University of Twente

Master Thesis

# Partitioning of Spatial Data in Publish-Subscribe Messaging Systems

Author:
Rik van Outersterp

Supervisors:
dr. ir. M.J. van Sinderen
dr. R.B.N. Aly
dr. ir. N.C. Besseling

August 26, 2016

**Abstract**

A global trend to gather and query large amounts of data manifests itself more and more in our daily lives. This data includes that of spatial data, which increases the need for big data systems that can process this type of data efficiently regardless of preferences of end-users or the amount of data. The growing trend of big spatial data increases the need for expansion of knowledge about how such systems can be scaled up to satisfy both the increasing supply and demand of such data.

Publish-subscribe messaging systems are capable of handling large amounts of data. Publishers send data to intermediary brokers, from which subscribers can retrieve the data. On these brokers data is stored in topics, which can be used to retrieve data efficiently based on a subscribers preferences. Topics can consist of multiple partitions in which the data is stored, which may improve load balancing and fault tolerance if multiple servers are used. A partitioning method defines in which partition data is stored on a broker. Partitioning of spatial data is a challenge, since spatial data is more complex than non-spatial data.

In this thesis we investigate how spatial partitioning methods can help to scale up publish-subscribe messaging systems for processing big spatial data. For this we present a case study consisting of Apache Kafka, an open-source publish-subscribe messaging system. In the case study the Kafka system processes road traffic data.

In our research we discuss existing spatial partitioning methods. We show how one of these, Voronoi, can improve processing of spatial data in a publish-subscribe system when compared to a system without spatial partitioning. In addition, we propose a new spatial partitioning method based on Geohash to overcome several drawbacks of Voronoi.

Our experiments show improvements in the areas of load balancing, transferred messages, and the amount of relevant messages when using spatial partitioning methods. We show that in general Geohash was able to achieve better results than Voronoi during our experiments. Only when the amount of partitions is more than half of what Geohash can define, Voronoi shows slightly better results.

# Contents

# Chapter 1

# Introduction

## 1.1 Spatial data

Spatial data refers to objects in multidimensional space. These data objects can vary from a single point to a multidimensional polygon. When the space in which data is located is geographical, spatial data is often referred to as geospatial data. Examples of geospatial data are tweet messages of Twitter with the sender's location, satellite images of the earth, or GPS data of cars on the highway.

Spatial data has some unique properties in comparison to non-spatial data [1], including the following.

**Complex structure.** Spatial data has a complex structure in which a data object can vary from a single point to a complex polygon in multiple dimensions, which usually cannot be stored efficiently in a traditional relational database.

**Dynamic.** Spatial data is often dynamic, because insertions and deletions are interleaved with updates of existing data, though this may not be specified within the insertion or deletion. Therefore structures that handle spatial data should support this behaviour without declining performance.

**Open operators.** Many spatial operators are not closed. This means that the outcome of operations cannot be predicted beforehand, e.g. an intersection of two polygons can return any number of single points, dangling edges, or disjoint polygons.

Due to these properties, traditional data processing systems cannot process spatial data with similar efficiency as non-spatial data [1]. This problem holds especially for systems that process a large amount of data, i.e. big data systems based on techniques as Hadoop or MapReduce. These systems do not support these properties by default and for them the only way to process spatial data with more efficiency is to treat it as non-spatial data or to use handling functions for spatial data around these non-spatial systems. An example of a handling function is Geohash, a technique that hashes the latitude and longitude coordinates into a character string, such that it can be used as an indexing technique within a key-value environment.

However, both approaches treat spatial data as non-spatial data or use handling functions that do not take advantage of the properties of spatial data and therefore are not able to achieve the same efficiency that can be accomplished when processing non-spatial data. This is a growing problem, since a global trend to gather and query large amounts of data, including spatial data, manifests itself more and more in our daily lives. Examples of such uses are projection of spatial data on a visual map or calculation of an estimated time of arrival (ETA) for road traffic. These use cases are described in more detail in Section 2.3.

