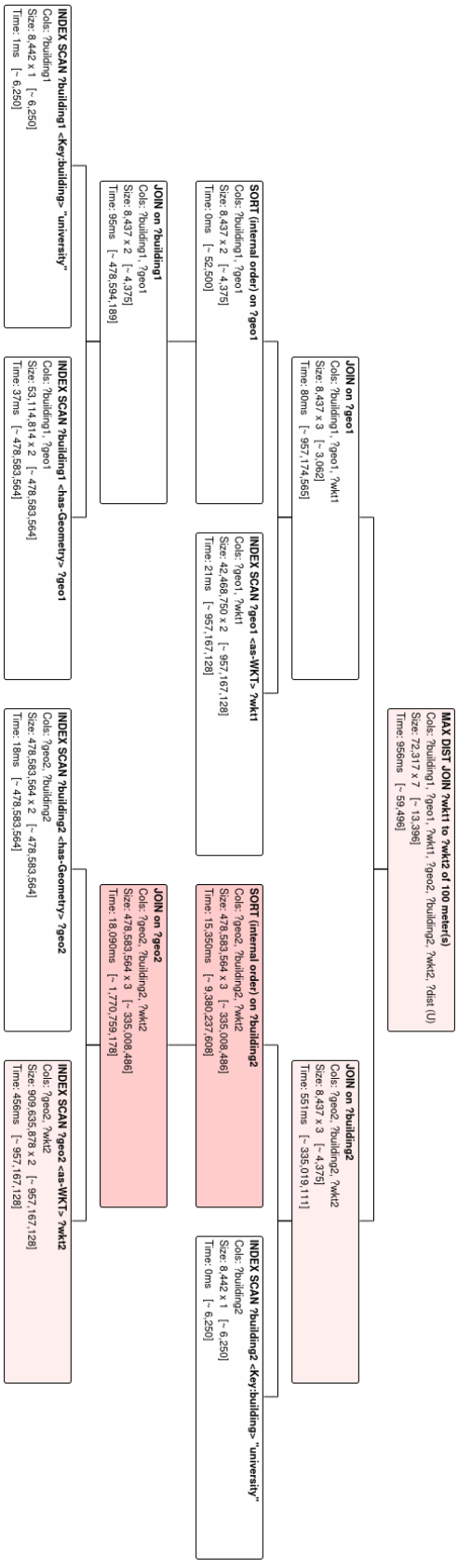


Figure A.25.: Analysis of the runtimes for the search for pairs of university buildings, which are at most 0.1 km meters apart from each other. The result has been calculated using the boundingBox algorithm with no midpoint approximation

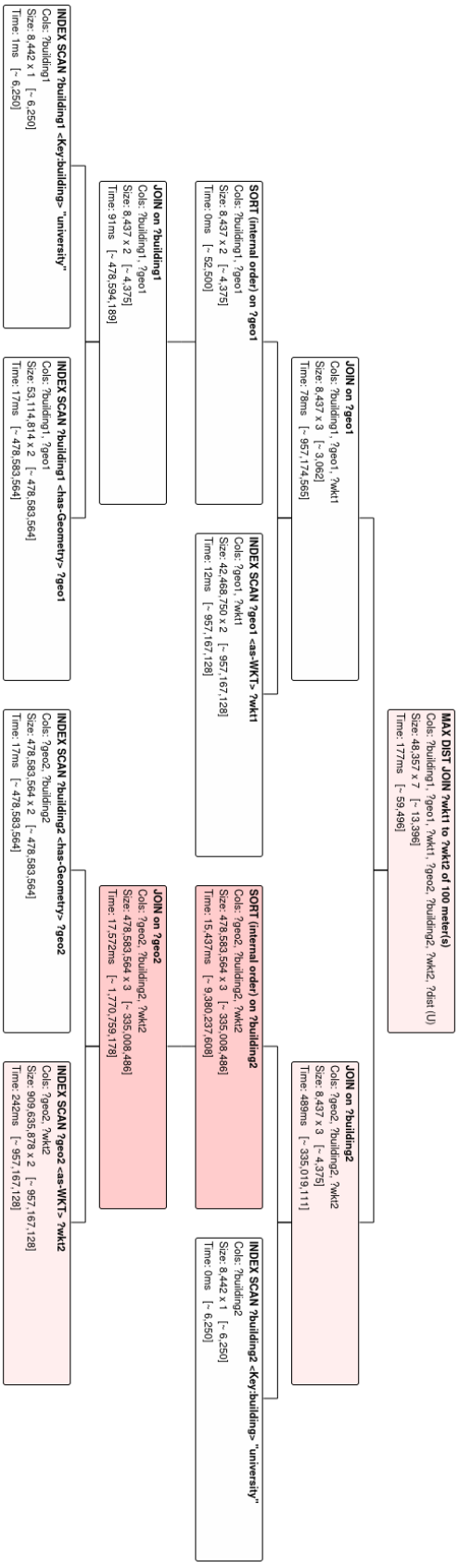


Figure A.26.: Analysis of the runtimes for the search for pairs of university buildings, which are at most 0.1 km meters apart from each other. The result has been calculated using the boundingBox algorithm with the midpoint approximation.

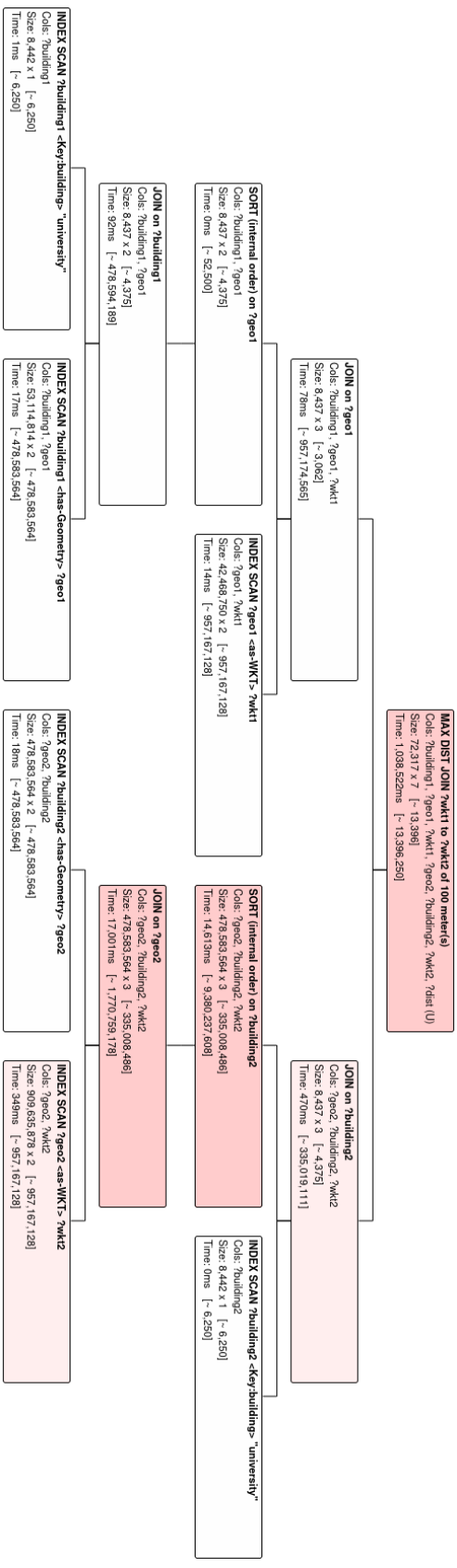


Figure A.27.: Analysis of the runtimes for the search for pairs of university buildings, which are at most 0.1 km meters apart from each other. The result has been calculated using the baseline algorithm with no midpoint approximation

A.3.1.6. University buildings at most 0.01 km apart from each other

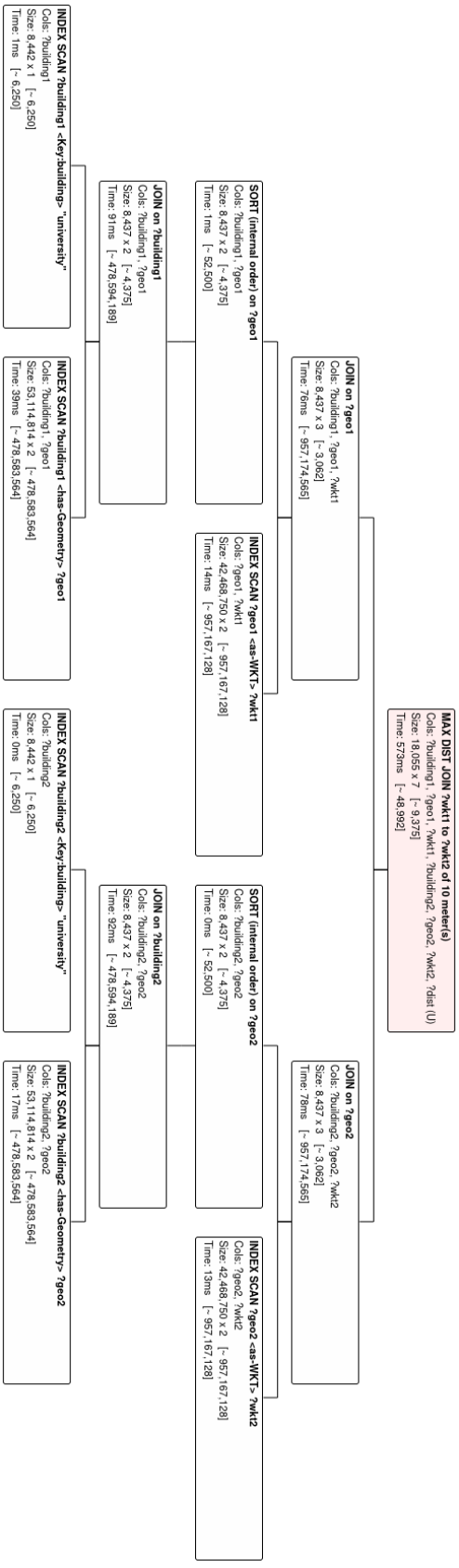


Figure A.28.: Analysis of the runtimes for the search for pairs of university buildings, which are at most 0.01 km meters apart from each other. The result has been calculated using the boundingBox algorithm with no midpoint approximation

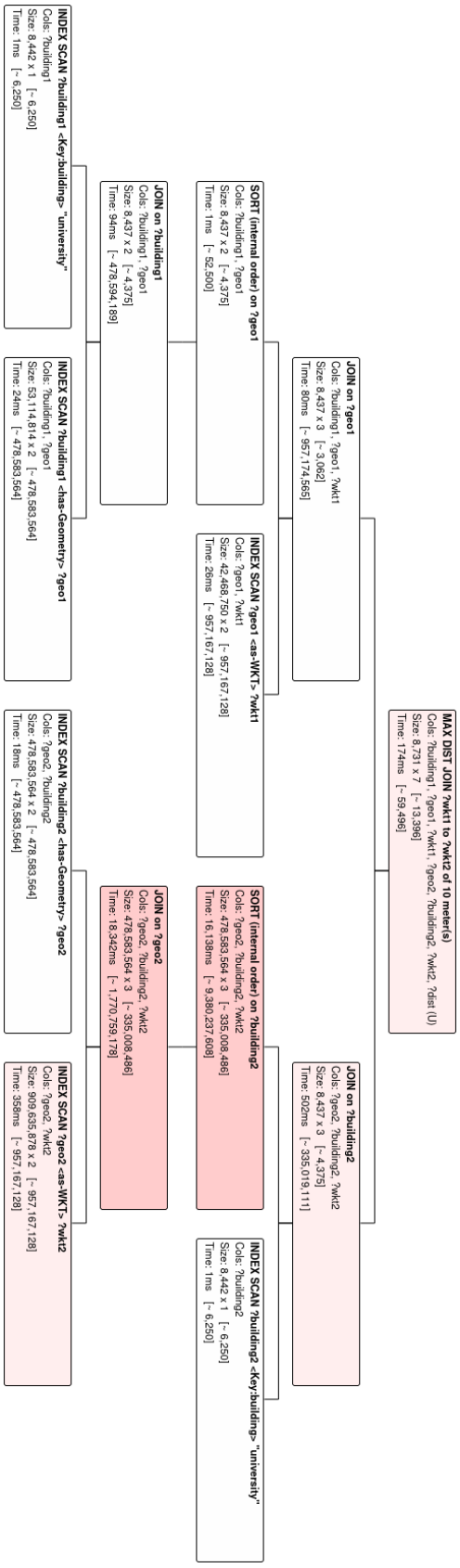


Figure A.29.: Analysis of the runtimes for the search for pairs of university buildings, which are at most 0.01 km meters apart from each other. The result has been calculated using the boundingBox algorithm with the midpoint approximation

