

Big Spatial Data Analysis and Processing

Master Thesis, August 2018
Department of Computer Science
University of Applied Science Rapperswil

Author: Philipp Koster
Advisor: Stefan Keller

Zusammenfassung

Big Data steht für solch grosse Datensätze, bei welchen traditionelle Ansätze und Technologien an ihre Grenzen kommen und Abfragen nicht mehr in zumutbarer Zeit ausgeführt werden können. Da allerdings heutzutage immer mehr Daten gesammelt und verarbeitet werden, haben sich verschiedene Technologien etabliert die das Speichern, Prozessieren, Analysieren und auch Visualisieren von Big Data effizienter machen.

Geospatial Big Data kann als Subset von Big Data verstanden werden, bei welchem die Daten geographische Attribute wie Punkte, Linien und Flächen beinhalten. Da der Anteil der geographischen Daten immer weiter wächst, wird das performante Prozessieren dieser immer wichtiger. Dafür existieren bereits Erweiterungen für verbreitete Big Data Technologien, dessen Funktionsumfang allerdings noch sehr klein ist.

Das Ziel dieser Thesis ist es, den Einsatz von Geospatial Big Data Technologien anhand zweier Anwendungsfälle zu evaluieren. Beim ersten Anwendungsfall werden aus OpenstreetMap Daten sogenannte Areas-of-Interest (AOIs) extrahiert, welche dem Leser einer Karte einen Eindruck über hochfrequentierte und sehenswerte Bereiche einer Stadt geben. Beim zweiten Anwendungsfall, der in Zusammenarbeit mit einem Industriepartner entwickelt wird, werden Berechnungen und Auswertungen auf einer grossen Anzahl GPS-Punkte ausgeführt. Beide Anwendungsfälle stossen an das Limit eines klassischen DBMS, welches im ersten Teil dieser Thesis anhand einer Implementierung mit PostgreSQL aufgezeigt wird. Im zweiten Teil wird ein geeigneter Big Data Technologie Stack eingeführt und dessen Potential und Limitierungen aufgezeigt.

Abstract

Big Data defines such large datasets, where traditional approaches and technologies reach their limits and analytical queries can no longer be executed in a reasonable time. However, as more and more data is collected and processed, different technologies have been established to make storing, processing, analyzing and visualizing Big Data more efficient.

Geospatial Big Data can be understood as a subset of Big Data containing geographic attributes such as points, lines and areas. As the amount of geographic data continues to grow everyday, high-performance processing is becoming increasingly relevant and necessary. There are different extensions for popular Big Data technologies, but their functional scope is still very limited.

The aim of this thesis is to evaluate the use of Geospatial Big Data technologies in two use cases. In the first use case, Areas-of-Interest (AOIs) are created from OpenstreetMap data. AOIs give the map reader an impression of certain areas in a city that are highly frequented and worth visiting. In the second use case, developed in cooperation with an industrial partner, analytical queries are executed on a large number of GPS points. Both applications reach the limit of a classic DBMS, which is demonstrated in the first part of this thesis by an implementation with PostgreSQL. The second part introduces a suitable Big Data technology stack and shows its potential strengths as well as limitations.

Management Summary

Nowadays, more and more data is being collected in various domains. To process such large amounts of data, new Big Data technologies like Hadoop or Apache Spark have evolved. These technologies split and distribute the calculations to the worker nodes of a cluster. The present thesis deals with the evaluation of Apache Spark in the processing of huge amount of data from the geospatial domain.

Goals

For the evaluation, two use cases are implemented. The evaluation of both the use cases is first done with the traditional PostgreSQL in combination with PostGIS, and later with Apache Spark along with a suitable geospatial extension. Both implementations are benchmarked and compared. The efficacy of the Big Data technology is finally evaluated.

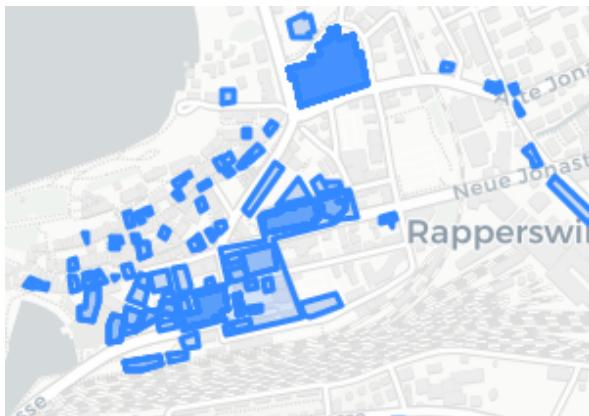
Results

The first use case involves generation of Areas-of-Interest (AOIs) using PostgreSQL along with PostGIS. For this, Points-of-Interests (POIs) taken from the OpenStreetMap data are clustered. The hulls of the clusters are then extended using a network centrality algorithm. Finally, the extended hulls are sanitized by removing irrelevant areas and merging overlapping regions. The generated AOIs are either accessible as GeoJSON file or can be generated with a web application.

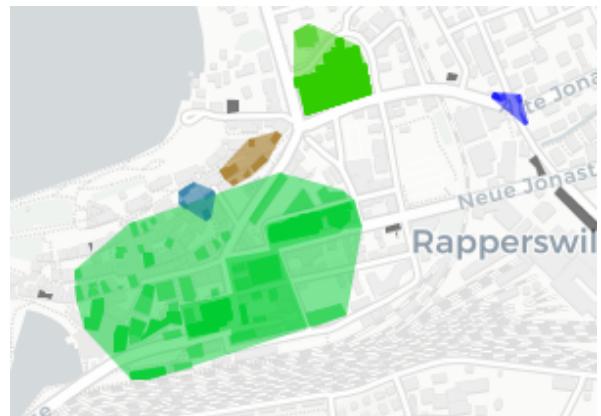
For the second use case, a problem encountered by an industrial partner was solved. They reported to have faced performance issues while using PostgreSQL and PostGIS. In a first step, their implementation was optimized and the performance could be improved to their satisfaction, simply by applying diligently sequenced PostgreSQL and PostGIS queries.

A geospatial extension of Apache Spark, namely GeoSpark, has also been evaluated in the thesis. While implementing the use cases, various pitfalls and bugs were encountered. Additionally, GeoSpark does not yet contain the necessary scope of functionality to implement a complex use case like the generation of AOIs. Therefore, only a relatively small subset of the PostGIS implementation could be re-implemented with GeoSpark.

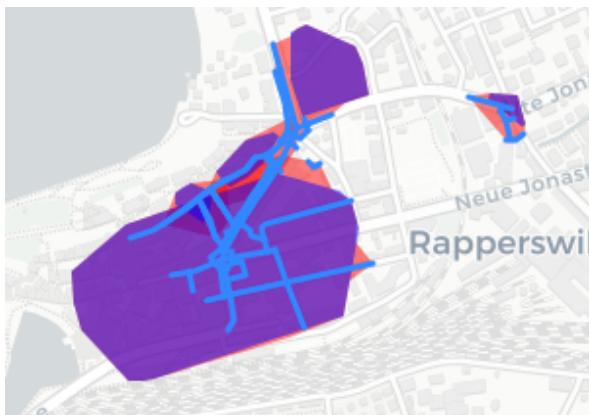
For the second use case, the GeoSpark implementation outperformed the PostGIS implementation, primarily due to its capability to parallelize the execution. However, it must be mentioned here, that the GeoSpark implementation was run with much stronger hardware. Moreover, Apache Spark intrinsically has a better memory utilization. Therefore, the comparison is not fair in all aspects.



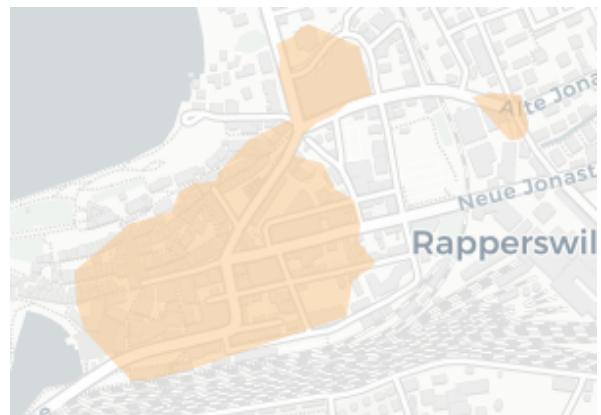
Select POIs by relevant tags.



Cluster POIs and draw hulls.



Extend hulls with network centrality.



Sanitize and export AOIs.

Illustration of Areas-of-Interest creation for Rapperswil.

Future Outlook

The use of GeoSpark is not yet production-ready for complex applications. However, it is under active development and the quality and scope of functionality is increasing continuously. Whether with GeoSpark or with some other similar solution, the ability to process geospatial data on a cluster will certainly gain importance in the near future.

The generated AOIs can be improved in many ways. The applied clustering algorithm can be refined and new data sources can be added. To improve the performance and provide faster queries, the extension through the network centrality needs to be enhanced, so that no external API needs to be called.

Declaration of Originality

I hereby declare that

- this thesis and the work reported herein was composed by and originated entirely from me unless stated otherwise in the assignment of tasks or agreed otherwise in writing with the supervisor;
- all information derived from the published and unpublished work of others has been acknowledged in the text and references are accordingly given in the bibliography; and
- no copyrighted material (such as images) has been used illicitly in this work.

Place, Date: Rapperswil, 29.06.2018

Name, Signature Philipp Koster



Contents

I. Introduction and Problem Statement	9
1. Overview	10
2. Use Case AOI	11
2.1. State of the Art	11
2.2. AOI with OpenStreetMap	15
3. Use Case Event Count	22
3.1. Data	23
4. Geospatial Big Data	25
4.1. Big Data	25
4.2. Apache Spark	25
II. Implementation with PostgreSQL and PostGIS	29
5. Use Case AOI	30
5.1. Implementation	30
5.2. Web Application	41
5.3. Benchmarks	45
6. Use Case Event Count	47
6.1. Existing Implementation	47
6.2. Benchmarks and Bottlenecks	49
6.3. Optimizations	54
6.4. Benchmarks after Optimizations	68
III. Implementation with Apache Spark	70
7. Evaluation	71
7.1. Criteria	71
7.2. Solutions	71
7.3. Summary	73
7.4. Data Source	74
8. Use Case AOI	75
8.1. Implementation	75
8.2. Compare Results	83
8.3. Benchmarks	84

9. Use Case Eventcount	86
9.1. Implementation	86
9.2. Benchmarks	88
IV. Summary and Conclusion	91
10. Summary of Results	92
11. Conclusion	93
V. Appendices	94

Part I.

Introduction and Problem Statement

1. Overview

Nowadays more and more data is collected. To process this huge amount of data, for example to perform analytical tasks, new technology stacks have evolved. This is necessary, since traditional technologies like PostgreSQL are not capable of performing queries on large datasets in reasonable time. These new technology stacks are generically known as BigData frameworks. The key concept is to perform a query not on one machine, but on a cluster of machines in parallel. Frameworks like MapReduce from Google, Apache Hadoop and Apache Spark are widely used and proven to be effective in practice.

Geospatial data refers to spatial datasets. Examples of spatial data are locations, streets or city boundaries, which could be represented as points, lines or polygons. Especially in the era of smart-phones and navigation devices, a lot of GPS information is collected and need to be processed and analysed.

The present thesis evaluates the use of a BigData framework, namely Apache Spark, for the particular case of geospatial data. Different extensions for Apache Spark exist, which can be used to process geodata. However, none of them have yet gained much popularity. Furthermore, there are hardly any benchmarks. With this in view, two use cases are implemented in the thesis.

In the first use case, Areas-of-Interest (AOIs) are generated from the data of OpenStreetMap. OpenStreetMap is a map of the world based on a collaborative, volunteered effort. AOIs are urban areas around a city or its neighbourhood with a high concentration of Points-of-Interest (POIs) typically located along a street of high spatial importance [15]. The calculation of AOI includes expensive spatial queries, for example the application of clustering algorithms.

The second use cases is implemented for an industrial partner. They capture a large amount of events which contain, besides other attributes, a timestamp and a GPS location. For analytical purpose they need to count the events that occurred inside certain areas. There can be hundreds of areas for one query. Since these analytical queries contains expensive spatial conditions, the runtime of the queries become unacceptable.

In the first part of this thesis, both use cases are described in more detail in Sections 2 and 3. Further, the Big Data Framework Apache Spark is introduced in Section 4.

In the second part of this thesis, the use cases are implemented with a traditional relational database, namely PostgreSQL and PostGIS. With these implementations, benchmarks are performed and documented. The benchmark results show the limitations and the bottlenecks of the implemented use cases.

In part three the same use cases are implemented with Apache Spark. Therefore, a suitable extension for Apache Spark which can handle geospatial queries is evaluated and used. The implementation is described in detail and the same benchmarks as with the PostgreSQL implementation are performed and compared.

The fourth part summarizes the results of the thesis and draws appropriate conclusions.

2. Use Case AOI

The objective of Areas-of-Interest (AOIs) is to give the viewers of a map an impression that there is a lot of activity in that area. For example, restaurants, bars, shops, museums are places of interest for tourists. Therefore, the AOIs on a map show tourists where it is worth going, for example, the old town, a street with a lot of restaurants or a shopping promenade.

Definition of AOI

“Urban area at city or neighbourhood level with a high concentration of POI, and typically located along a street of high spatial importance” [15]

This Section first describes the state-of-the-art of AOIs. Next, an approach to generate AOIs based on the data of OpenStreetMap is described.

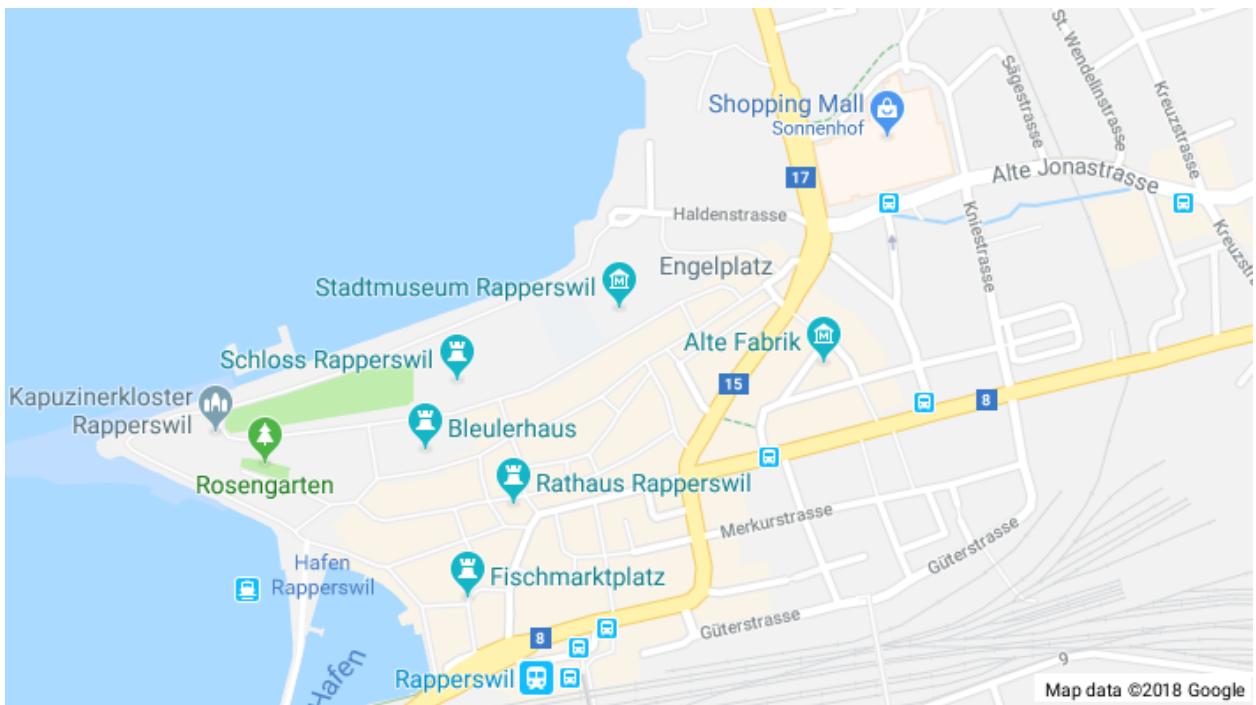
2.1. State of the Art

2.1.1. Google Maps

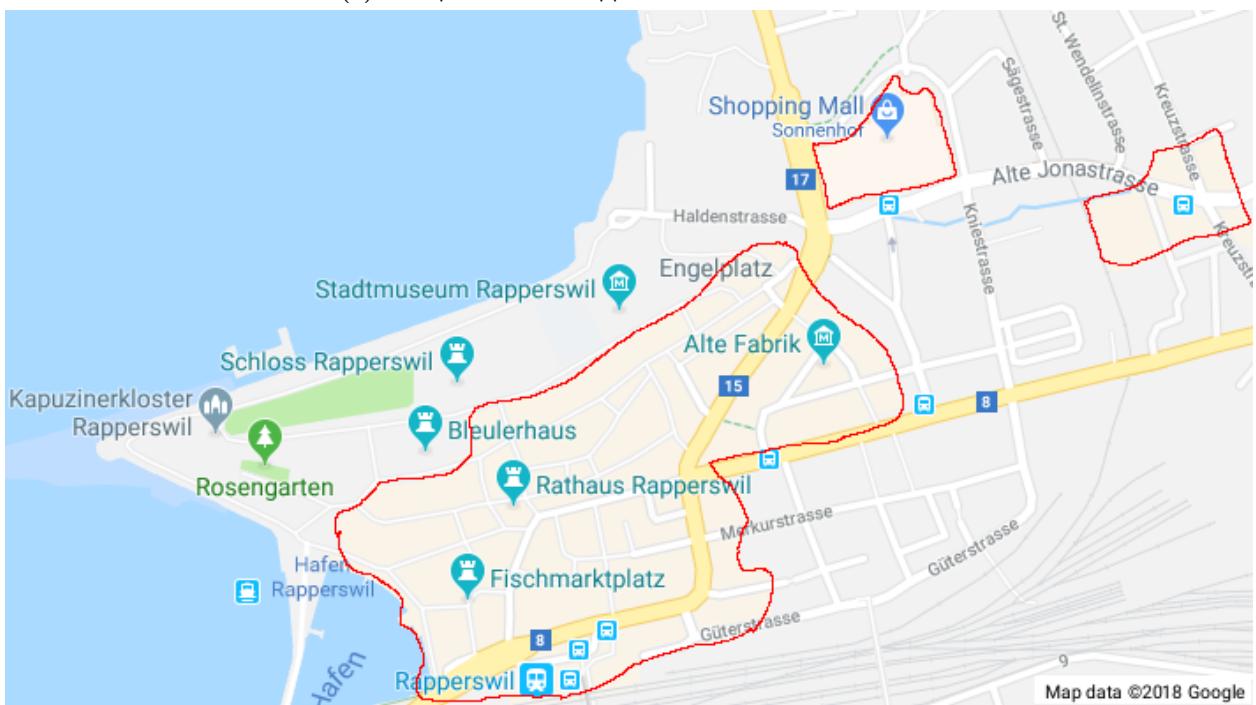
The most popular AOIs are the ones in Google Maps (GMaps)¹. They were introduced in mid 2016 and represent “places where there’s a lot of activities and things to do” with an orange shade [12]. Figure 2.1a shows an example of GMaps with AOIs in the town Rapperswil. As one can see, the old town is for example marked as AOI. Since they can be overlooked very easily, the AOIs are marked in Figure 2.1b with a red line.

One characteristic of the AOIs in GMaps is that they are different on higher zoom levels. When the zoom level is greater or equal 17, single buildings are marked with pale orange, instead of whole areas. This is illustrated in Figure 2.2.

¹They are considered the most popular because they probably have the largest audience. Although, most people never noticed them



(a) GMaps AOIs of Rapperswil with zoom level 16.



(b) Same AOIs, but marked with a red line, since they can be overlooked very easily.

Figure 2.1.: AOIs in Google Maps (GMaps)

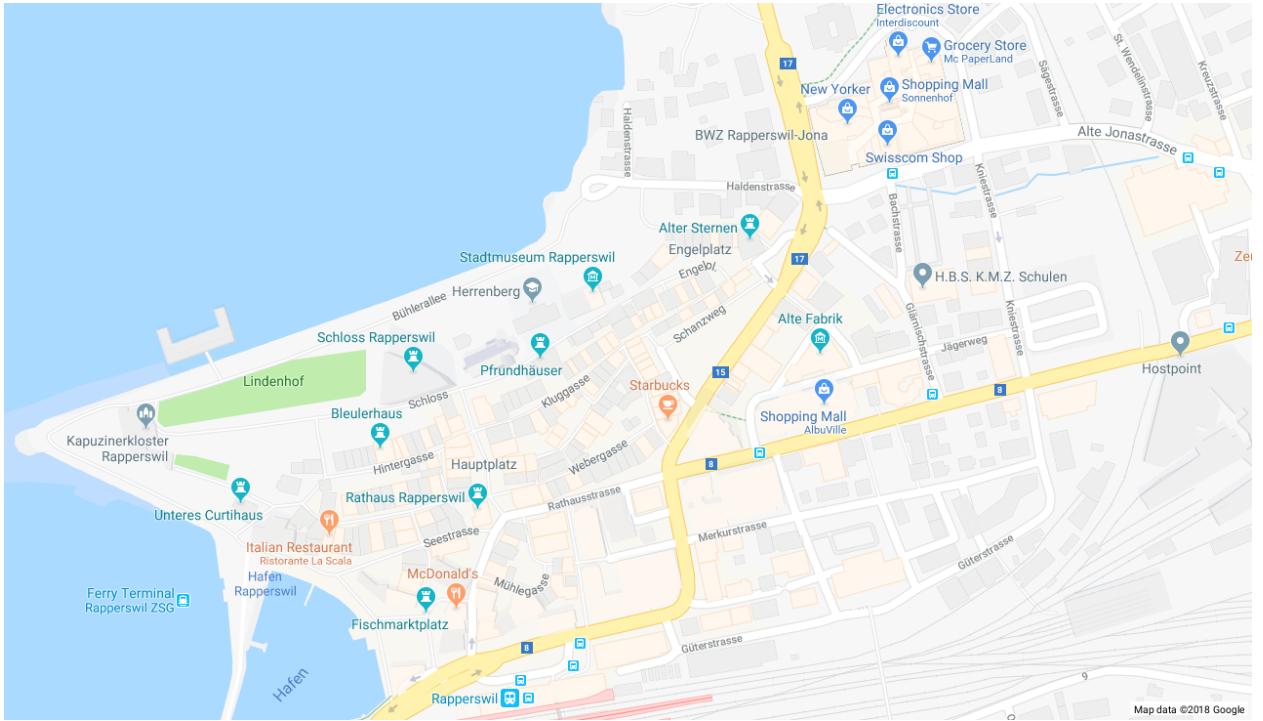


Figure 2.2.: GMaps POIs of Rapperswil with zoom level 17.

The way in which AOIs are generated on GMaps is unknown, since their algorithm, for good reason, is not publicly available. Therefore, one can only assume how AOIs in GMaps are constructed. This has already been evaluated by [15]. Their compilation of observations which could be verified are:

- AOIs are selected based on POI class
- AOIs are spatially clustered from POI points and buildings with some influence of street lines (AOI buildings are aggregated and sometimes look like they are buffered symmetrically on both street sides, even when on the other side there are no AOIs)
- AOIs are selected based on density of surrounding POIs (i.e. a certain amount of POIs must exist within a given radius)
- There is a aggregation or extrapolation of some levels performed on small scaled POIs to make the entire building that houses them into an AOI.

Other assumptions are more difficult to verify, but quite possible, for example:

- AOIs are selected based on (mostly unconscious) analysed user tracks from mobile app
- AOIs are selected based on user reviews and likes.

2.1.2. TopPlace by AVUXI

TopPlace is a product developed by AVUXI. They call it, “a worldwide location rating system which ranks the popularity of every place on Earth” [2]. Among others, TopPlace offers worldwide heat maps. By selecting one category from Sightseeing, Eating, Shopping and Nightlife, a map overlay shows the concentration of the most relevant areas. Figure 2.3 shows the heat maps of Barcelona for the different categories.



Figure 2.3.: Heat maps of TopPlace by AVUXI

AVUXI says they are using more than 60 sources to determine AOIs [2]. In one of their blog posts, they list a few data sources [1]:

- Open Data: Geo-data of each city, in order to have a reference point on which published census data can be retrieved and used.
- Open Data: Census information on immigration and housing.
- Open Data: Industrial and green areas
- Public Data: Information on housing prices.
- AVUXI Data: Categorized GeoPopularity data grouped by census areas.
- AVUXI Data: Heat maps of areas of interest by basic traveler activities: Eating, Shopping, Sightseeing and Nightlife

Even though this information about the AOIs of TopPlace exists, it is not clear how TopPlace generates their AOIs. This lack of clarity is comprehensible since the generated maps are of business value to AVUXI.

AVUXI market their AOIs as superior to the AOIs of GMaps, because with their categories they are more tailored to the different needs of map readers.

It is striking, however, that AOIs exists only in bigger cities and no data is available in small cities or villages. Another downside of their AVUXI's TopPlace is, that it is a paid product and costs at least 49 € per month (as on May 2018).

2.2. AOI with OpenStreetMap

Section 2.1 listed already existing and available AOIs. But they have the downsides, that their creation is not comprehensible or are not free to use. Therefore, the generation of AOIs based on open data, like OpenStreetMap, is desirable.

2.2.1. Approach

The following approach to generate AOIs with OpenStreetMap (OSM) is strongly based on the existing whitepaper "Areas-of-Interest for OpenStreetMap (AOI for OSM)" by Stefan Keller and Kang Zi Jing [15]. Their proposed approach contains the following steps:

- Select relevant POIs
- Cluster selected POIs
- Create areas based on clusters
- Extend areas based on the network centrality
- Sanitize AOIs
- Export AOIs

The individual steps are described in more detail below.

2.2.1.1. Select Relevant POIs

The data of OpenStreetMap contains so-called points of interests (POI). As pointed out in the OpenStreetMap documentation, a POI is not necessarily interesting, since, for example, post boxes are relatively interesting/ uninteresting depending on the context [19]. Some examples for POIs are:

- Churches, schools, town halls, distinctive buildings
- Post offices, shops, postboxes, telephone boxes
- Pubs
- Tourist attractions

Figure 2.4 shows POIs related to food and drinks in Rapperswil using the web application Open-PoiMap², which allows to visualize POIs.

²Available at <http://openpoimap.org>

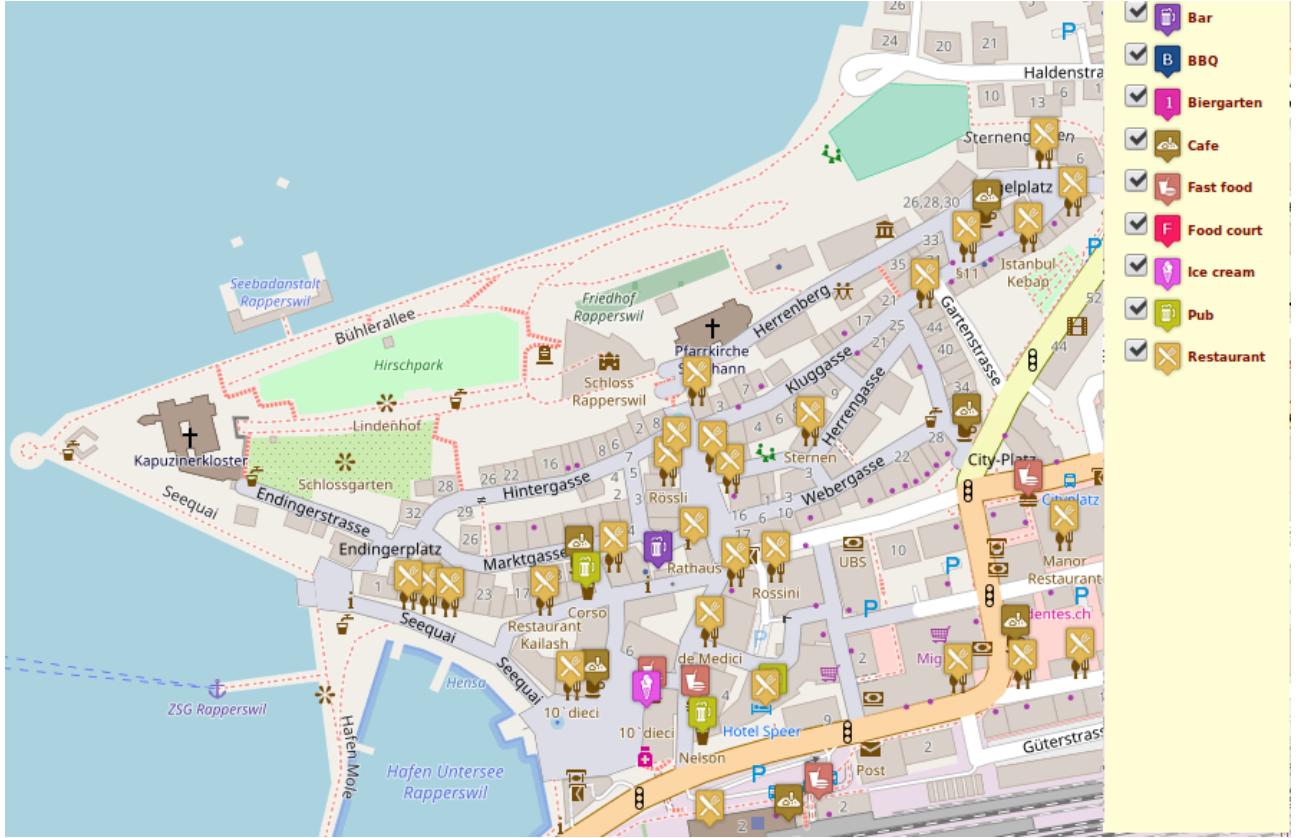


Figure 2.4.: POIs of Rapperswil, related to food and drink, visualized with openpoimap.org.

Objects in OpenStreetMap have tags. For example, a node or polygon which has the tag *amenity* with the value *pub* maps a pub. Therefore, the POIs are selected based on their tags. A list for AOIs relevant tags can be found in the Appendix A.1.

2.2.1.2. Cluster Selected POIs

After the POIs have been selected, they are clustered in the next step. The clustering algorithm Density-Based Spatial Clustering of Applications with Noise (DBSCAN) is suitable for this purpose.

The DBSCAN clustering algorithm takes two arguments, *minPts* and a radius ϵ . If two points are within a maximal distance of ϵ they are called directly reachable. If in the radius ϵ of a point are at least *minPts* points, it is a core point. All core points, at least one, and points which are directly reachable from core points build a cluster. Points which are not reachable from a core point, are called outliers. This is illustrated in Figure 2.5 where the red points are core points, the yellow points are directly reachable from a core point and the blue point is an outlier. The arrows indicate the reachability between the points [28].

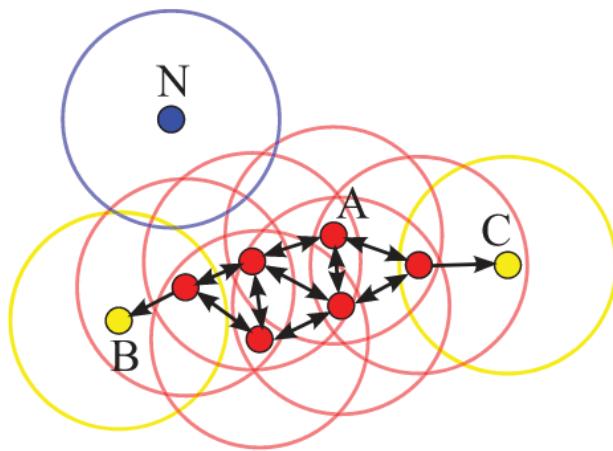


Figure 2.5.: Illustration of the DBSCAN clustering algorithm. Source: [28]

Figure 2.6 illustrates the created clusters when DBSCAN was applied to multiple polygons. As one can see, nearby reachable polygons are clustered, which is indicated by the numbers 0, 1 and 2. Red polygons are noise and therefore not part of a cluster.