## 1.2 Big spatial data systems

To overcome the problem of less efficiency, extensions such as Hadoop-GIS and SpatialHadoop were developed. These systems are specialized in processing big spatial data and are capable of processing spatial data with similar efficiency as their traditional counterparts process non-spatial

data. They achieve this by focusing on how to process spatial data in an efficient way by improving partitioning and indexing techniques.

According to [2], a design for systems to handle big spatial data efficiently should consist of four main system components: language, indexing, query processing, and visualization. These components are described in more detail below and an example of such a system is visualized in Figure 1.1.

**Language.** A high level language allows non-technical users to interact with the system.

**Indexing.** Although it is referred to as the indexing component, it is not only responsible for indexing data, but also for partitioning data. Because partitioning and indexing are closely related, they are often used interchangeable in the literature. We define partitioning data as how data can be divided such that it is stored efficiently, and indexing data as organizing data such that it can be retrieved efficiently. For example, in a distributed environment a two-level structure can be used, where data is first partitioned across the machines, and then indexed locally on each machine.

**Query processing.** The query processing component consists of the spatial operations that are executed on the spatial data using the constructed spatial indexing and/or partitioning.

**Visualization.** Data output can be visualized by generating an image, such as a map or a graph.

Although [2] proposes that big spatial data systems should contain of all four components, we argue that effective scaling up of spatial data systems to big spatial data systems is dependent on only two of these components, namely the query processing and indexing components. Both components are responsible for processing data as efficiently as possible, while the other two components (language and visualization) are not necessarily required: systems can work without a high level language (although this might require trained users for interacting with the system) and query results do not always have to be visualized in an image, e.g. the previously mentioned use case of calculating an estimated time of arrival (ETA). Missing one or more of these components can also be seen in Hadoop-GIS. According to [3], Hadoop-GIS consists of only three of these components, namely the language, query processing, and indexing component.

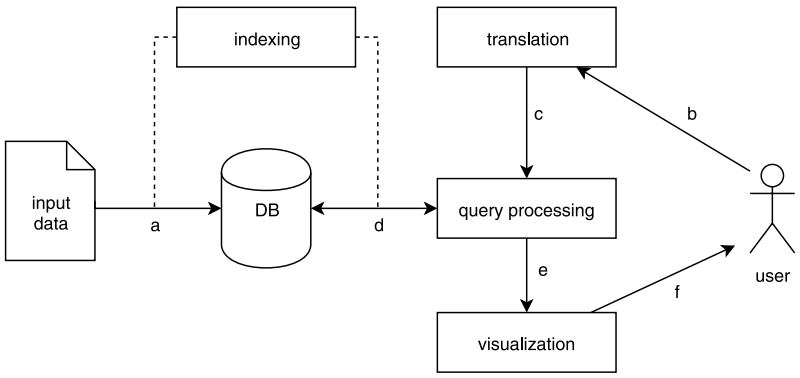



Figure 1.1: (a) The indexing component is responsible for storing the input data into the database. (b) A user sends a request in a high level language. (c) The request in high level language is translated to a query. (d) Querying the database using the indexing component. (e) Query results are sent to be visualized. (f) A visualization of the results is returned to the user.

## 1.3 Publish-subscribe messaging systems

The growing availability of (spatial) data everyday tends to require a filter to select only data that a consumer is interested in, based on their preferences. Such a filter can be accomplished by using a publish-subscribe messaging system. In this type of system, data from multiple senders, called publishers, can be sent to multiple receivers, called subscribers, through an intermediary message broker, as shown in Figure 1.2. Data is stored in a topic, which can divide it again over partitions for load balancing and fault tolerance if multiple servers are used.