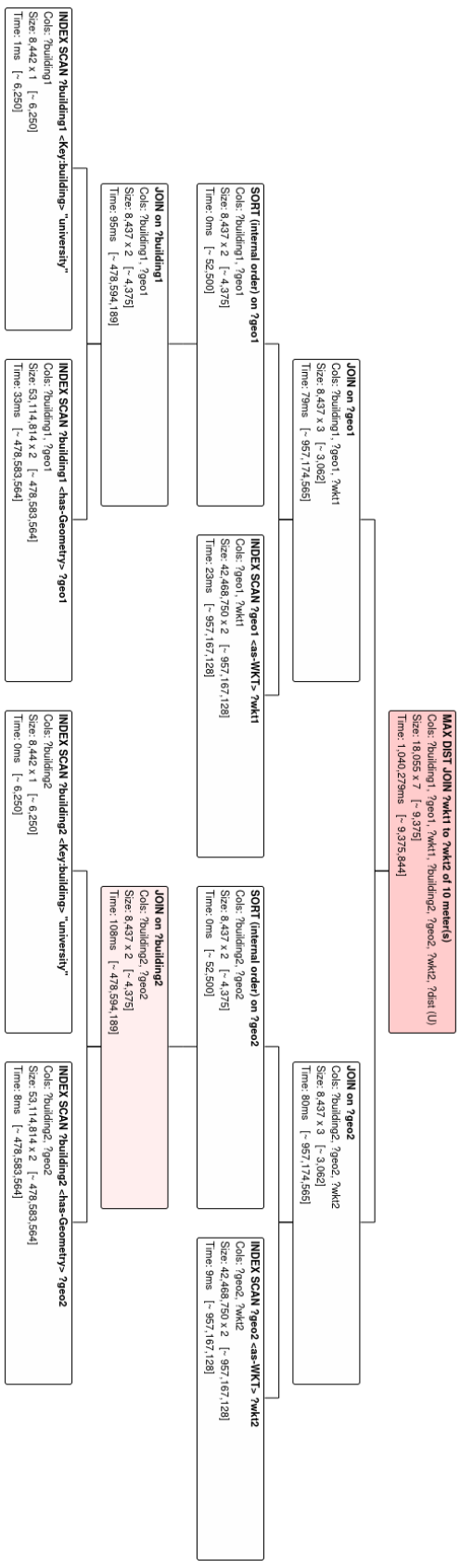


Figure A.30.: Analysis of the runtimes for the search for pairs of university buildings, which are at most 0.01 km meters apart from each other. The result has been calculated using the baseline algorithm with no midpoint approximation

A.3.1.7. University buildings at most 25 km from the university library freiburg away

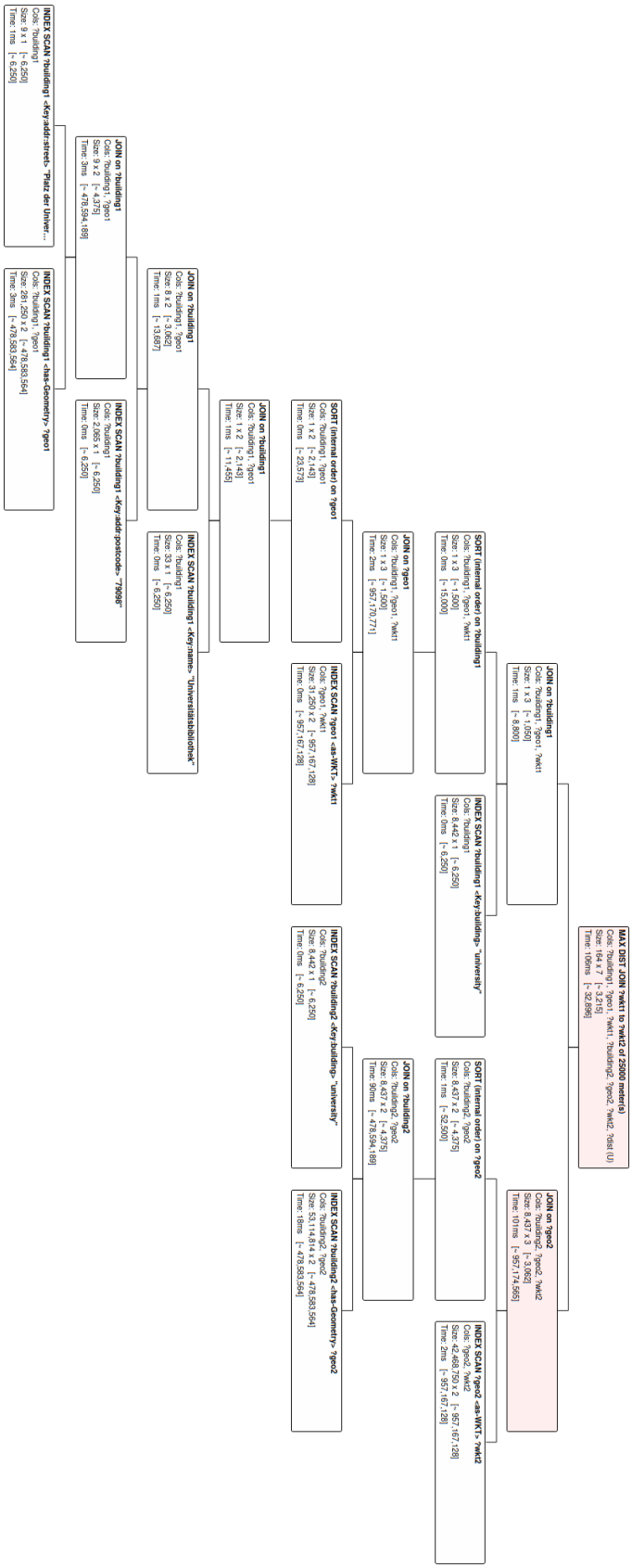


Figure A.31.: Analysis of the runtimes for the search of university buildings, which are at most 25 km meters apart from the university library freiburg. The result has been calculated using the boundingBox algorithm with no midpoint approximation

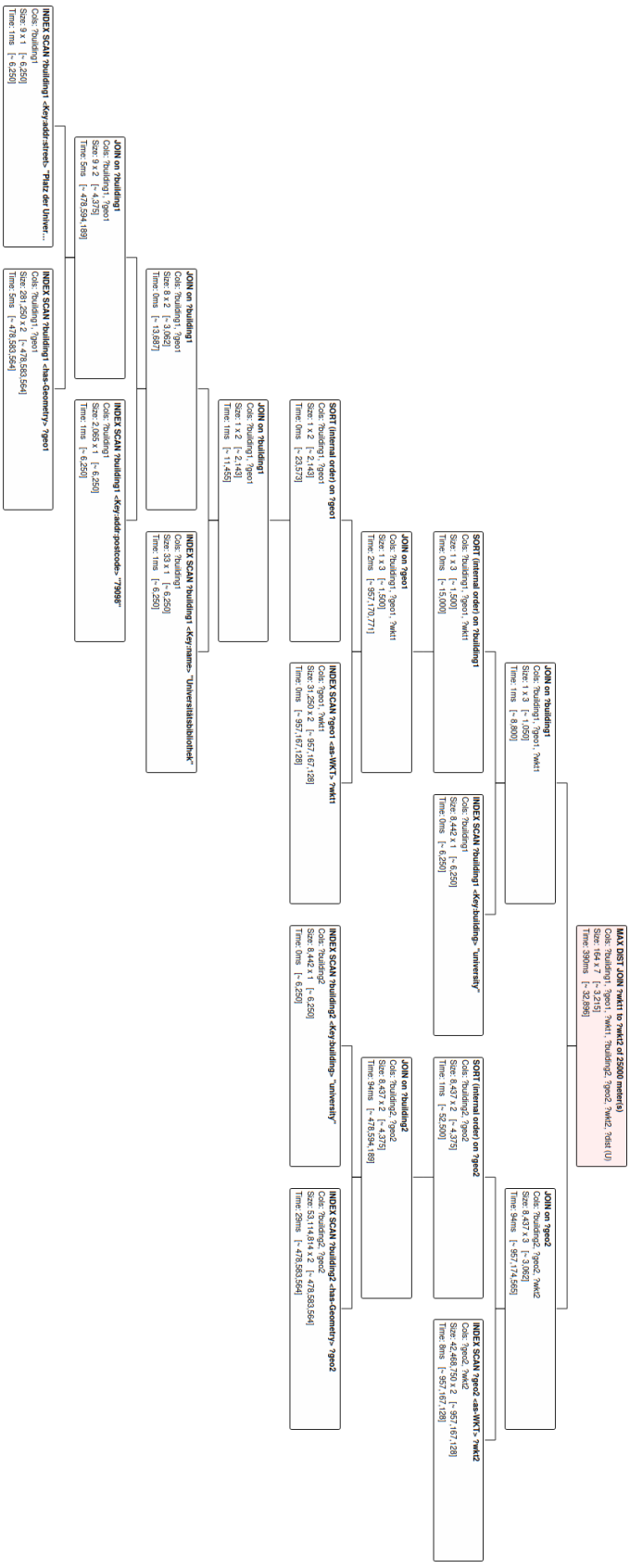


Figure A.32.: Analysis of the runtimes for the search of university buildings, which are at most 25 km meters apart from the university library freiburg. The result has been calculated using the boundingBox algorithm with the midpoint approximation

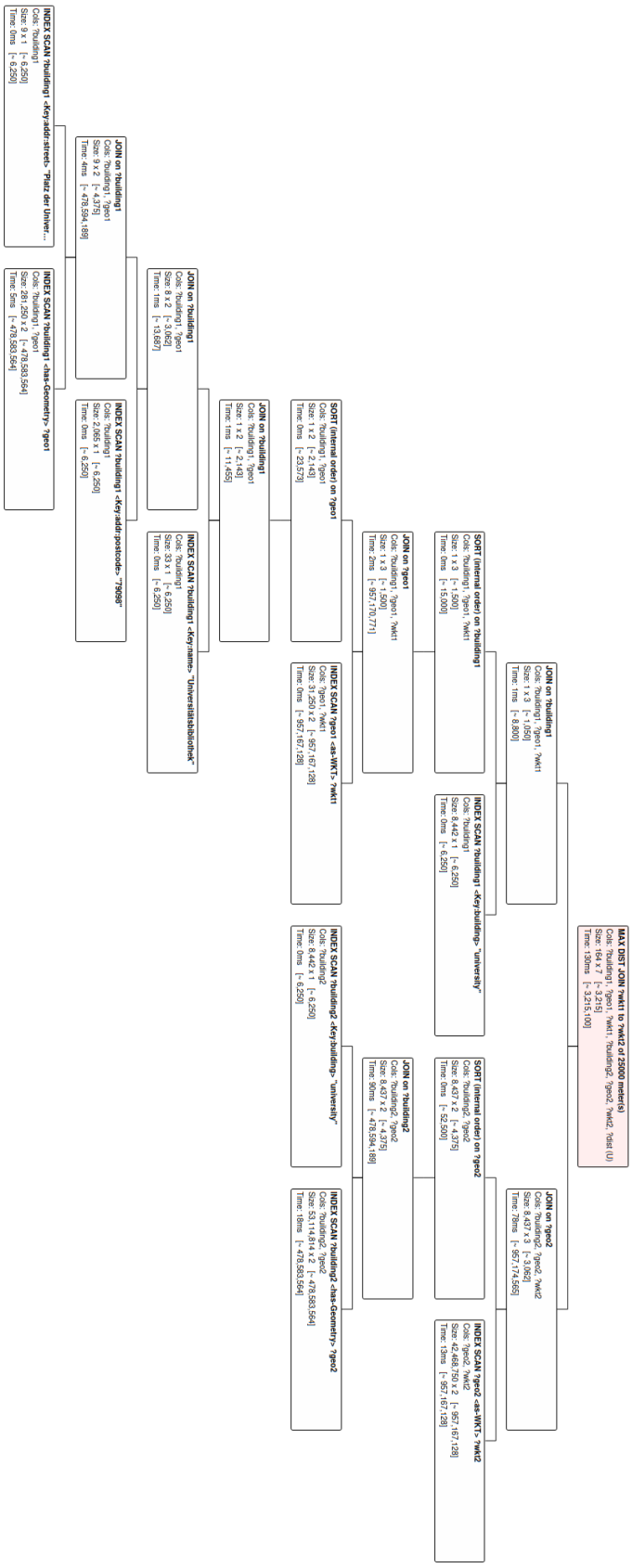


Figure A.33.: Analysis of the runtimes for the search of university buildings, which are at most 25 km meters apart from the university library freiburg. The result has been calculated using the baseline algorithm with no midpoint approximation

A.3.2. Restaurants near the university library freiburg and near a tram stop

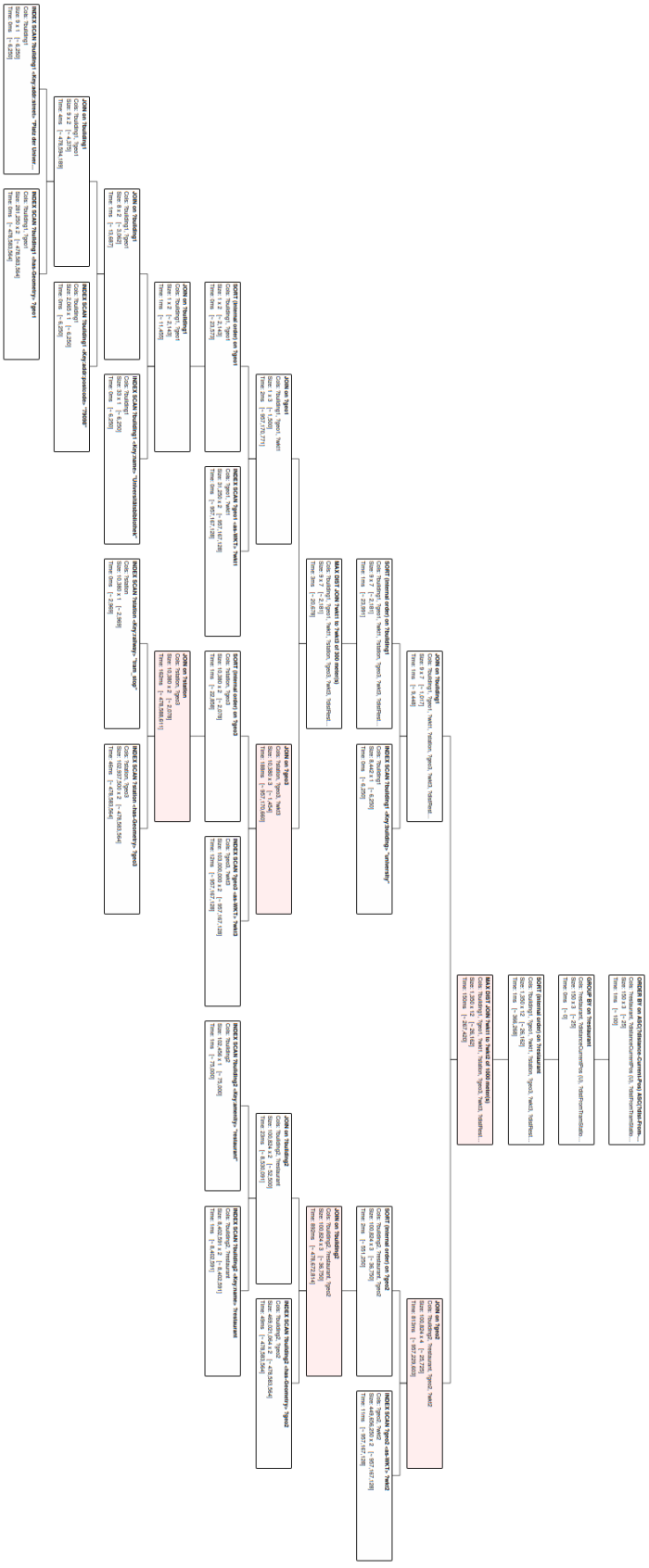


Figure A.34.: Analysis of the runtimes for the search of restaurants near the university library which are also near a tram stop. The result has been calculated using the boundingBox algorithm with no midpoint approximation