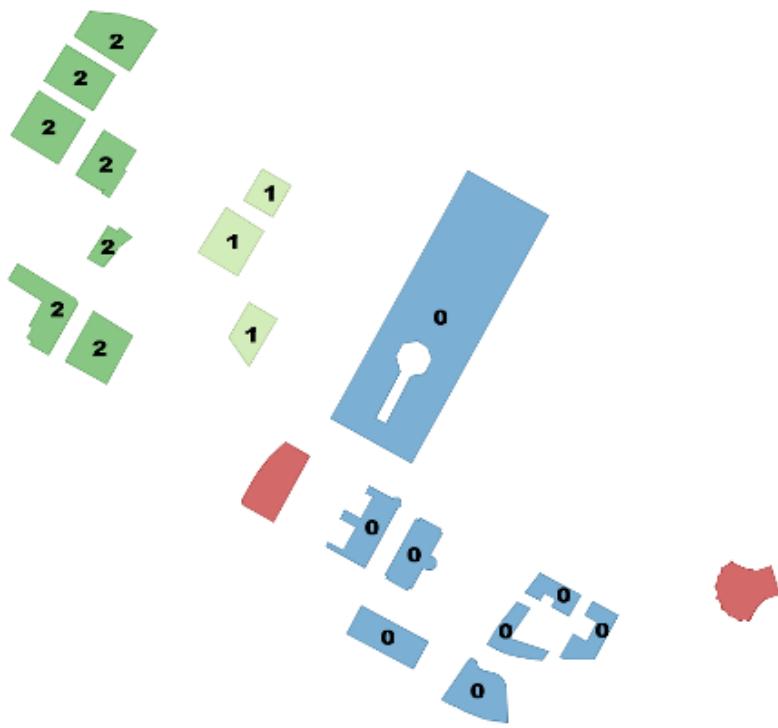


Figure 2.6.: DBSCAN clustering on multiple polygons. Three clusters 0, 1 and 2 are created. Red polygons are not density reachable and therefore not part of a cluster.

Suitable values for minPts and ϵ of the DBSCAN algorithm are evaluated in Section 5.1.3.

2.2.1.3. Create Areas Based on Clusters

After the POIs are clustered, areas can be drawn around the clusters. Either a convex, or a concave hull may be suitable. Figure 2.7 shows a convex and a concave hull around some selected POIs in Rapperswil. Which of the hulls look better—is a matter of personal discretion. In the scope of this thesis only convex hulls are considered. The resulting hulls can be seen as very basic AOIs.



Figure 2.7.: Example for different hulls around certain POIs in Rapperswil.

2.2.1.4. Extend Areas Based on the Network Centrality

The resulting AOIs are only very basic, since they simply express Areas-of-Interest based on concentrations of Points-of-Interest. One way to improve the expressiveness of these AOIs is to take the centrality of the street network into account. This adds value for viewers of the map, as POIs are naturally connected by streets.

The idea is, to express the importance of streets, based on how *central* they are. To quantify this, concepts from the graph theory can be used. In network analysis and graph theory, the concept of centrality expresses how important an edge is. Different measures of centrality exist, which calculates centrality from different considerations. For example, degree centrality, closeness centrality, betweenness centrality and many more.

The closeness centrality defines the centrality of a vertex in a graph as the sum of distances of the shortest paths to all other vertices in the same graph. This is expressed as equation 2.1 where $d(y, x)$ equals the distance of the shortest path between y and x . The centrality is expressed as the reciprocal of the sum, since larger distances to all other vertices equals less centrality.

$$C(x) = \frac{1}{\sum_y d(y, x)}. \quad (2.1)$$

To apply the closeness centrality to a street network, a graph based on the nodes of the network can be created. Nodes in a street network are intersection points of the streets. Figure 2.8 illustrates the street network and the corresponding nodes of Rapperswil.



Figure 2.8.: Street network and the corresponding nodes of Rapperswil.

On the created graph, the closeness centrality algorithm can be applied. The results are the calculated values for the centrality of all nodes. This is illustrated in Figure 2.9.



Figure 2.9.: Results of closeness centrality algorithm. Nodes are colored by their relative centrality, from lowest in dark purple to highest in bright yellow.

Alternatively, the nodes can be seen as edges and the streets as vertices of a graph. This is the inverse of the previous graph, where the centrality of the streets (instead of the intersections) are calculated by applying the closeness centrality algorithm. The resulting graph is illustrated in Figure 2.10.



Figure 2.10.: Results of closeness centrality algorithm on line graph. Streets are colored by their relative centrality, from lowest in dark purple to highest in bright yellow.

The general idea of combining the network centrality with the AOIs from Section 2.2.1.3 is, to slightly extend the AOIs to certain zones where important roads are located. Figure 2.11 illustrates this by overlaying an example AOI in Rapperswil with the corresponding network centrality graph. Sections where the AOI could be extended slightly, are highlighted with red.

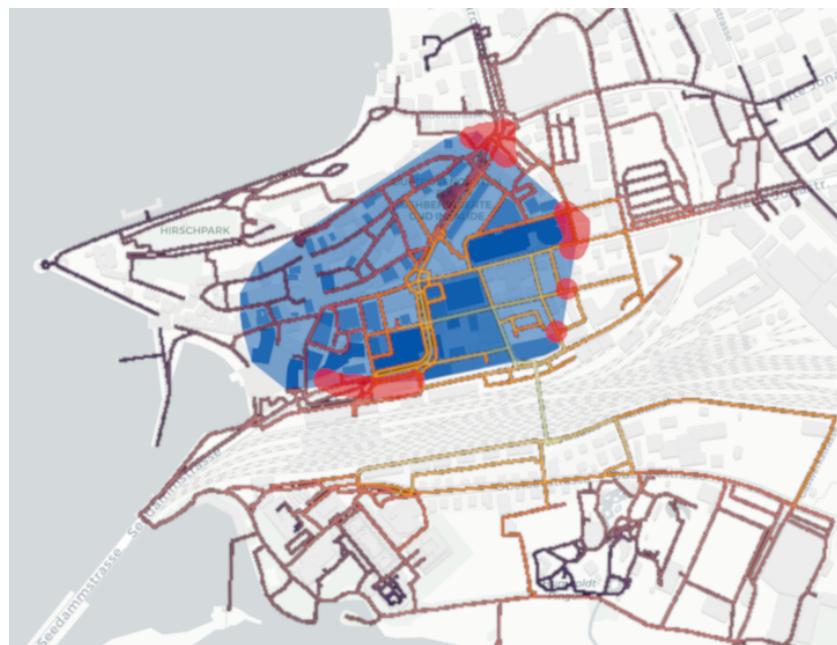


Figure 2.11.: Example AOI of Rapperswil overlaid with network centrality. Zones where the AOI could be extended are marked red.

2.2.1.5. Sanitize AOIs

To improve the significance of the AOIs, certain areas need to be excluded. For example, lakes and rivers are useless as AOIs. Figure 2.12 shows this with an example of a river which is included in a AOI of Zürich.

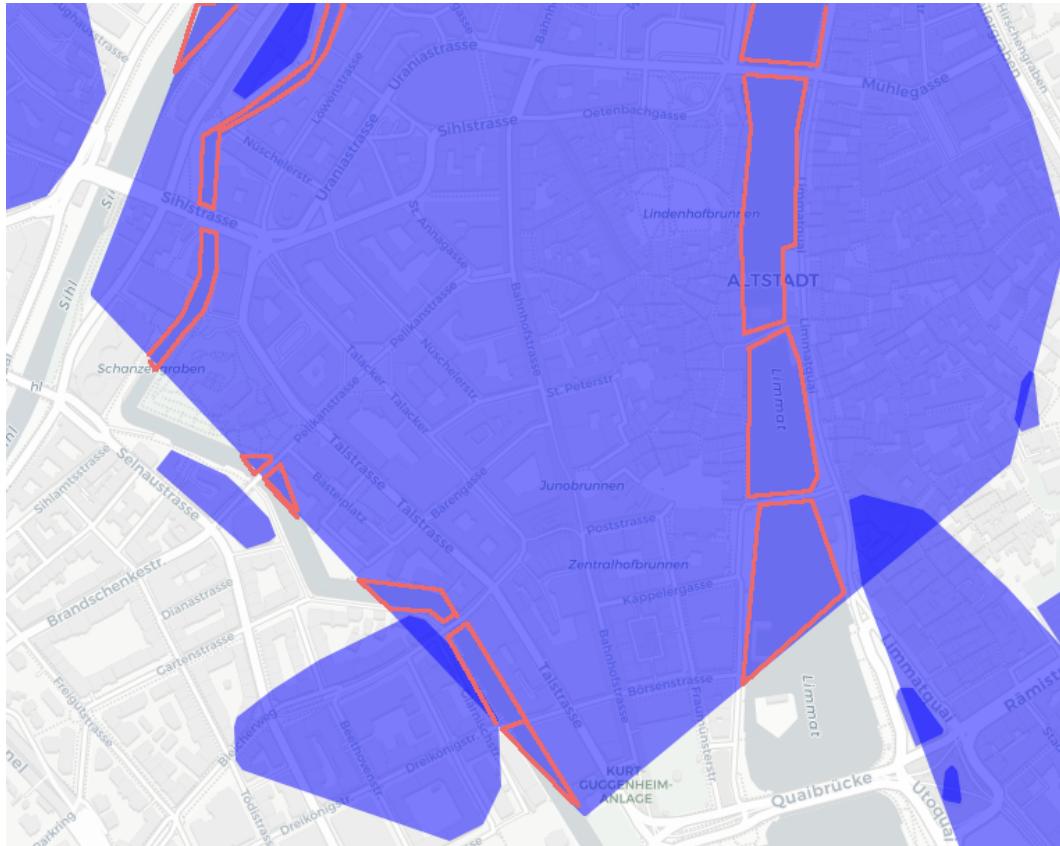


Figure 2.12.: An example AOI which includes rivers. Areas which should be excluded are marked red.

2.2.1.6. Export AOIs

After the AOIs have been sanitized, they are ready to export. A suitable format is GeoJSON which encodes geographic data structures. It is specified with RFC 7946³.

³For more details see <https://tools.ietf.org/html/rfc7946>

3. Use Case Event Count

As already mentioned in Section 1, this use case is a request of an industrial partner. The task is to count all events which have taken place within a given polygon and a specified range of time. An example would be to count all events which have taken place at the main railway station in Zürich, between the 1st and 14th of August 2017.

To count the occurrence of events inside a given polygon and within a range of time, all events are filtered by time stamp and locations. Since the use of GPS causes the accuracy of the captured location to vary between a few meters and hundreds of meters, an event is considered to have occurred inside a polygon if at least 50 percent of the area of an event is inside the polygon. As Figure 3.1 shows, the area of an event is represented by a circle with the location as the center and the accuracy as the radius of the circle. Figure 3.2 shows the different possible cases of the intersection of an event area and a polygon.

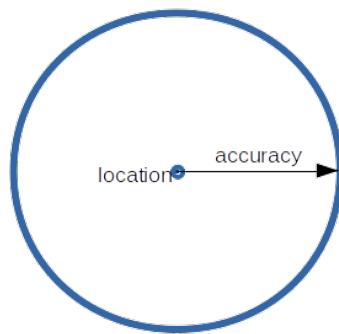


Figure 3.1.: The area of an event is represented as a circle with the even location as the center and its accuracy as the radius of the circle.

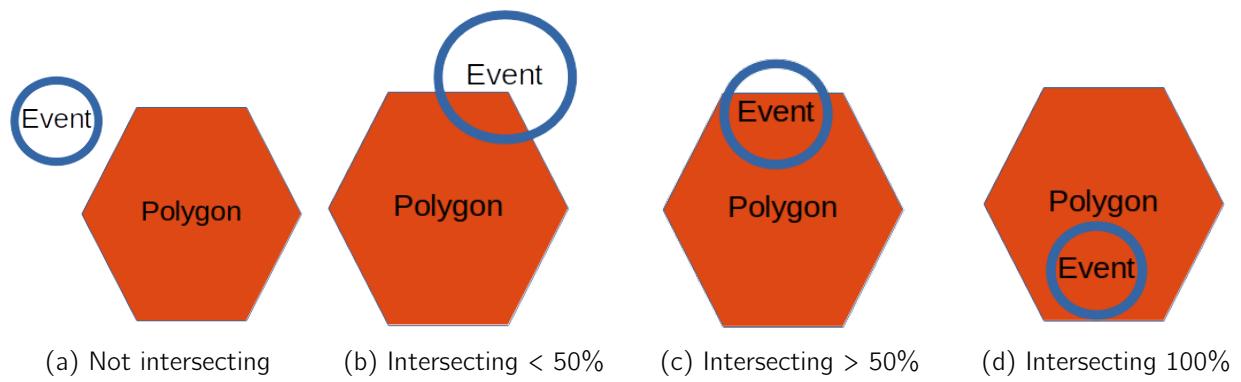


Figure 3.2.: Different cases of an event area intersecting a polygon.

3.1. Data

This section describes the nature of data that are processed and analyzed for the use case. Table 3.1 lists all attributes of the data with the corresponding data type and an example value.

Table 3.1.: Attributes of the data with the corresponding data type and an example value

Attribute	Data Type	Example Value
id	bigint	13
consumer_id	varchar (64)	30f2795dbeaafef586cefd63bfe...
occurred_at	timestamp	20170711 08:53:15+00
location	geometry (Point, 4326)	0101000020E6100000D64C08670...
horizontal_accuracy	double	65
session_id	bigint	99

Brief explanations for some important attributes are provided below.

location

The location attribute is a point geometry with the WGS84 geographic coordinate reference system¹. This is because GPS is used to record the location.

horizontal_accuracy

The horizontal accuracy attribute is also part of the captured location. The value represents in meters with what accuracy it was possible to capture the location. This depends on the quality of the GPS, Wifi and phone signal. Figure 3.3 shows the distribution of those values which are special cases or have more than 100000 records.

¹The WGS84 CRS has the EPSG identifier 4326

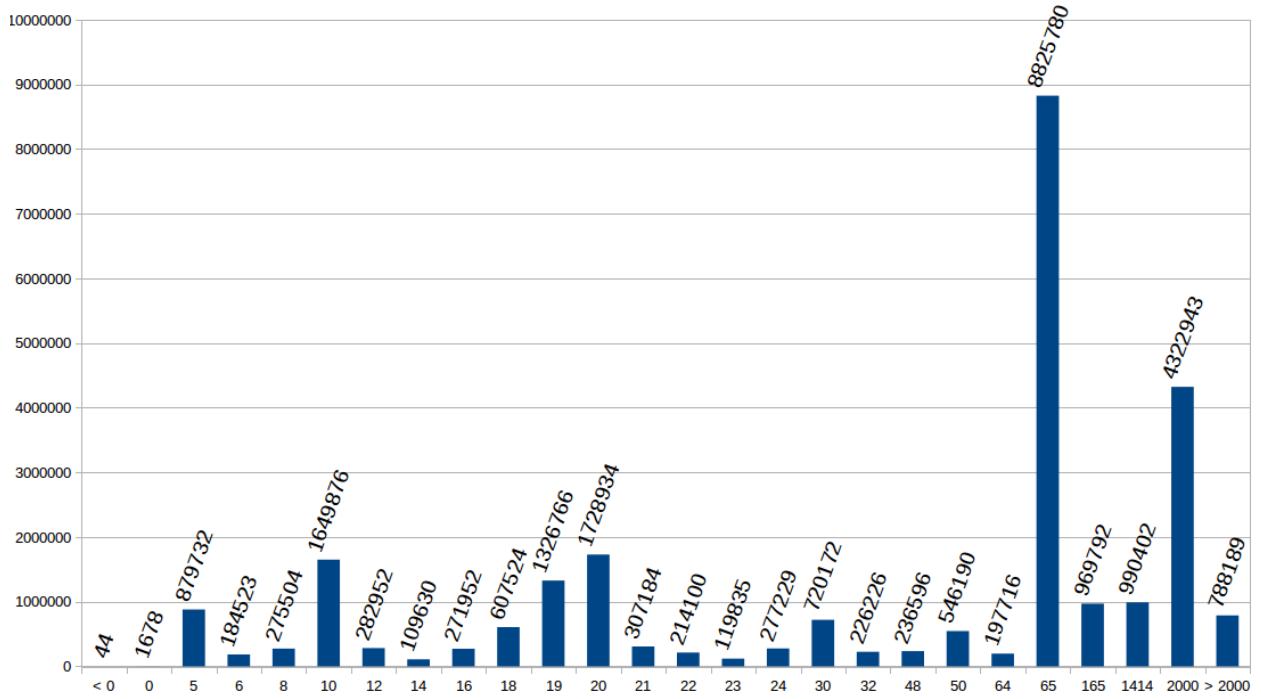


Figure 3.3.: Distribution of horizontal_accuracy attribute (special cases and values with more than 100000 records).

Obviously, an accuracy less than 0 makes no sense, therefore it is wrong data. For values equal to 0, the same applies, because the typical accuracy of GPS-enabled smartphones is 4.9 meters and increases near buildings and trees [13]. Therefore, it is even questionable if values lower than 5 meters are meaningful. Since Android returns 0 if no horizontal accuracy is available, 0 values should be ignored [11].

The largest spike in the distribution is at 65 meters. This can be explained by the plausible use of Assisted GPS (AGPS). AGPS combines the initial GPS signal with the cellular network location. This leads to a much lower power consumption, but only a slightly lower accuracy. Additionally, it is much faster and works better in the vicinity of buildings. The accuracy of GPS is minimum 5 meters, whereas the accuracy of AGPS is up to 65 meters [4].

4. Geospatial Big Data

4.1. Big Data

Big Data is only loosely defined. General speaking, Big Data covers datasets which are so big that traditionally used software is not able to handle it in a viable amount of time. Often Big Data is defined with the following three main characteristics [21]:

- Volume
- Velocity
- Variety

Big Data covers large *volumes* of data. Moreover, it is important to maintain an acceptable *velocity* even with growing data sets. *Variety* refers to the fact that Big Data is often not homogeneous but varies in content, for example, in a collection of data from different data sources.

4.2. Apache Spark



Figure 4.1.: Logo of Apache Spark.

Apache Spark is an open-source cluster computing framework to do tasks such as ETL, analytics or machine learning on large volumes of data. Apache Spark aims to hold as much data as possible in-memory, which leads to a high speedup compared to other cluster computing technologies like Apache Hadoop or Google MapReduce¹.

Figure 4.2 shows the architecture of an Apache Spark cluster. It uses a master/ worker architecture. The *Driver Program*, which executes the program, talks to the *Cluster Manager*, which is the master. The master then coordinates the execution of the program on the *Worker Nodes*. Apache Spark supports multiple cluster managers, such as Apache Mesos, Hadoop YARN or Kubernetes. Additionally it ships with an own cluster manager, which is called Standalone.

¹MapReduce is a programming model and an associated implementation, whereas Apache Hadoop is a open-source implementation of MapReduce.

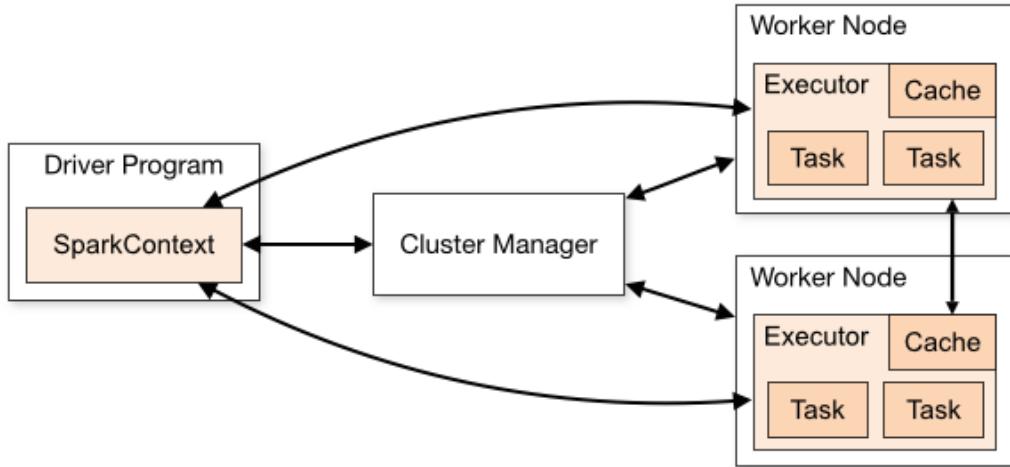


Figure 4.2.: Spark architecture.

4.2.1. Language and API

Apache Spark is written in Scala, but developer APIs for different languages exists. The most popular choices are Scala and Python but one can even use Java or R. The choice of the language depends on the context of the use case. Performance wise, Scala outperforms Python². Python on the other hand has a lower learning curve and a large variety of existing libraries. Therefore, the best suitable language must be evaluated based on the developer capabilities and project needs.

Besides the choice of the language, different APIs exists from which any one can be chosen. The common APIs for Apache Spark are RDDs, DataFrames and Datasets. Since the understanding of the concepts of these APIs is important, they are summarized shortly.

4.2.1.1. RDDs

Resilient distributed dataset (RDD) was the first API available for Apache Spark. RDDs can be described as fault-tolerant immutable collection of data distributed across worker nodes. Listing 4.1 shows an example use of RDD with Scala. A text file is first read, the resulting elements are transformed and filtered, and the filtered elements are counted in the end. It is very important to understand that the first three lines of the code example do not trigger an execution of the code on the cluster. The call of the map and the filter functions are so-called **transformations**. Only with the last line, the counting of the filtered elements, the code is sent to the workers of the cluster and executed there. Functions which triggers an execution are called **actions**. This behaviour is described as lazy execution.

```

1  val rdd = sc.textFile(...) // reading a text file
2  val transformed_rdd = rdd.map(element => transform(element))
3  val filtered_rdd = transformed_rdd.filter(element => element == " ")
4  val count = filtered_rdd.count()

```

Listing 4.1: Example use of RDDs.

²... which can be overridden when Apache Spark DataFrames are properly used.

One has to be aware that an RDD does not exist only on one place, but is distributed over multiple worker nodes. Apache Spark takes care of the distribution of the data to the different worker nodes, and coordinates the execution of the transformations and actions on the worker nodes.

As already mentioned, RDD stands for resilient distributed dataset. They are resilient because each transformation on an RDD is recorded in a graph. This means, it is possible for Apache Spark to recreate each state of an RDD at any point of time. Therefore, if an error occurs in one transformation, the affected RDD can be recreated.

The biggest problem with RDD is that Apache Spark is not able to optimize the transformations applied to RDDs. For example, when the transform function in Listing 4.1 is very costly, it could be preferable to do the filtering of the elements first and then apply the transform on each element. But Apache Spark does not know enough about the data contained in the RDD, wherefore it is not able to do this optimization. This handicap is solved with DataFrames and Datasets.

4.2.1.2. DataFrames

DataFrames were released with Apache Spark 1.3 and are built on top of RDDs. As a result, they are also a distributed collection of data. But with DataFrames the relevant data is organized into named columns, similar to a relational database management system (DBMS) such as PostgreSQL. Since Apache Spark now has more information about the data contained in the DataFrame, it is able to optimize the queries and create a suitable query plan.

Another big advantage of DataFrames is that a more intuitive syntax can be used. With RDDs it was necessary to pass lambda functions to the transformations. With DataFrames a more intuitive set of functions exists. Listing 4.2 shows an example of the simpler syntax of DataFrames compared to RDDs.

```
1 // calculate average per key on RDD
2 rdd.map(lambda (x, y): (x, (y, 1))).reduceByKey(lambda x, y: (x[0] + y[0], x[1] +
3     + y[1])).map(lambda (x, (y, z)): (x, y / z))
4
5 // does the same on a DataFrame
df.groupBy("name").agg(avg("age"))
```

Listing 4.2: Same operation with RDDs and DataFrames.

DataFrames can also make use of SparkSQL, which allows to use a query language identical to SQL. Listing 4.3 shows an example use of SparkSQL with DataFrames.

```
1 df.createOrReplaceTempView("my_data")
2
3 spark.sql("SELECT name, avg(age) FROM my_data GROUP BY 1")
```

Listing 4.3: Use of SparkSQL with DataFrames.

The biggest disadvantage of DataFrames is, that they are not type safe. The attributes are only referred by their string names and therefore DataFrames are not type safe in compilation time. This leads to exceptions during runtime. However, this problem is solved with Datasets.

4.2.1.3. Datasets

Datasets are the last and the newest API of Apache Spark. They've been introduced with version 1.6. With Apache Spark 2.0 the APIs of DataFrames and Datasets are actually merged, whereas DataFrames corresponds to untyped Datasets. Besides that, a typed version of Datasets exists.

The key benefit of Datasets is the type safety of the API. Therefore, type mismatches can be detected at compile time, which can be a big advantage in developing code.

Since Python is a dynamically typed language, it does not have the support for the Dataset API. Therefore, the Dataset API is only available in Scala and Java.

4.2.1.4. Summary

As already mentioned, the choice of a suitable API depends on the use case. RDDs need to specifically instruct Apache Spark how to do something, whereas it is sufficient for DataFrames and Datasets to indicate what to do— while Apache Spark chooses the best way to do it. The latter allows better optimization. On the other hand, it is not possible for DataFrames or Datasets to work with unstructured data, when it would be necessary to use RDDs. When handling structured data, it is most advisable to use Datasets, which ensures type safety during compilation and optimization of computing resources.

4.2.2. Geo Spatial Data Analytics

Apache Spark is not suitable for out of the box spatial data analytics. For that, an extension of Apache Spark is necessary. An appropriate extension for the use cases of this thesis is discussed in Section 7.

Part II.

Implementation with PostgreSQL and PostGIS

5. Use Case AOI

5.1. Implementation

In this section the implementation of generating the AOIs using PostgreSQL and PostGIS is described in detail. The theory used in the approach is introduced in Section 2.

The following technology stack is used for the implementation:

- Python 3.6
- PostgreSQL 10
- PostGIS 2.4

More details can be found in the source code. The full implementation runs inside docker containers. The corresponding `docker-compose.yml` can be found in the Appendix A.2.

5.1.1. Work with OpenStreetMap

To work with the data of OpenStreetMap, the data needs to be imported to PostgreSQL. Multiple tools exists to do this. A widely used tool is `osm2pgsql`¹. It is as is a command-line based program that converts OpenStreetMap data to postGIS-enabled PostgreSQL databases [20]. OpenStreetMap data in the PBF format can be imported with the following bash command:

```
1 osm2pgsql --create --slim --database gis /data/switzerland-latest.osm.pbf
```

Listing 5.1: Import switzerland-latest.osm.pbf to PostgreSQL database

With the above command the OpenStreetMap elements of Switzerland, like streets, buildings, lakes etc. are imported to the database. The geometries are converted to PostGIS compatible geometries. Additionally, all tags on the elements are imported too, and made available for PostgreSQL queries.

Since only the OpenStreetMap elements with certain tags are relevant for the POI clustering, one can import a subset of elements. This is not possible with `osm2pgsql`, but for example with `Osmfilter`² or `Osmosis`³ makes it possible. The procedure for using `osmfilter` can be found in the Appendix A.3.

5.1.2. Pre-Cluster POIs

The clustering of the POIs will be done locally with different parameters of the clustering algorithm. This is necessary because the same clustering algorithm cannot be used for small as well as large cities. For example, five nearby POIs have a different meaning in a sparsely populated mountain village in Switzerland as compared to a densely populated city like New York.

¹<https://github.com/openstreetmap/osm2pgsql>

²<https://wiki.openstreetmap.org/wiki/Osmfilter>

³<https://wiki.openstreetmap.org/wiki/Osmosis>

Adequate parameters will be evaluated in the next Section 5.1.3. But first it is necessary to pre-cluster the POIs to have separated areas for clustering in the next step. Therefore, one can simply cluster all POIs using DBSCAN with a large value for ϵ . The resulting clusters will separate the POIs coarsely. Figure 5.1 shows the resulting clusters when using the PostgreSQL query in Listing 5.2. This totally results in $\tilde{2500}$ clusters for Switzerland.

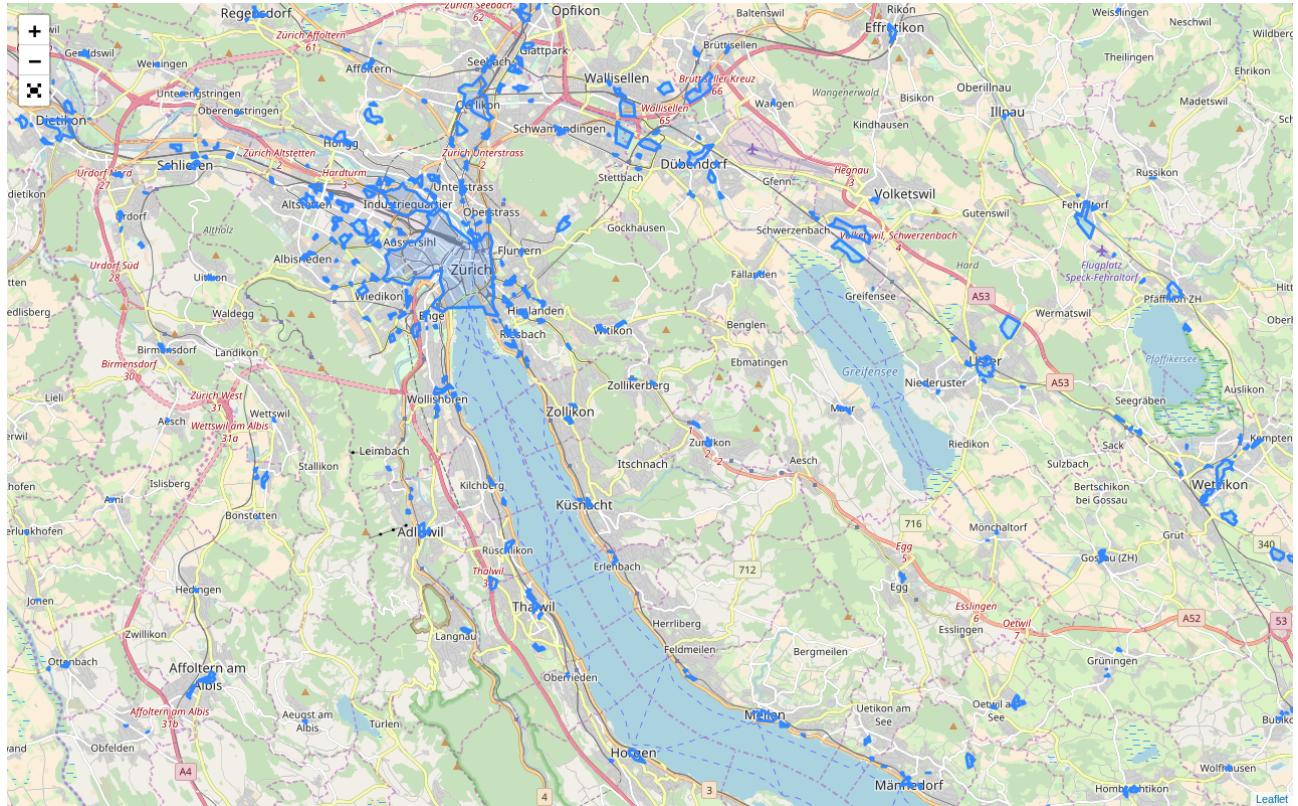


Figure 5.1.: Resulting clusters of pre-clustering POIs.

```

1 WITH clustered_pois AS (
2     SELECT geometry, ST_ClusterDBSCAN(geometry, 100, 3) over () as cid FROM pois
3 )
4 SELECT ST_ConvexHull(ST_Union(geometry)),
5     ST_Area(ST_ConvexHull(ST_Union(geometry))),
6     COUNT(geometry)
7 FROM clustered_pois AS hull WHERE cid > 0 GROUP BY cid

```

Listing 5.2: Query to pre-cluster POIs.

5.1.3. DBSCAN Local Adaption

As already described in Section 2.2.1.2 the DBSCAN clustering algorithm takes two parameters, viz. ϵ which defines the maximal distance of two points to be reachable and $minPts$ which defines how many points must be reachable by a core point.