Publish-subscribe messaging systems use asynchronous communication as messages are temporarily stored on an intermediary message broker and not sent directly from publishers to subscribers. Because of this, a publish-subscribe messaging system is said to be a loosely coupled environment. A major advantage of a loosely coupled environment is its scalability: publishers and subscribers are completely unaware of each other's existence and as they do not have to consider each other by keeping track of each other's status, location, and preferences, the amount of publishers and subscribers can be higher than in peer-to-peer like systems. However, when publish-subscribe systems are scaled up towards large-scale sizes, their scalability will get limited by the message broker and its ability to handle large amounts of data.

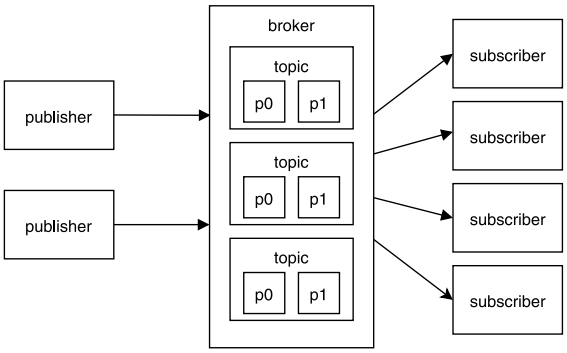



Figure 1.2: Schematic view of publish-subscribe messaging systems.

## 1.4 Research problem & questions

From the previous sections the following can be concluded. Spatial data in itself has properties that makes it more difficult to process by traditional systems that do not take advantage of these properties, thus traditional systems process spatial data with less efficiency than non-spatial data. This really becomes a problem since there is a growing trend of gathering and processing big spatial data. For this data to be processed efficiently, big spatial data systems are developed, which aim to process spatial data as efficiently as traditional big data systems process traditional data. We argued that only two of the proposed main components for big spatial data systems are required in all cases for scaling up spatial data systems, namely the query processing and indexing components. An existing type of system that can handle big data and is scalable for growing data needs, is that of publish-subscribe messaging systems. However, these systems are originally not designed for big spatial data and it is therefore unclear how publish-subscribe systems can be scaled up in order to be able to process the growing amounts of big spatial data.

In this research we investigate if, and if so how, the upscaling of publish-subscribe messaging systems can be improved for big spatial data processing. We do this by applying a required component of big spatial data systems in a publish-subscribe messaging system. For this research we chose to apply only one of the most important components of big spatial data systems as a first contribution to the research field of spatial data in publish-subscribe messaging systems. We decided to limit our focus to the indexing component to investigate its effectiveness through a spatial partitioning method, because an optimized partitioning method improves the query performance and therefore the interaction with the system. Thus, we want to investigate if spatial partitioning can be applied to publish-subscribe messaging systems, and if so, which methods can be applied and how these perform in comparison to a system without spatial partitioning applied.

We formalize the research problem into the following research question as the objective for this research.

*How can spatial partitioning methods help to scale up publish-subscribe messaging systems for processing big spatial data?*

In order to provide an answer to this question, we first answer the following subquestions.

*How can spatial partitioning be applied in publish-subscribe messaging systems?*

Publish-subscribe messaging systems and how spatial partitioning can be applied in them, are discussed in more detail in Chapter 3.

*Which spatial partitioning methods exist that can be applied in publish-subscribe messaging systems?*
In Chapter 3 we present several spatial partitioning methods and discuss their applicability.

*What is a suitable use case to evaluate applicable methods and how do these methods perform in this use case in comparison to a system without spatial partitioning?*
In Chapter 2 we present a case study with multiple use cases for spatial data. One of these use cases is used in the experiments as described in Chapter 6. Models of the evaluated methods (Voronoi and Geohash) are described in Chapters 4 and 5.

Our research is intended to provide insight in how spatial partitioning methods can help to improve the scalability of publish-subscribe messaging systems in order to process big spatial data. As the amount of available spatial data increases, it increases the need for expansion of knowledge about how such systems can be scaled up to satisfy both the increasing supply and demand of this data. We provide this insight by investigating the applicability of an existing method and how it performs against a proposed spatial partitioning method called Geohash.

## 1.5 Outline

The outline of the thesis is as follows.