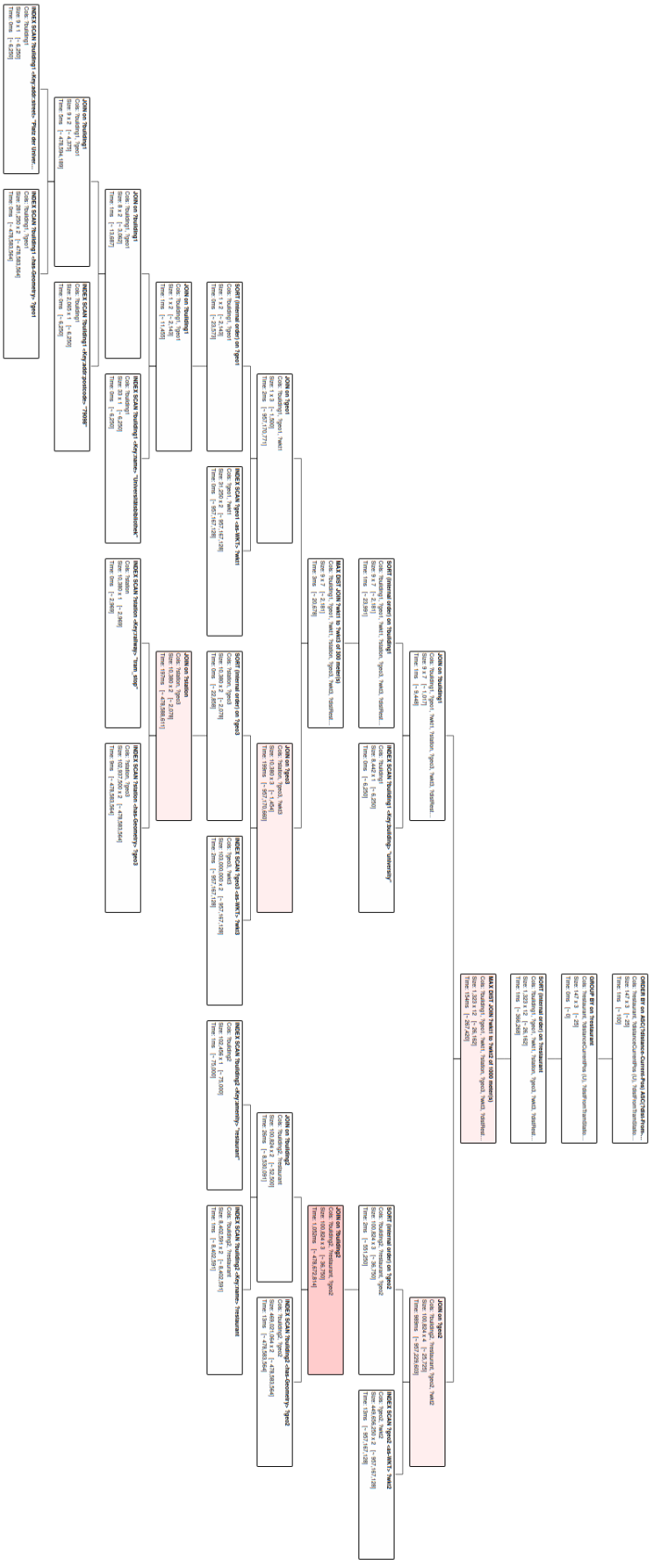


Figure A.35.: Analysis of the runtimes for the search of restaurants near the university library which are also near a tram stop. The result has been calculated using the boundingBox algorithm with the midpoint approximation

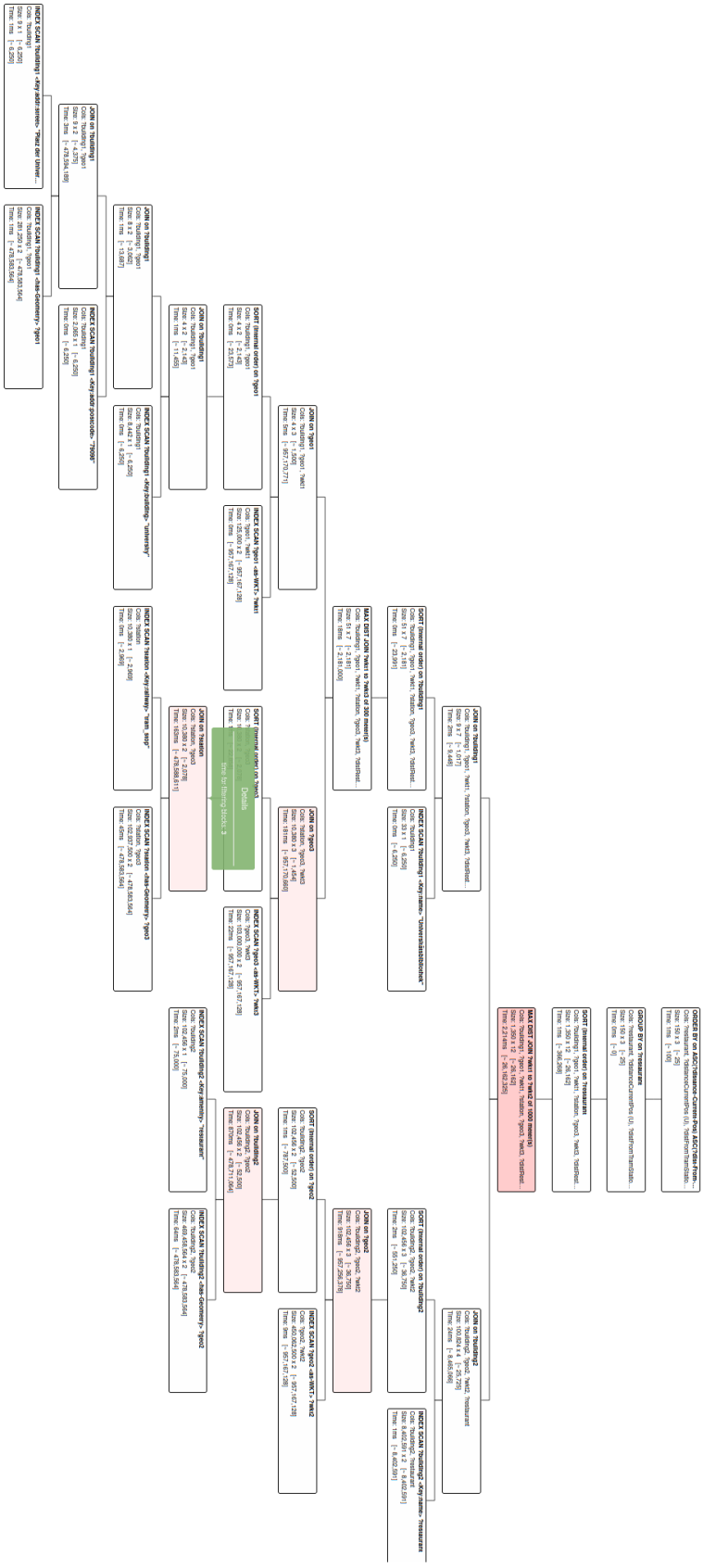


Figure A.36.: Analysis of the runtimes for the search of restaurants near the university library which are also near a tram stop. The result has been calculated using the baseline algorithm with no midpoint approximation

A.3.3. Antenna query

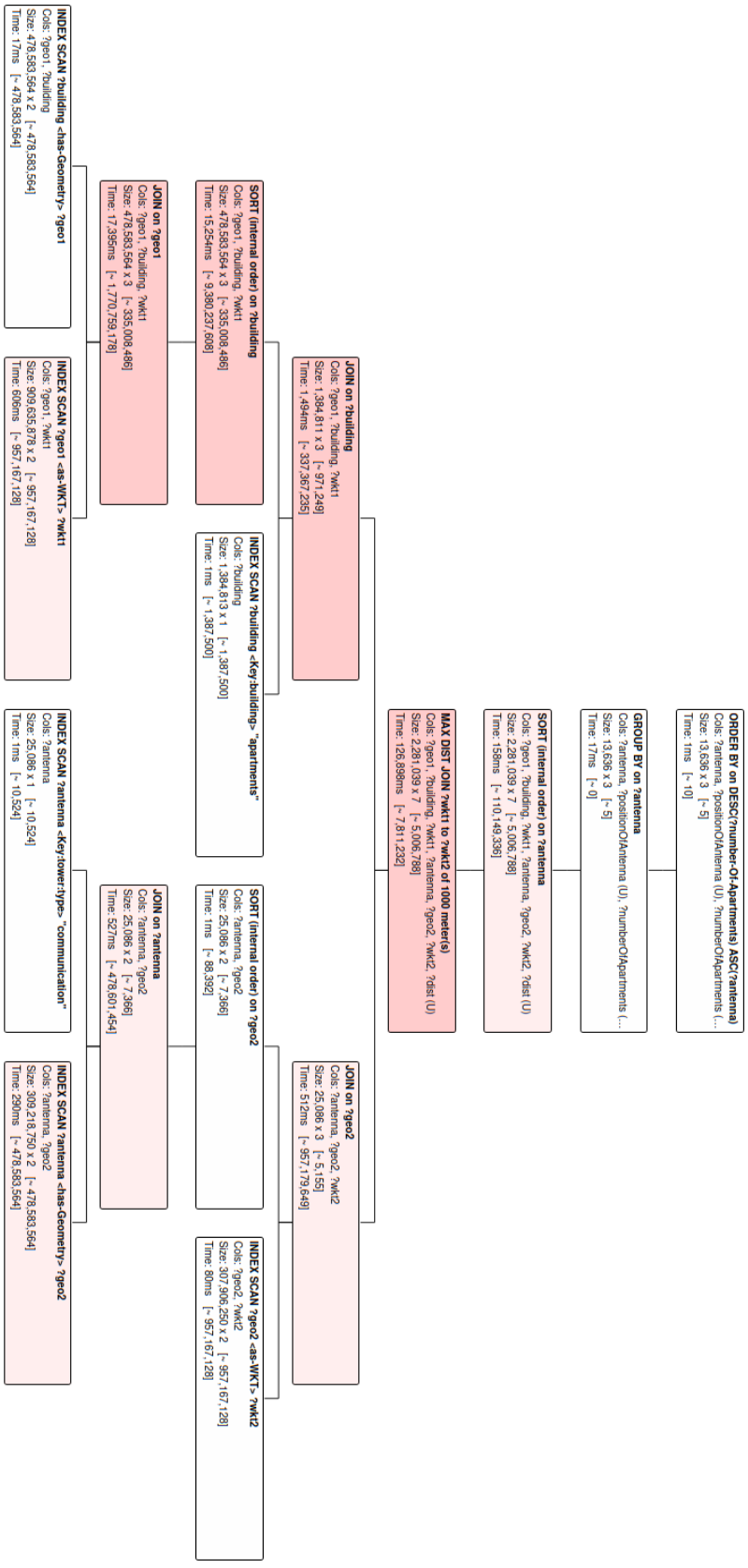


Figure A.37.: Analysis of the runtimes for the search of antennas and the amount of apartments near them. The result has been calculated using the boundingBox algorithm with no midpoint approximation