For ϵ a value of 35 meters has been evaluated. The evaluation has been done by comparing the resulting clusters for different values of ϵ ⁴. Figure 5.2 shows the resulting clusters for Zürich for different values of ϵ .



Figure 5.2.: Comparing resulting clusters for different values of ϵ in the DBSCAN algorithm

The optimal value of $\epsilon = 35$ is evaluated as the best value since the resulting clusters are neither too small as with the value 30 (Figure 5.2a), nor too large as with $\epsilon \geq 40$ (Figures 5.2c and 5.2d).

Besides ϵ , a suitable value for $minPts$ must also be evaluated. This is done in the same manner, by comparing the different outputs. Figure 5.3 shows the resulting clusters for Zürich.

⁴Admittedly, deciding the best value for ϵ by comparing the outputs is a very subjective choice.

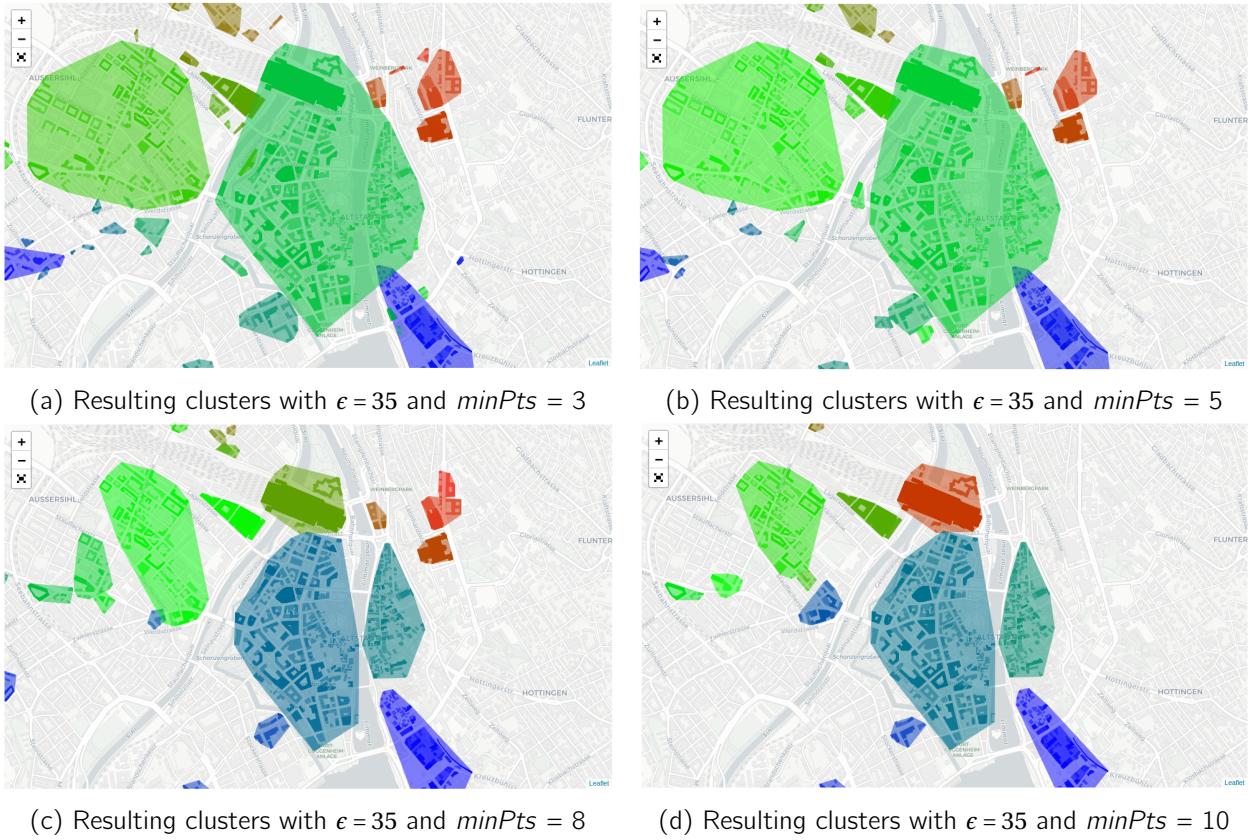


Figure 5.3.: Comparing resulting clusters for Zürich with different values of *minPts* in DBSCAN

With a small value like 3, a lot of small clusters result, which is not very useful for AOIs. The number and size of the clusters gets better after increasing the value for *minPts*. The value 8 seems to be the most suitable in this context.

However, the values $\epsilon = 35$ and *minPts* = 8 have been evaluated for one city, Zürich. Figure 5.4a shows the resulting clusters when the same values have been used for a small city like Stäfa. Only one cluster results, which is obviously not good. The reason for this is, that Stäfa has fewer POIs and therefore the same values as for Zürich cannot be applied here. By comparing different results, the values $\epsilon = 35$ and *minPts* = 3 have been evaluated. Interestingly, ϵ has the same value for both cities and only *minPts* differs. Figure 5.4b shows the resulting clusters for Stäfa with the new values for the DBSCAN algorithm.



(a) Clusters for Stäfa with $\epsilon = 35$ and $minPts = 8$

(b) Clusters for Stäfa with $\epsilon = 35$ and $minPts = 3$

Figure 5.4.: Comparing resulting clusters for Stäfa with different values of $minPts$ in DBSCAN

Table 5.1 shows the evaluated values for other cities.

Table 5.1.: Evaluated DBSCAN parameters for different cities.

City	evaluated ϵ	evaluated $minPts$
Zürich	35	8
Bern	35	5
Genf	35	5
Stäfa	35	3
Rüti	35	3
Rapperswil	35	3
Hombrechtikon	35	3

As one can see, the value $\epsilon = 35$ is suitable for all cities. Only the parameter $minPts$ differs. With this insight, a formula can be defined, which returns a suitable value for $minPts$ based on the size of the city and the amount of POIs.

For finding a suitable formula $y = f(x_1, x_2)$, where x_1 is the size of the city, x_2 the amount of POIs and y the resulting value for $minPts$, linear regression can be used, based on example values. The example values are listed in Table 5.2.

Table 5.2.: Input values for linear regression to find suitable formula for $minPts$

City	Size in m^2	Amount of POIs	$minPts$
Zürich	12514226	2071	8
Bern	3995449	664	5
Genf	3893063	382	5
Rüti	472063	69	3
Rapperswil	444458	104	3
Stäfa	363218	50	3
Hombrechtikon	67687	11	3

A multiple linear regression can be easily done online⁵. Equation (5.1) shows the resulting formula.

⁵For example, <http://www.xuru.org/rt/MLR.asp>

Since the formula is derived from only 7 samples, it is doubtful that it is applicable for every area on earth. Especially since it are only Swiss cities, whereas bigger cities in other countries will have a different spatial density of POIs. However, for a proof of concept it is adequate and can be refined in the future. It is also possible to derive a formula for calculating the ϵ the same way.

$$y = 2.91 - 5.34 \times 10^{-7} \cdot x_1 + 5.74 \times 10^{-3} \cdot x_2 \quad (5.1)$$

This formula can now be applied to all pre-clusters created in Section 5.1.2. Therefore the pre-clustered POIs are inserted into the table `preclusters`. Later the `minPts` value of each row is updated based on the area and number of POIs. The query is shown in Listing 5.3. As one can see, the value of `minPts` is limited to the minimum of 2.

```

1  CREATE TABLE preclusters (
2    id SERIAL,
3    hull geometry,
4    area integer,
5    pois_count integer,
6    dbscan_minPts integer
7  );
8
9
10 -- pre-cluster POIs and insert into preclusters table
11 -- ...
12
13 UPDATE preclusters SET dbscan_minPts = GREATEST(2, round((2.91 - 5.34 * 10^(-7) * ↴
14   area + 5.74 * 10^(-3) * pois_count
15 )));

```

Listing 5.3: Set DBSCAN parameter `minPts` for preclustered POIs.

5.1.4. Cluster POIs

The POIs are now pre-clustered and each pre-cluster has an assigned `minPts` value for the DBSCAN algorithm. With this, the POIs can now be clustered for each pre-cluster. Later a convex hull is generated for each resulting cluster. Listing 5.4 shows the corresponding query.

```

1 WITH clusters AS (
2   SELECT geometry,
3     ST_ClusterDBSCAN(geometry, eps := 35, minpoints := preclusters.^
4       dbscan_minpts) over () AS cid
5   FROM pois, preclusters
6   WHERE ST_Within(geometry, preclusters.hull)
7 )
8   SELECT cid, ST_ConvexHull(ST_Union(geometry))
9   FROM clusters
10  WHERE cid IS NOT NULL
11  GROUP BY cid

```

Listing 5.4: Clustering POIs of each pre-cluster.

5.1.5. Network Centrality

As described in Section 2.2.1.4 hulls of the clustered POIs are extended based on the network centrality of the paths and streets inside these hulls. Therefore, the street network is fetched as graph first. Figure 5.5 shows the hulls and the corresponding street network. As one can see, the street network is larger than the hulls. This is on purpose, since the network centrality calculated later on is more meaningful if it includes the outer streets around the hull too.

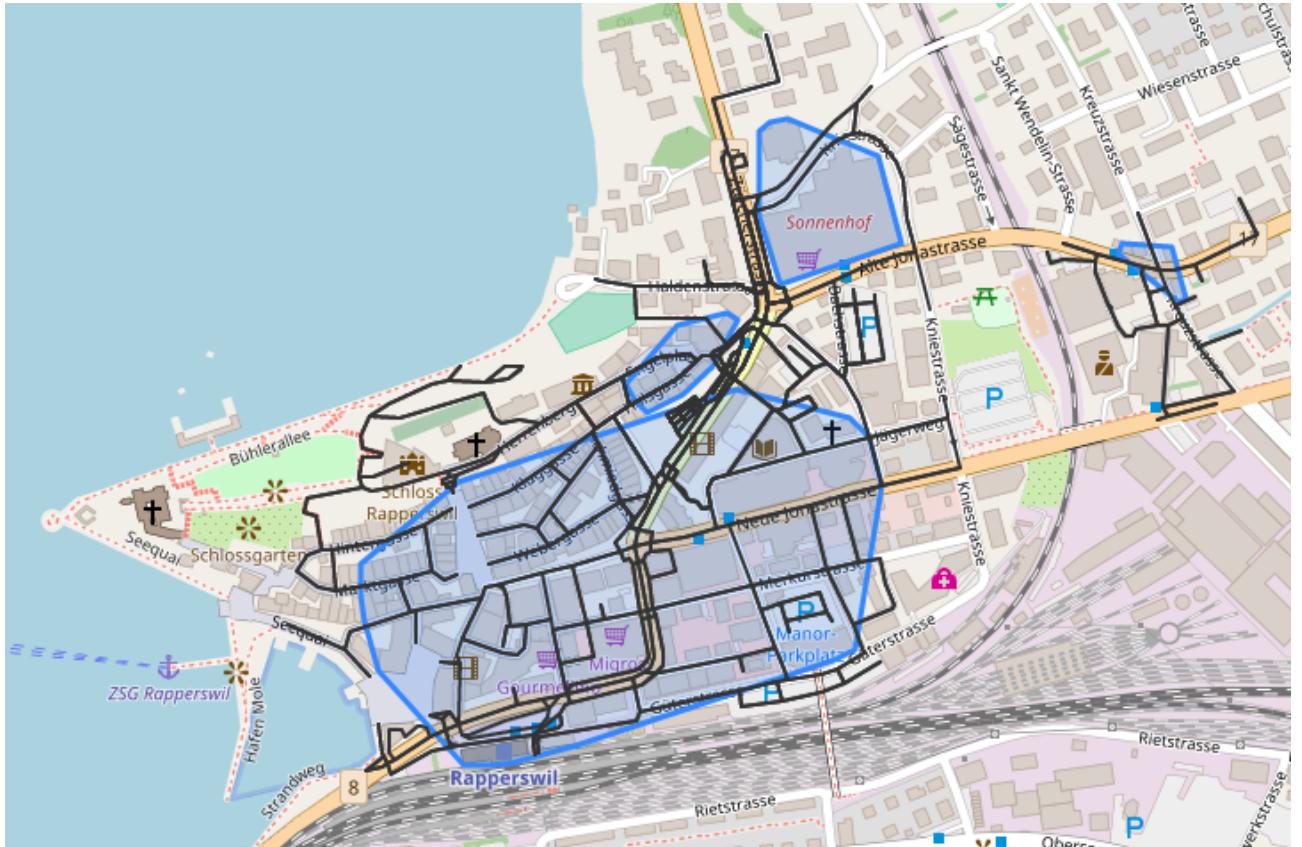
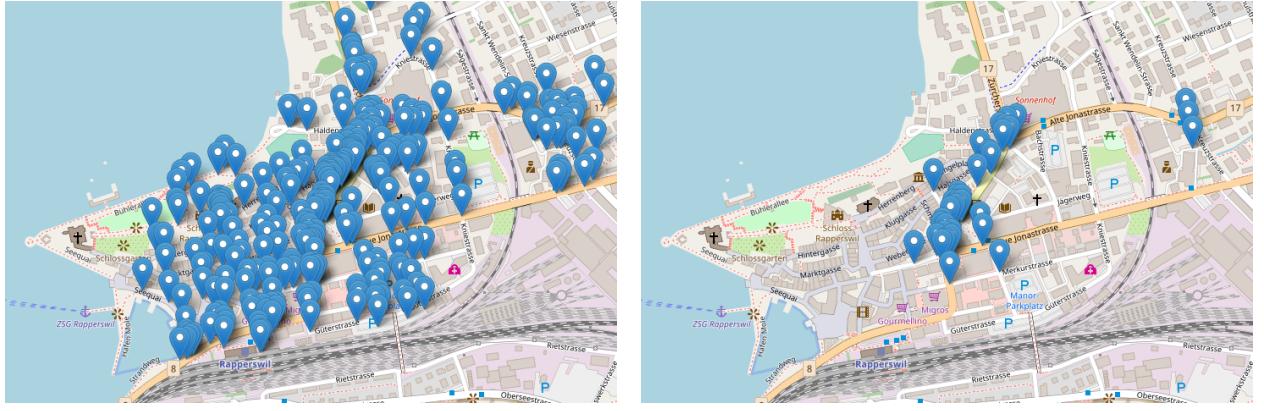


Figure 5.5.: Street network per POI cluster hull of Rapperswil.

The street network is fetched as graph using the library osmnx⁶. Next, the network centrality is calculated on this graph, using networkx⁷. After the centrality of each node is calculated using the closeness centrality, the nodes are filtered and only the top 10% of the nodes are taken for further processing. Figure 5.6 illustrates the resulting nodes.

⁶<https://github.com/gboeing/osmnx>

⁷<https://networkx.github.io/>



(a) All nodes of street network graph

(b) The 10% most central nodes

Figure 5.6.: Nodes of street network graph.

Now, all streets of these nodes are selected. These selected streets not only contain streets which are fully inside the hulls but also streets which are outside the hull. The streets outside are cut with a 50 meter buffer around the hull. This is illustrated in Figure 5.7.

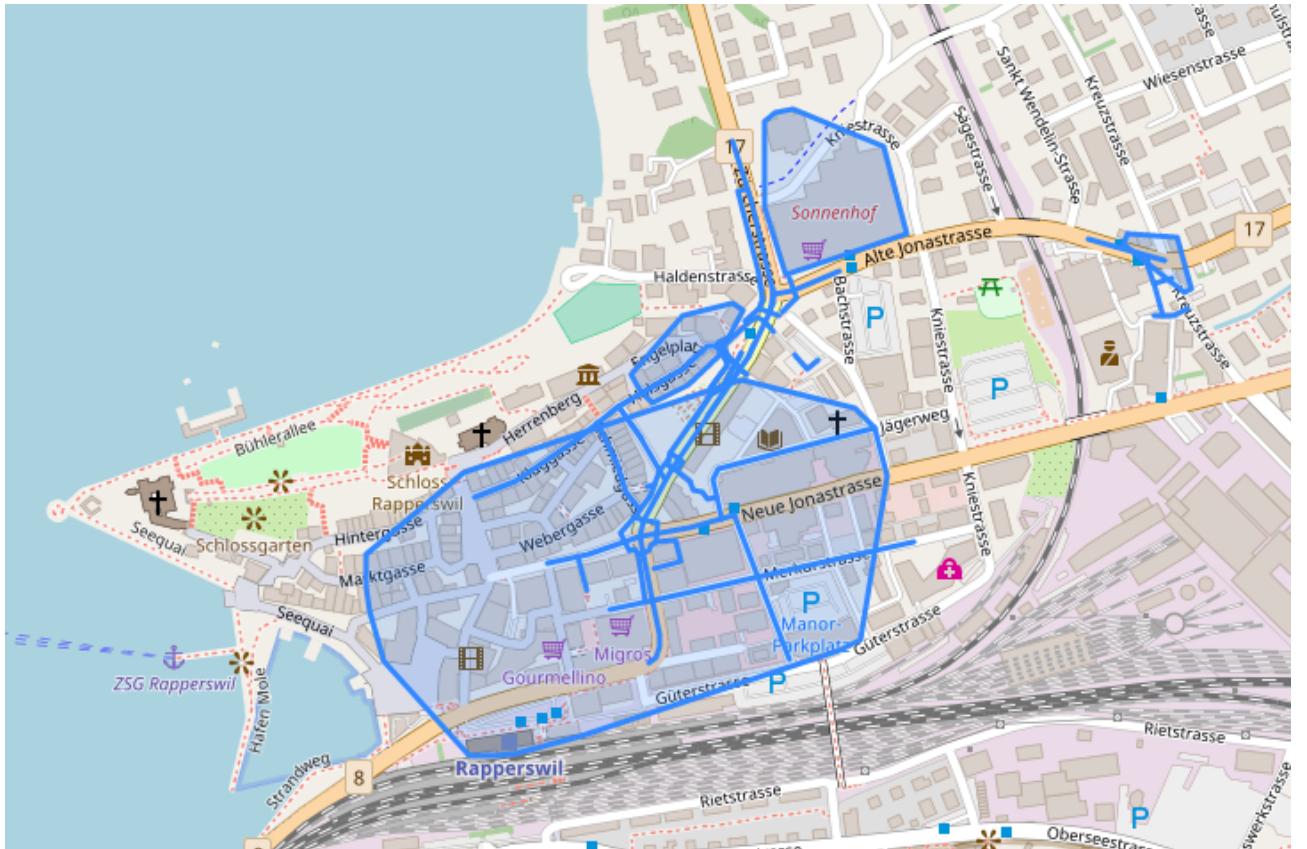


Figure 5.7.: 10% of the most central streets, cut with a 50 meter threshold outside the hull.

Finally, the hulls can be extended by combining them with the cut-off central streets. This can be done easily, by drawing a concave hull around the original hull and the extending streets. Figure 5.8 illustrates the resulting hulls. The extension is marked red.

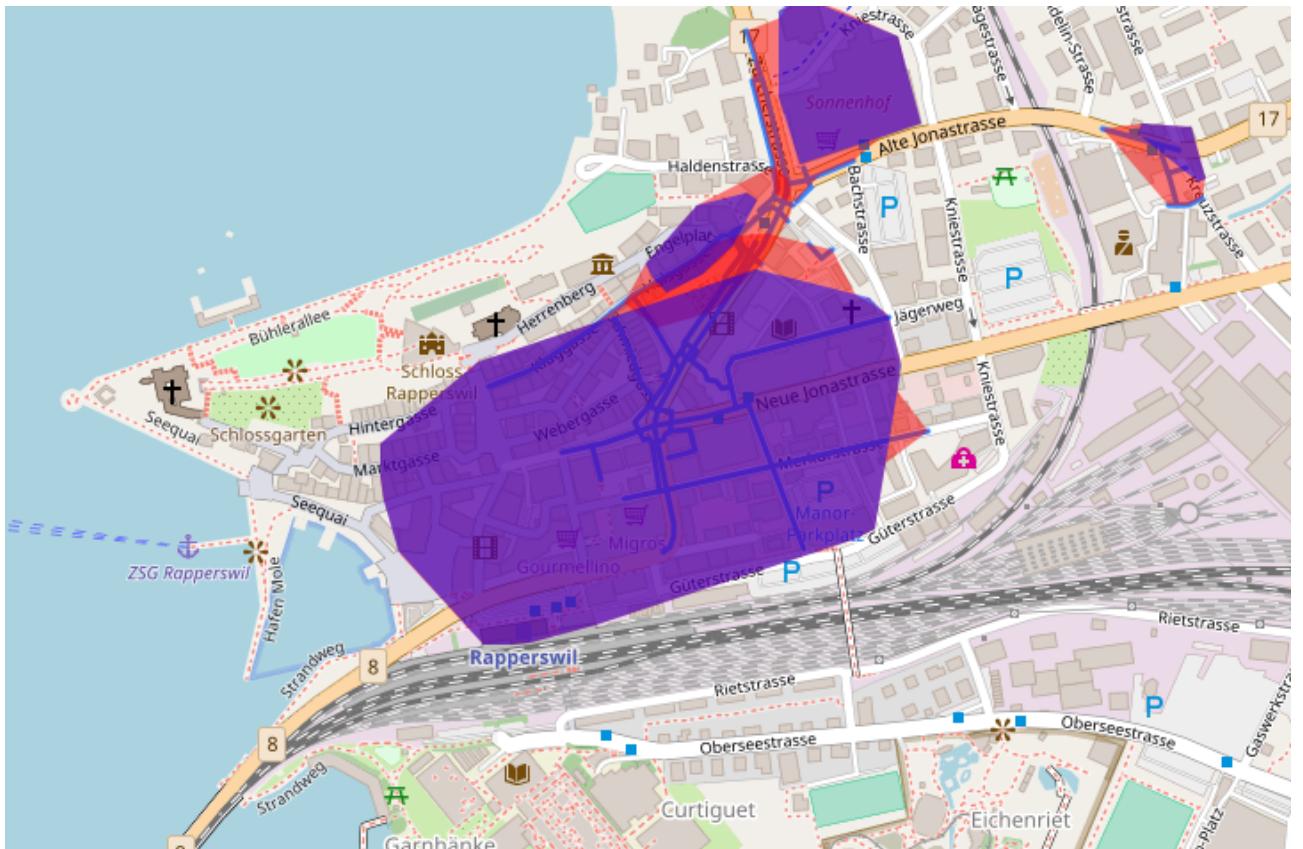


Figure 5.8.: Hulls extended with network centrality. The extensions are marked red.

Listing 5.5 shows the Python code for fetching the street graph and calculating the most central node ids. The PostgreSQL query which extends the hulls is shown in Listing 5.6.

```

1 # hulls = hulls of clustered POIs
2 central_nodes = []
3
4 for hull in hulls.geometries:
5     # fetch street graph using osmnx
6     # the buffer of 50 corresponds to 50 meteres
7     street_graph = osmnx.graph_from_polygon(hull.buffer(50), network_type='all')
8
9     # calculate closeness centrality of all nodes in street graph using networkx
10    closeness_centrality = networkx.closeness_centrality(street_graph)
11
12    # sort nodes by its centrality
13    sorted_nodes = sorted(closeness_centrality.items(), key=operator.itemgetter(1), reverse=True)
14
15    # and take the first 10 %
16    central_nodes += [node[0] for node in sorted_nodes[:len(sorted_nodes) // 10]]
17
18
19 # concat all node ids to a comma separated string
20 central_nodes_ids = ', '.join([f'{key}' for key in central_nodes])

```

Listing 5.5: Fetch street graph and calculate most central nodes.

```

1 WITH hulls AS ({hulls_query}),
2 intersecting_lines AS (
3     SELECT hulls.cid, ST_Intersection(way, ST_Buffer(hulls.geometry, 50)) AS ↵
4         geometry FROM planet_osm_line, hulls
5     WHERE osm_id = ANY(
6         SELECT id FROM planet_osm_ways
7         WHERE nodes && ARRAY[{central_nodes_ids}]::bigint []
8     )
9     AND ST_DWithin(planet_osm_line.way, hulls.geometry, 50)
10 )
11 SELECT ST_ConcaveHull(ST_Union(geometry), 0.99) AS geometry FROM (
12     SELECT cid, geometry FROM hulls
13     UNION
14     SELECT cid, geometry FROM intersecting_lines
15 ) AS hull_and_lines
16 GROUP BY cid

```

Listing 5.6: Extending hulls based on most central nodes.

5.1.6. Exclude Water

To exclude water from the AOIs, one can select all OpenStreetMap elements which have a corresponding tag. Therefore all elements which have a tag for the keys `waterway` or `water` are selected. As a special case, waterways or water inside a tunnel are excluded⁸. The polygons of these elements can then be subtracted from the AOIs. Figure 5.9 illustrates this step. Listing 5.7 shows the corresponding query.



Figure 5.9.: Exclude water from AOIs.

⁸For example underneath the train station in Zürich flows a river.

```

1 WITH hulls AS ({hulls_query})
2   SELECT ST_Difference(hulls.geometry , (
3     SELECT ST_Union(way) AS geometry FROM planet_osm_polygon
4     WHERE (water IS NOT NULL OR waterway IS NOT NULL)
5       AND (tunnel IS NULL OR tunnel = 'no')
6       AND st_intersects(way, hulls.geometry)
7   )) AS geometry
8   FROM hulls

```

Listing 5.7: Exclude water from AOIs

5.1.7. Sanitize AOIs

As last step, the AOIs needs to be sanitized. This includes:

- Merge overlapping polygons
- Remove invalid polygons
- Remove polygon artifacts with no value

An examples for this is shown in Figure 5.10. The corresponding PostgreSQL query is shown in Listing 5.8.



Figure 5.10.: Sanitize AOIs

```

1 WITH aois AS ({aois_query})
2   SELECT ST_Simplify((ST_Dump(ST_Union(geometry))).geom, 5) AS geometry
3   FROM aois
4   WHERE ST_IsValid(geometry)

```

Listing 5.8: Sanitize AOIs

5.1.8. Export AOIs

After sanitizing the AOIs in Section 5.1.7, they are ready to be exported. For this, the gdal library, a translator library for raster and vector geospatial data formats⁹ can be used. It includes the cli command ogr2ogr¹⁰ which allows to export a PostgreSQL + PostGIS table to GeoJSON. The corresponding bash command is shown in Listing 5.9.

```
1 ogr2ogr -f GeoJSON /path/to/aois.geojson -sql "SELECT ST_Transform(hull, 4326) ↵
   FROM aois"
```

Listing 5.9: Export AOIs as GeoJSON

5.2. Web Application

To quickly visualize and test the AOIs, a web application has been implemented. This section summarizes the used technologies and illustrates the web interface. The source code of the implementation is not described in detail. The documentation to setup and run the web application can be found in the Appendix A.3.

For the web application flask¹¹, a Python microframework is used. GeoPandas¹² allows to query a PostgreSQL + PostGIS database for geometries. To render the maps, the Python library folium¹³ is used. One can pass a GeoJSON to folium for rendering. Since GeoPandas can convert geometries to GeoJSON, one can pass a GeoPandas DataFrame directly to folium. For map rendering the CartoDB Basemaps(Positron)¹⁴ are used.

⁹<http://www.gdal.org/>

¹⁰<http://www.gdal.org/ogr2ogr.html>

¹¹<http://flask.pocoo.org/>

¹²<https://github.com/geopandas/geopandas/>

¹³<https://github.com/python-visualization/folium>

¹⁴<https://github.com/CartoDB/CartoDB-basemaps>

AOIs with OpenStreetMap

Either [browse all AOIs of Switzerland \(overlay on GMaps\)](#) or generate them for a specific location.

The screenshot shows the user interface for generating AOIs. At the top, there are two input fields: 'Location' (containing 'Rapperswil') and 'Custom Location' (containing 'e.g. 47.3720, 8.5417'). Below these is a 'Tags' section containing a JSON-like code block:

```
{  
  "landuse_tags": [  
    "retail"  
  ],  
  "amenity_tags": [  
    "pub",  
    "bar",  
    "cafe",  
    "restaurant",  
    "pharmacy",  
    "bank",  
    "fast_food",  
    "food_court",  
    "  
  ]
```

Below the tags is a 'Hull Algorithm' section with two radio buttons: 'Convex' (selected) and 'Concave(0.99)'. There is also a checkbox labeled 'Explain steps when generating AOIs (slower)'. A 'Generate AOIs' button is at the bottom.

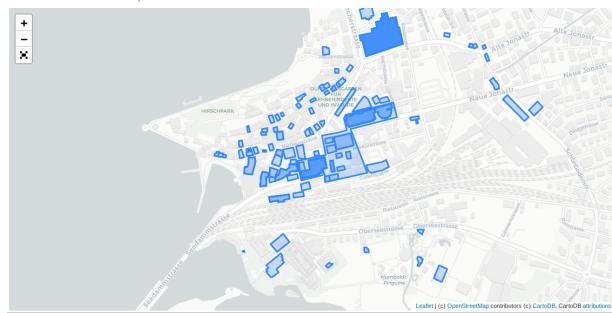
Figure 5.11.: Interface of the web application.

Figure 5.11 shows the main interface of the web application. One can either select a location from the location-dropdown or insert a custom location. The tags are shown readonly for a better understanding. As a second option, one can chose between the convex and concave hull algorithm for the AOIs. When selecting the checkbox *Explain steps when generating AOIs (slower)* the output contains each step of the AOI generation, as shown in Figure 5.12. If the checkbox is not checked, the output only contains the final AOIs, as shown in Figure 5.13.

Alternatively, one can browse all AOIs of Switzerland either visualized with Folium or as overlay on a Google Maps. The overlay on Google Maps allows to compare the AOIs quickly. This is illustrated in Figure 5.14.

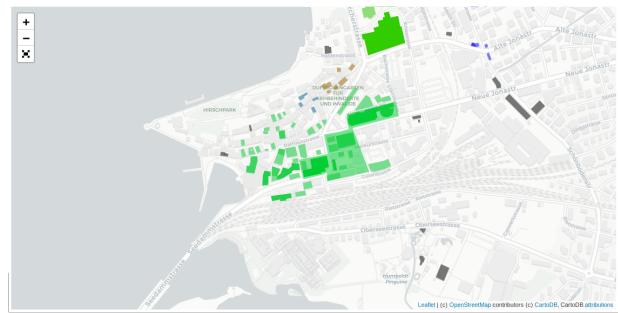
1. Get all polygons with tags

- Select polygons which:
- have a given tag
 - contain a node with a given tag (and building = true)
 - have not the attribute access = private



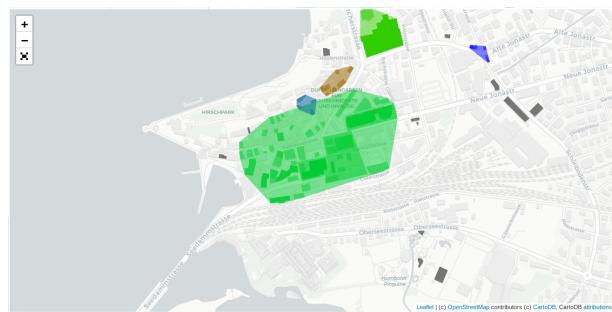
2. Cluster polygons with DBSCAN

Cluster polygons by using DBSCAN algorithm, whereas the parameters minPts and eps are different depending on the location of the polygons (local adaption).



3. Create hulls around clusters

Convex or concave hulls are possible and results obviously in different hulls. This can be selected one page before.