Chapter 2 presents a case study for this research that consists of a publish-subscribe messaging system. From this case study we define a suitable use case for the remainder of this research.

Chapter 3 discusses publish-subscribe messaging systems in more detail and how spatial partitioning methods can be applied in these systems. In this chapter we also present several partitioning methods.

Chapter 4 defines one of the presented spatial partitioning methods (Voronoi) in a model that is used in the experiments of this research.

Chapter 5 proposes a new spatial partitioning method based on Geohash. This method is defined in a model that is used in the experiments of this research.

Chapter 6 discusses the experiments of this research and found results.

Chapter 7 concludes this thesis by answering the research questions and discusses future work.

# Chapter 2

# Case Study

This chapter presents the case study for this research. For this case study the Connect 2 system of Simacan is used. Simacan, a subsidiary of OVSoftware, is an IT company in the Netherlands that is specialized in gathering geospatial road traffic data and making it accessible and useful for its clients. In order to accomplish these goals, Simacan has developed several tools for its clients to access the gathered data in a user-friendly interface, e.g. easily readable traffic updates or visual representation of traffic on a map. For these products Simacan gathers roughly three types of geospatial traffic data:

- **Traffic data per segment or measuring point.** These data consists of traffic velocity, intensity, or incidents in predefined road segments or at road measuring points.
- **Floating car data.** These data consists of a single vehicle's velocity, position, and direction.
- **Other traffic related data.** This includes weather information and charge points for electric or hybrid vehicles.

Currently, the focus of Simacan is the segment or measuring point traffic data, which is gathered from multiple public and private sources, including TomTom, the Dutch National Data Warehouse for Traffic Information (NDW), and the Dutch Ministry of Infrastructure and Environment (Rijkswaterstaat).

Simacan aims to advance the functionality of its current system, Connect 1. Concrete examples of advanced functionality are storing data from a larger covered geographical area and from a larger time period (increasing the amount of historic data). Mainly due to the use of a relational database, the current system lacks the scalability capacity to achieve these aims: calculations made on the data would require to much resources and time to complete. Therefore they are limited to the scope of the Netherlands and a history of up to four hours back. They also aim at (a more advanced) support of routes, which can be defined by a client. In this scenario only relevant data regarding a defined route is pushed (in real-time) to the client, instead of, for example, publishing a list of all traffic related events in the Netherlands.

In their aim to achieve these goals a new system, Connect 2, is in development. In the next sections this system is described in detail.

## 2.1 Background

In this section we discuss several techniques that are used in Simacan's Connect 2.

### 2.1.1 Apache Kafka

Apache Kafka, hereafter referred to as Kafka, is a publish-subscribe messaging system rethought as a distributed commit log [4]. It was originally developed by LinkedIn, but donated to Apache, although most developers of Kafka are still from LinkedIn.

The general concept of Kafka can be described as follows. Publishers send messages to one or more intermediary message brokers and subscribers can retrieve messages that are stored on this brokers. In Kafka publishers are called producers and subscribers are called consumers. Similar

to other publish-subscribe messaging systems, messages are categorized in topics. Each topic can consist of multiple partitions in which the messages are stored, see Figure 2.1. Compared to other messaging systems, Kafka consists of so-called *dumb* brokers: they only store the messages and do not care who reads them (they also do not push the messages to consumers) or even who publish them for that matter.

Kafka runs on Apache ZooKeeper, an open-source server that includes various standard services in order to let Kafka be as simple as possible [5]. ZooKeeper can be used to configure Kafka and its components. It also keeps track of the state of all components within Kafka, e.g. which broker a replication leader is or which consumers are still alive and communicating with the cluster.

We will now describe each component of Kafka in more detail.

**Producers.** Producers publish messages to a topic. During this process, producers are responsible for choosing the partition on which a message needs to be be stored. By default this is determined by a hash-mod of a key, but producers can use a custom partitioning method. When such a key is not given, the message will be sent to a random partition. Because of the producers being responsible for choosing a partition for each message, they are responsible for load balancing messages to the various brokers, although this can only be achieved by a partitioning method that is designed for load balancing. It does not matter if a producer cannot see the corresponding broker initially: all brokers can be discovered by a producer from any single broker.