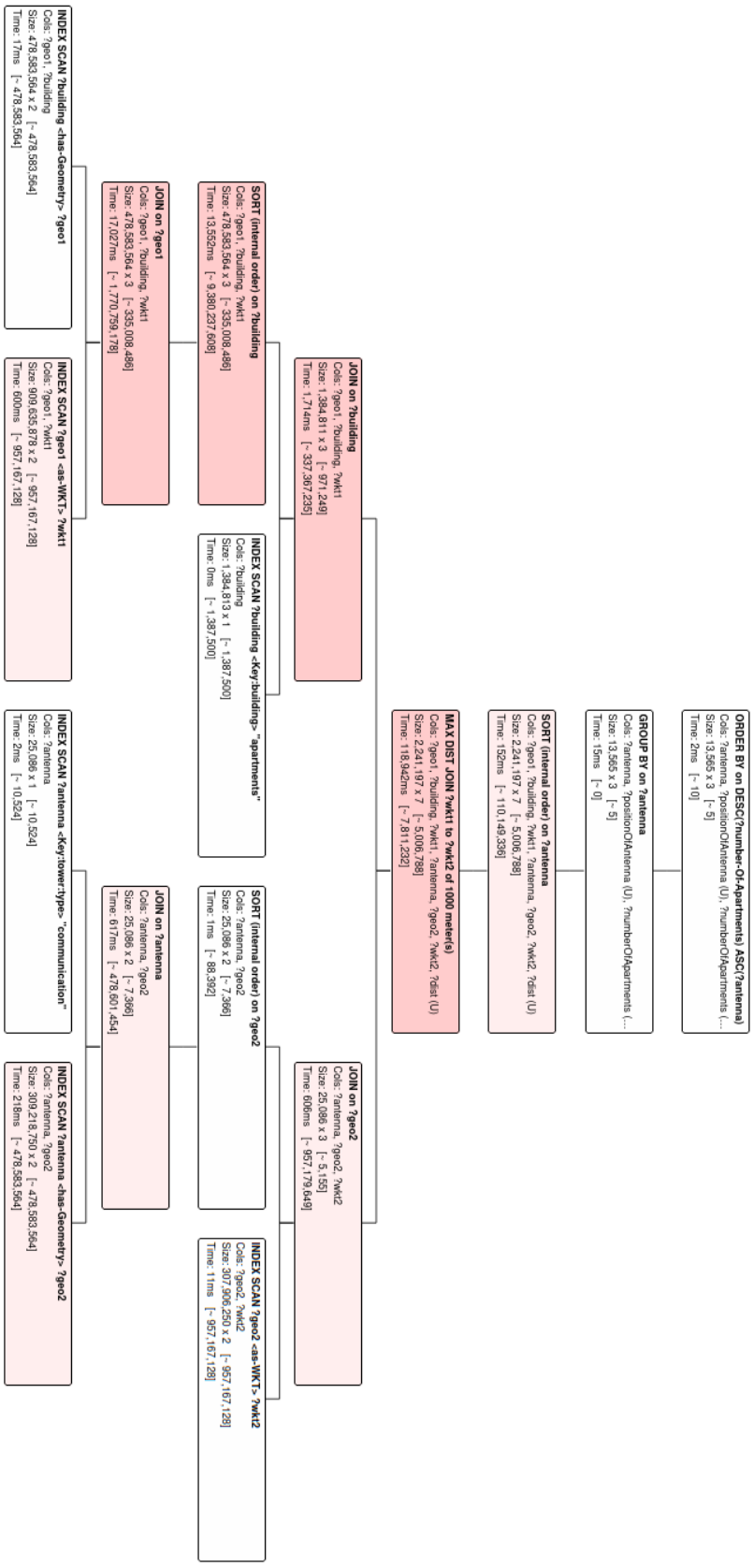


Figure A.38.: Analysis of the runtimes for the search of antennas and the amount of apartments near them. The result has been calculated using the boundingBox algorithm with the midpoint approximation

A.4. Timing analysis of the code

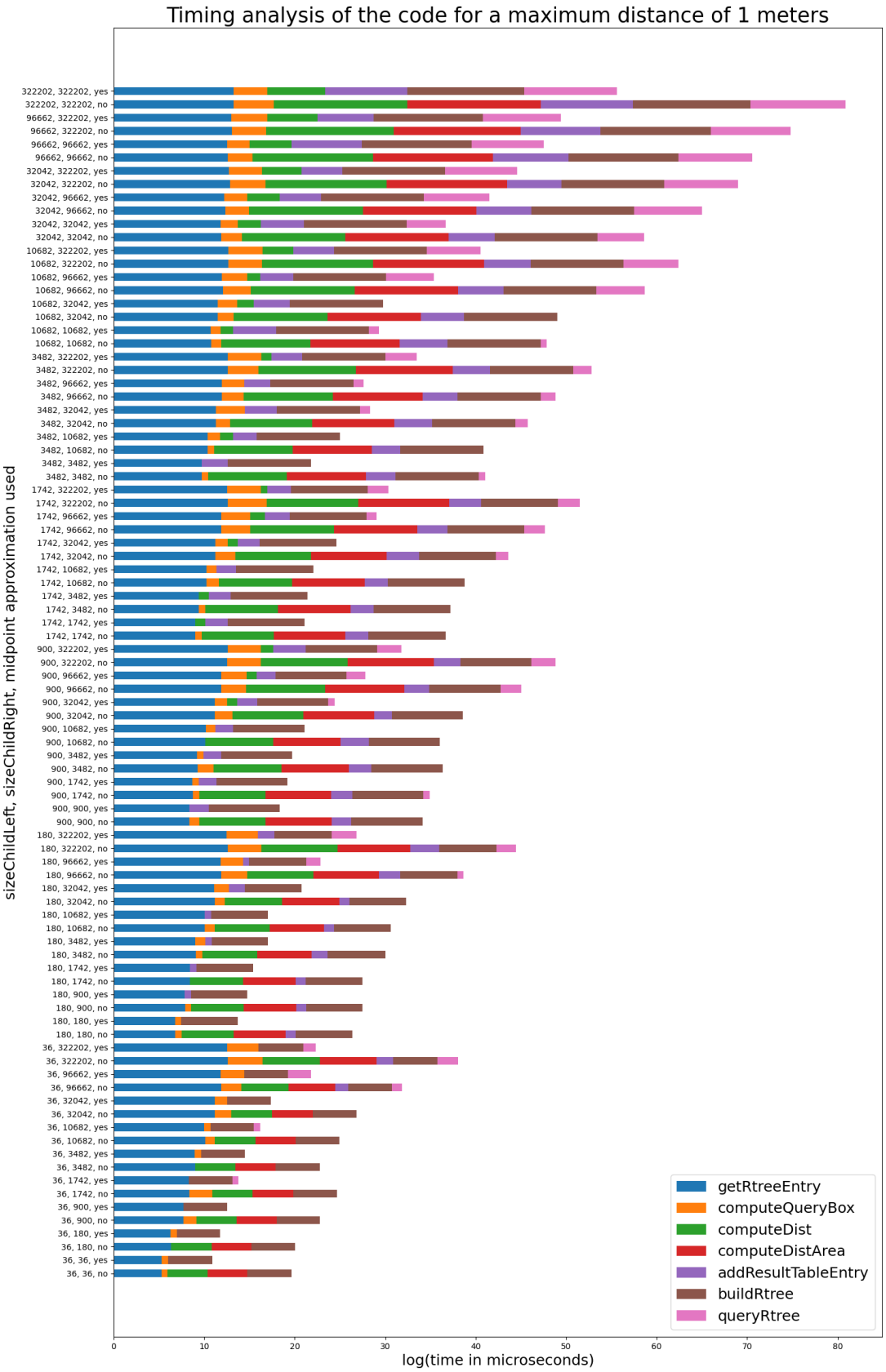


Figure A.39.: This figure shows the times spent in different methods for the query with a maximum distance of 1 meter. The x axis is logarithmic

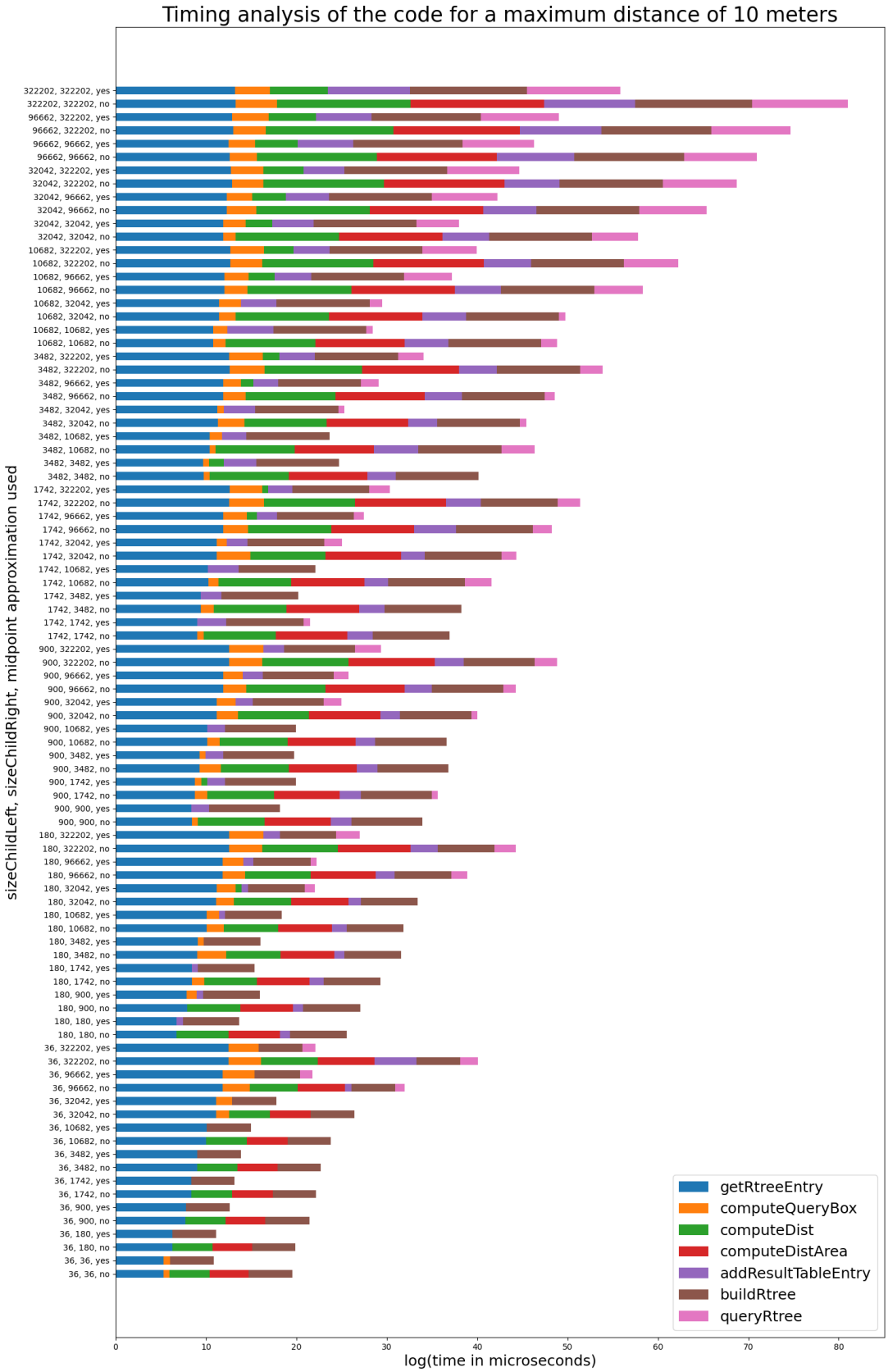


Figure A.40.: This figure shows the times spent in different methods for the query with a maximum distance of 10 meter. The x axis is logarithmic

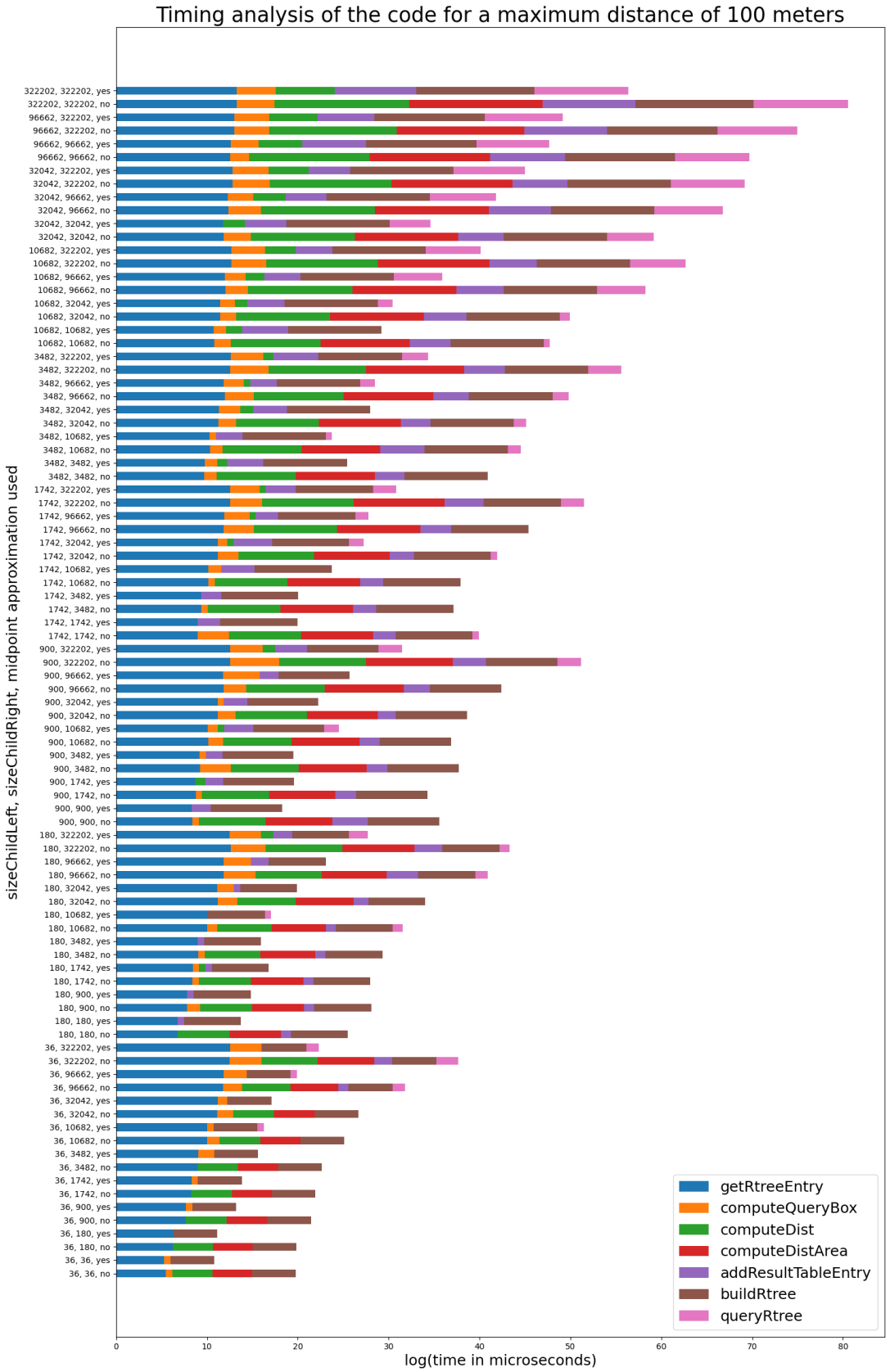


Figure A.41.: This figure shows the times spent in different methods for the query with a maximum distance of 100 meter. The x axis is logarithmic