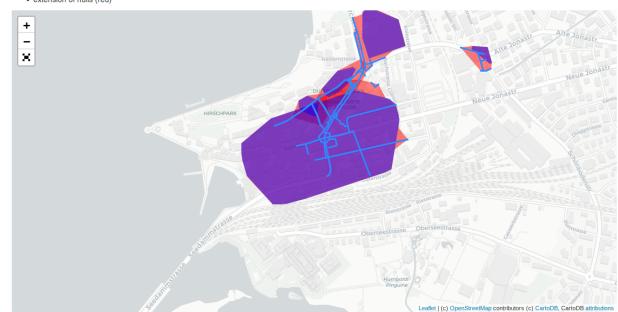
4. Extend using network centrality

Therefore:

- calculate closeness centrality of street graph for each hull (inkl buffer)
- select 10% of the most central streets and ways
- cut streets which are leaving the hull after 50 meters
- extend hulls by drawing concave hull around hull and selected and cut streets

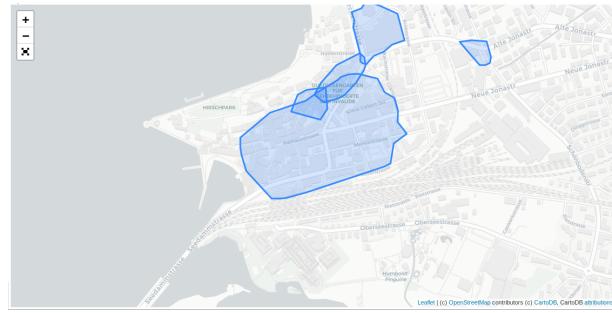
Illustration:

- hulls before (voiced)
- 10% most central streets (blue)
- extension of hulls (red)



5. Exclude waterways and water

If present...



6. Sanitize

- union overlapping polygons
- simplify polygons slightly
- remove invalid polygons



Figure 5.12.: Explained steps of AOI generation of web application

OpenStreetMap AOI

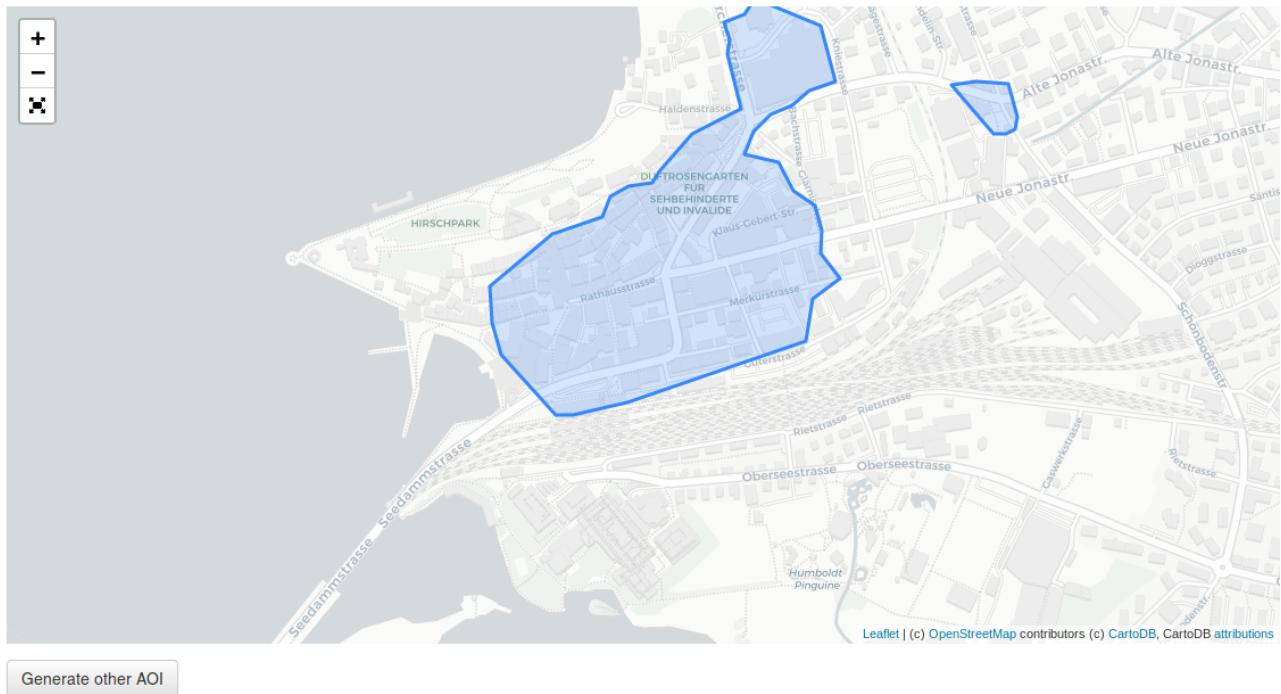
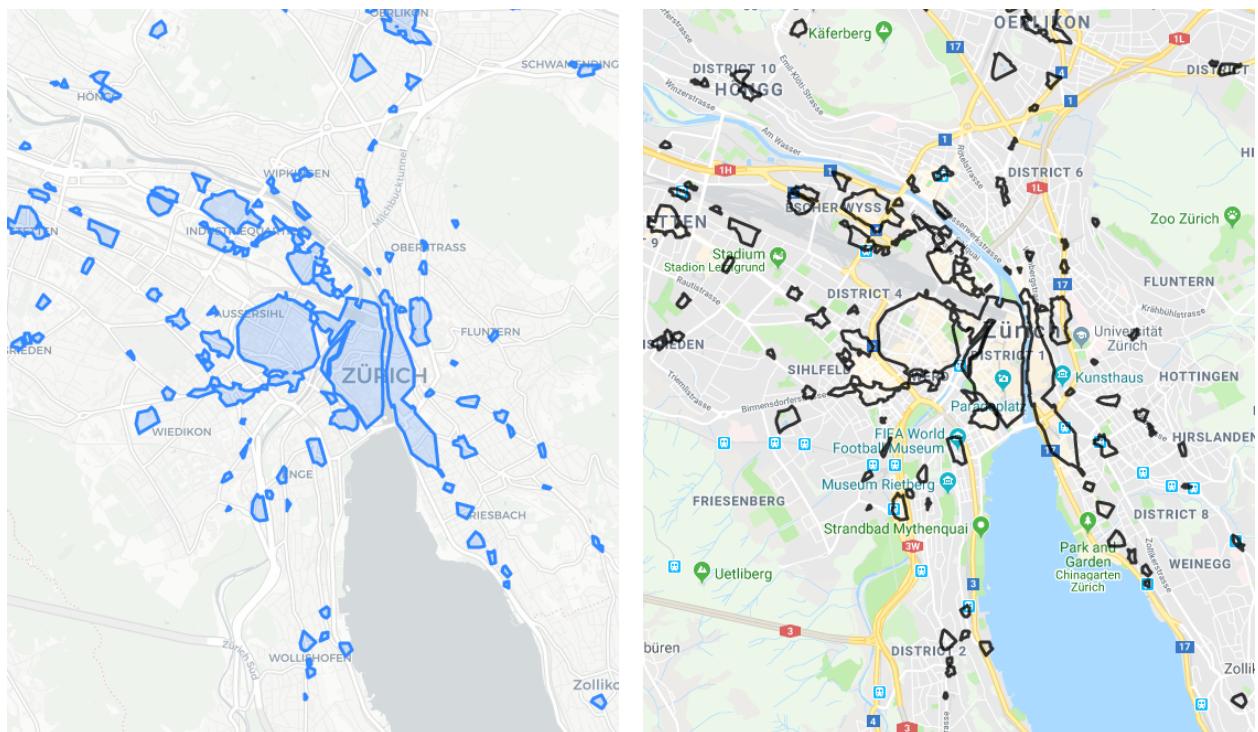


Figure 5.13.: Generated AOIs of web application



(a) On map generated with Folium

(b) On Google Maps

Figure 5.14.: Overlay of all generated AOIs.

5.3. Benchmarks

In this section the implementation from Section 5.1 is benchmarked and the major bottlenecks are identified.

5.3.1. Hardware

The benchmarks are executed on a HP EliteBook 6930p with the following specification:

- Intel(R) Core(TM)2 Duo CPU P8600 @ 2.40GHz
- 6 GB memory
- SSD Storage

PostgreSQL was configured with the recommended parameters of PGtune¹⁵. The parameters can be found in the Appendix A.4.

5.3.2. Results

Table 5.3 contains the runtime of the different phases when generating the AOIs for Switzerland. The OpenStreetMap data of Switzerland is downloaded from Geofabrik¹⁶ and has a size of 278 megabytes.

The runtime of the import are not listed, since it is not relevant for this use case. About the performance of osm2pgsql a lot of literature and benchmarks exists¹⁷.

After the OSM data is imported and the POIs are pre-clustered, generating the AOIs is done with one big query, which covers all steps described in Section 5.1, except calculating the network centrality. Calculating the network centrality is done beforehand. Accordingly, the runtime of the network centrality is listed separately and the runtime of the other steps are summarized.

Table 5.3.: Runtimes when generating AOIs for Switzerland

Step	Runtime
Pre-Clustering	30s
Generating AOIs without network centrality	23s
Generating AOIs with network centrality	59min
Export AOIs	25s

As one can see, the generation of the AOIs are much faster when it does not include the network centrality. This is because calculating the network centrality includes fetching the street graph and performing expensive calculations on the graph.

The pre-clustering and generating of the AOIs without network centrality takes up almost the same time. The low runtime is due to the small amount of POIs which are processed. Switzerland contains only 43301 POIs with the defined tags.

¹⁵<https://pgtune.leopard.in.ua>

¹⁶<http://download.geofabrik.de/europe.html>

¹⁷<https://wiki.openstreetmap.org/wiki/Osm2pgsql/benchmarks>

When calculating the AOIs for a certain area, for example when using the web application, the runtime depends on the selected area. Table 5.4 contains the runtimes for various areas. Again, the network centrality is the most expensive part to calculate. The runtime with the network centrality is not listed, if it exceeded a certain threshold time.

Table 5.4.: Generating AOIs for different areas

Area	With Network Centrality	Without Network Centrality
Stäfa	7s	80ms
Rapperswil	7.3s	200ms
Zürich	157s	1.1s
Canton of Bern		3.8s
Canton of Zürich		4.7s

5.3.3. Summary

The major bottleneck is the calculation of the network centrality algorithm. This is due to osmnx, which fetches the street graph from an external source. But even if this fetch is cached and osmnx only has to read the graph from the file system, calculating the network centrality is still slow, because it includes complex calculations.

Besides that, the bottleneck when calculating the AOIs for Switzerland concerns the missing ability of PostGIS to parallelize queries. To improve this, one could split the query manually in multiple queries and then run them in parallel. This would add some complexity to the creation of the queries but it would be a viable solution.

If generating the AOIs need to be done periodically, for example once a week, then it can be said that the performance is already good enough, except the network centrality. But if an API need to be provided to generate AOIs for larger areas in real time, the performance gets critical.

6. Use Case Event Count

6.1. Existing Implementation

This section describes the existing implementation with PostgreSQL 9.6 and PostGIS 2.4. Even though the industrial partner provided their own implementation, it was written from scratch. This way it was possible to reduce the implementation to the most important parts and remove dependencies which are not necessary for the use case.

The most relevant part of the implementation is the querying of the events, since it is the slowest part according to the industrial partner. The following two are the input parameters:

- The **area** to query the events for. These areas are provided in the GeoJSON format.
- The **date range** to query the events for. The range is provided with a start- and end-date.

With these input parameters, the events are queried with the following two steps:

1. Creating Areas

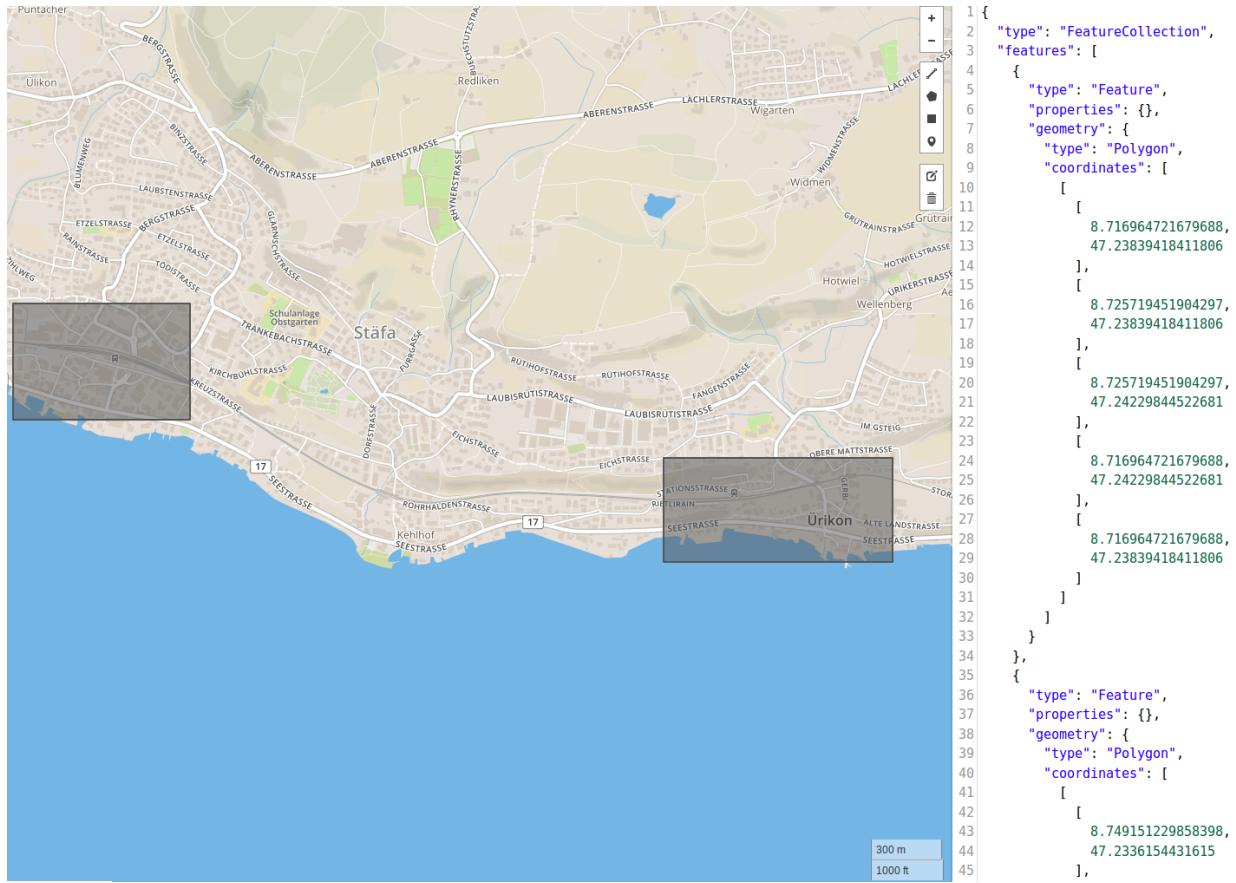
Since the areas are provided as GeoJSON, it is necessary to make them accessible to the PostgreSQL query. Therefore, the GeoJSON is parsed with Python and inserted into a new PostgreSQL table, which is named `areas`. Listing 6.1 shows the create query of the table.

```
1 CREATE TABLE areas(  
2     area geometry(Polygon, 4326),  
3     buffered geometry(Polygon, 4326)  
4 );
```

Listing 6.1: Query to create areas table.

The table consists of an `area` and a `buffered area`. The buffered area is simply the area buffered with 500 meters.

Each feature in the input GeoJSON is now collected in the `areas` table. Figure 6.1 shows an example for an input GeoJSON and the resulting rows of the `areas` table.



(a) Input GeoJSON

	area geometry(Polygon,4326)	buffered geometry(Polygon,4326)
1	0103000020E610000001000000050000000000	0103000020E610000001000000250000000450
2	0103000020E610000001000000050000000000	0103000020E610000001000000250000000450

(b) Resulting rows in areas table

Figure 6.1.: Each feature of the input GeoJSON results in one row in the areas table.

2. Querying Events

After the GeoJSON have been included into the areas table, the events can be queried. As already described in Section 3, the query should return all events that occurred within the input date range and inside an area more than half of which lies within the input area. Listing 6.2 shows the corresponding query.

```

1 SELECT app_id, consumer_id, session_id, occurred_at
2   FROM events_view, areas
3 WHERE occurred_at BETWEEN '{from_date}' AND '{to_date}'
4   AND ST_Within(location, areas.buffered)
5   AND ST_Area(ST_Intersection(areas.area, ST_Transform(ST_Buffer(
6     ST_Transform(ST_MakeValid(location), 3857), accuracy), 4326))) / 
7     ST_Area(ST_Transform(ST_Buffer(ST_Transform(ST_MakeValid(location),
8     3857), accuracy), 4326)) > 0.5

```

Listing 6.2: Query to get events.

The query contains three conditions:

The first condition filters on the basis of date range.

The second condition filters events which are inside the buffered area.

The last condition gets the area of the intersection of the event and the input areas in m^2 . This value is divided by the area of the event in m^2 . If the result is larger than 0.5, which means at least 50% of the event area is inside an input area, then the event is taken into consideration.

The second condition would actually not be necessary, because the last would filter the events anyway. But the second condition performs much better than the last one, since indexes can be used by PostgreSQL. Therefore, it is advisable to keep all three due to performance reasons.

The query returns the relevant events which can be evaluated in a succeeding step. However, the evaluation of the events is not relevant for the use case of this thesis.

6.2. Benchmarks and Bottlenecks

As pointed out by our industrial partner, the query for the events described in Section 6.1 works, but gets slower with more data. To verify this, multiple benchmarks are performed.

A database server with PostgreSQL 9.6 was provided by our industrial partner. It has 2 CPUs, 8 GB of memory and 175 GB SSD storage.

6.2.1. Areas

Different benchmarks are accordingly defined to measure different aspects. Each benchmark is defined by a GeoJSON which contains the relevant areas to query the data for. Table 6.1 names and illustrates the different benchmark areas.

Name	GeoJSON
Multiple small areas	

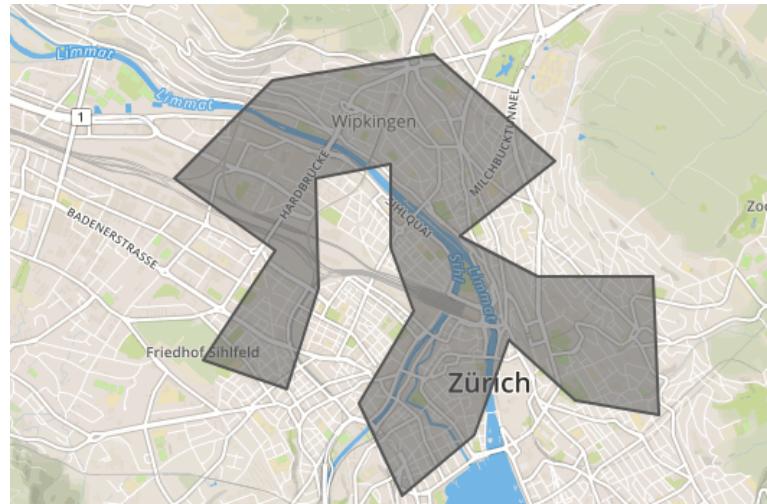
One big area



Multiple big areas



One big complex area



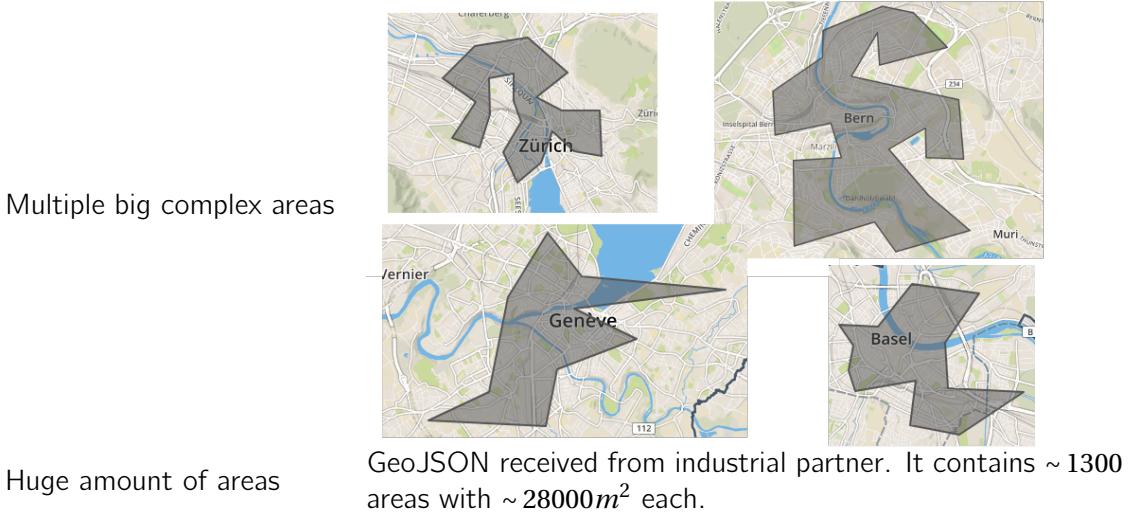


Table 6.1.: Different areas for the benchmarks.

To run the benchmarks against different amounts of data, the date range of the query can be varied. Table 6.2 lists the number of events per date range, and inside the different benchmark areas, where an event is considered to be inside a benchmark area if at least 50% of the buffered event area intersects with the benchmark area.

Table 6.2.: Count of events intersecting the different benchmark areas with at least 50%

Benchmark Area / Weeks	2 weeks	4 weeks	6 weeks	8 weeks	10 weeks
Multiple small areas	2927	5111	8535	12276	16103
One big area	63194	122747	206004	304931	402162
Multiple big areas	80146	159044	265003	393098	519971
One big complex area	31466	60759	101315	149818	196843
Multiple big complex areas	57487	112547	187805	278051	367776
Huge amount of small areas	18544	36833	61587	92446	123726

6.2.2. Indexes

PostgreSQL indexes have a huge impact on the performance of the queries. The following indexes exists:

- btree index on events.occurred_at
- gist index on events.location
- gist index on areas.area
- gist index on areas.buffered

6.2.3. Runtimes

Figure 6.2 contains the runtimes (in seconds) of the query described in Section 6.1. The query is executed with all benchmark areas and increasing time ranges. Each query is executed five times and the average of the runtime is documented. Since PostgreSQL does some caching, the documented runtimes do not correspond to the runtimes on a cold-start. However, its effect is neglected, as it applies for all benchmarks.

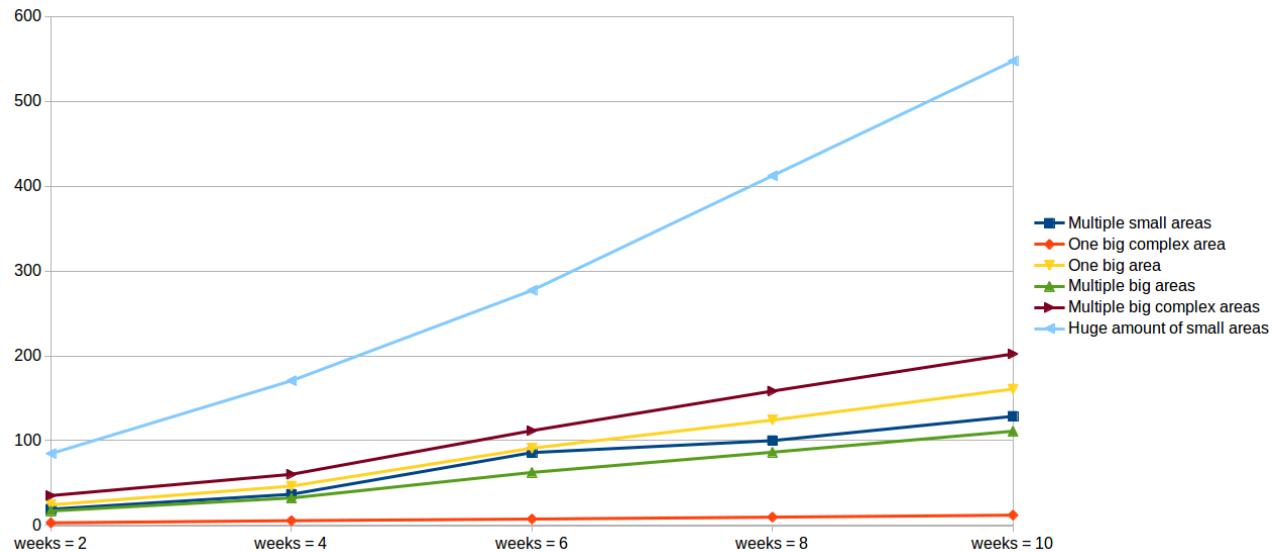


Figure 6.2.: Runtimes in seconds for different benchmark areas and increasing amount of data.

One can very well see that the runtimes are longer when more weeks of data are queried. The longest runtime has the benchmark area “Huge amount of small areas” which takes almost ten minutes when querying for 10 weeks of data.

6.2.4. Bottlenecks

As documented in Section 6.2.3, the execution become slower with more data being processed. Figure 6.3 shows the output of the PostgreSQL command `EXPLAIN ANALYZE`¹ for the query, querying for the benchmark area “Huge amount of small areas” and one week.

The following conclusions can be drawn from the output:

- PostgreSQL makes use of the indexes on the occurred_at and location columns
- The index occurred_at column is very fast
- PostgreSQL performs a bounding box check with the location of the event and the buffered area
- Calculating the intersection and checking if it is 50% of the events area takes the most time.

Now that the bottlenecks are identified, optimizations can be applied to improve the performance of the query. This is done in Section 6.3.

¹The output is visualized with explain.depesz.com

#	exclusive	inclusive	rows x	rows	loops	node
1.	179.495	35,737.431	↑ 4.0	8,033	1	→ Nested Loop (cost=0.58..138,900.27 rows=32,186 width=3,376) (actual time=1.958..35,737.431 rows=8,033 loops=1)
2.	205.366	205.366	↓ 1.1	292,170	1	→ Index Scan using events_view_occurred_at on events_view (cost=0.44..11,006.27 rows=254,992 width=133) (actual time=0.047..205.366 rows=292,170 loops=1) Index Cond: ((occurred_at >= '2017-03-31 22:00:00+00':timestamp with time zone) AND (occurred_at <= '2017-04-06 22:00:00+00':timestamp with time zone))
3.	35,352.570	35,352.570	↓ 0.0	0	292,170	→ Index Scan using areas_buffered on areas (cost=0.14..0.49 rows=1 width=3,243) (actual time=0.117..0.121 rows=0 loops=292,170) Index Cond: (buffered ~ events_view.location) Filter: (((st_area(st_intersection(area, st_transform(st_buffer(st_transform(st_makevalid(events_view.location), 3857), events_view.accuracy), 4326))) / st_area(st_transform(st_buffer(st_transform(st_makevalid(events_view.location), 3857), events_view.accuracy), 4326))) > 0.5)::double precision) AND _st_contains(buffered, events_view.location)) Rows Removed by Filter: 1

Figure 6.3.: Output of PostgreSQL command EXPLAIN ANALYZE when querying benchmark "Huge amount of small areas" for one week of data.

6.3. Optimizations

This section contains all optimizations applied to the PostgreSQL and PostGIS implementations. In this context, it is important to weigh up the cost and return of single optimizations, since one could invest almost any amount of time needed to improve the performance. Section 6.4 contains the benchmark results performed after the optimizations have been applied.

6.3.1. Preprocessing Events

The original query contained, among others, the following steps:

- Calculate buffered area of event with `ST_Buffer(location, accuracy)`
- Calculate area of event in m^2 with `ST_Area(area)`
- Change CRS of locations with `ST_Transform(location, 3857)`

To improve the performance of the queries, these steps can be done once for each event in the preprocessing stage. Later, the queries are executed on the preprocessed events. This reduces the runtime of the queries.

Additionally only events having a meaningful accuracy should be queried. As described in Section 3.1 the test data contains events with an accuracy less than 0 and more than 1000. Both are not practical and should therefore be ignored.

6.3.2. Utilizing Convex Hull properties instead of Buffers on Areas

As described in Section 6.1, the query contains a condition which filters all events that are inside a 500 meters buffer of the input areas (`ST_Within(location, areas.buffered)`). This condition is used because it is faster than calculating and comparing the intersections. However, the condition can be improved since it returns events for which it is impossible to have more than 50% intersection of the areas. A better approach is, to filter the events by using the convex hull property of the areas. If the location of an event is not inside the convex hull of an area, it can under no circumstances have more than 50% intersection. The new condition can filter out more events in the initial stage, so that less events need to be checked against the expensive intersection condition.

6.3.3. PostGIS ST_Buffer

The PostGIS Function `ST_Buffer` takes an input geometry and returns a geometry extended by a given radius. Figure 6.4 shows an example for a point, buffered with a radius of 100 m.

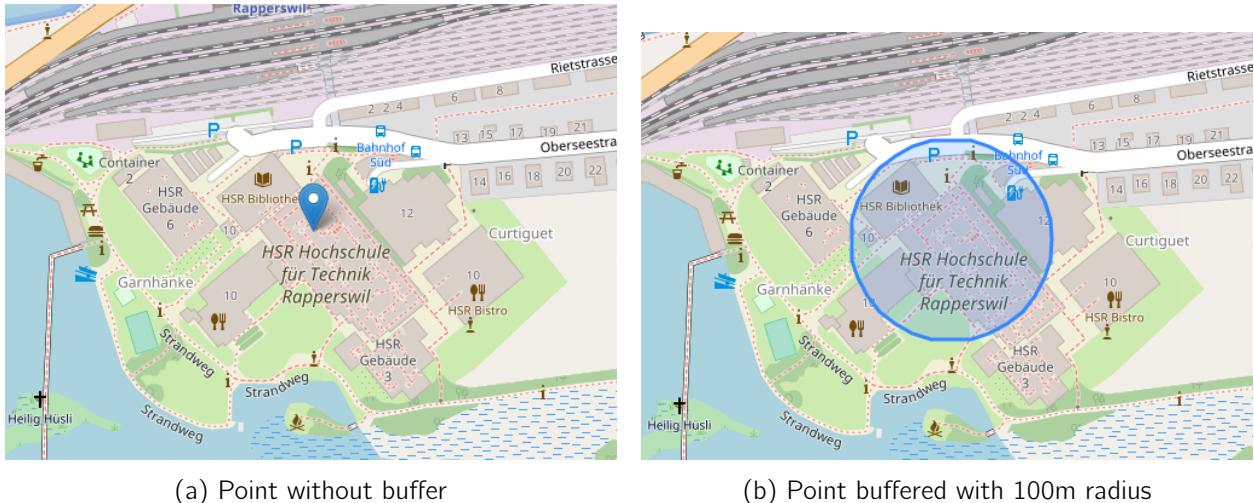


Figure 6.4.: Example for use of PostGIS function `ST_Buffer`

It is important to mention that the buffer is only an approximated circle, which is actually represented by a polygon using 32 points. With the `ST_Buffer` function it is possible to customize the style of the buffer, such as the number of points used to approximate the circle. By adjusting the parameter `quad_segs` from the default value of 8 to a value of 2, the number of segments is reduced by a factor of 4, which results in 8 instead of 32 points to approximate the circle. Figure 6.5 visualizes this.