Producers have three levels of acknowledgement when sending messages to a broker. Requesting more acknowledgements improves the fault tolerance, but it also requires more communication and resources as messages are not dismissed after sending until enough acknowledges are received.

- **No acknowledgement (0).** This means that a producer cannot guarantee that messages are actually received by a broker.
- **Acknowledgement from n replicas (1..n).** In this case a producer can guarantee that a message will be received by 1..n brokers.
- **Acknowledgement from all replicas (-1).** This means that a producer can guarantee that a message is stored on all brokers it should be stored on.

**Consumers.** Consumers subscribe to topics and process published messages from those topics that they pull from the brokers. Since the brokers are *dumb*, consumers have to maintain track of their state (offset) themselves, i.e. consumers have to know which messages they did read and which not. Consumers can be grouped into consumer groups. For consumers within a consumer group applies that a consumer group reads a message only once. Thus consumers divide the load by retrieving messages for the whole group.

**Broker.** A broker is a server in a Kafka cluster and stores messages that are published by producers. A machine can consist of multiple brokers. A Kafka broker is not the same as a usual message broker, because it does not send a message to the designated receiver: it remains *dumb* by receiving a message, store it in the topic and partition it is told to, send a copy of the message when requested by a consumer, and delete the message after a certain time or to make space for new messages. A broker is responsible for persisting the messages for a configurable amount of time, but it is dependent on the disk capacity. Messages are stored in an append-only log. When producers send a message, a sequential write is performed by the broker for the new message. Analogue to this, a sequential read is performed to determine all messages that have to be send to a consumer from a certain offset. This is actually one of the key features of Kafka, as sequential reads may be faster than random reads, especially when stored in the page cache. Note that messages are not deleted when read by consumers: messages are stored by brokers for a configurable amount of time, and this only depends if there is enough disk space available.

**Partitioning.** Messages are categorized in different topics. Topics can be defined manually in the Kafka cluster using ZooKeeper or automatically when data is published to a non-existing topic. For each topic multiple partitions can exist. Partitions are limited to the available space of the broker, since a partition cannot be divided over multiple brokers, let alone machines. A partition is basically a commit log consisting of published messages, sorted by the time they were received by the broker. Each message within a partition has a unique identifier, called the *offset*, which is

used by consumers to maintain track of which messages they have read. The messages within a topic are not totally ordered, but messages within a partition are order on the time of arrival.

A disadvantage of Kafka is that the number of partitions cannot be easily changed, since they are determined when creating a topic. In addition, existing messages cannot be moved to a new partition where they may belong in a new partitioning method, leaving two options. The first option is that publishers send those messages again to the new partition, leaving it up to the consumer to handle any duplicates. This however raises not only the problem of how to handle duplicates, but also how many messages should be send again. Another option is to leave it completely to the consumers to use both partitioning methods for a while to retrieve messages from the old partition. Again, this raises the problem how long this situation is required before unnecessary messages are retrieved, let alone how these should be handled.

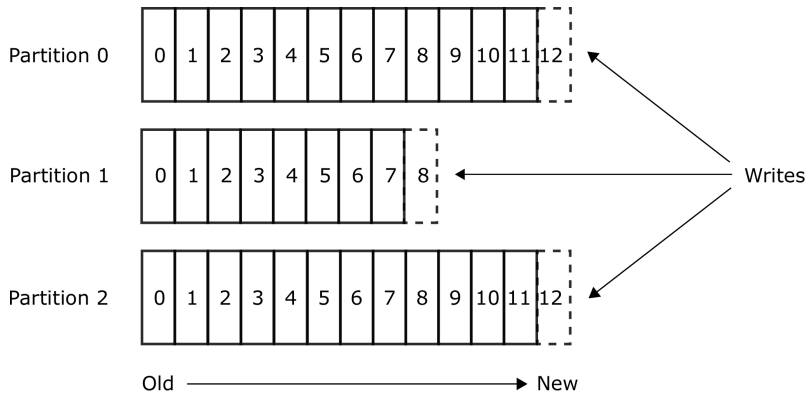



Figure 2.1: Anatomy of a topic.