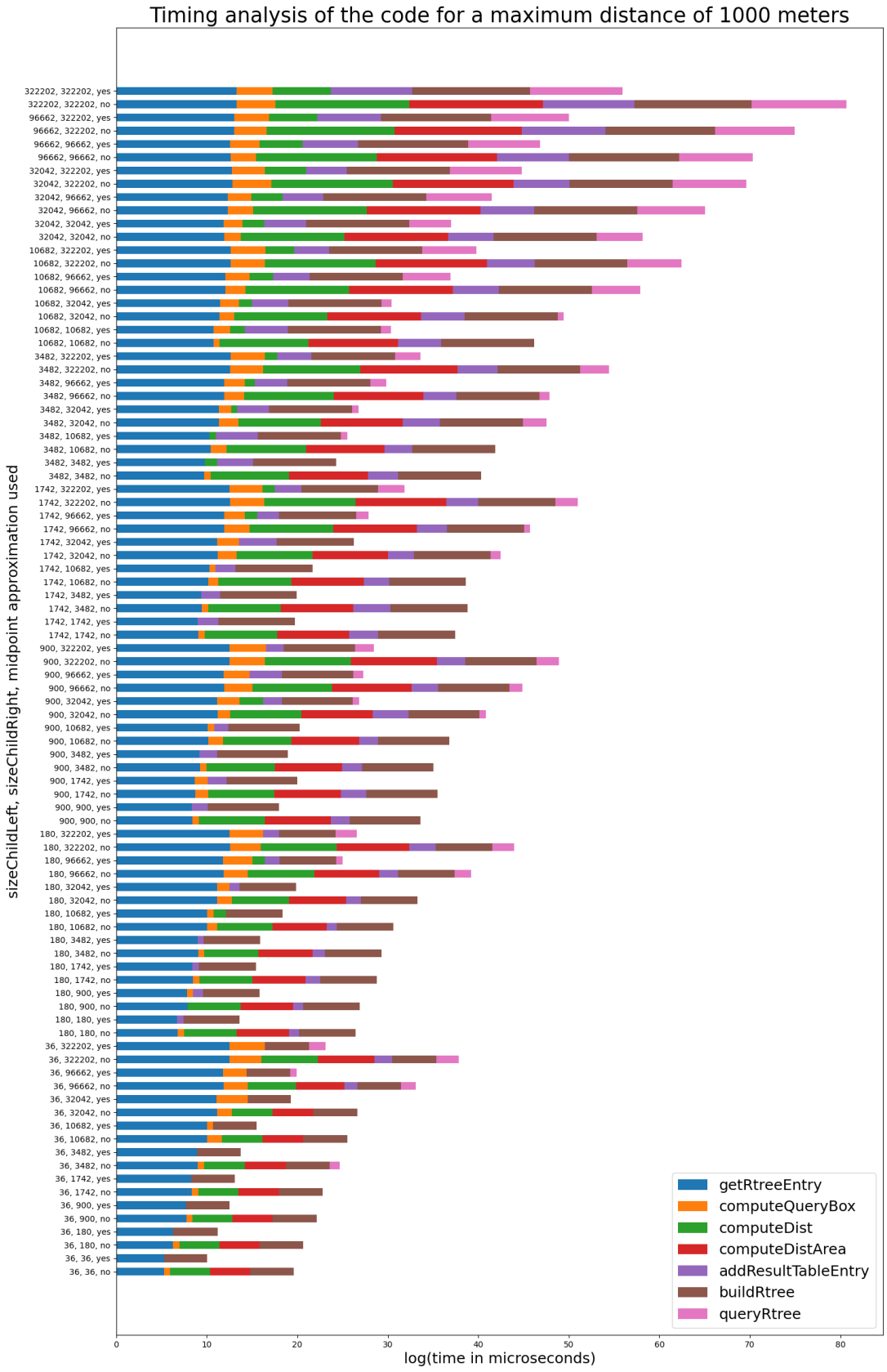


Figure A.42.: This figure shows the times spent in different methods for the query with a maximum distance of 1000 meter. The x axis is logarithmic

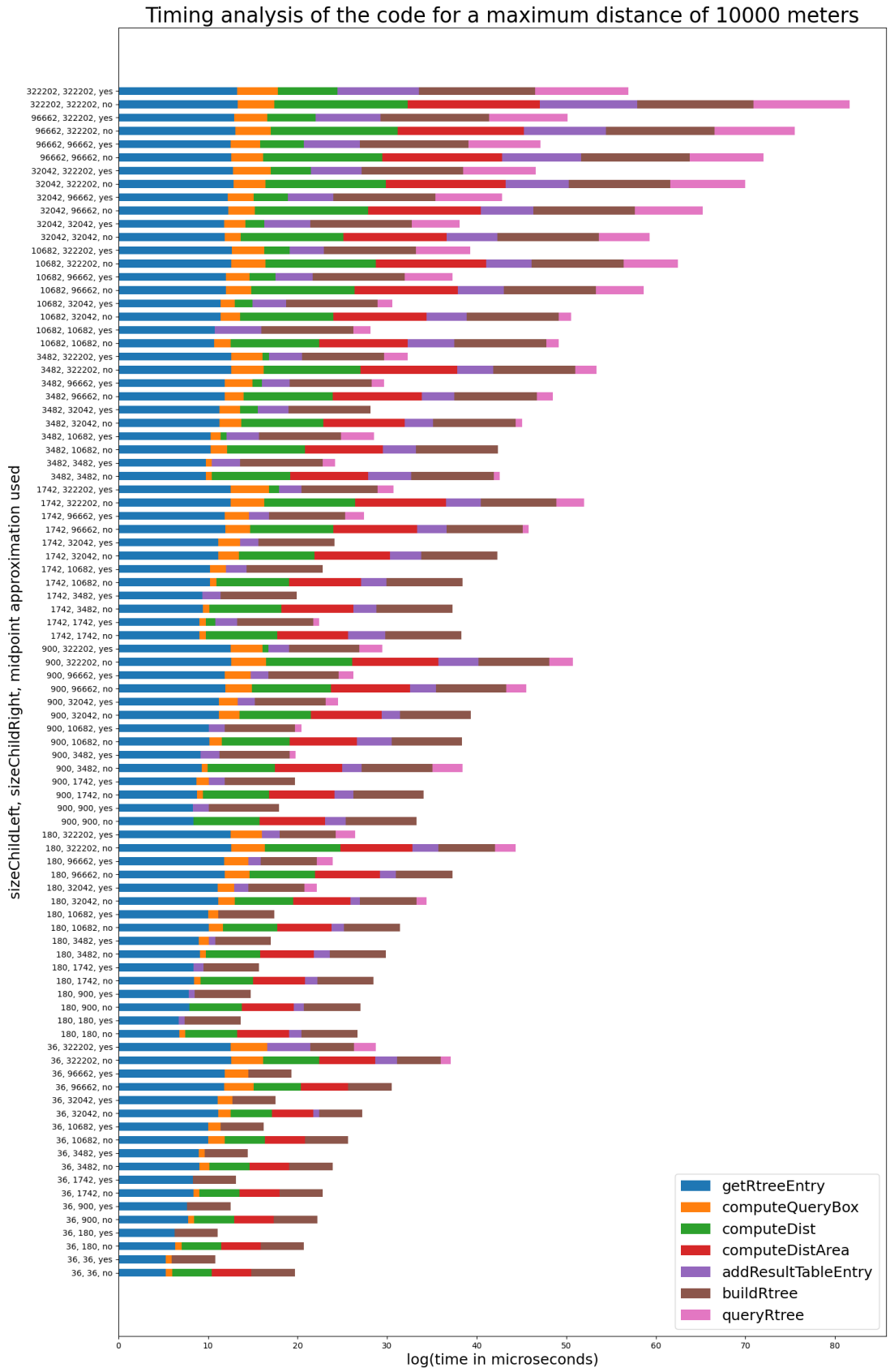


Figure A.43.: This figure shows the times spent in different methods for the query with a maximum distance of 10000 meter. The x axis is logarithmic

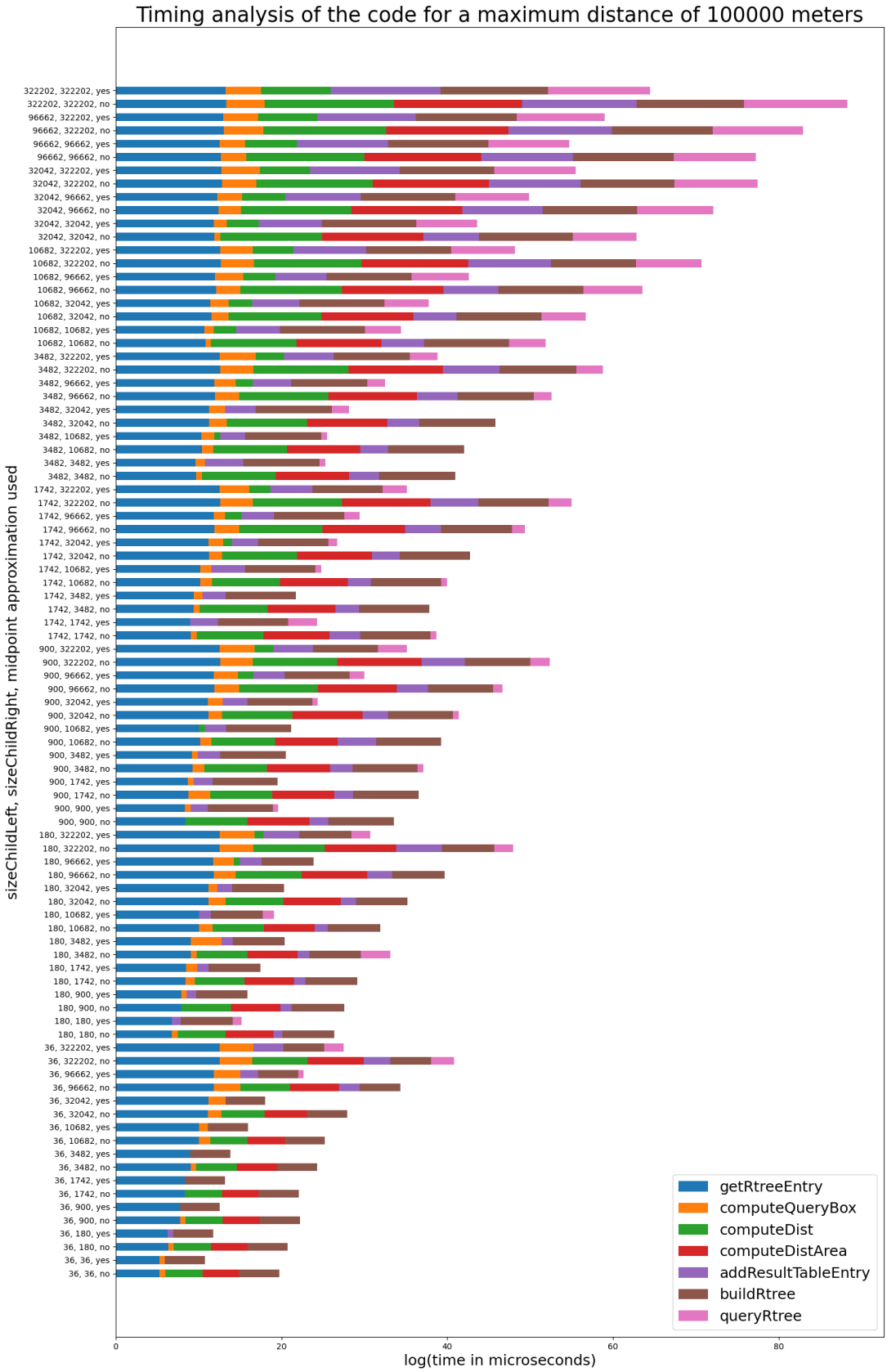


Figure A.44.: This figure shows the times spent in different methods for the query with a maximum distance of 100000 meter. The x axis is logarithmic