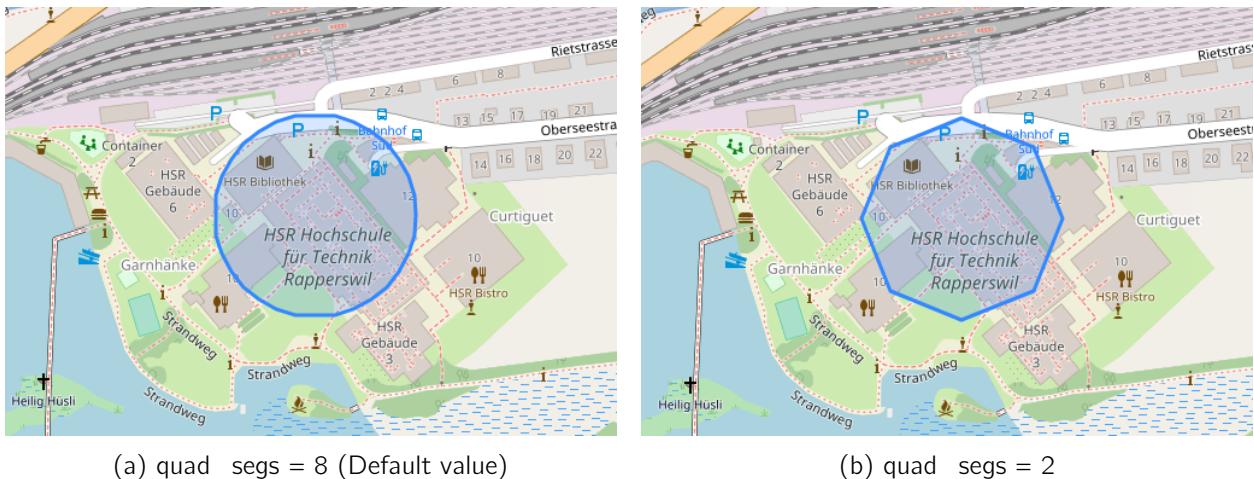


Figure 6.5.: Adjusting the `quad_segs` parameter of `ST_Buffer`

As one can see, the area of the polygon varies depending on how accurately the circle is approximated. The area of a perfect circle is $A = \pi r^2$. This results in $A \approx 31416$ when $r = 100$. Table 6.3 shows the decreasing accuracy of the area when decreasing the value of the `quad_seg` parameter²

²As one can guess, a circle approximated with 4 points is a square, where the resulting area is $A = 2r^2$

Table 6.3.: Resulting areas with varying value of quad_segs parameter.

quad_segs	Area of resulting polygon	Difference to area of circle
8	~ 31214 m ²	~ 0.6 %
6	~ 31058 m ²	~ 1.1 %
4	~ 30614 m ²	~ 2.5 %
3	~ 30000 m ²	~ 4.5 %
2	~ 28284 m ²	~ 9.9 %
1	~ 20000 m ²	~ 36.3 %

Since the size of the event areas are varying, it may be expected that the count of events intersecting the benchmark areas would vary too. Table 6.4 shows the counts for the benchmark area *Multiple big areas* for 10 weeks of data in case of different values of quad_segs. As one can see, its impact on the results is really small.

Table 6.4.: Impressions of 10 weeks with frequency cap of 10.

quad_segs = 8	quad_segs = 4	quad_segs = 3	quad_segs = 2	quad_segs = 1
449821	449821	449824	449866	449866

The decreasing complexity of the polygons has a positive impact on the performance of the calculation of the intersections, shown in Figure 6.6. The benchmarks are executed with 4 weeks of data.

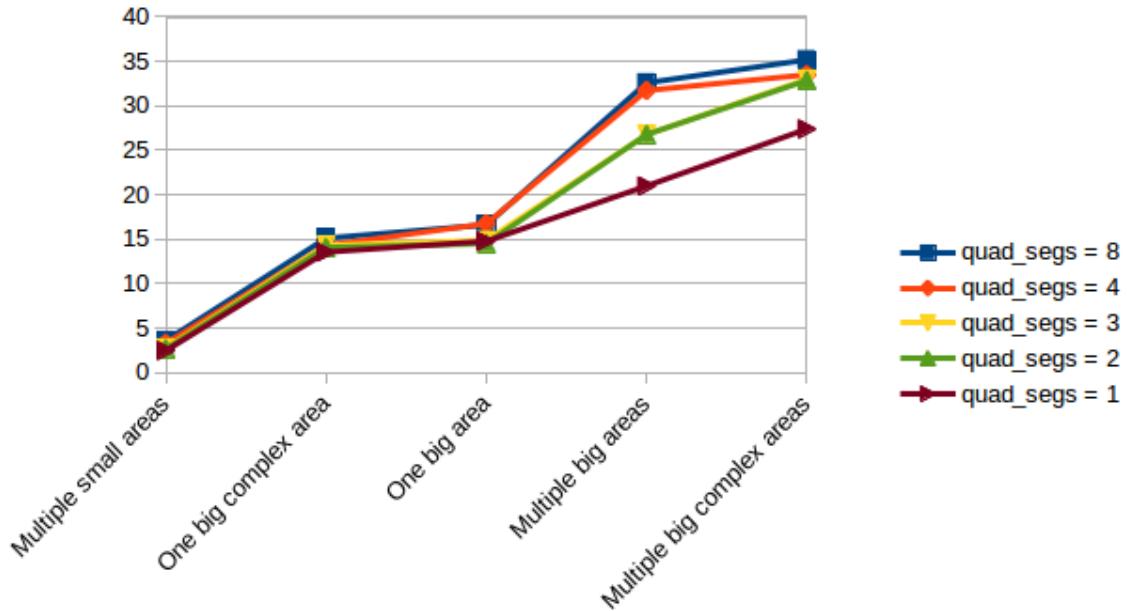


Figure 6.6.: Runtimes in seconds for different values of quad_segs parameter and benchmark areas.

Since the impact on the final results is very minor and there is a positive impact on the performance, the lowest value of 1 for the quad_segs parameter is considered to be optimal.

A further improvement of this approach could be possible by increasing the radius of the buffer slightly when using a small value for the quad_segs parameter. The resulting area would overlap more with

the original circular area (or when using a larger value for the quad_segs parameter). However, this is not implemented in this thesis.

6.3.4. Simplify Areas

One way to improve the performance is to reduce the complexity of the areas. Simpler area polygons lead to faster querying of the events. For example, in Figure 6.7 one can see one event area in red, and the corresponding simplified area in blue. This simplification was done with the PostGIS function *ST_Simplify* with a tolerance of 100, which uses the Douglas-Peucker algorithm [23].

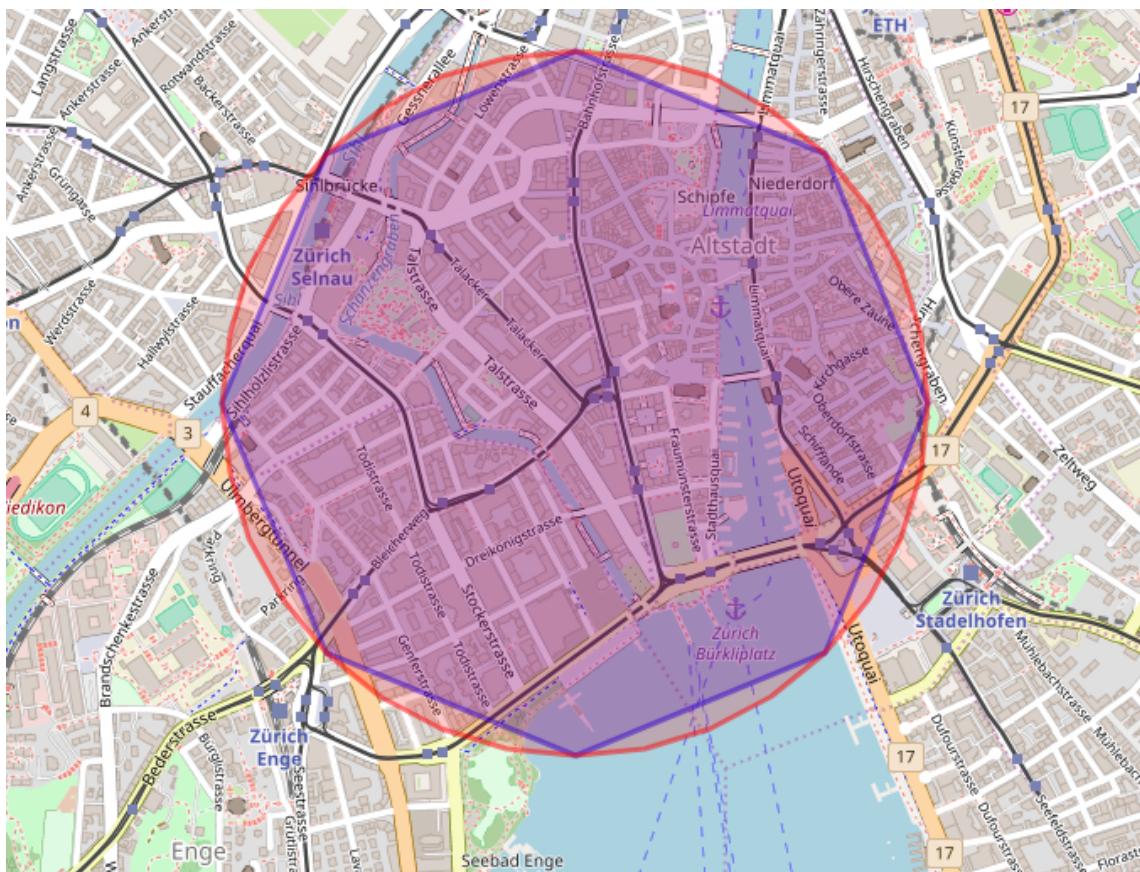


Figure 6.7.: The blue event area results when running *ST_Simplify* with a tolerance of 100 on the red event area.

As one can see in Figure 6.7 the resulting area uses fewer points to express the polygon and is therefore simpler. On the downside, the resulting area is smaller than the original. This has an impact on the results. Benchmarks have shown that this impact is quite high. Hence a very low tolerance value is suggested. A low value, for example 10, will have a small impact on the area by reducing the complexity of the polygons slightly if possible.

6.3.5. Cluster Events

Calculating the intersections is the bottleneck of the algorithm. Therefore, a longer runtime results with more data. One approach to improve the runtime is, to reduce the number of events which need to be queried. This can be done by clustering the events to reduce the amount of records to query.

The idea of clustering the events is simple: Merge nearby events together, which share almost the same area. Since the area of events is calculated with their accuracy, events are only allowed to be clustered, when they have the same accuracy. For example, Figure 6.8a shows two events which share the same location, but since their area is totally different, it makes no sense to cluster them. Neither does it make sense to cluster events which share the same accuracy, but only have a partly common area, as shown in Figure 6.8b. On the other hand, Figure 6.8c shows two events which share almost the same area and can therefore be clustered safely.

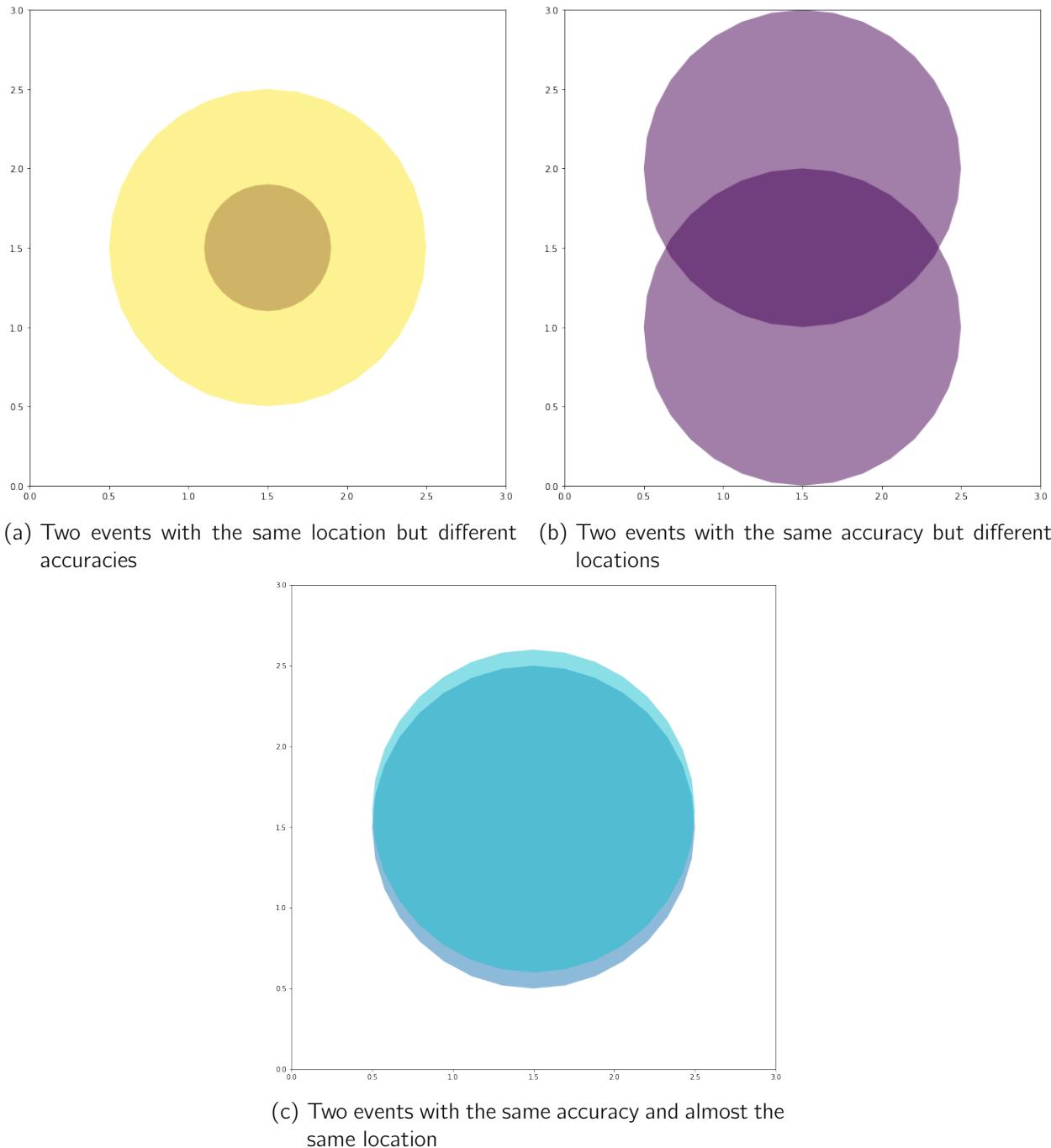


Figure 6.8.: Different cases of an event area intersecting a polygon.

A certain threshold in percent must be evaluated, which defines how much percent of the area must

be shared by both, so as to cluster them.

Since overlapping areas of events have the same radius, the only difference is the center of the area. Figure 6.9 shows the areas of two events, where the difference in location of one event is m , horizontally and vertically. For simplicity, the areas are visualized as squares instead of circles.

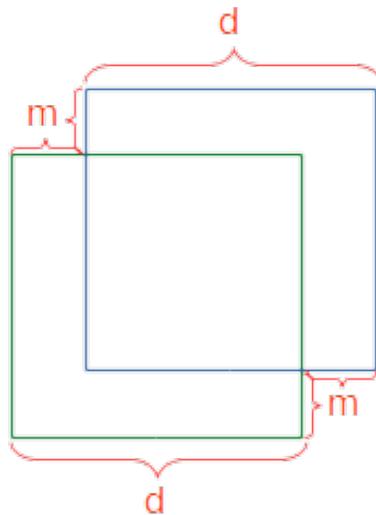


Figure 6.9.: Two areas of overlapping areas.

The position of the blue event in Figure 6.9 differs from the position of the area of the red event by a distance m horizontally as well as vertically. Therefore, the area of intersection P can be calculated as $P = (d - m)^2$. The area of one event is $A = d^2$. One can get the percentage of the intersecting area by dividing P by A . For example, if one wants the areas of the events to overlap by 50%, the maximum difference in the location can be found with the following equation:

$$\begin{aligned} P/A &= 0.5 \\ (d - m)^2/d^2 &= 0.5 \\ m &\approx 0.3d \end{aligned} \tag{6.1}$$

The maximum allowable difference between the locations of two events is therefore 0.3 times the diameter of the event area d . It's not surprising that m is dependent on the diameter of the area, since a larger area allows a larger distance between the events locations, as shown in Figure 6.10.

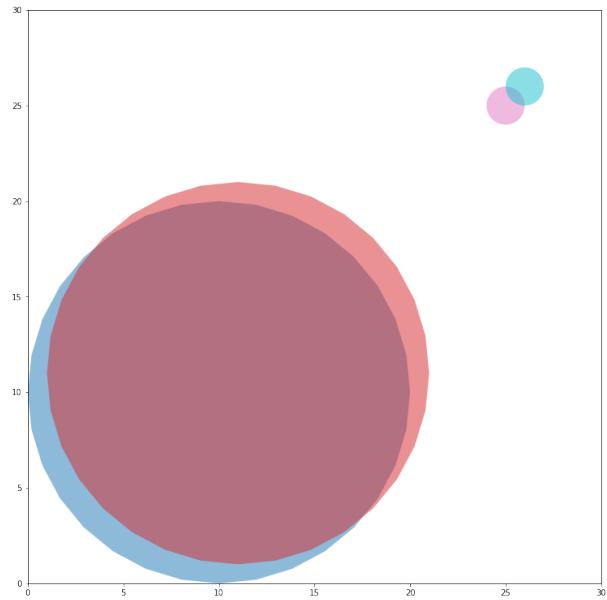


Figure 6.10.: One large and one small event area moved by one meter.

Now that the maximum difference of the event locations is defined by m , the easiest way to cluster the events is to snap the events to a grid with a grid size of m . For example, when $m=5$, the grid size is 5. Therefore, all locations of the events are snapped horizontally and vertically to the nearest multiple of 5, as shown in Figure 6.11.

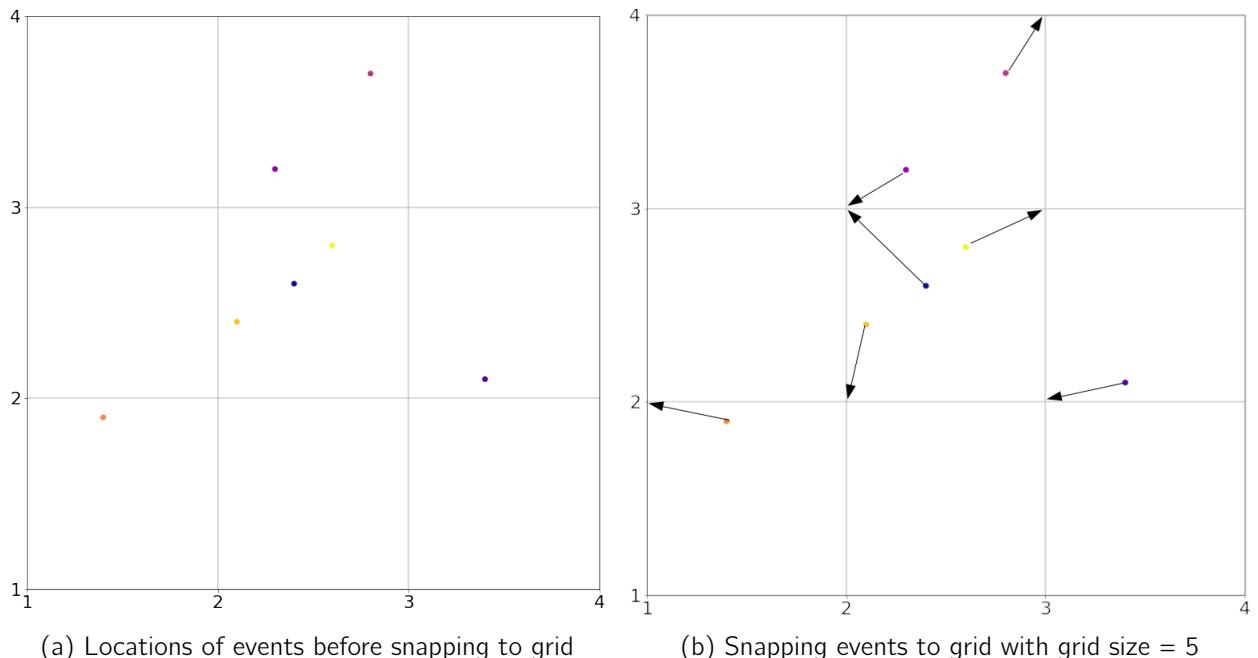


Figure 6.11.: Example for snapping event locations to grid.

When the location of the events are snapped to the grid, all events which share the same location belong to one cluster. The cluster itself has a location too. However, the location of the clusters is not its location after snapping to the grid, but the centroid of all original locations of the events the cluster contains. Figure 6.12 shows the locations and areas of two events in blue and green.

Additionally, is the resulting location and area of the cluster shown in red. This way, the difference of the area of the events and the cluster is reduced.

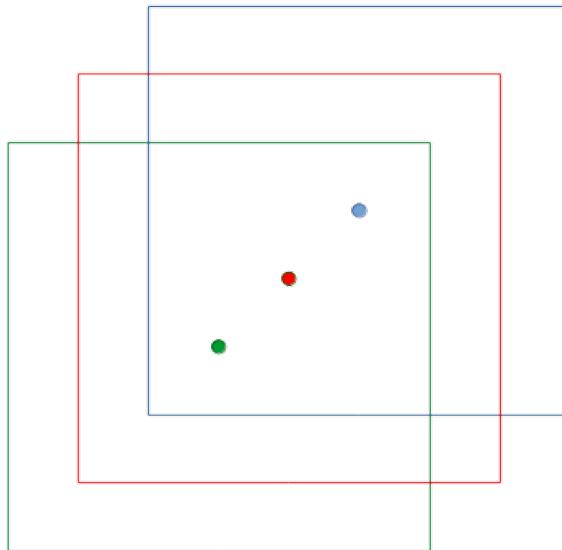


Figure 6.12.: Location and area of events (green, blue) and location and area of resulting cluster (red)

6.3.5.1. Implementation

Thankfully PostGIS contains all the necessary utilities to cluster the events. To query the events for the clusters the following query can be executed:

```

1 FOR distinct_accuracy IN (SELECT DISTINCT(accuracy) FROM events_view)
2   SELECT occurred_at::date,
3         ST_Centroid(ST_Union(array_agg(location))) AS location,
4         array_agg(id) AS event_ids
5   FROM events_view
6   WHERE accuracy = distinct_accuracy
7   GROUP BY occurred_at::date,
8           ST_SnapToGrid(location, (distinct_accuracy * 2 * M))
9 END LOOP;

```

The loop over the distinct accuracies is necessary, because events should only be clustered with other events if they share the same accuracy. Therefore, the clustering is processed for each accuracy separately.

As one can see, the locations of the events are snapped to the grid with the PostGIS function *ST_SnapToGrid*. The function takes the location and the size of the grid as first and second parameters. The size of the grid is determined based on the accuracy of the event. Since the accuracy of an event is the radius, it is multiplied by two to get the diameter. Then, the diameter is multiplied with a predefined factor of *M*. Following the calculations in eqn. (6.1) the value *M* = 0.0125 ensures that the clustered events share at least 95% of their area. The ideal value for *M* will be evaluated in the next Section.

After the locations of the events are snapped to the grid, all events are grouped by their dates and locations. This leads to an array of events for every date and location. One could also cluster the events over the full date range and not separately for each date. However, this would later lead

to difficulties when querying the clusters, since querying for an indexed date column is extremely efficient.

The location of the new cluster is the center of the locations of the clustered events. This is done with the PostGIS function *ST_Centroid*.

After the events have been clustered, the query to fetch the matches must be adopted to query the clusters instead of the events directly. This results in the new query, shown in Listing 6.3

```

1 WITH event_ids AS (
2     SELECT unnest(event_ids) AS ids
3     FROM clustered_events AS clusters, areas
4     WHERE occurred_at BETWEEN '{from_date}' AND '{to_date}'
5         AND ST_Intersects(clusters.location, areas.area)
6         AND (ST_Covers(areas.area, clusters.area)
7             OR ST_Area(ST_Intersection(areas.area, clusters.area)) > half_area)
8     )
9     ),
10    matches AS (
11        SELECT DISTINCT ON (session_id) app_id, consumer_id, session_id, occurred_at
12        FROM events_view, event_ids
13        WHERE id = ANY(ARRAY[event_ids.ids])
14    )
15    SELECT app_id, consumer_id, array_agg(occurred_at)
16    FROM matches
17    GROUP BY app_id, consumer_id

```

Listing 6.3: Query events from clustered events

As one can see, is the query is almost the same. The only difference is, that the *clustered_events* table is queried, instead of the *events_view*. Since the clustered events contain the grouped events as array, the queried arrays are unnested to get the corresponding events from the *events_view*.

6.3.5.2. Benchmarks

This section contains the impact of the clustering approach on the amount of events to query, the runtime of the benchmarks and the difference of the results. With the clustering approach, a small difference in the results must be accepted, since clustering nearby events introduces a small error. The maximum tolerable error is evaluated at the end of this section.

Amount of Clusters The amount and size of the resulting clusters is dependent on the factor p , which defines how much of the event areas must intersect, so as to cluster them. Figure 6.13 contains the amount of resulting clusters for different values for p . It is important to note that clusters with only one event are counted too.

Table 6.5.: Resulting count of events when querying different clusters

	No Clustering	90%	70%	50%	30%	10%
Multiple small areas	16143	16415	16399	16397	16391	16411
One big complex area	196553	199293	199293	199241	199249	199326
One big area	402222	408002	408089	408111	408057	407930
Multiple big areas	520006	527827	527836	527838	527840	527978
Multiple big complex areas	367323	372586	372516	372574	372593	372633
Huge amount of small areas	127556	129322	129151	128795	128990	129267

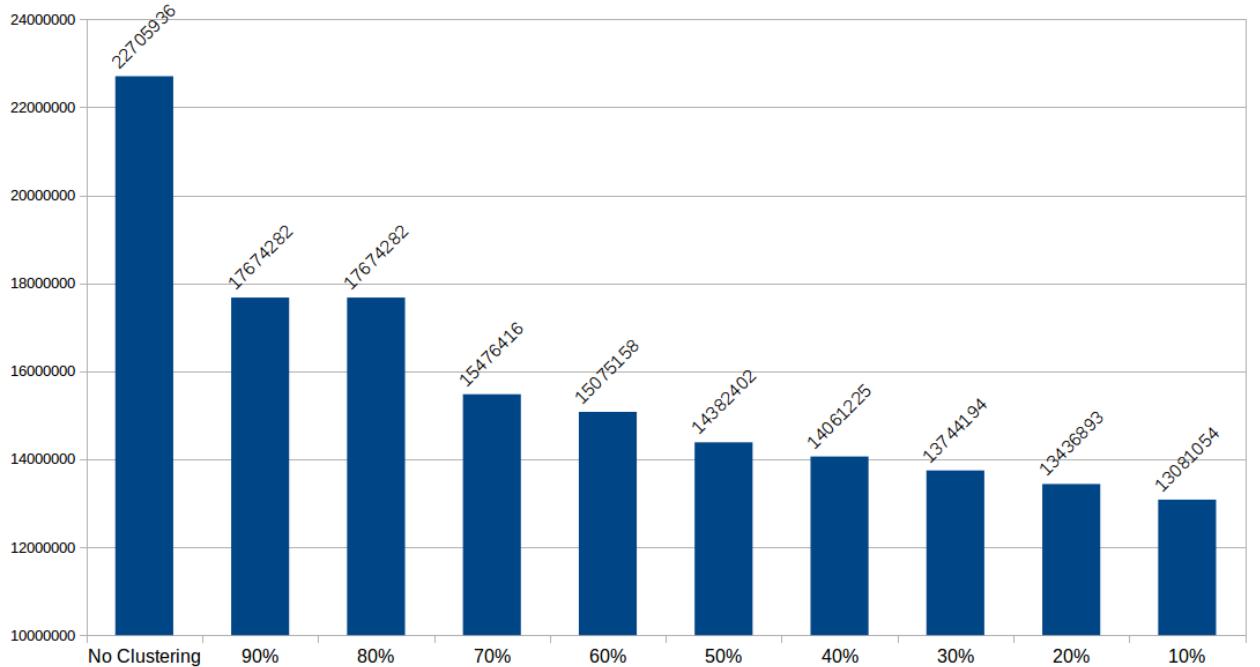


Figure 6.13.: Resulting amount of clusters for different values of p

Benchmark Results As already mentioned, the clustering introduces an error by grouping nearby events which have almost the same area. After clustering, some events may be counted as intersecting a benchmark area by more than 50%, even though the original event was not. Table 6.5 lists the count of events for each benchmark area, with events of 10 weeks. The full range of benchmark values can be found in the Appendix B.1.

As expected, the clustering has an impact on the count of the events. As one can see, the largest difference happens when comparing no clustering with a 90% clustering. The count increases about 1.4% for all benchmark areas. Interestingly, the difference in counts for 90% to 10% clustering is negligible. This can be seen even better when visualizing the count of events in one benchmark area as graph in Figure 6.14.

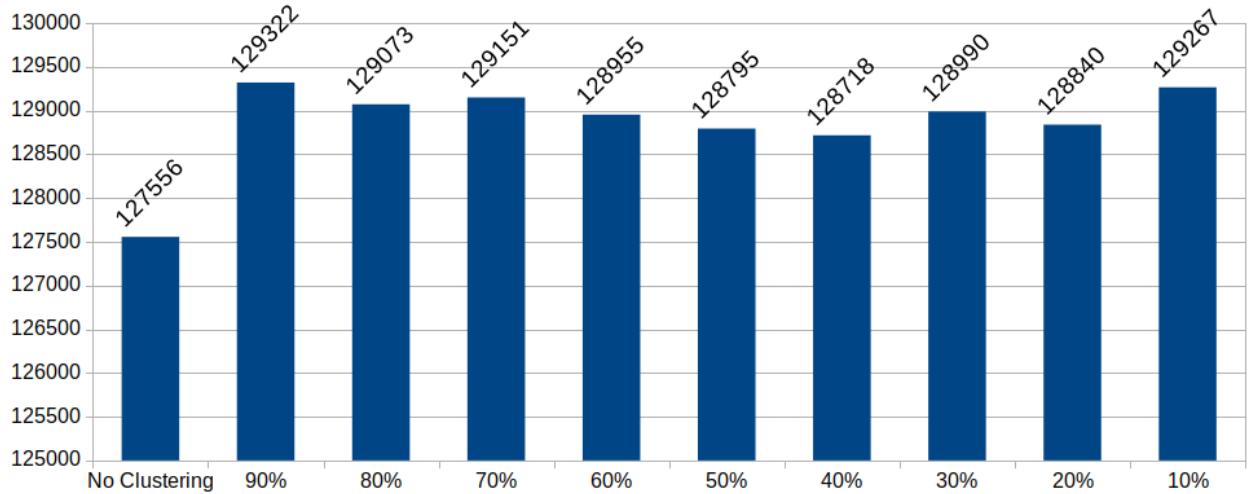


Figure 6.14.: Resulting count of events for benchmark area "Huge amount of small areas"

The small dimension of error, even with very low values of p , leads to the conclusion that the test data is very sparse. Since the clustering is done separately for every date, it is to be expected that the error gets bigger when more data is present for a given date. This is shown in Figure 6.15 and Figure 6.16 where one can see the resulting event counts when the clusters were built per week or month, instead of per day.

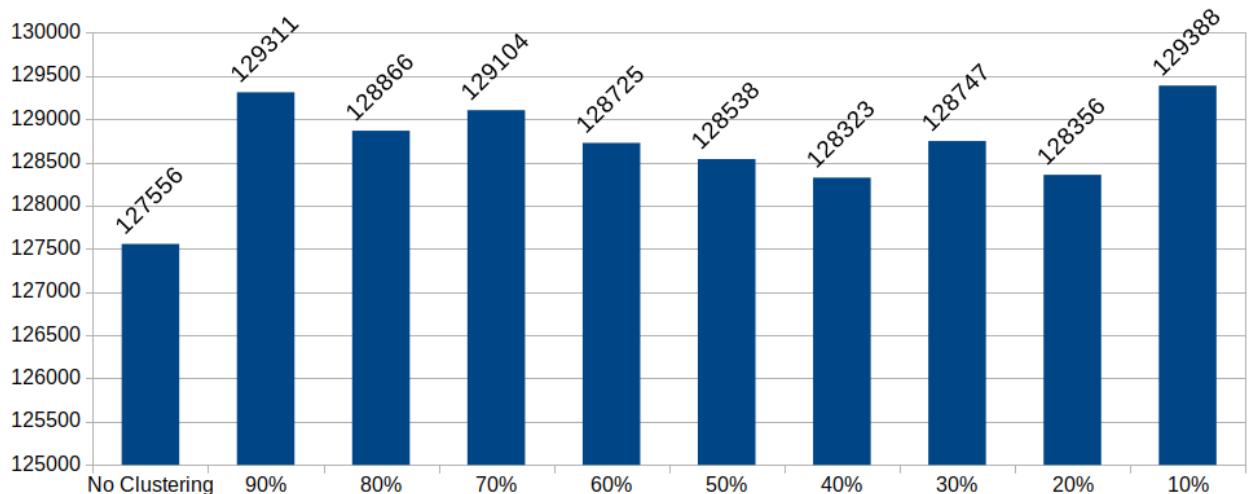


Figure 6.15.: Resulting count of events for benchmark area "Huge amount of small areas" with clusters created per week instead of day.

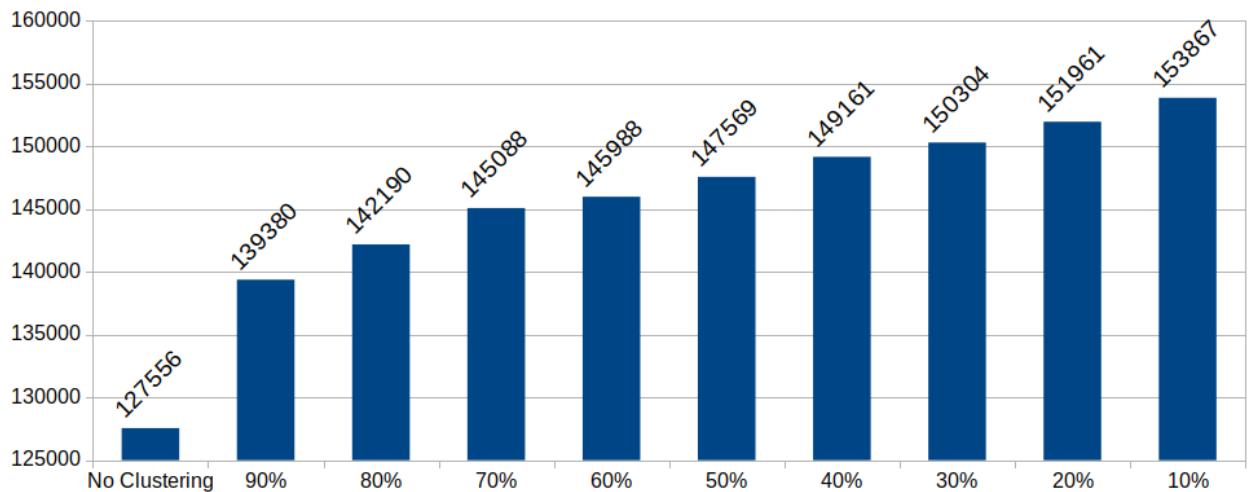


Figure 6.16.: Resulting count of events for benchmark area "Huge amount of small areas" where clusters created per month instead of day.

Still, the count of clusters per week are not very different from the clusters per day, since the error between no clustering and clusters with 10% is only 1.3%. First with clusters per month the error gets larger, with an error of 8.5% between no clustering and a 90% clustering. But one has to bear in mind that clusters per week deals with a dataset of $7 \times 21\text{M} = 147\text{M}$ records and clusters per month equals a dataset of $30 \times 21\text{M} = 630\text{M}$ records. Additionally, the "simulated" record is not comparable with a real data with so many records, since the distribution of the events would be different.

Benchmark Runtime Since the amount of events is reduced after clustering, the runtime decreases considerably. The fact that less data can be queried faster has already been shown with the original implementation in Section 6.2.3.

6.3.5.3. Conclusion

By reducing the amount of data to query, the runtime decreases appreciably. The error introduced is tolerable. For the benchmarks, the recommended value of p for the clustering algorithm is 0.4, which says that two events must at least intersect 40% in order to cluster them together. This sounds like a small intersection, but it must be calculated on a relative-basis since the area of the cluster is built based on the centroid of the event locations. One must keep in mind, that it will be necessary to evaluate which value for p is the best, based on the data. If other amounts of data or data with a different distribution is used, it may be necessary to evaluate a new suitable value for p .

6.3.6. Reduce Accuracy Distribution

In Section 3.1 the *accuracy* attribute has been described. The amount of different values is very large. This distribution of different values can be reduced, which leads to advantages for the clustering algorithm, since only events with the same accuracy are clustered.

For example, events with the accuracies 10, 11 and 12 are now using the accuracy 10. Therefore, no events with the accuracies 11 and 12 exists anymore. This leads to new areas for the events, which accuracy was altered. But this is negligible if the difference in the area is not too large. How large the

difference in the area can be tolerable, depends on the accuracy of the event. The larger the accuracy, the more room for adjustment.

When defining that only accuracies are summarized which resulting event areas differ by only 10%, the following formula can be used to calculate the maximal difference in the accuracy, based on the accuracy.

$$\begin{aligned} \text{original_area} &= \text{accuracy}^2 * \pi \\ \text{new_area} &= (\text{accuracy} - \text{max_shift})^2 * \pi \\ 1 - \frac{\text{new}}{\text{original_area}} &= 0.1 \\ \text{max_shift} &\approx 0.05 * \text{accuracy} \end{aligned}$$

With the formula $\text{max_shift} \approx 0.05 * \text{accuracy}$ the maximal shift for each accuracy can be calculated. The value is round off to the nearest integer, with a minimal value of 1. The resulting value is named *reduction_factor*. With this reduction factor, the new reduced accuracy can be calculated with this formula:

$$\text{reduced_accuracy} = \text{accuracy} - (\text{accuracy} \bmod \text{reduction_factor})$$

After applying this formula to all accuracies, the different accuracy values for events with accuracies between 1 and 1000 are reduced from 1000 different values to 148 different values. This has a positive impact on the clustering algorithm.

6.3.7. Parallel Queries

PostgreSQL version 9.6 and higher supports parallel queries. With parallel queries, the workload of one big query is divided into multiple sub queries, which can be executed in parallel [25]. Unfortunately, PostgreSQL, even in version 10, combined with PostGIS 2.4, is not good in parallelizing spatial queries [22]. Forks for PostgreSQL 10 and PostGIS 2.4 exists, which adds better support for parallel queries with PostGIS, but it is not yet possible in a stock PostgreSQL.

Instead of using the capabilities of PostgreSQL to parallelize, it is possible to split a big query into sub queries before sending them to PostgreSQL. For the queries of this use case, this is quite simple, since queries getting slow means the defined date range is large. Therefore, the splits of the queries can be done based on the date range.

For example, instead of querying events for a range of four weeks, one can split the query in four subqueries, where each subquery queries for one week of data. The results can then merged together before processing them further.

Figure 6.17 shows the benchmark results for querying events with different number of subqueries. As one can see, the runtime only improves significantly when switching from one to two queries. This is most likely due to the fact, that the host system, where the benchmarks are executed, only have two CPU cores³.

³Since the query to the database is an I/O query, it should release the global interpreter lock of Python for other Threads. However, this must be supported by the used library psycopg2. It is maybe worth to test other libraries for better multi-threading capabilities.

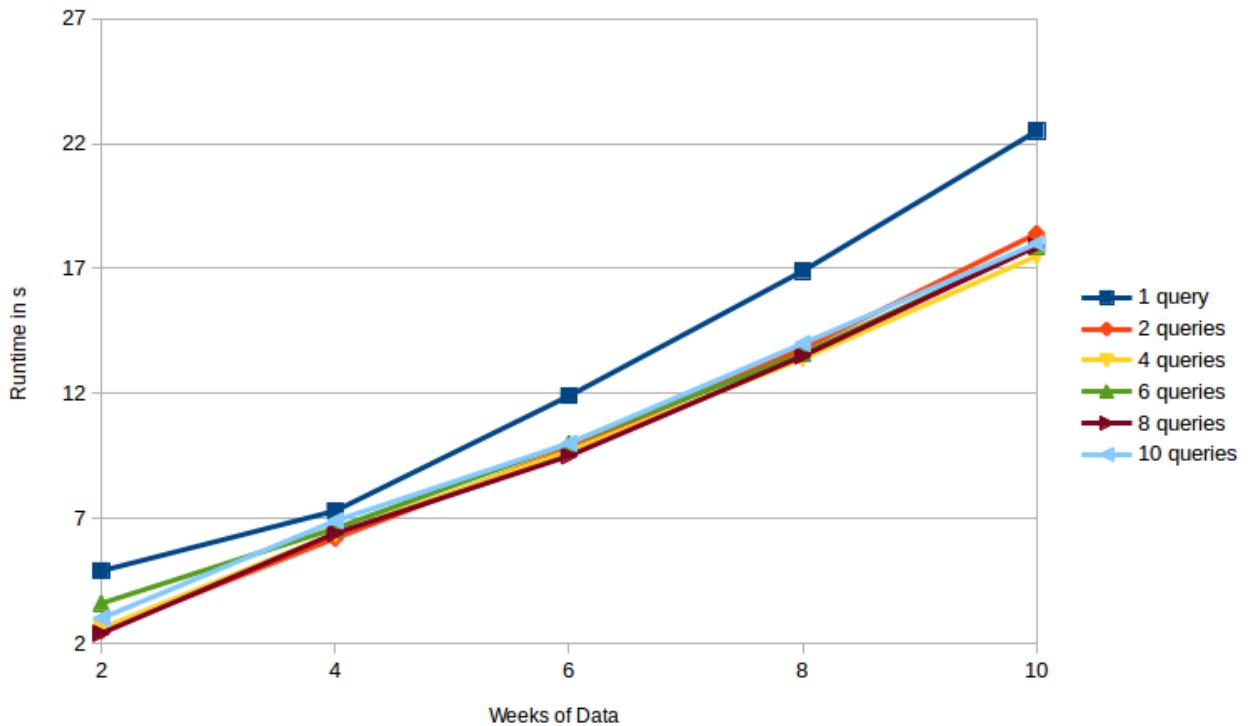


Figure 6.17.: Benchmarks when splitting query in multiple sub queries and executing them parallel.

6.3.8. Multicolumn Index

As described in Section 6.2.2 two indexes are created on the events table. A `btree` index on the `occurred_at` column and a `gist` index on the `location` column. However, it is not guaranteed that PostgreSQL will use both indexes for querying the events. To ensure that both queries are executed on both indexes, and as benchmarks have shown better performance, creating one multicolumn index is more suitable. To create a multicolumn index which mixes a `btree` and a `gist` index, the PostgreSQL extension `btree_gist` is necessary [24]. Listing 6.4 shows the query to create this multicolumn index.

```

1 CREATE EXTENSION IF NOT EXISTS btree_gist;
2 CREATE INDEX clusters_occurred_at_location ON clusters USING gist(occurred_at, ↴
    location);
```

Listing 6.4: Creating multicolumn index on occurred_at and location

6.4. Benchmarks after Optimizations

This section contains the benchmark results after all optimizations of Section 6.3 have been applied. This includes:

- Preprocessing events
- Using convex hull instead of buffered areas
- Simplifying input areas with a tolerance of 5
- quad_segs = 1 for buffering
- Reduction of accuracies by a factor of 3
- Clustering of events with p = 40
- Amount of threads = amount of weeks

Figure 6.18 shows the runtimes for the benchmarks, when querying the events with the applied optimizations.

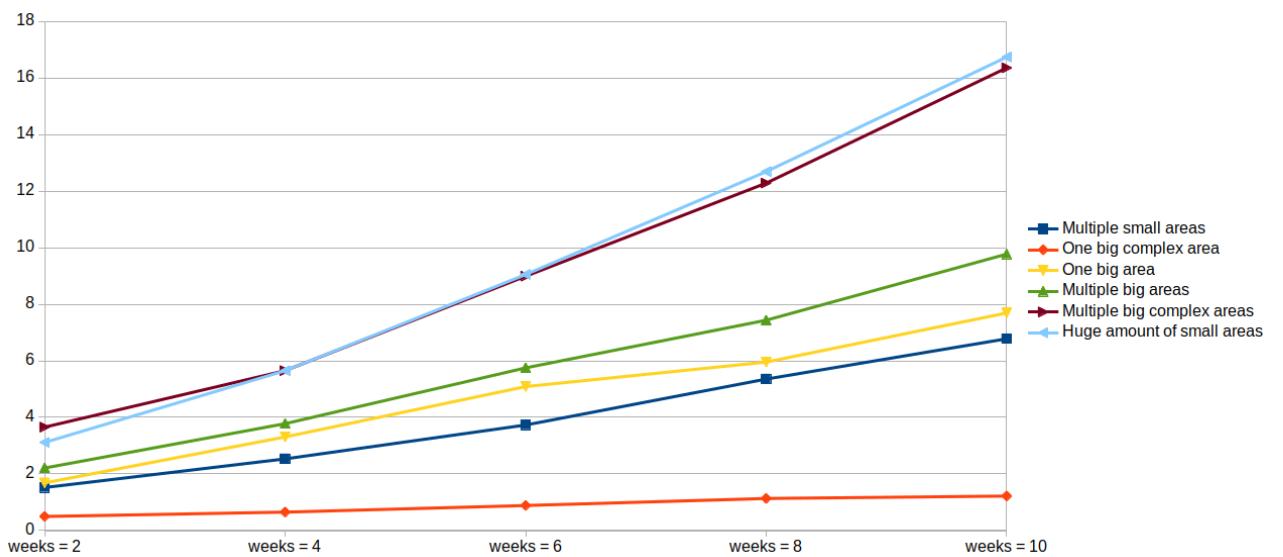


Figure 6.18.: Benchmark results for different benchmark areas and increasing amount of data.

The queries are executing significantly faster than before. Table 6.6 compares the results of the original query benchmarks from Section 6.2.3 with benchmark results of the optimized queries. For the comparison the runtimes for 10 weeks of data are shown. Additionally the speedup ($\frac{t_1}{t_2}$) is listed.

Table 6.6.: Comparing the benchmark results of the original and the optimized queries.

	Not optimized	Optimized	Speedup
Multiple small areas	129s	7s	18.42
One big complex area	12s	1s	12.00
One big area	161s	8s	20.12
Multiple big areas	111s	10s	11.10
Multiple big complex areas	202s	16s	12.62
Huge amount of small areas	547s	17s	32.17

6.4.1. Further Optimizations

As already mentioned, one could invest almost infinite time to improve the performance of the implementation with PostgreSQL and PostGIS further. A promising approach is the ability of PostgreSQL to parallelize queries for using multiple CPU cores. This feature is named Parallel Query and was introduced with PostgreSQL 9.6 and has got multiple improvements with PostgreSQL 10.0. However, PostgreSQL is not yet able to parallelize all kind of queries and PostGIS is a special case by itself. There are efforts to extend PostGIS accordingly, but this is not yet production ready [26] [27] but it is definitely worth to keep an eye on this.

Another approach to scale the implementation is sharding. With sharding, the data and queries gets distributed across multiple machines for faster processing. For example, the company [citusdata.com](https://www.citusdata.com)⁴ provides this for PostgreSQL databases [3]. If and how this works with PostGIS queries was not tested.

⁴<https://www.citusdata.com/>

Part III.

Implementation with Apache Spark

7. Evaluation

There exists many solutions to extend big data frameworks as Hadoop or Apache Spark with spatial data types. In this section these solutions are described and categorized on the basis of certain criteria. Finally, the most suitable solution is selected for the implementation.

7.1. Criteria

The criteria for the evaluation, listed in order of their priority, are:

- Compatibility with Apache Spark
- Spatial Analysis Methods with Vector Data
- Support for SparkSQL
- Spatial Partitioning
- Efficient Spatial Joins
- Compatible with pySpark

The focus of this thesis is a solution which works with Apache Spark. Due to the use cases, spatial analysis methods on vector (rather than raster) data is of utmost importance. The support for SparkSQL has been given a high priority, as it leads to a very convenient API, especially when coming from PostgreSQL with PostGIS. Spatial partitioning and efficient spatial joins are preconditions for an efficient and scalable data processing. The last criterion is compatibility with pySpark and therefore providing an API for Python. Python is popular in data analysis and geospatial data processing, therefore a Python API would be a plus. But especially with SparkSQL support, Python bindings are less relevant. Therefore, the criterion is given a low prioritization.

7.2. Solutions

Based on the above criteria, the existing solutions are shortly summarized.

7.2.1. GeoWave

GeoWave is a software library that connects the scalability of distributed computing frameworks and key-value stores with modern geospatial software to store, retrieve and analyze massive geospatial datasets [10]. It supports the key-value stores Apache Accumulo and Apache HBase.

For querying, multiple options exist. The most common option is to use a GeoServer¹ Plugin. Alternatively a CLI, a RDD API or SparkSQL can be used. Even though a SparkSQL API exists, it does not yet cover all methods necessary for the present use cases. For example, it does not contain a method to return the intersection of two geometries. But since the RDD API implements such method, it could be extended accordingly.

¹<http://docs.geoserver.org/>

The focus of GeoWave is on raster data, though vector data is also supported.

7.2.2. GeoMesa

GeoMesa is an open source suite of tools that enables large-scale geospatial querying and analytics on distributed computing systems. GeoMesa provides spatio-temporal indexing on top of the Accumulo, HBase, Google Bigtable and Cassandra databases for massive storage of point, line, and polygon data [5].

The functionality of GeoMesa overlaps to a good extent with the functionality of GeoWave [5], though GeoMesa has a stronger focus on vector data. Additionally its documentation is more detailed than that of GeoWave.

GeoMesa has a SparkSQL API². It supports spatial partitioning and efficient joins with indexes. Similar to GeoWave, no method exists in GeoMesa to return the intersection of two geometries. Though, it is possible to extend the functionality to include this feature.

7.2.3. GeoTrellis

GeoTrellis is a Scala library and framework that uses Spark to work with raster data [9]. Even though its main focus is on raster data, a Scala wrapper around JTS exists, which provides vector data types and operations. As a distributed vector-feature store, GeoMesa or GeoWave can be used. But both integrations are still in an experimental level.

GeoTrellis has Python bindings through a project called GeoPySpark³.

7.2.4. STARK

STARK is a framework that tightly integrates with Apache Spark and adds support for spatial and temporal data types and operations [30]. It was implemented as part of a comparison of existing solutions for spatial data processing on Apache Hadoop and Apache Spark [14].

It supports spatial partitioning and indexes. However, its functional scope is very limited. It has no Spark SQL API and no documentation besides the very basic readme of the GitHub repository.

7.2.5. Magellan

Magellan extends Apache Spark with spatial data types and operations [17]. It supports spatial indexes but no spatial partitioning yet. Even though a SparkSQL API exists, it covers only a small amount of operations (such as intersects, within and contains).

Magellan has almost no documentation, and with just two code contributors, very little progress has been made.

Python bindings existed once, but got broken with the newest release of Apache Spark. Therefore, it does not support pySpark anymore.

²<http://www.geomesa.org/documentation/user/spark/sparksql.html>

³<https://github.com/locationtech-labs/geopyspark>

7.2.6. GeoSpark

GeoSpark extends Apache Spark with a set of out-of-the-box Spatial Resilient Distributed Datasets (SRDDs) that efficiently load, process, and analyze large-scale spatial vector data across machines [7].

It supports spatial partitioning and indexes for joins. Additionally, it has an extension for SparkSQL. Unfortunately, the spatial partitioning is not yet available for SparkSQL, where it is necessary to switch to SRDDs.

Some of the documentation is still somewhat brief, but covers most of the functionalities. No Python bindings are available.

7.2.7. LocationSpark

LocationSpark is a spatial data processing system built on top of Apache Spark [16]. Since the last commit was more than one year ago, it is considered to be not maintained anymore.

7.2.8. Combine with GeoPandas

One could use an existing library such as GeoPandas to extend Apache Spark. GeoPandas is an open source project to work with geospatial data in python [6]. Although this would be possible, no spatial partitioning or indexes would be supported. Additionally, no SparkSQL API would be available with this solution.

7.3. Summary

Table 7.1 and 7.2 lists a summary of the criteria for each solution. If a criteria is only partly fulfilled, it is marked with additional parentheses.

Based on these results, GeoSpark is chosen for implementing the use cases of this thesis. GeoWave and GeoTrellis were dismissed for the focus on raster data. GeoMesa would be an interesting solution, but due to its additional dependency on Apache Accumulo, was dismissed too.

Table 7.1.: Evaluation summary Part 1

	GeoWave	GeoMesa	GeoTrellis	GeoSpark
Compatibility with Apache Spark	✓	✓	✓	✓
Spatial Analysis Methods with Vector Data	(✓)	✓	(✓)	✓
Support for SparkSQL	(✓)	(✓)	(✓)	✓
Spatial Partitioning	✓	✓	✓	(✓)
Efficient Spatial Joins	✓	✓	✓	✓
Comprehensive Documentation	(✓)	✓	✓	(✓)
Compatible with pySpark	X	X	✓	X

Table 7.2.: Evaluation summary Part 2

	STARK	Magellan	LocationSpark	GeoPandas
Compatibility with Apache Spark	✓	✓	✓	✓
Spatial Analysis Methods with Vector Data	✓	✓	✓	✓
Support for SparkSQL	✗	(✓)	✗	✗
Spatial Partitioning	✓	✓	✗	✗
Efficient Spatial Joins	✓	✓	✓	✗
Comprehensive Documentation	✗	✗	✗	(✓)
Compatible with pySpark	✗	✗	✗	✓

7.4. Data Source

Apache Spark can read from various data sources. It is even possible to use a JDBC driver to connect to PostgreSQL. Other data sources are files with different data formats, for example text files like CSV or TSV. Alternatively more specific data formats like Avro⁴ (row-based) or Parquet⁵ (column-based) exist. In terms of reading-speed and compressions, they are more suitable than plain text files.

The data source, is it a file or a JDBC connection must be accessible by each worker node of the Apache Spark cluster. Files are therefore typically provided on a distributed file system, like the Hadoop Distributed File System (HDFS)⁶ or an object storage like Amazon S3⁷.

Since the Parquet file format is very common and has excellent performance benchmarks, it is chosen for the implementation of both use cases.

⁴<https://avro.apache.org/>

⁵<https://parquet.apache.org/>

⁶HadoopDistributedFileSystem

⁷<https://aws.amazon.com/s3/>

8. Use Case AOI

8.1. Implementation

This Section describes the implementation of the AOI use case with Apache Spark extended with GeoSpark. It is based on the scope of the PostgreSQL and PostGIS implementation. Differences and limitations are documented accordingly.

8.1.1. Work with OpenStreetMap

Multiple options exist to work with OpenStreetMap data using Apache Spark. For example, with the tool OpenStreetMap Parquetizer,¹ PBF files can be converted to Parquet files.

Alternatively, one can read directly from a PostgreSQL database. However, this is slower than reading from Parquet files.

For this implementation, the OpenStreetMap data is read from the PostgreSQL database and written to Parquet files. This allows a better control of the data layout, than using a tool like OpenStreetMap Parquetizer.

Listing 8.1 shows the corresponding lines of Scala code, to read the data from the PostgreSQL database and write it to a Parquet file. Only the relevant tags are written to the Parquet file.

```
1 // opts = connection parameter for database
2 sparkSession.read
3     .format("jdbc")
4     .options(opts)
5     .option("dbtable", "planet_osm_point")
6     .load
7     .createOrReplaceTempView("points")
8
9 sparkSession.sql("SELECT way AS geometry, amenity, leisure, landuse, shop, access
10    FROM points")
11        .write
12        .parquet("out/points.parquet")
```

Listing 8.1: Read points from PostgreSQL database and write to Parquet file.

Listing 8.2 shows the code to read from the created Parquet file and how the rows can be accessed using SparkSQL.

```
1 sparkSession.read
2     .parquet("out/points.parquet")
3     .createOrReplaceTempView("points")
4
5 val poisDataFrame = sparkSession.sql("SELECT ST_GeomFromWKB(geometry) FROM points
6     WHERE array_contains(array('cafe', 'restaurant'), 'shop')")
```

¹<https://github.com/adrianulbona/osm-parquetizer>

Listing 8.2: Read from Parquet file.

8.1.2. Pre-Cluster POIs

As in the PostgreSQL + PostGIS implementation, it is necessary to pre-cluster the POIs as done in Section 5.1.2. Therefore, the DBSCAN clustering algorithm is used. However, GeoSpark does not come with an implementation of the DBSCAN clustering algorithm, for which an additional dependency is necessary.

Helmut Neukirchen [18] benchmarked multiple DBSCAN implementations for Apache Spark. Unfortunately, the results are devastating. DBSCAN on Spark² only returns a proximity of the clusters based on bounding box calculation. Neukirchen notes that this leads to completely wrong clusters. RDD DBSCAN³ is not able to scale to more than 16 to 32 cores. And Spark DBSCAN⁴ is marked as experimental, and even slow with more than 900 cores. However, the documentation of the Spark DBSCAN implementation is better than the documentation of the RDD DBSCAN implementation. Additionally, the benchmarks of Neukirchens are more then 2 years old and the corresponding implementations have been developed. Therefore Spark DBSCAN is used for this thesis.

The implementation only works on points and not on polygons, as in the PostgreSQL implementation. The points are partitioned using a density based partitioner⁵. Then, the neighbors of each point are identified. Points are neighbors if they lie within the distance ϵ . For points which are near the border of a partition, neighbors across partitions are identified too. Then, the points are clustered for each partition. And finally, clusters which share neighbors across partitions are merged. Figure 8.1 illustrates the approach.

²<https://github.com/mraad/dbscan-spark>

³<https://github.com/irvingc/dbscan-on-spark>

⁴https://github.com/alitouka/spark_dbSCAN

⁵More details about the density based partitioning can be found here: https://github.com/alitouka/spark_dbSCAN/wiki/Density-based-partitioning

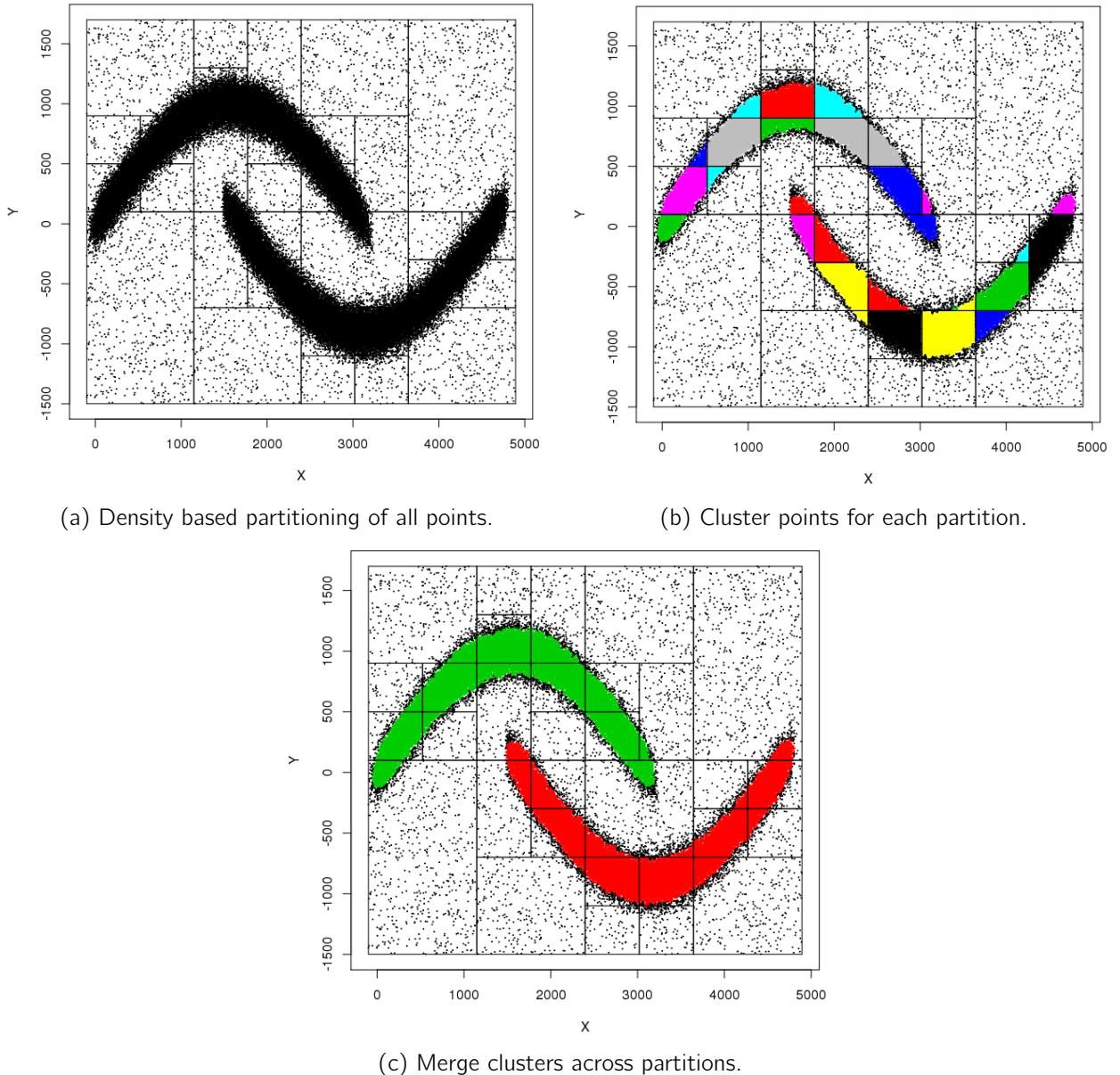
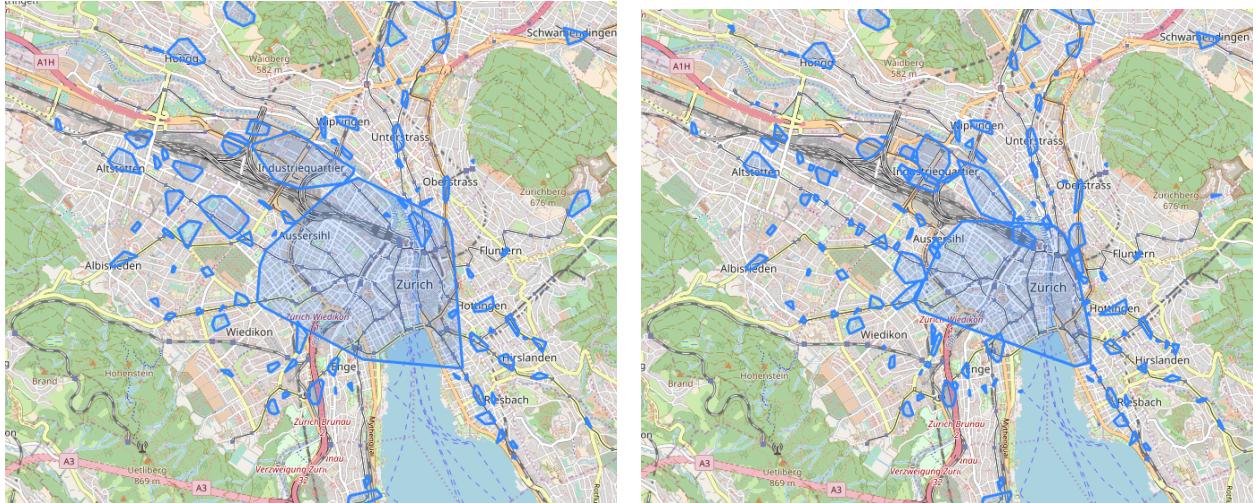


Figure 8.1.: Illustrations of DBSCAN algorithm on Spark. Source: [29]

The resulting clusters, when applied to the points in Switzerland, are shown in Figure 8.2a. Besides that, Figure 8.2b shows the same clusters from the PostgreSQL + PostGIS implementation. Since the Apache Spark clustering algorithms only work on points and not on whole polygons, different shapes of clusters are formed.



(a) Apache Spark implementation.

(b) PostgreSQL + PostGIS implementation

Figure 8.2.: Resulting clusters of DBSCAN clustering of Switzerland.

Since the DBSCAN algorithm is not part of GeoSpark, the clustering cannot be applied directly on the GeoSpark geometry instances. Therefore, transformation of the data from GeoSpark types to the DBSCAN clustering algorithm types and vice versa, is necessary. Due to the implementational overhead, this could have negative impacts on the benchmarks.