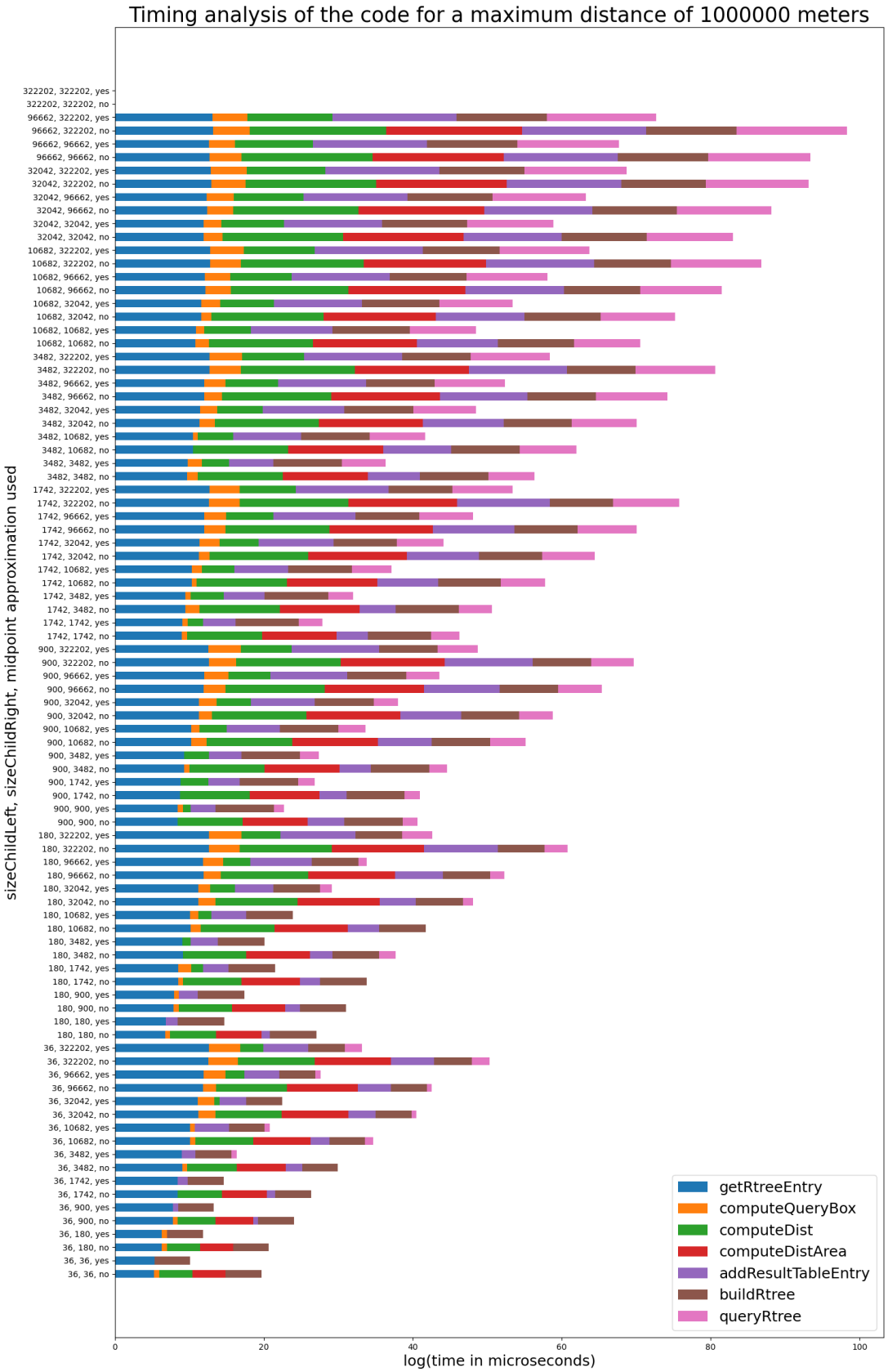


Figure A.45.: This figure shows the times spent in different methods for the query with a maximum distance of 1000000 meter. The x axis is logarithmic

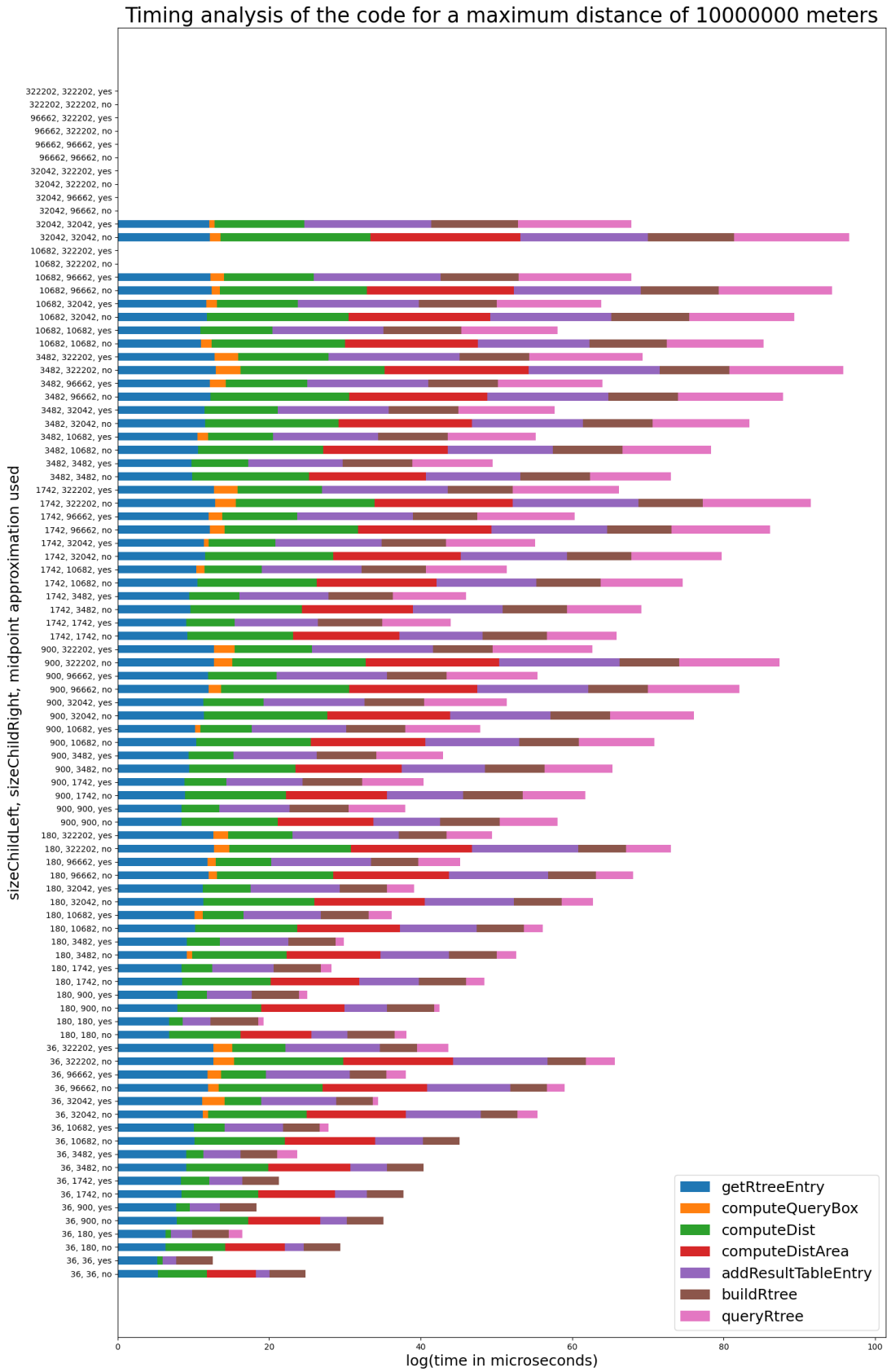


Figure A.46.: This figure shows the times spent in different methods for the query with a maximum distance of 10000000 meter. The x axis is logarithmic

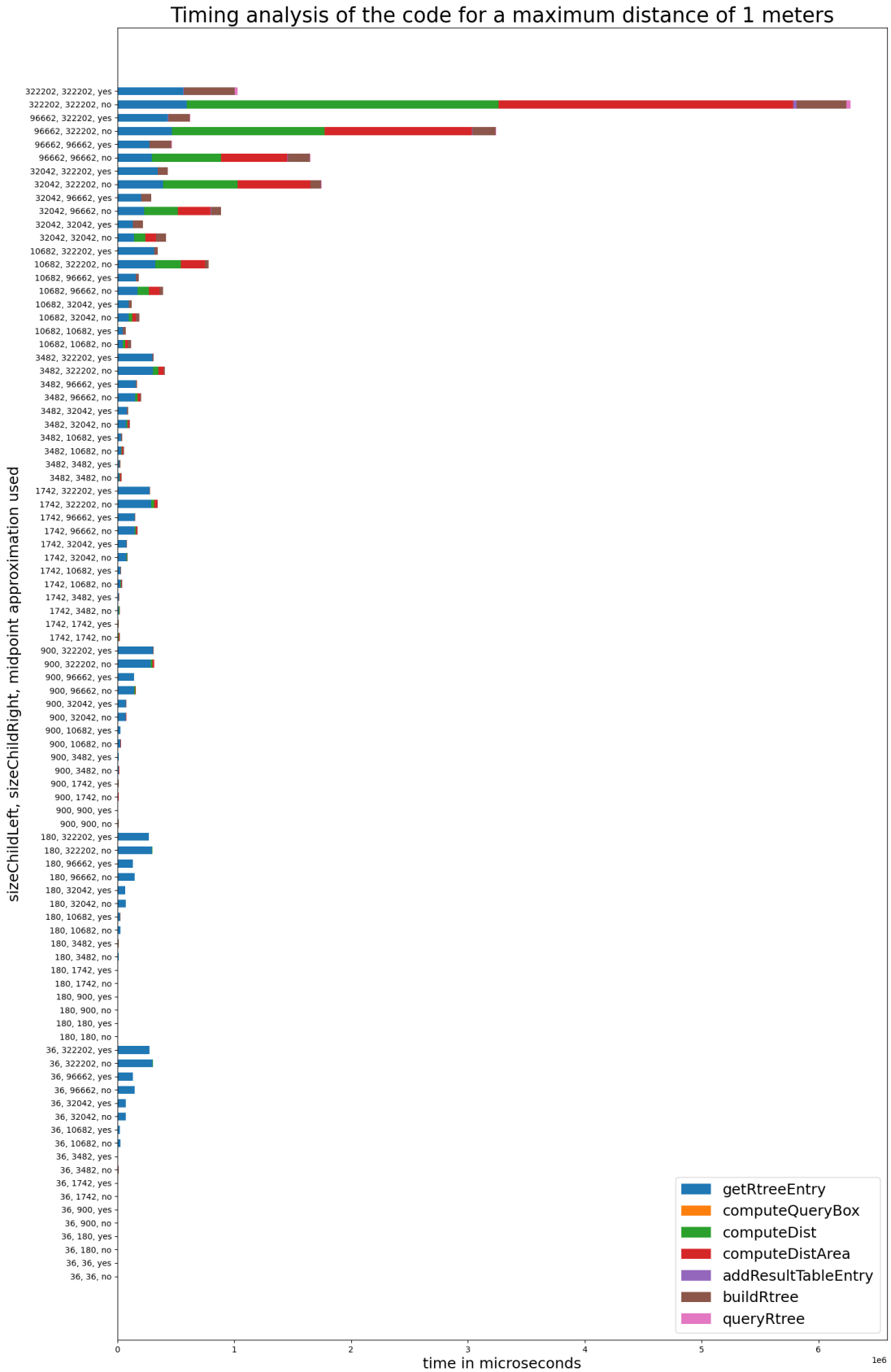


Figure A.47.: This figure shows the times spent in different methods for the query with a maximum distance of 1 meter

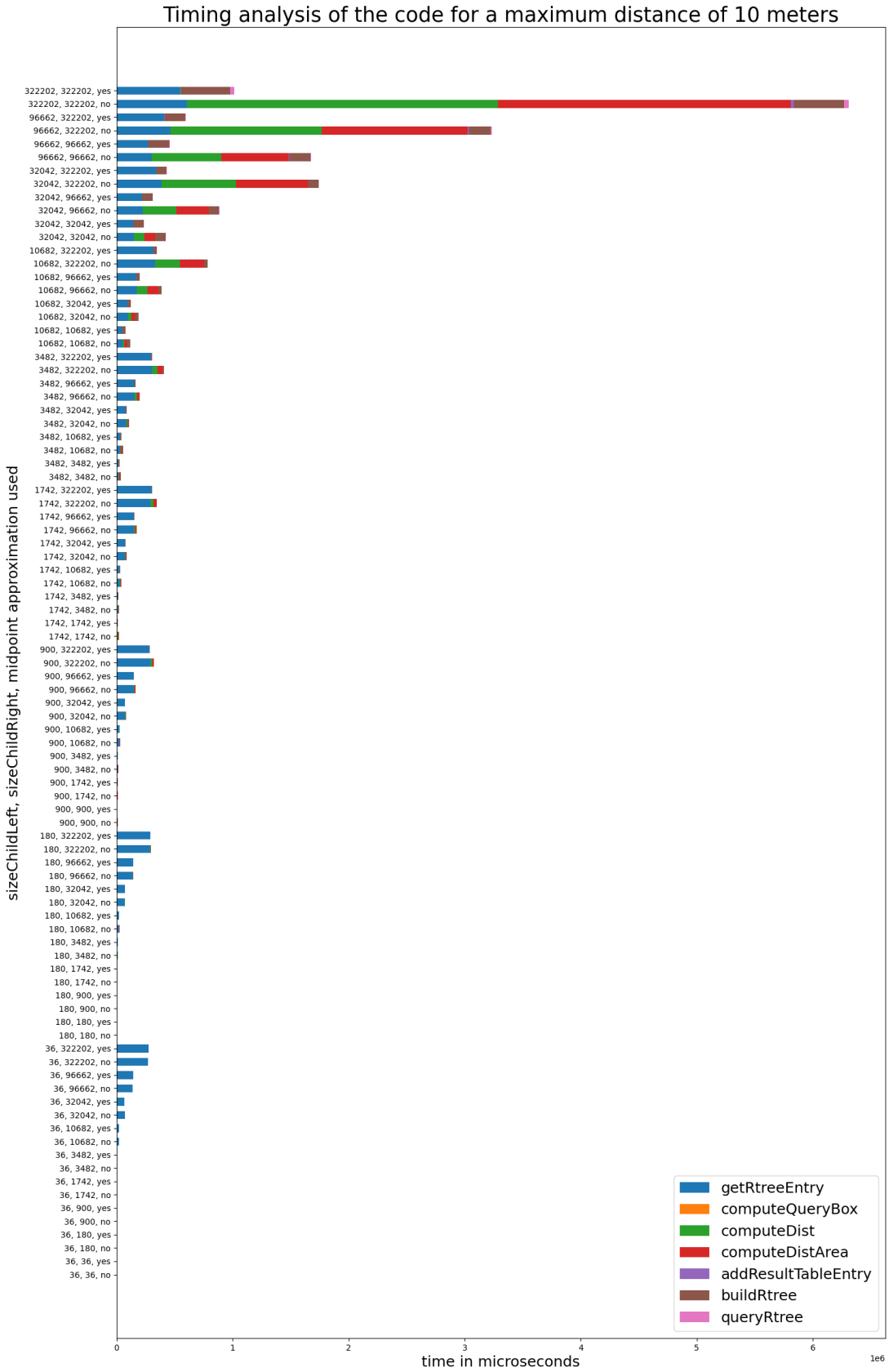


Figure A.48.: This figure shows the times spent in different methods for the query with a maximum distance of 10 meter