Listing 8.3 shows the Scala code to pre-clusters the POIs. The code is explained with comments in detail.

```

1 // map GeoSpark geometries to Point class of DBSCAN clustering algorithm
2 val pois = poisDataFrame.rdd.map(row => new Point(row.getAs[Geometry](1).✓
3   getCoordinates()(0).y, row.getAs[Geometry](1).getCoordinates()(0).x))
4
5 // init clustering settings, eps = 100 meters, minPts = 3
6 val clusteringSettings = new DbscanSettings().withEpsilon(100).withNumberOfPoints✓
7   (3)
8
9 // cluster pois
10 val model = Dbscan.train(pois, clusteringSettings)
11
12 // map clustered pois to Spark Rows
13 val preclusteredPois = model.clusteredPoints.map { point => Row(point.coordinates✓
14   (0), point.coordinates(1), point.clusterId)}
15
16 // create DataFrame from spark rows
17 val schema = new StructType(Array(StructField("x", DoubleType), StructField("y",✓
18   DoubleType), StructField("cluster_id", LongType)))
19 sparkSession.createDataFrame(preclusteredPois, schema).createOrReplaceTempView("✓
20   preclustered_pois")
21
22 // finally, the rows can be casted back to GeoSpark geometries
23 sparkSession.sql("SELECT ST_Point(CAST(x AS Decimal(24,20)),CAST(y AS Decimal✓
24   (24,20))) AS geometry, cluster_id FROM preclustered_pois").✓
25   createOrReplaceTempView("preclustered_pois")

```

Listing 8.3: Pre-clustering of the POIs.

8.1.3. DBSCAN Local Adaption

As in the PostgreSQL + PostGIS implementation, a *minPts* parameter for the DBSCAN algorithm is calculated for each pre-cluster, based on the area and the amount of POIs. To calculate the area of the pre-clusters, a convex hull of the clustered points must be calculated. GeoSpark contains the function `ST_ConvexHull` to calculate the convex hull of a polygon. Unfortunately, GeoSpark does not contain a function to aggregate the clustered points to one polygon. In PostgreSQL one would use `ST_Union`, which exists in GeoSpark as `ST_Union_Aggr` but only works on polygons [8]. Alternatively, the function `ST_Envelope_Aggr` exists, which can be used as aggregations of points, but only returns an envelop. Listing 8.4 shows the corresponding Spark SQL queries to draw the hulls of the pre-clusters and calculate the *minPts* value. The formula to calculate the *minPts* value was derived in Section 5.1.3.

```
1 sparkSession.sql(
2     """SELECT cluster_id, ST_ConvexHull(ST_Envelope_Aggr(geometry)) AS hull,
3         |ST_Area(ST_ConvexHull(ST_Envelope_Aggr(geometry))) AS area,
4         |COUNT(geometry) AS pois_count
5     |FROM preclustered_pois
6     |GROUP BY cluster_id""".stripMargin).createOrReplaceTempView("preclusters")
7
8 sparkSession.sql(
9     """SELECT cluster_id,
10        |GREATEST(2, round((-5.342775355 * 0.0000001 * area + 5.738819175 * ✓
11        |0.001 * pois_count + 2.912834423)) AS dbscan_minPts
12     |FROM preclusters""".stripMargin).createOrReplaceTempView("preclusters")
```

Listing 8.4: Draw hulls of the pre-clusters and calculate *minPts* value.

8.1.4. Cluster POIs

After the POIs have been pre-clustered, the POIs in each pre-cluster can now be clustered individually. However, it is not possible to use the chosen DBSCAN algorithm implementation from Section 8.1.2. This is because the implementation only works with RDDs and therefore nested RDDs would be required, which is not allowed with Apache Spark.

Since running the clustering algorithm on one pre-cluster is not performance-critical and it is not necessary to do this in a distributed manner, one can use a non-distributed implementation of the DBSCAN algorithm. A widespread implementation is part of the Apache Commons, a project focused on reusable Java components⁶.

Listing 8.5 shows the Scala code which uses the `DBSCANClusterer` of Apache Commons to cluster the POIs of each pre-cluster. Comments inside the code explains it in more detail.

```
1 class OwnPoint(x: Double, y: Double) extends Clusterable {
2     override def getPoint: Array[Double] = {
3         return Array(x, y)
4     }
5 }
6
7 val pois = preclusteredPoisWithMinPts.groupBy(poi => poi.getLong(0)).map {
8     case (cluster_id, pois) => {
9         val minPts = pois.last.getDouble(1).asInstanceOf[Int]
10        val eps = 35 // 35 meters
11    }
12 }
```

⁶<https://commons.apache.org/>

```

11
12 // map the pois to an own class which can be used by the DBSCANclusterer
13 val points = pois.map(row => new OwnPoint(row.getAs[Point](2).getCoordinates(),
14                                         ()(0).x,
15                                         row.getAs[Point](2).getCoordinates(),
16                                         ()(0).y))
17
18 // cluster the points
19 val clusters = clusterer.cluster(points.asJavaCollection)
20
21 // map the resulting clusters to an array of tuples (cluster_id, points)
22 // the cluster_id is derived by combining the pre-cluster_id and the new cluster_id
23 // the points are mapped back to tuples of doubles (x, y)
24 clusters.toArray().map(cluster => (cluster_id << 3 + clusters.indexOf(cluster),
25                                     ),
26                                     cluster.asInstanceOf[Cluster[OwnPoint]].getPoints.toArray().map(point => point
27                                     .asInstanceOf[OwnPoint].getPoint()))
28 }.reduce(_++_) // merge resulting lists
29 .map { case (cluster_id, points) => points.map(point => (point(0), point(1),
30                                         cluster_id)) } // map to (x, y, cluster_id)
31 .reduce(_++_) // merge resulting lists
32 .map { clustered_poi => Row(clustered_poi._1, clustered_poi._2, clustered_poi._3)} // map to Spark rows

```

Listing 8.5: Draw hulls of the pre-clusters and calculate minPts value.

The variable `pois` now contains an array of Apache Spark rows, where each row contains a POI with the coordinates and a cluster id. These rows can then be converted back to an Apache Spark DataFrame to be used with SparkSQL and GeoSpark. This is shown in Listing 8.6.

```

1 // a schema is necessary to convert a RDD to a DataFrame
2 val clustered_pois_schema = new StructType(Array(StructField("x", DoubleType),
3                                                 StructField("y", DoubleType), StructField("cluster_id", LongType)))
4 sparkSession.createDataFrame(sparkSession.sparkContext.parallelize(pois),
5                             clustered_pois_schema).createOrReplaceTempView("clustered_pois")

```

Listing 8.6: Convert POIs RDD to DataFrame.

Finally, the clustered POIs are available in SparkSQL and the hull can be drawn using GeoSpark. The only aggregation function in GeoSpark to combine polygons calculates a rectangle around the polygons. The corresponding code is shown in Listing 8.7.

```

1 // the POIs need to be cast to an GeoSpark point using ST_Point
2 sparkSession.sql("""SELECT ST_Point(CAST(x AS Decimal(24,20)),
3                                     |           CAST(y AS Decimal(24,20))) AS geometry,
4                                     |           cluster_id FROM clustered_pois""".stripMargin).createOrReplaceTempView("clustered_pois")
5
6 // finally the hull can be drawn for each cluster
7 val aois = sparkSession.sql("SELECT ST_ConvexHull(ST_Envelope_Aggr(geometry)) AS aoi
8                           FROM clustered_pois GROUP BY cluster_id")

```

Listing 8.7: Draw hulls with GeoSpark.

Due to the use of the `ST_Envelope_Aggr` function, the resulting AOIs are rectangular. This is illustrated in Figure 8.3. Even though the AOIs are not useful, benchmarking the implementation will be possible. The results are compared with the AOIs of the PostgreSQL + PostGIS implementation in Section 8.2.

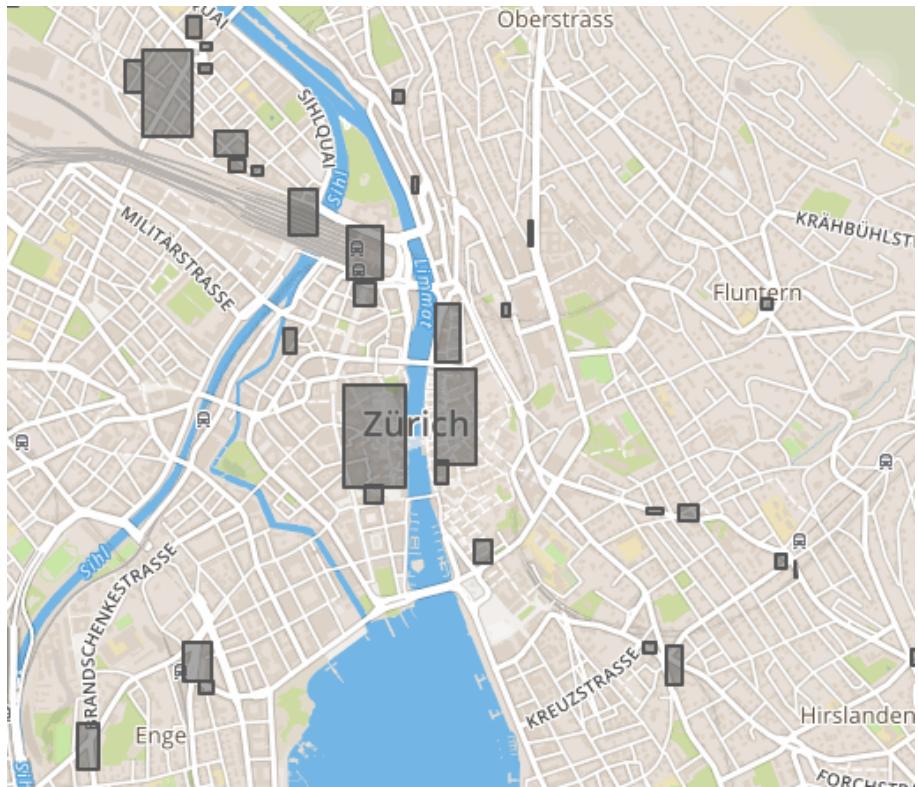


Figure 8.3.: AOIs generated with GeoSpark.

8.1.5. Network Centrality

The used library for the PostgreSQL + PostGIS implementation was osmnx. Since it is a Python library, it cannot be used with the Apache Spark implementation. Eventually GraphX⁷ could have been used for the task, but as it does not contain anything related to network centrality, it would be necessary to extend it accordingly. This has not been done as part of this thesis.

8.1.6. Exclude Water and Sanitize AOIs

Unfortunately, GeoSpark does not implement anything like the `ST_Difference` function of PostGIS, used in Section 5.1.6. Therefore, this step is skipped. The same applies for the `ST_Simplify` function of PostGIS, used in Section 5.1.7.

⁷<http://spark.apache.org/graphx/>

8.1.7. Export AOIs

GeoSpark provides a function `saveAsGeoJSON` to save geometries as GeoJSON. Since it is only available on RDDs, the AOIs DataFrame must be converted to a RDD first. The corresponding code is shown in Listing 8.8.

```
1 val spatialAoisRDD = new SpatialRDD[Geometry]
2 spatialAoisRDD.rawSpatialRDD = Adapter.toRdd(aoisDataFrame)
3 spatialAoisRDD.saveAsGeoJSON("out/aois.export")
```

Listing 8.8: Export AOIs as GeoJSON

Due to the nature of Apache Spark, files can not be exported as one file, but the AOIs of each partition are written to its own file. This is illustrated in Figure 8.4. To merge the files, one could use for example a simple bash command.

```
$ ls out/aois.export/
part-00000 part-00014 part-00028 part-00042 part-00056 part-00070 part-00084 part-00098 part-00112 part-00126 part-00140 part-00154 part-00168 part-00182 part-00196
part-00001 part-00015 part-00029 part-00043 part-00057 part-00071 part-00085 part-00099 part-00113 part-00127 part-00141 part-00155 part-00169 part-00183 part-00197
part-00002 part-00016 part-00030 part-00044 part-00058 part-00072 part-00086 part-00100 part-00114 part-00128 part-00142 part-00156 part-00170 part-00184 part-00198
part-00003 part-00017 part-00031 part-00045 part-00059 part-00073 part-00087 part-00101 part-00115 part-00129 part-00143 part-00157 part-00171 part-00185 part-00199
part-00004 part-00018 part-00032 part-00046 part-00060 part-00074 part-00088 part-00102 part-00116 part-00130 part-00144 part-00158 part-00172 part-00186 SUCCESS
part-00005 part-00019 part-00033 part-00047 part-00061 part-00075 part-00089 part-00103 part-00117 part-00131 part-00145 part-00159 part-00173 part-00187
part-00006 part-00020 part-00034 part-00048 part-00062 part-00076 part-00090 part-00104 part-00118 part-00132 part-00146 part-00160 part-00174 part-00188
part-00007 part-00021 part-00035 part-00049 part-00063 part-00077 part-00091 part-00105 part-00119 part-00133 part-00147 part-00161 part-00175 part-00189
part-00008 part-00022 part-00036 part-00050 part-00064 part-00078 part-00092 part-00106 part-00120 part-00134 part-00148 part-00162 part-00176 part-00190
part-00009 part-00023 part-00037 part-00051 part-00065 part-00079 part-00093 part-00107 part-00121 part-00135 part-00149 part-00163 part-00177 part-00191
part-00010 part-00024 part-00038 part-00052 part-00066 part-00080 part-00094 part-00108 part-00122 part-00136 part-00150 part-00164 part-00178 part-00192
part-00011 part-00025 part-00039 part-00053 part-00067 part-00081 part-00095 part-00109 part-00123 part-00137 part-00151 part-00165 part-00179 part-00193
part-00012 part-00026 part-00040 part-00054 part-00068 part-00082 part-00096 part-00110 part-00124 part-00138 part-00152 part-00166 part-00180 part-00194
part-00013 part-00027 part-00041 part-00055 part-00069 part-00083 part-00097 part-00111 part-00125 part-00139 part-00153 part-00167 part-00181 part-00195
```

(a) Files in export directory.

```
$ cat out/aois.export/part-00194
{"type": "Feature", "geometry": {"type": "Polygon", "coordinates": [[[7.556307291793928, 46.49073457890784], [7.556307291793928, 46.49123621339556], [7.55702100328716, 46.49073457890784], [7.556307291793928, 46.49073457890784]]]}, "properties": {"UserData": ""}}
{"type": "Feature", "geometry": {"type": "Polygon", "coordinates": [[[7.7177898845717054, 47.288749387420026], [7.7177898845717054, 47.289050700651686], [7.718503326570354, 47.288749387420026], [7.7177898845717054, 47.288749387420026]]]}, "properties": {"UserData": ""}}
{"type": "Feature", "geometry": {"type": "Polygon", "coordinates": [[[7.909693385457025, 47.349741672390806], [7.911217197673477, 47.349741672390806], [7.911217197673477, 47.3488327788060686], [7.909693385457025, 47.3488327788060686]]]}, "properties": {"UserData": ""}}
{"type": "Feature", "geometry": {"type": "Polygon", "coordinates": [[[9.284270010894174, 46.8562010863139856], [9.284270010894174, 46.8562010863139856]]]}, "properties": {"UserData": ""}}
{"type": "Feature", "geometry": {"type": "Polygon", "coordinates": [[[8.441372066146572, 47.080645565073254], [8.440171467769344, 47.080645565073254]]]}, "properties": {"UserData": ""}}
{"type": "Feature", "geometry": {"type": "Polygon", "coordinates": [[[7.647250126043863, 47.5252539968064], [7.646899513588472, 47.5252539968064]]]}, "properties": {"UserData": ""}}
{"type": "Feature", "geometry": {"type": "Polygon", "coordinates": [[[6.111602129549433, 46.20514700359484], [6.111602129549433, 46.20514700359484]]]}, "properties": {"UserData": ""}}
{"type": "Feature", "geometry": {"type": "Polygon", "coordinates": [[[7.39293838733394, 46.94091657328481], [7.39293838733394, 46.94122906233117], [7.393168834452931, 46.94122906233117], [7.393168834452931, 46.94091657328481]]]}, "properties": {"UserData": ""}}
{"type": "Feature", "geometry": {"type": "Polygon", "coordinates": [[[8.723936420114141, 47.24021587586861], [8.723936420114141, 47.24021587586861], [8.723936420114141, 47.24021587586861]]]}, "properties": {"UserData": ""}}
```

(b) GeoJSON content of one file.

Figure 8.4.: Resulting files when exporting GeoJSON with GeoSpark.

8.2. Compare Results

Figure 8.5 compares certain AOIs of the GeoSpark implementation in Section 8.1 with the AOIs of the PostGIS implementation from Section 5.1.

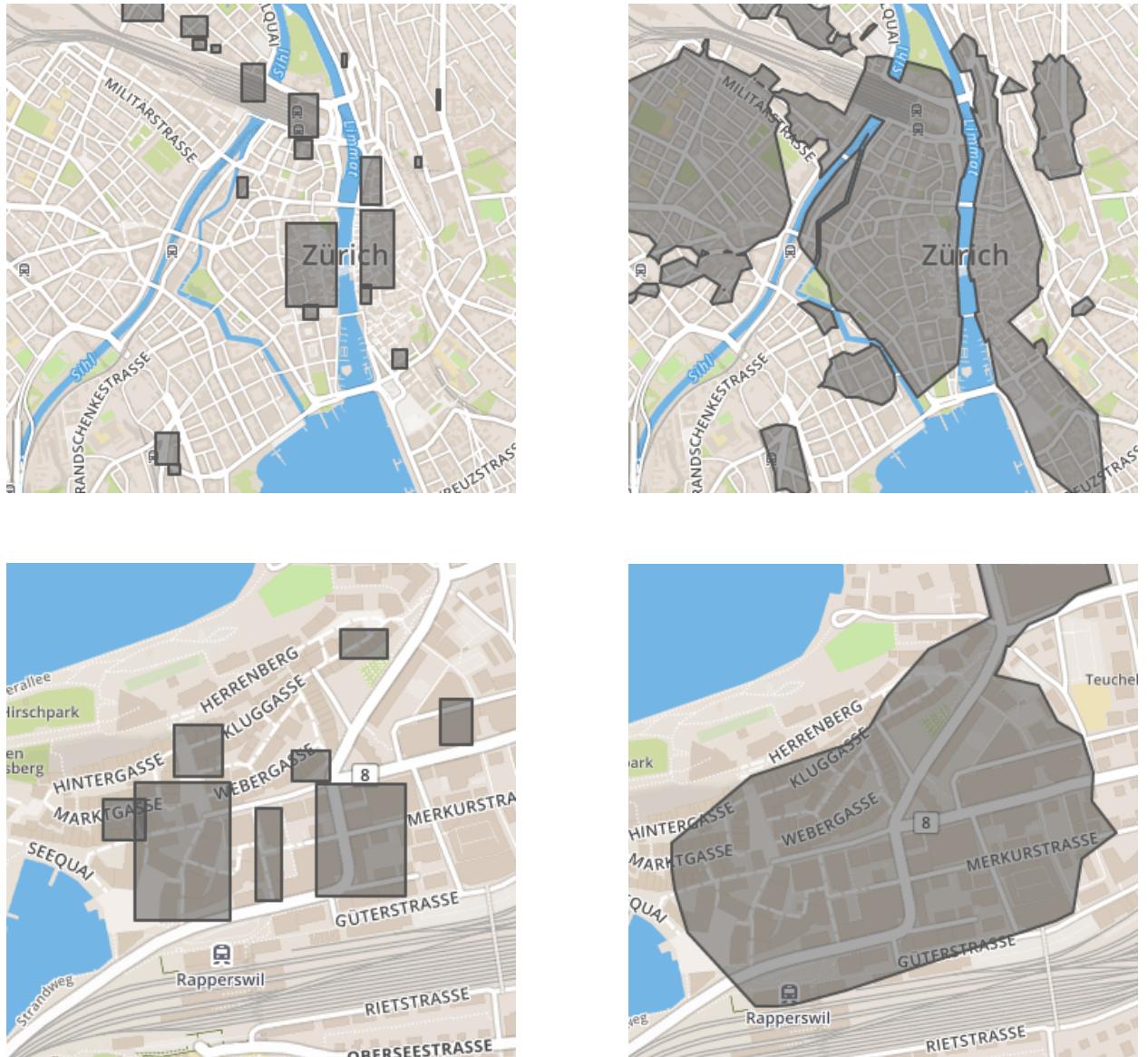


Figure 8.5.: GeoSpark AOIs (left) compared to PostgreSQL AOIs (right).

As one can see, the AOIs from the GeoSpark implementation are not really useful, compared to the AOIs of the PostGIS implementation. Due to multiple missing functions in GeoSpark, it was not possible to implement the same functionality. The AOIs generated with GeoSpark are much smaller, since it was not possible to apply the DBSCAN clustering to polygons, but only to points. This leads to a huge difference in the clusters.

8.3. Benchmarks

In this section the implementation from Section 8.1 is benchmarked.

8.3.1. Hardware

The benchmarks are executed on a cluster with one master and three worker nodes, with 4 CPU cores and 8 GB memory each. It is only a virtual cluster, since it is configured using Docker and all nodes are running on the same server. This is not an ideal implementation, as it does not show the problem of network traffic. On the other hand, each node can be configured and scaled easily by editing the Docker configuration. The server, where the cluster is running on, has 32 CPU cores and 540 GB of memory and is therefore capable of running various setups of nodes. The Docker configuration can be found in the Appendix C.1.

8.3.2. Results

As explained in Section 5.3.2, the benchmarks for the data of Switzerland is processed. The benchmarks are executed on the Parquet files, generated as described in Section 8.1.1. The steps to generate the Parquet files are not benchmarked, since the `osm2pgsql` import of the PostgreSQL + PostGIS implementation was not benchmarked either.

Table 8.1 contains the runtime of the different phases when generating the AOIs for Switzerland. To benchmark the capability for scaling of the Apache Spark implementation, the runtimes with one, two and three worker nodes are listed.

Table 8.1.: Runtimes when generating AOIs for Switzerland with Apache Spark

	1 worker node	2 worker nodes	3 worker nodes
Pre-Clustering	36 s	34 s	33 s
Generating AOIs without network centrality	3 s	3 s	3 s
Export AOIs	2 s	2 s	2 s

Figure 8.6 compares the runtime of the PostgreSQL + PostGIS and Apache Spark + GeoSpark implementation.

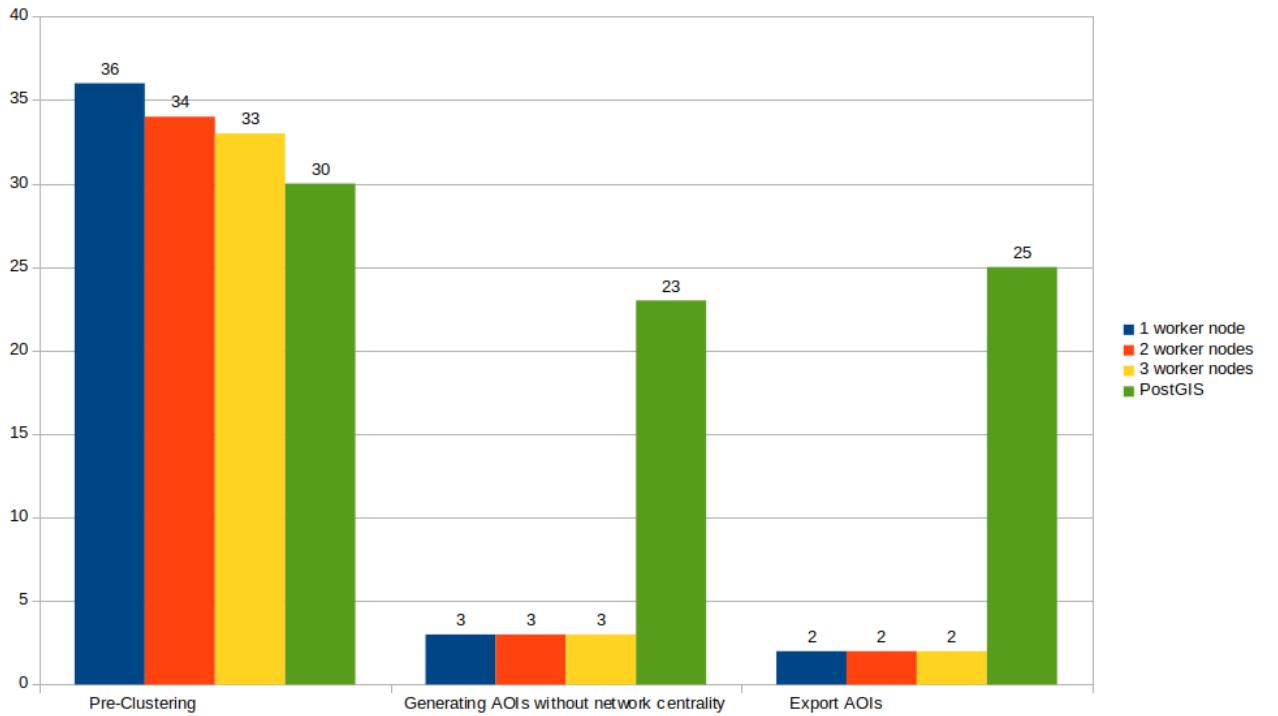


Figure 8.6.: Runtime in seconds of Spark and PostgreSQL implementations.

In the illustration, one can see multiple things. First, generating the AOIs with Spark is much faster than with PostgreSQL. However, this must be put into perspective since due to the missing functionality of GeoSpark, the implementation is much simpler and does not operate on complex polygons but only on rectangles. Therefore, a comparison is not meaningful.

For the runtimes of the pre-clustering phase, one can see that the Spark implementation is slower than the PostgreSQL implementation. This is even more surprising, when taking into account that the Spark implementation is performing on a cluster with three worker nodes, each with 8 GB memory, and 4 CPU cores (total: 24 GB of memory and 16 CPU cores), whereas the PostgreSQL implementation runs on an older working machine with 2 CPU cores and 6 GB of memory. The explanation for this lies in the higher complexity of the implementation due to the missing DBSCAN functionality of GeoSpark. Therefore, the data types must be converted multiple times between different data types while processing, which hinders Apache Spark to distribute the computations properly on all worker nodes.

What additionally can be seen in the pre-clustering phase is that the Apache Spark implementation scales when adding more worker nodes. The speedup is not huge, but as one can see, that Apache Spark is able to distribute these computations. Even though, the reduction in the runtime is grossly proportional to the costs of the additional hardware.

In the export-phase one can see the potential of Apache Spark. Exporting the geometries can be perfectly parallelized to all worker nodes and their cores. But again, one has to take into account the fact that the geometries which are exported are less complex. But even with more complex geometries, Apache Spark would outperform the current PostgreSQL implementation.

9. Use Case Eventcount

9.1. Implementation

This Section describes the implementation of the Eventcount use case with Apache Spark extended with GeoSpark. It is based on the scope of the PostgreSQL and PostGIS implementation. Differences and limitations are documented accordingly.

9.1.1. Data Source

For the implementation of this use case, the data is read from PostgreSQL and converted to Parquet files. Then the events are initially read from the Parquet file to the memory. All following code examples will read the data from a Parquet file.

Listing 9.1 shows the corresponding lines of Scala code, to read the events from the PostgreSQL database and write it to a Parquet file.

```
1 // opts = connection parameter for database
2 def writeEventsParquet() = {
3     sparkSession.read
4         .format("jdbc")
5         .options(opts)
6         .option("dbtable", "events")
7         .load
8         .createOrReplaceTempView("events")
9
10    // convert event location to GeoSpark geometry with ST_GeomFromWKB
11    val events = sparkSession.sql("SELECT ST_GeomFromWKB(location) AS location, "
12        "accuracy, occurred_at, consumer_id, session_id FROM events")
12
13    events.write.parquet("events.parquet")
```

Listing 9.1: Read events from PostgreSQL database and write to Parquet file.

The second data input, viz. the areas, are not provided via the database, but as GeoJSON. Unfortunately, GeoSpark is not able to read a GeoJSON file. It can read the GeoJSON format, but only if each line of the file represents one feature. Therefore, it is necessary to convert the GeoJSON files accordingly and the files can be read as shown in Listing 9.2.

```
1 sparkSession.read
2     .format("csv")
3     .option("delimiter", "\n")
4     .option("header", "false")
5     .load(path)
6     .createOrReplaceTempView("areas_csv")
7
8 sparkSession.sql("SELECT ST_GeomFromWKT(areas._c0) AS area FROM areas")
9     .createOrReplaceTempView("areas_csv")
```

Listing 9.2: Read areas from CSV file where each line contains a GeoJSON feature.

9.1.2. Querying Events

A straightforward implementation with GeoSpark is, to use the provided SparkSQL API extensions. Listing 9.3 shows the corresponding code. The data of the events and areas is read from Parquet files. With both DataFrames a view for SparkSQL is created. And in the end are the events queried using SparkSQL. As one can see, the query is identical to the PostgreSQL + PostGIS query. The functions ST_Intersects, ST_Area and ST_Intersection are provided by GeoSpark¹.

```
1 var eventsDataFrame = sparkSession.read.parquet("events.parquet")
2 eventsDataFrame.cache()
3 eventsDataFrame.createOrReplaceTempView("events")
4
5 var areasDataFrame = sparkSession.read.parquet("areas.parquet")
6 areasDataFrame.createOrReplaceTempView("areas")
7
8 val count = sparkSession.sql(
9 """SELECT * FROM events, areas
10 | WHERE occurred_at >= "2017-04-01 00:00:00"
11 | AND occurred_at <= "2017-04-15 00:00:00"
12 | AND ST_Intersects(ST_Buffer(events.location, events.accuracy), areas.area)
13 | AND ST_Area(ST_Intersection(ST_Buffer(events.location, events.accuracy), ↴
14 | areas.area)) / ST_Area(ST_Buffer(events.location, events.accuracy)) > 0.5
14 """.count()
```

Listing 9.3: Query events with GeoSpark Spark SQL

The implementation with SparkSQL is very elegant. Unfortunately, there are two problems with this. First, the application of the GeoSpark function ST_Area to the output of the function ST_Buffer has a bug and always returns 0.² As a solution, the area of the events must already be calculated on the PostgreSQL database, when converting the events to the Parquet file. This results in a new attribute `area` on the events.³

The second problem is, that the GeoSpark SparkSQL API does not yet support caching for the internally used SpatialRDDs⁴. To be able to use caching, the RDD API must be used directly.

Listing 9.4 shows the code for preparing the events RDD before any query. On line 1 the Parquet file is read as before. On line 3, the DataFrame is converted to a SpatialRDD, with a method provided by GeoSpark. After calling the `analyze` method on the SpatialRDD, one can do a spatial partitioning with the `spatialPartitioning` method. The method takes a parameter for a grid type and the number of partitions. Finally, on line 8, the spatial partitioned RDD can be cached.

```
1 val eventsDataFrame = sparkSession.read.parquet("events.parquet")
2 val spatialEventsRDD = new SpatialRDD[Geometry]
3 spatialEventsRDD.rawSpatialRDD = Adapter.toRdd(eventsDataFrame)
4 spatialEventsRDD.analyze()
5 spatialEventsRDD.spatialPartitioning(GridType.KDBTREE, numberOfPartitions)
6 spatialEventsRDD.spatialPartitionedRDD.cache()
```

Listing 9.4: Prepare events with GeoSparks SpatialRDDs

¹These functions are defined by the Open Geospatial Consortium (OGC) to achieve interoperability between different spatial databases. However, GeoSpark does not yet provide all of these functions.

²This has been reported to the maintainer.

³This step can be equated with the preprocessing of the events in Section 6.3.1

⁴Not to be confused with the caching of the event DataFrame (as done in Listing 9.3 on line 2), which is possible, but does not bring any benefit since GeoSpark uses internally an own version of RDDs (SpatialRDDs).