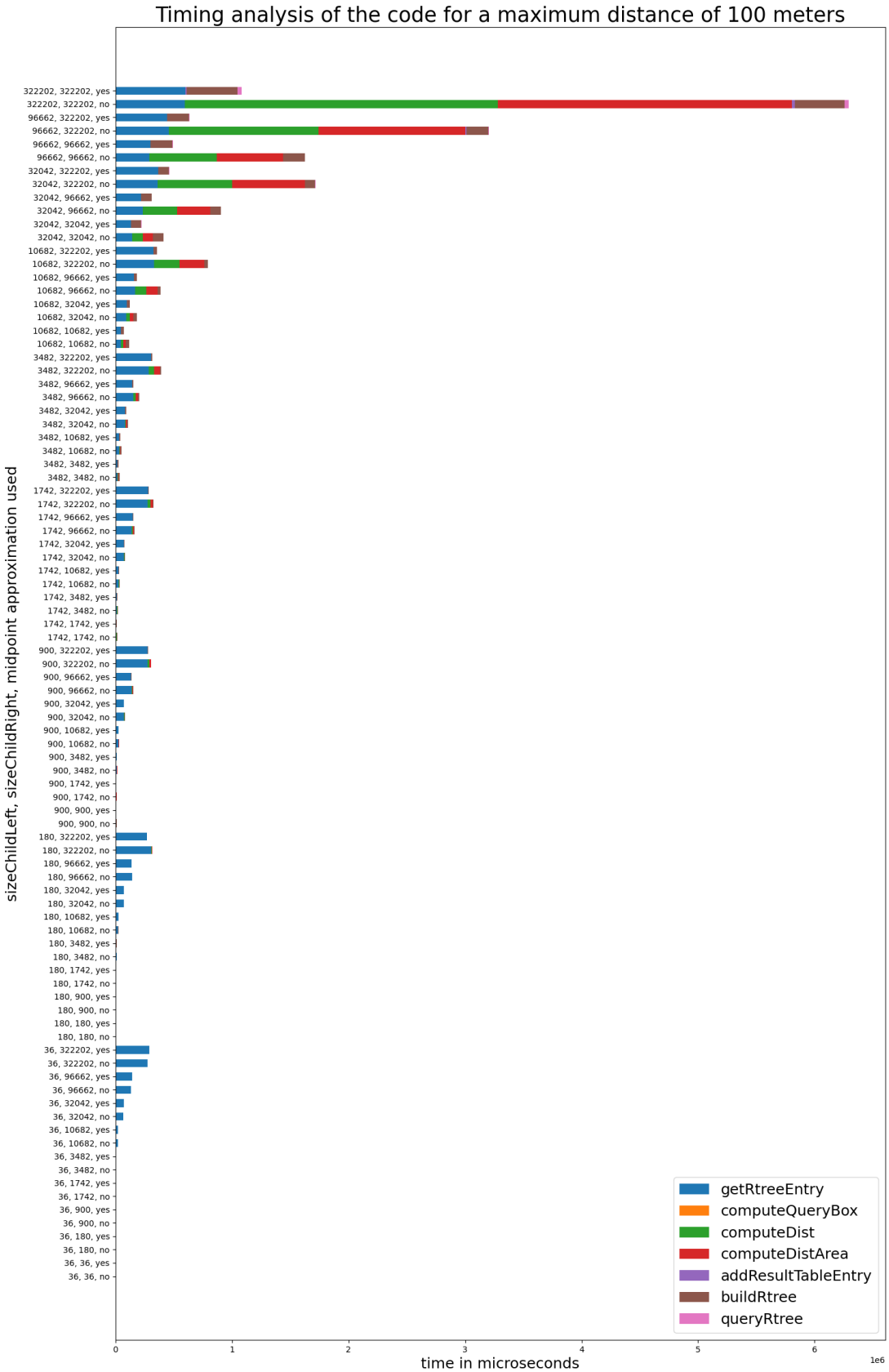


Figure A.49.: This figure shows the times spent in different methods for the query with a maximum distance of 100 meter

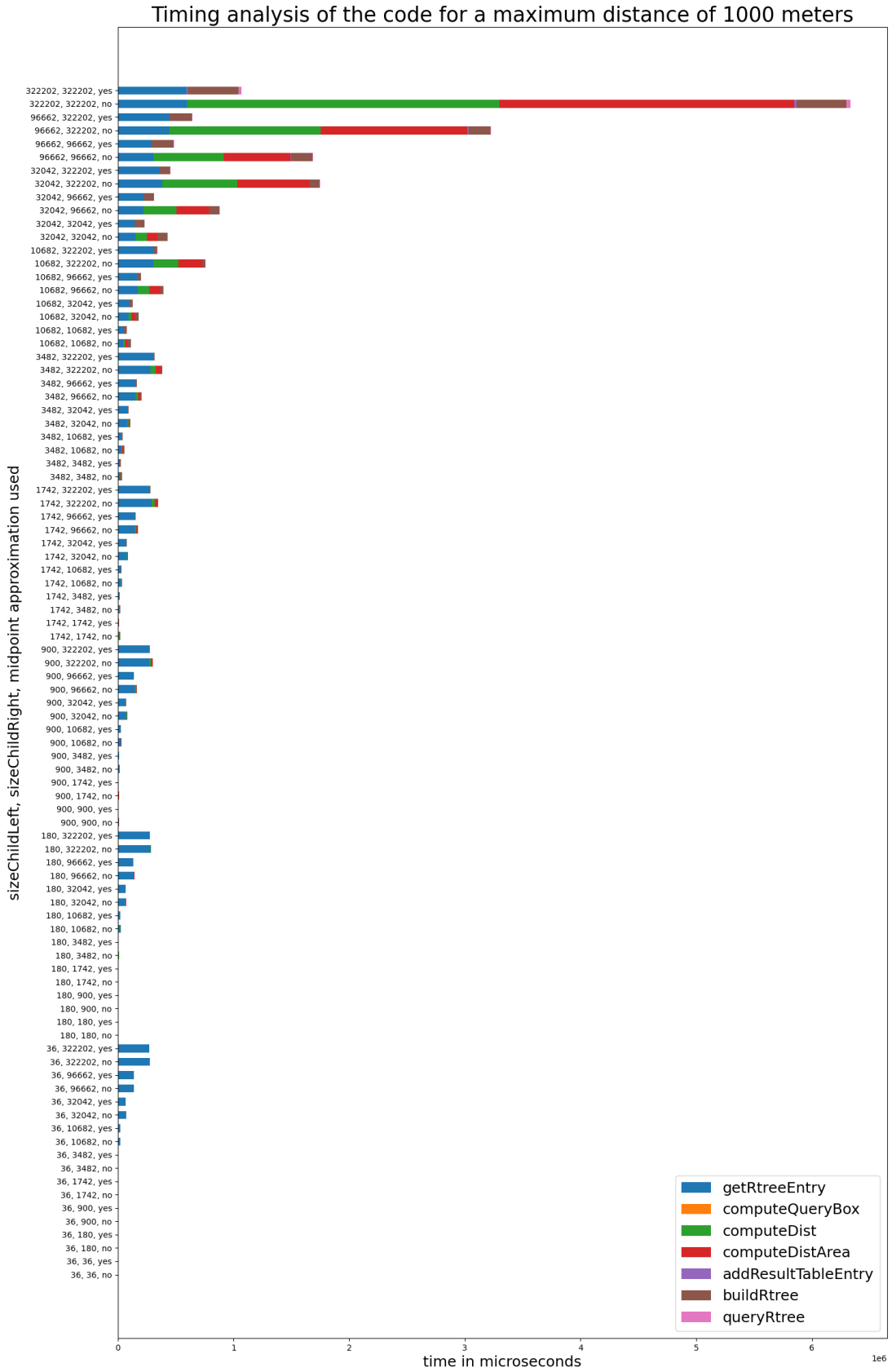


Figure A.50.: This figure shows the times spent in different methods for the query with a maximum distance of 1000 meter

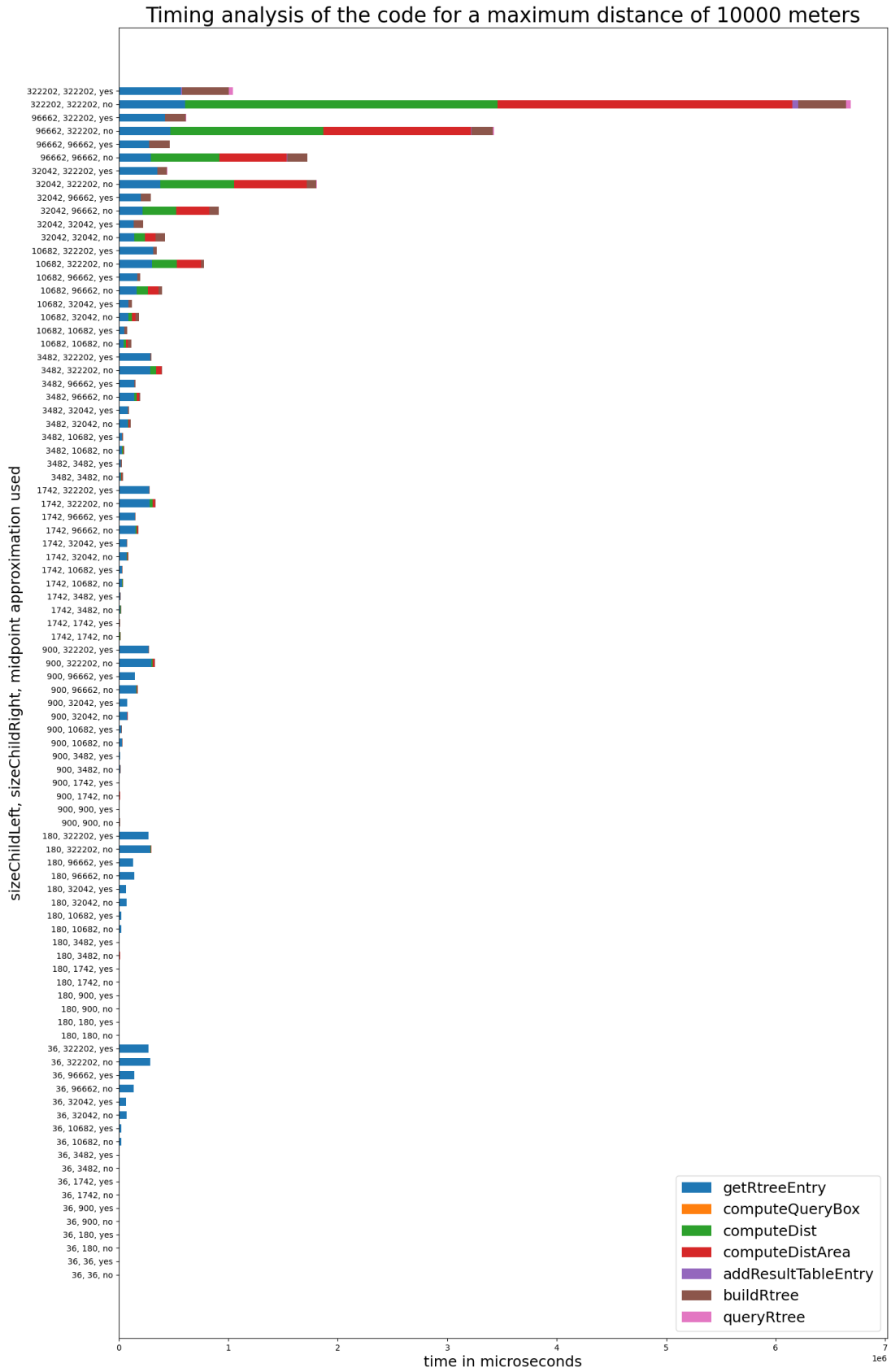


Figure A.51.: This figure shows the times spent in different methods for the query with a maximum distance of 10000 meter

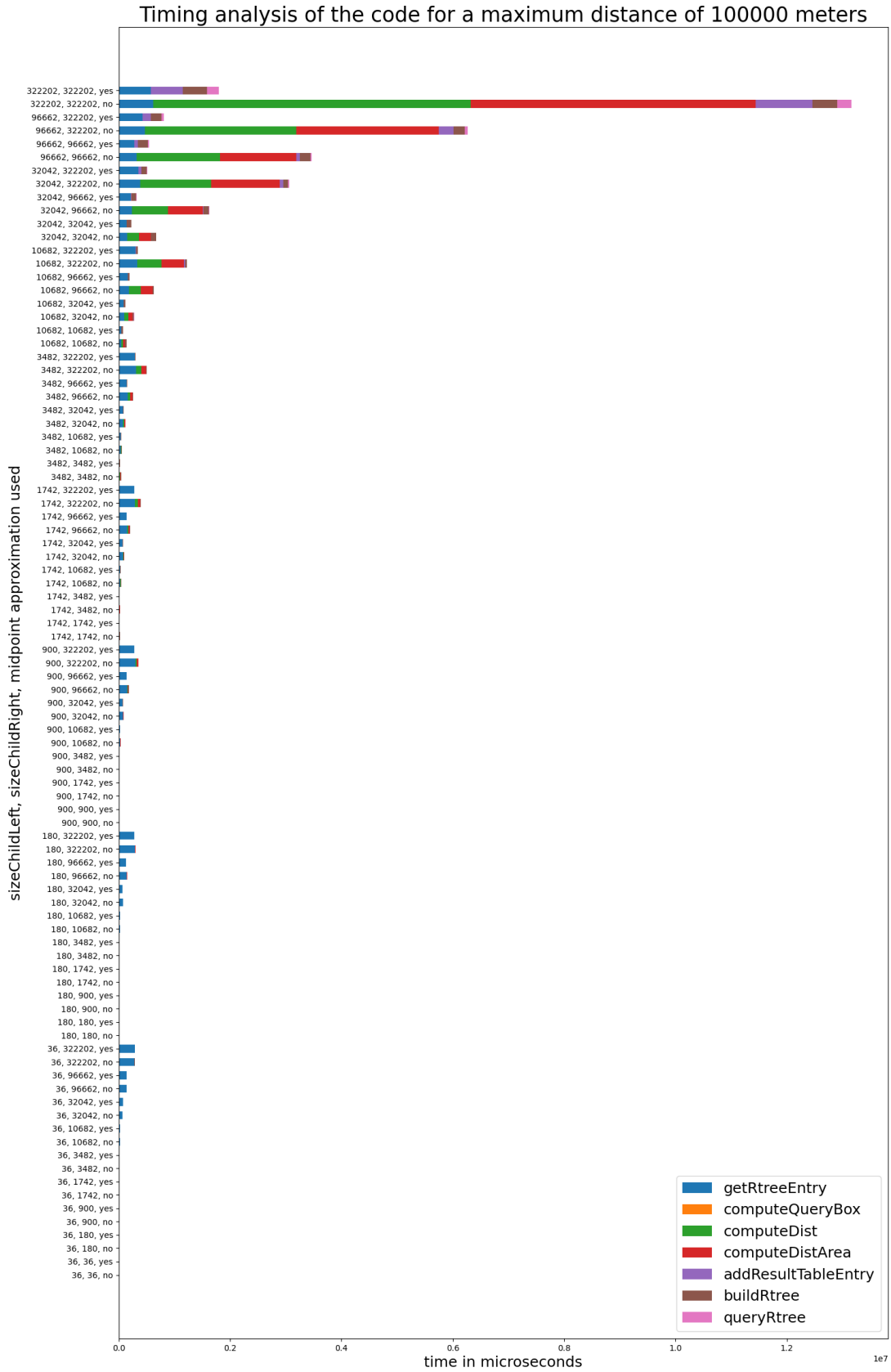


Figure A.52.: This figure shows the times spent in different methods for the query with a maximum distance of 100000 meter