Now that the events are properly partitioned and cached, the Spark SQL query from Listing 9.3 must be reproduced using the RDD API. The corresponding code is shown in Listing 9.5.

```

1 // join all events and areas which intersects each other (even with less than ↴
2   50%)
3 val intersecting = JoinQuery.SpatialJoinQueryFlat(spatialEventsRDD, ↴
4   spatialAreasRDD, false, true)
5
6 val events = intersecting.rdd.filter(geometryPair => {
7   // read and parse the timestamp from the user data
8   val occurred_at = Timestamp.valueOf(geometryPair._2.getUserData.toString.split(
9     "\t")(0))
10
11   // filter with the timestamp
12   occurred_at.after(from_timestamp) && occurred_at.before(to_timestamp)
13 })
14 // and finally filter for all events which intersects more than 50%
15 .filter(geometryPair => geometryPair._1.intersection(geometryPair._2).getArea / (
16   geometryPair._2.getArea) > 0.5)

```

Listing 9.5: Query events with GeoSpark RDDs

To make a spatial join with the RDD API, one has to use the `JoinQuery.SpatialJoinQueryFlat` method. It takes two `SpatialRDD`s as arguments and returns one `SpatialPairRDD` with pairs of two geometries that intersect. This reveals another oddity of GeoSpark. All additional attributes of one of the input `SpatialRDD`s, for example the `occurred_at` timestamp, are lost after this join. Only the geometry attribute remains. To carry additional data attributes with the geometries, one has to prepare them differently when creating. Listing 9.6 shows the difference for the initial import query. The attributes can be added to the `ST_GeomFromWKB` and are carried as user data.

```

1 // old query
2 val events = sparkSession.sql("SELECT ST_GeomFromWKB(location) AS location, ↴
3   accuracy, occurred_at, consumer_id, session_id FROM events")
4
5 // new query
6 val events = sparkSession.sql("SELECT ST_GeomFromWKB(location, accuracy, ↴
7   occurred_at, consumer_id, session_id) AS location FROM events")

```

Listing 9.6: Carrying attributes of events as user data of the geometry.

9.2. Benchmarks

In this section the implementation from Section 9.1 is benchmarked. For the cluster, the one in Section 8.3.1 is used.

The same benchmarks are applied as in Section 6.2. By default the largest area “Huge amount of small areas” is used for benchmarking.

The time to read all events from the Parquet files into memory, including the spatial partitioning, takes about 4 minutes. This time is not captured with the benchmarks, since in a production use, the events will be loaded into memory initially only once. It takes about 11 GB memory for all events. This value would be smaller if it isn’t necessary to calculate the area on the PostgreSQL database, due to the bug in GeoSpark.

Choosing the amount of partitions is essential when using GeoSpark with Apache Spark. It depends on the amount of data and the amount of worker nodes. Figure 9.1 illustrates the runtimes for different constellations.

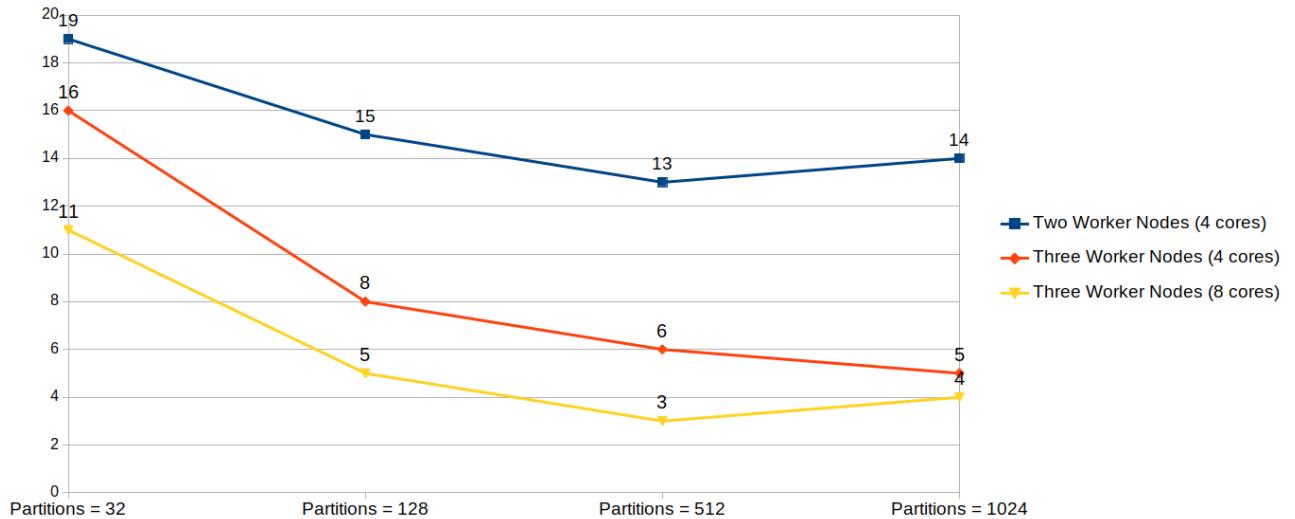


Figure 9.1.: Runtimes in seconds querying benchmark area “Huge amount of small areas” for different number of partitions and worker nodes. Figures in parenthesis are the number of CPU cores per node.

The runtimes show that every cluster with different number of worker nodes or cores benefits from more partitions in the beginning, but the positive effect stagnates when increasing the partitions further. The performance even decreases with too many partitions. This is because when Apache Spark reads from a partition a serialization overhead is necessary.

It can be additionally seen from Figure 9.1, that the runtime benefits from more worker nodes and more CPU cores per worker node. This seems obvious, but it requires Apache Spark with GeoSpark to be able to parallelize. Accordingly, the implementation can make use of the distribution execution. For the further benchmarks, a three worker nodes cluster with 8 CPU cores each and 512 partitions are used.

Figure 9.2 shows the runtimes for the GeoSpark implementation when increasing the amount of weeks. The runtimes almost remain the same, even with more data. This is, because the “expensive” join query is executed in the beginning, and the events are later filtered by date⁵.

⁵“Expensive” is written in quotations, since due to the spatial partitioning GeoSpark can execute it efficiently

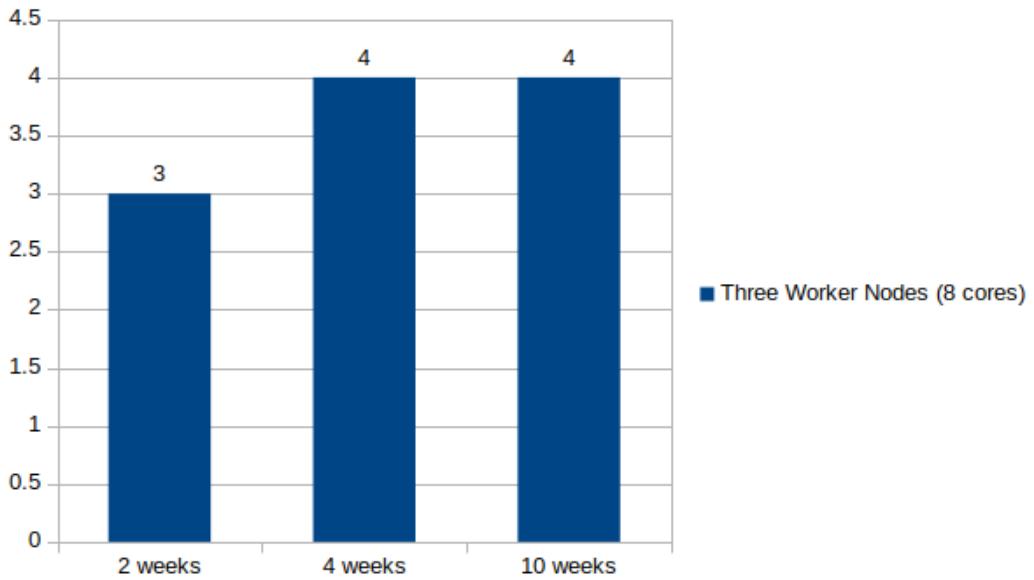


Figure 9.2.: Runtimes in seconds querying benchmark area “Huge amount of small areas” for different number of weeks.

Figure 9.3 shows the runtimes for all benchmark areas for the GeoSpark and the PostGIS implementation. As one can see, GeoSpark outperforms the PostGIS implementation, especially with more data. GeoSpark is clearly better in parallelizing the join query. However, when reading this results, one must keep in mind that the GeoSpark solution is running on much stronger hardware then the PostGIS solution. For example, the PostGIS implementation could be improved too, if it would run purely in memory.

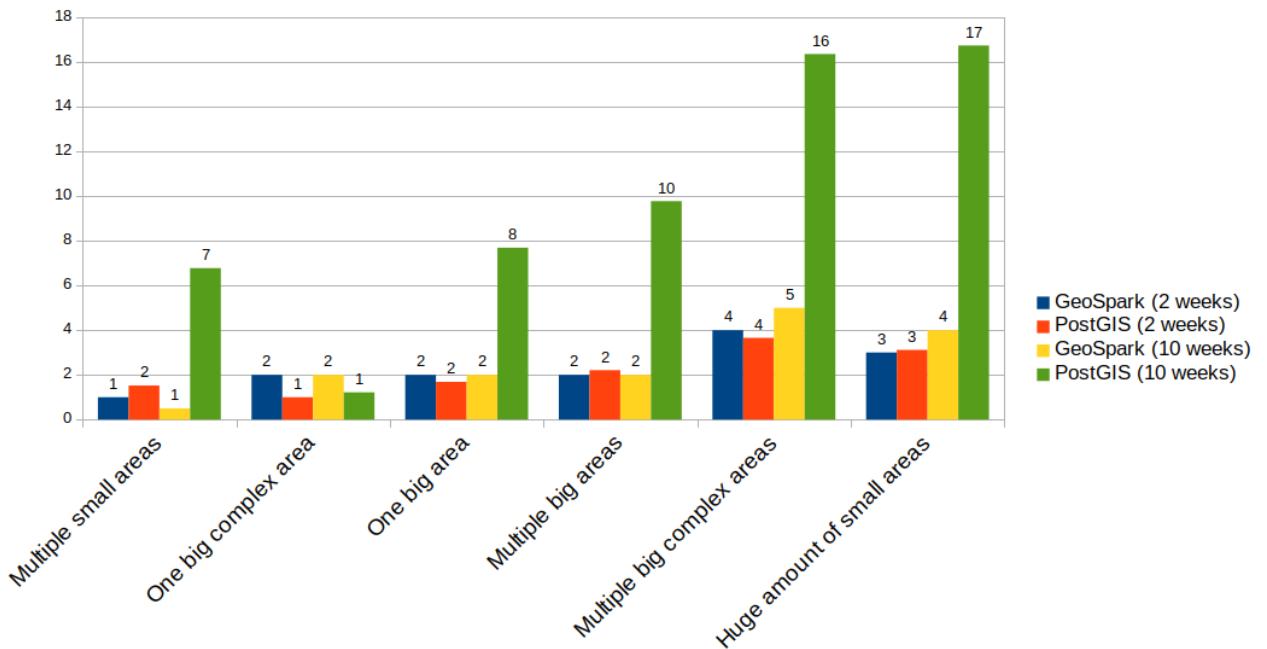


Figure 9.3.: Comparing runtimes in seconds for querying different benchmark areas.

Part IV.

Summary and Conclusion

10. Summary of Results

This chapter summarizes the results of this thesis.

An approach to generate Areas-of-Interest (AOIs) with OpenStreetMap data was implemented with PostgreSQL and PostGIS, with the results presented in Chapter 5. One can either generate the AOIs for whole countries or the full planet, or use a web application to generate the AOIs for specific areas. The web application visualizes and explains each step of the AOI generation, which leads to a high transparency and comprehensibility. The final implementation was benchmarked and bottlenecks identified.

In Chapter 6 a use case of an industrial partner encountering performance issues with PostgreSQL and PostGIS was re-implemented with the same technology stack. After identifying the performance bottlenecks, several approaches were applied to optimize the runtimes. The resulting implementation runs up to 30 times faster and proves to be a satisfying solution for the industrial partner.

In Chapters 7 to 9 a suitable extension for Apache Spark to handle spatial data types was evaluated. Based on the functional scope, GeoSpark was chosen. The PostGIS implementations of both use cases were then re-implemented using Apache Spark and GeoSpark.

In the case of the AOI use case, many limitations of GeoSpark were encountered. Basic operations contained bugs and many necessary operations were missing. Therefore, only a very basic implementation with incomplete results could be realized. Nevertheless, the implementation was benchmarked and the runtimes and generated AOIs compared with the corresponding PostGIS implementation.

The second use case (from the industrial partner) was simpler. Here, GeoSpark was found to be more suitable for its implementation. It was possible to achieve better runtimes than with the PostGIS implementation, especially in case of more volume of data.

11. Conclusion

The bottleneck in generating the AOIs with PostgreSQL and PostGIS was the application of the network centrality. The relevant library osmnx used here did not seem to be efficient enough for this task and need to be improved or replaced. Excluding the network centrality, it is possible to generate the AOIs for the whole planet and make the resulting GeoJSON publicly available. Through the web application it would be possible for an user to understand how the AOIs are generated.

The procedure for creation of AOIs still has a lot of room for improvement. This can be done, for example by fine-tuning the DBSCAN parameters of the local adaption. Besides that, more data sources could be considered while generating the AOIs, as for example Twitter Posts, Instagram Photos or FourSquare reviews. The possibilities are endless.

For the industrial partner, their existing implementation with PostgreSQL and PostGIS was optimized to their satisfaction. The solution with Apache Spark and GeoSpark was reported to be inappropriate for their purpose, as it would require too costly hardware and an extensive stack of new technologies. Since the runtimes with the optimized version are sufficient, they would be able to use it accordingly. There are options to further improve the solution. For example, one could consider running PostgreSQL in memory or making use of Sharding. Additionally, improvements on parallelization of PostgreSQL and PostGIS could also be considered.

GeoSpark appears to be still immature for the use of a complex task like generating AOIs. For simpler tasks it was seen to be able to distribute the calculations and make use of a distributed cluster. Nevertheless, many bugs were encountered and it still seems premature for a productive environment. However, GeoSpark is under active development and the quality and scope of functionality increases continuously. Be it GeoSpark or a different solution like GeoMesa, in the future it hopefully will be possible to make complex queries, as with PostGIS, on a cluster.

Part V.

Appendices

A. Use Case AOI

A.1. Tags for POIs

The selected tags for the relevant POIs as JSON.

```
1   'landuse_tags': [
2     'retail'
3   ],
4   'amenity_tags': [ 'pub', 'bar', 'cafe', 'restaurant', 'pharmacy', 'bank',
5   'fast_food', 'food_court', 'ice_cream', 'library', 'ferry_terminal',
6   'clinic', 'doctors', 'hospital', 'pharmacy', 'veterinary', 'dentist',
7   'arts_centre', 'cinema', 'community_centre', 'casino', 'fountain',
8   'nightclub', 'studio', 'theatre', 'dojo', 'internet_cafe', 'marketplace',
9   'post_office', 'townhall'
10  ],
11  'shop_tags': [ 'mall', 'bakery', 'beverages', 'butcher', 'chocolate',
12  'coffee', 'confectionery', 'deli', 'frozen_food', 'greengrocer',
13  'healthfood', 'ice_cream', 'pasta', 'pastry', 'seafood', 'spices', 'tea',
14  'department_store', 'supermarket', 'bag', 'boutique', 'clothes', 'fashion',
15  'jewelry', 'leather', 'shoes', 'tailor', 'watches', 'chemist', 'cosmetics',
16  'hairdresser', 'medical_supply', 'electrical', 'hardware', 'electronics',
17  'sports', 'swimming_pool', 'collector', 'games', 'music', 'books', 'gift',
18  'stationery', 'ticket', 'laundry', 'pet', 'tobacco', 'toys'
19  ],
20  'leisure_tags': [ 'adult_gaming_centre', 'amusement_arcade', 'beach_resort',
21  'fitness_centre', 'garden', 'ice_rink', 'sports_centre', 'water_park'
22  ]
```

A.2. docker-compose.yml

```
1 version: '3'
2 services:
3   webapp:
4     build: ./webapp
5     env_file: ./webapp/.env
6     environment:
7       - PGHOST=postgres
8       - PGUSER=postgres
9       - PGDATABASE=gis
10    volumes:
11      - ./webapp:/webapp/
12      - ./data/:/data/
13    ports:
14      - "5000:5000"
15    depends_on:
16      - postgres
17
18  notebooks:
19    build: ./notebooks
20    environment:
21      - PGHOST=postgres
22      - PGUSER=postgres
23      - PGDATABASE=gis
24    env_file:
25      - .env
26    volumes:
27      - ./data/:/data/
28      - ./notebooks/notebooks://home/jovyan/
29    ports:
30      - "8888:8888"
31    depends_on:
32      - postgres
33
34  postgres:
35    image: mdillon/postgis:10
36    environment:
37      - POSTGRES_PASSWORD=
38    volumes:
39      - ./postgres/storage/:/var/lib/postgresql/data/
40      - ./data/alter_config.sh:/docker-entrypoint-initdb.d/alter_postgres_config.sh
41    ports:
42      - "54320:5432"
```

A.3. Web Application Readme

```
1 To run the web application locally, the following steps are necessary:  
2  
3 Build the docker image:  
4 docker-compose build webapp  
5  
6 Init the database:  
7 docker-compose run --rm webapp bash import_osm.sh  
8  
9 This will import the file 'data/switzerland.osm.pbf' to the database. To change  
10 the file, edit the import script 'webapp/import_osm.sh'.  
11  
12 Since not all OSM elements are necessary for generating AOIs, one can filter  
13 the 'osm.pbf' file before importing it. Therefore, osmfilter can be used. An  
14 example is provided in the file 'pbf_filter_example.sh'.  
15  
16 Prepare the POIs:  
17 docker-compose run --rm webapp bash setup_pois.sh  
18  
19 Start the webapp container:  
20 docker-compose up -d webapp  
21  
22 Now one can access the web interface at http://localhost:5000.
```

A.4. PostgreSQL Configuration

```
1 # DB Version: 10  
2 # OS Type: linux  
3 # DB Type: web  
4 # Total Memory (RAM): 6 GB  
5 # CPUs num: 2  
6 # Hard drive type: ssd  
7  
8 ALTER SYSTEM SET  
9   max_connections = '200';  
10 ALTER SYSTEM SET  
11   shared_buffers = '1536MB';  
12 ALTER SYSTEM SET  
13   effective_cache_size = '4608MB';  
14 ALTER SYSTEM SET  
15   maintenance_work_mem = '384MB';  
16 ALTER SYSTEM SET  
17   checkpoint_completion_target = '0.7';  
18 ALTER SYSTEM SET  
19   wal_buffers = '16MB';  
20 ALTER SYSTEM SET  
21   default_statistics_target = '100';  
22 ALTER SYSTEM SET  
23   random_page_cost = '1.1';  
24 ALTER SYSTEM SET  
25   effective_io_concurrency = '200';  
26 ALTER SYSTEM SET  
27   work_mem = '7864kB';  
28 ALTER SYSTEM SET  
29   min_wal_size = '1GB';
```

```
30 | ALTER SYSTEM SET
31 |   max_wal_size = '2GB';
32 | ALTER SYSTEM SET
33 |   max_worker_processes = '2';
34 | ALTER SYSTEM SET
35 |   max_parallel_workers_per_gather = '1';
36 | ALTER SYSTEM SET
37 |   max_parallel_workers = '2';
```

B. Use Case Event Count

B.1. Benchmark Results

Table B.1.: Resulting count of events when querying different cluster.

	No Clustering	90%	80%	70%	60%
Multiple small areas	16143	16415	16415	16399	16399
One big complex area	196553	199293	199240	199293	199326
One big area	402222	408002	408041	408089	408079
Multiple big areas	520006	527827	527838	527836	527889
Multiple big complex areas	367323	372586	372533	372516	372553
Huge amount of small areas	127556	129322	129073	129151	128955

Table B.2.: Resulting count of events when querying different cluster.

	50%	40%	30%	20%	10%
Multiple small areas	16397	16381	16391	16389	16411
One big complex area	199241	199246	199249	199222	199326
One big area	408111	408038	408057	408109	407930
Multiple big areas	527838	527871	527840	527924	527978
Multiple big complex areas	372574	372468	372593	372617	372633
Huge amount of small areas	128795	128718	128990	128840	129267

C. Apache Spark

C.1. Spark Cluster

The cluster is setup through the Portainer web interface¹, using the following configuration (irrelevant and private information are skipped):

```
1 version: '3.4'
2 services:
3   master:
4     image: gettyimages/spark:2.3.0-hadoop-2.8
5     volumes:
6       - data:/data/
7     command: bin/spark-class org.apache.spark.deploy.master.Master -h 0.0.0.0
8     networks:
9       - default
10      - web
11     ports:
12       - "4040:4040"
13     expose:
14       - "8080"
15       - "4040"
16       - "7077"
17       - "6060"
18     restart_policy:
19       max_attempts: 3
20     replicas: 1
21     placement:
22       constraints:
23         - node.role == manager
24   logging:
25     driver: "json-file"
26     options:
27       max-size: "256K"
28 worker:
29   image: gettyimages/spark:2.3.0-hadoop-2.8
30   volumes:
31     - data:/data/
32   environment:
33     SPARK_WORKER_INSTANCES: 1
34     SPARK_WORKER_CORES: 4
35     SPARK_WORKER_MEMORY: 8g
36   command: bin/spark-class org.apache.spark.deploy.worker.Worker spark://master:7077
37   networks:
38     - default
39   deploy:
40     restart_policy:
41       max_attempts: 3
42       replicas: 3
```

¹<https://github.com/portainer/portainer>

```
43     placement:
44       constraints:
45         - node.role == manager
46   logging:
47     driver: "json-file"
48     options:
49       max-size: "256K"
```

Bibliography

- [1] AVUXI. *Taming Open Data – the process behind Top Areas*. URL: <https://www.avuxi.com/blog/news/the-process-behind-top-areas> (visited on 05/01/2018).
- [2] AVUXI. *TopPlace*. URL: <http://www.avuxi.com/topplace> (visited on 05/01/2018).
- [3] Citusdata. *Sharding a multi-tenant app with Postgres*. URL: <https://www.citusdata.com/blog/2016/08/10/sharding-for-a-multi-tenant-app-with-postgres/> (visited on 06/08/2018).
- [4] Developerlife.com. *Android Location Providers – gps, network, passive – Tutorial – developer-life.com*. URL: <https://developerlife.com/2010/10/20/gps/> (visited on 05/22/2018).
- [5] GeoMesa. *locationtech/geomesa: GeoMesa is a suite of tools for working with big geo-spatial data in a distributed fashion*. URL: <https://github.com/locationtech/geomesa> (visited on 05/28/2018).
- [6] GeoPandas. *GeoPandas 0.3.0 — GeoPandas 0.3.0 documentation*. URL: <http://geopandas.org/> (visited on 05/28/2018).
- [7] GeoSpark. *GeoSpark*. URL: <http://datasystemslab.github.io/GeoSpark/> (visited on 05/28/2018).
- [8] GeoSpark Wiki. *Aggregate function - GeoSpark*. URL: <http://datasystemslab.github.io/GeoSpark/api/sql/GeoSparkSQL-AggregateFunction/> (visited on 06/21/2018).
- [9] GeoTrellis. *locationtech/geotrellis: GeoTrellis is a geographic data processing engine for high performance applications*. URL: <https://github.com/locationtech/geotrellis> (visited on 05/28/2018).
- [10] GeoWave. *GeoWave User Guide*. URL: <http://locationtech.github.io/geowave/userguide.html%7B%5C%7Dwhat-is-geowave> (visited on 05/28/2018).
- [11] Google. *Location | Android Developers*. URL: [https://developer.android.com/reference/android/location/Location.html%7B%5C%7DgetAccuracy\(\)](https://developer.android.com/reference/android/location/Location.html%7B%5C%7DgetAccuracy()) (visited on 04/22/2018).
- [12] Google Blog. *Discover the action around you with the updated Google Maps*. 2016. URL: <https://blog.google/products/maps/discover-action-around-you-with-updated/> (visited on 05/01/2018).
- [13] GPS.gov. *GPS.gov: GPS Accuracy*. URL: <https://www.gps.gov/systems/gps/performance/accuracy/%7B%5C%7Dhow-accurate> (visited on 04/21/2018).
- [14] Stefan Hagedorn and T U Ilmenau. “Big Spatial Data Processing Frameworks : Feature and Performance Evaluation”. In: *Proceedings of the 20th International Conference on Extending Database Technology (EDBT)* (2017), pp. 490–493. DOI: [10.5441/002/edbt.2017.52](https://doi.org/10.5441/002/edbt.2017.52).
- [15] Kang and Stefan. *Areas-of-Interest for OpenStreetMap (AOI for OSM) – White Paper Draft v.01 - HackMD*. URL: <https://md.coredump.ch/s/HkT2Up6Hz%7B%5C%7D> (visited on 05/01/2018).
- [16] LocationSpark. *merlintang/SpatialSpark: LocationSpark: A Distributed In-Memory Data Management System for Big Spatial Data*. URL: <https://github.com/merlintang/SpatialSpark> (visited on 05/28/2018).

- [17] Magellan. *harsha2010/magellan: Geo Spatial Data Analytics on Spark*. URL: <https://github.com/harsha2010/magellan> (visited on 05/28/2018).
- [18] Helmut Neukirchen. "Survey and Performance Evaluation of DBSCAN Spatial Clustering Implementations for Big Data and High-Performance Computing Paradigms". In: (2016).
- [19] OpenStreetMap. *Points of interest - OpenStreetMap Wiki*. URL: <https://wiki.openstreetmap.org/wiki/Points%7B%5C%7Dof%7B%5C%7Dinterest> (visited on 05/01/2018).
- [20] OpenStreetMap Wiki. *Osm2pgsql - OpenStreetMap Wiki*. URL: <https://wiki.openstreetmap.org/wiki/Osm2pgsql> (visited on 05/04/2018).
- [21] Ahmed Oussous et al. "Big Data technologies: A survey". In: *Journal of King Saud University - Computer and Information Sciences* (2017). ISSN: 22131248. DOI: [10.1016/j.jksuci.2017.06.001](https://dx.doi.org/10.1016/j.jksuci.2017.06.001). URL: <http://dx.doi.org/10.1016/j.jksuci.2017.06.001>.
- [22] Paul Ramsey. *Patching Plain PostgreSQL for Parallel PostGIS Plans — CARTO Blog*. URL: <https://carto.com/blog/inside/postgres-parallel/> (visited on 04/24/2018).
- [23] PostGIS Wiki. *ST_Simplify*. URL: <https://postgis.net/docs/ST%7B%5C%7DSimplify.html> (visited on 04/25/2018).
- [24] PostgreSQL Wiki. *PostgreSQL: Documentation: 9.6: btree_gist*. URL: <https://www.postgresql.org/docs/9.6/static/btree-gist.html> (visited on 04/26/2018).
- [25] PostgreSQL Wiki. *PostgreSQL: Documentation: 9.6: Parallel Query*. URL: <https://www.postgresql.org/docs/9.6/static/parallel-query.html> (visited on 04/24/2018).
- [26] Ramsey. *Parallel PostGIS · Paul Ramsey*. URL: <http://blog.cleverelephant.ca/2016/03/parallel-postgis.html> (visited on 06/08/2018).
- [27] Ramsey. *Parallel PostGIS II · Paul Ramsey*. URL: <http://blog.cleverelephant.ca/2017/10/parallel-postgis-2.html> (visited on 06/08/2018).
- [28] Erich Schubert et al. "DBSCAN Revisited, Revisited". In: *ACM Transactions on Database Systems* 42.3 (July 2017), pp. 1–21. ISSN: 03625915. DOI: [10.1145/3068335](https://doi.acm.org/citation.cfm?doid=3129336.3068335). URL: <http://dl.acm.org/citation.cfm?doid=3129336.3068335>.
- [29] Spark_dbscan. *How It Works · alitouka/spark_dbscan Wiki*. URL: <https://github.com/alitouka/spark%7B%5C%7D dbscan/wiki/How-It-Works> (visited on 06/19/2018).
- [30] STARK. *dbis-ilm/stark: A framework for Spatio-Temporal Data Analytics on Spark*. URL: <https://github.com/dbis-ilm/stark> (visited on 05/28/2018).

D. List of Figures

2.1.	AOIs in Google Maps (GMaps)	12
2.2.	GMaps POIs of Rapperswil with zoom level 17.	13
2.3.	Heat maps of TopPlace by AVUXI	14
2.4.	POIs of Rapperswil, related to food and drink, visualized with openpoimap.org.	16
2.5.	Illustration of the DBSCAN clustering algorithm. Source: [28]	17
2.6.	DBSCAN clustering on multiple polygons. Three clusters 0, 1 and 2 are created. Red polygons are not density reachable and therefore not part of a cluster.	17

2.7. Example for different hulls around certain POIs in Rapperswil.	18
2.8. Street network and the corresponding nodes of Rapperswil.	19
2.9. Results of closeness centrality algorithm. Nodes are colored by their relative centrality, from lowest in dark purple to highest in bright yellow.	19
2.10. Results of closeness centrality algorithm on line graph. Streets are colored by their relative centrality, from lowest in dark purple to highest in bright yellow.	20
2.11. Example AOI of Rapperswil overlaid with network centrality. Zones where the AOI could be extended are marked red.	20
2.12. An example AOI which includes rivers. Areas which should be excluded are marked red.	21
3.1. The area of an event is represented as a circle with the even location as the center and its accuracy as the radius of the circle.	22
3.2. Different cases of an event area intersecting a polygon.	22
3.3. Distribution of horizontal_accuracy attribute (special cases and values with more than 100000 records).	24
4.1. Logo of Apache Spark.	25
4.2. Spark architecture.	26
5.1. Resulting clusters of pre-clustering POIs.	31
5.2. Comparing resulting clusters for different values of ϵ in the DBSCAN algorithm	32
5.3. Comparing resulting clusters for Zürich with different values of $minPts$ in DBSCAN	33
5.4. Comparing resulting clusters for Stäfa with different values of $minPts$ in DBSCAN	34
5.5. Street network per POI cluster hull of Rapperswil.	36
5.6. Nodes of street network graph.	37
5.7. 10% of the most central streets, cut with a 50 meter threshold outside the hull.	37
5.8. Hulls extended with network centrality. The extensions are marked red.	38
5.9. Exclude water from AOIs.	39
5.10. Sanitize AOIs	40
5.11. Interface of the web application.	42
5.12. Explained steps of AOI generation of web application	43
5.13. Generated AOIs of web application	44
5.14. Overlay of all generated AOIs.	44
6.1. Each feature of the input GeoJSON results in one row in the areas table.	48
6.2. Runtimes in seconds for different benchmark areas and increasing amount of data.	52
6.3. Output of PostgreSQL command EXPLAIN ANALYZE when querying benchmark "Huge amount of small areas" for one week of data.	53
6.4. Example for use of PostGIS function ST_Buffer	55
6.5. Adjusting the quad_segs parameter of ST_Buffer	55
6.6. Runtimes in seconds for different values of quad_segs parameter and benchmark areas.	56
6.7. The blue event area results when running ST_Simplify with a tolerance of 100 on the red event area.	57
6.8. Different cases of an event area intersecting a polygon.	58
6.9. Two areas of overlapping areas.	59
6.10. One large and one small event area moved by one meter.	60
6.11. Example for snapping event locations to grid.	60
6.12. Location and area of events (green, blue) and location and area of resulting cluster (red)	61
6.13. Resulting amount of clusters for different values of p	63
6.14. Resulting count of events for benchmark area "Huge amount of small areas"	64

6.15. Resulting count of events for benchmark area "Huge amount of small areas" with clusters created per week instead of day.	64
6.16. Resulting count of events for benchmark area "Huge amount of small areas" where clusters created per month instead of day.	65
6.17. Benchmarks when splitting query in multiple sub queries and executing them parallel.	67
6.18. Benchmark results for different benchmark areas and increasing amount of data.	68
8.1. Illustrations of DBSCAN algorithm on Spark. Source: [29]	77
8.2. Resulting clusters of DBSCAN clustering of Switzerland.	78
8.3. AOIs generated with GeoSpark.	81
8.4. Resulting files when exporting GeoJSON with GeoSpark.	82
8.5. GeoSpark AOIs (left) compared to PostgreSQL AOIs (right).	83
8.6. Runtime in seconds of Spark and PostgreSQL implementations.	85
9.1. Runtimes in seconds querying benchmark area "Huge amount of small areas" for different number of partitions and worker nodes. Figures in parenthesis are the number of CPU cores per node.	89
9.2. Runtimes in seconds querying benchmark area "Huge amount of small areas" for different number of weeks.	90
9.3. Comparing runtimes in seconds for querying different benchmark areas.	90