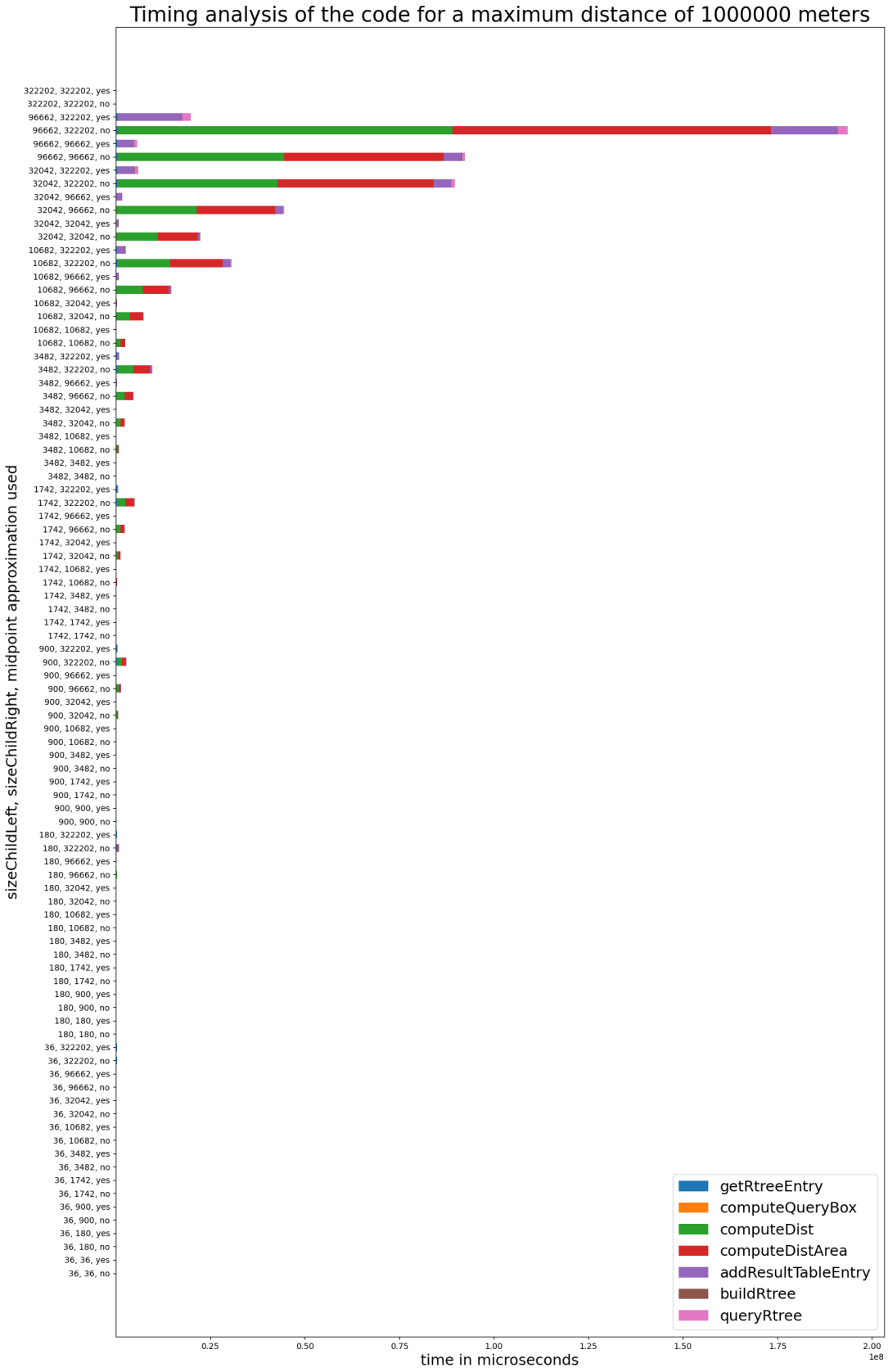


Figure A.53.: This figure shows the times spent in different methods for the query with a maximum distance of 1000000 meter

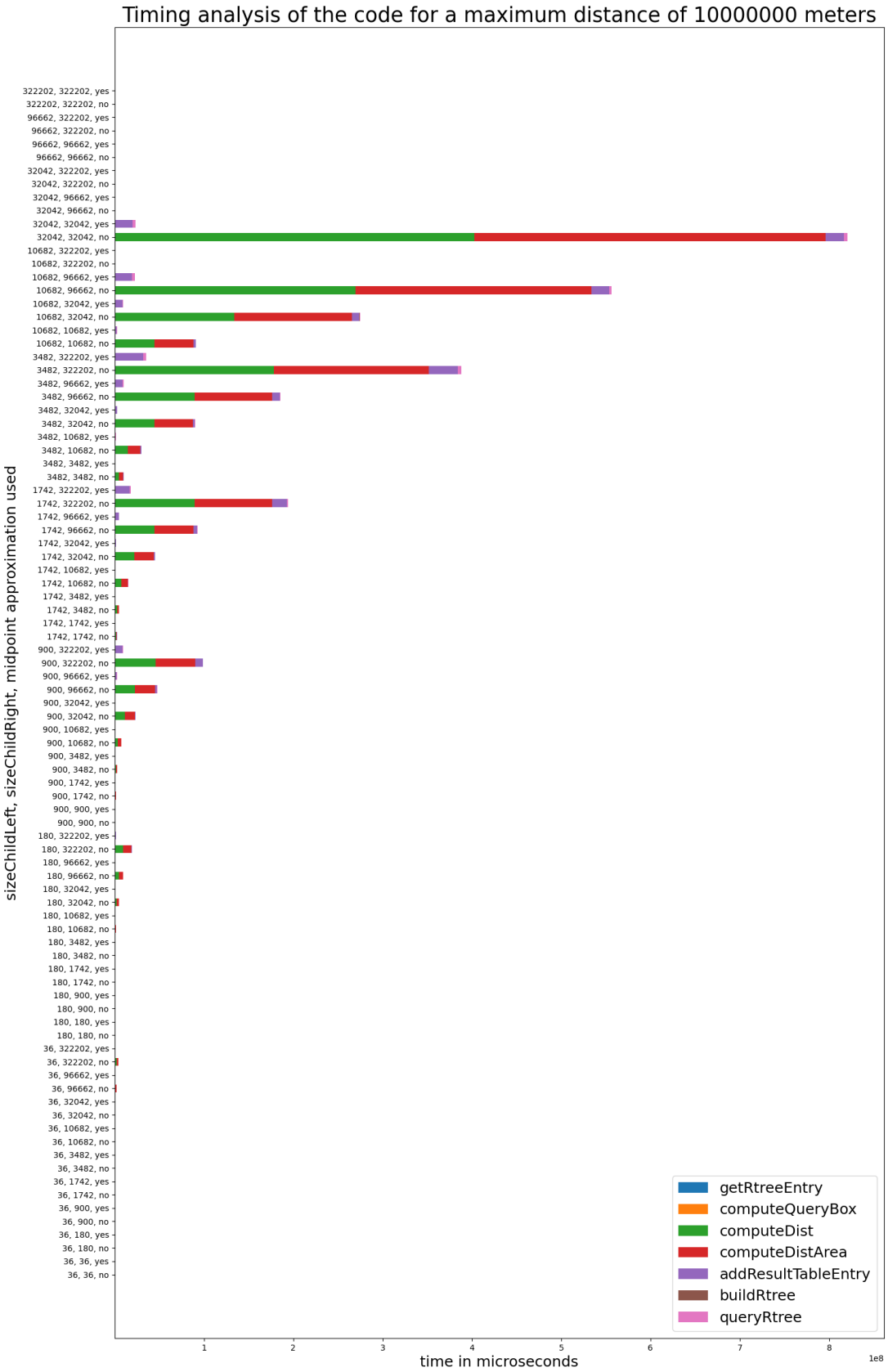


Figure A.54.: This figure shows the times spent in different methods for the query with a maximum distance of 10000000 meter

Bibliography

[1] OpenStreetMap Foundation, "Openstreetmap." [Online]. Available: https://www.openstreetmap.org/

[2] Hannah Bast and Björn Buchhold, "Qlever: A query engine for efficient sparql+text search," 2017.

[3] Hannah Bast, Patrick Brosi, Johannes Kalmbach and Axel Lehmann, "An efficient rdf converter and sparql endpoint for the complete openstreetmap data," 2021.

[4] Antonin Guttman, "R-trees: a dynamic index structure for spatial searching," 1984.

[5] Google, "S2 Geometry Library," https://s2geometry.io/, n.d.

[6] Open Geospatial Consortium, "OGC GeoSPARQL - A Geographic Query Language for RDF Data," Open Geospatial Consortium, Tech. Rep. OGC 22-047r1, 2024. [Online]. Available: https://docs.ogc.org/is/22-047r1/22-047r1.html

[7] The Apache Software Foundation, "Apache jena geosparql documentation." [Online]. Available: https://jena.apache.org/documentation/geosparql/index.html

[8] ontotext, "Graphdb." [Online]. Available: https://graphdb.ontotext.com/documentation/10.8/index.html

[9] Hannah Bast, Johannes Kalmbach, Theresa Klumpp, Florian Kramer, Niklas Schnelle, "Efficient and effective sparql autocompletion on very large knowledge graphs," 2022.

[10] Wikidata contributors, "Wikidata: The Free Knowledge Base," 2025. [Online]. Available: https://www.wikidata.org/

[11] Google, Inc., "Freebase: An Open Knowledge Graph," 2016. [Online]. Available: https://developers.google.com/freebase/

[12] "Knowledge graphs," 2023. [Online]. Available: https://ad-publications.cs.uni-freiburg.de/CHAPTER_knowledge_graphs_BKKK_2023.pdf

[13] W3C, "Sparql 1.1 query language." [Online]. Available: https://www.w3.org/TR/sparql11-query/

[14] OpenStreetMap Foundation, “Openstreetmap wiki - elements.” [Online]. Available: https://wiki.openstreetmap.org/wiki/Elements

[15] Office of Geomatics, “World geodetic system 1984 (wgs 84).” [Online]. Available: https://earth-info.nga.mil/?dir=wgs84&action=wgs84

[16] OpenStreetMap Foundation, “Openstreetmap wiki - xml.” [Online]. Available: https://wiki.openstreetmap.org/wiki/OSM_XML

[17] Barend Gehrels, Bruno Lalande, Mateusz Loskot, Adam Wulkiewicz, Oracle and/or its affiliates, “boost geometry index rtree.” [Online]. Available: https://www.boost.org/doc/libs/1_87_0/libs/geometry/doc/html/geometry/reference/spatial_indexes/boost___geometry___index___rtree.html

[18] Cmglee, “WGS84 mean Earth radius.” [Online]. Available: https://en.wikipedia.org/wiki/File:WGS84_mean_Earth_radius.svg

[19] Peter Mercator, “Latitude and longitude graticule on a sphere.” [Online]. Available: https://en.wikipedia.org/wiki/File:Latitude_and_longitude_graticule_on_a_sphere.svg

[20] Wim Pijls, “Some properties related to mercator projection,” *The American Mathematical Monthly*, vol. 108, no. 6, pp. 537–543, 2001. [Online]. Available: https://doi.org/10.1080/00029890.2001.11919781

[21] “Projection cylindrique.” [Online]. Available: https://commons.wikimedia.org/wiki/File:Projection_cylindrique.jpg

[22] Lars H. Rohwedder, “Normal Mercator map 85deg.” [Online]. Available: https://de.wikipedia.org/wiki/Datei:Normal_Mercator_map_85deg.jpg

[23] Strebe, “Equirectangular projection SW.” [Online]. Available: https://en.wikipedia.org/wiki/File:Equirectangular_projection_SW.jpg

[24] Wikipedia, “Liste der Groß- und Mittelstädte in Deutschland.” [Online]. Available: https://de.wikipedia.org/wiki/Liste_der_Gro%C3%9F-_und_Mittelst%C3%A4dte_in_Deutschland