Partitioning of the topics can have two major advantages:

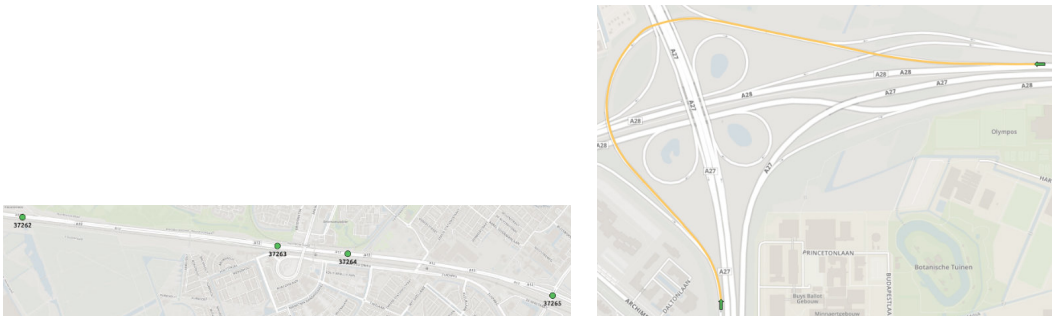
- **Load balancing.** Using an equal partition size and distribution of the partitions over the brokers, load balancing can be optimized, especially when storing data. In this case, each broker has to store the same amount of data, which costs the same amount of disk writes, I/O transactions, RAM, and disk space.

- **Predictability.** Using an equal partition size and distribution of the partitions over the brokers, predictability in several areas can be improved. First and foremost is the predictability of the required storage per partition on a broker. Having equal sized partitions on brokers increases the ability to determine how long data can be stored on a broker. This can be useful when not only the latest data is requested, but also older (historical) data. Considering historical data it can increase storage efficiency, since no larger partition can exist that limits the time frame of historical data.

  Secondly, the prediction of estimated data transfer improves. When a partition is accessed, all messages from a certain offset on that partition are sent. Any additional filtering on the messages is left to be handled by the subscriber. When the partitions have similar size, a subscriber can predict the amount of bandwidth and temporarily storage that is needed for the messages.

**Replication.** Partitions can be replicated on other brokers for fault tolerance. In that case a partition has one broker acting as the leader and zero or more brokers as followers, called replicas. The leader handles all read/write requests, while all replicas passively follow the leader's actions. When a leader fails, one of the replicas will become leader through a selection process handled by ZooKeeper. It should be clear that for optimal fault tolerance the replicas exist on different physical machines.

Replication is not only an advantage for partitioning as partitions can be replicated on other brokers for fault tolerance. It also shows the drawback of having many partitions. Since replication requires data transfers within a cluster, the more partitions there are the more resources are used to synchronize data, and thus less for the transfer of messages from a producer or to a consumer.




(a) (b)

Figure 2.2: Examples of location referencing by TMC (a) and OpenLR (b) [6].

### 2.1.2 Location referencing

Location referencing is the process of describing a location [6]. The description of a location can vary from a single street name to a point given in 3-dimensional coordinates that is as accurate as possible. These examples already show that methods and precision can widely vary. Regarding geospatial traffic data GPS coordinates seems the way to go at first glance, but when a road segment must be described it shows some drawbacks. A single GPS coordinate on a crossing of roads does not give any information about which road it refers to, let alone the direction. In order to overcome these drawbacks, several methods are developed. In this section two of the most well-known methods are introduced: Traffic Message Channel and OpenLR.

#### Traffic Message Channel

Traffic Message Channel (TMC) is a widely used location reference method for traffic data [6]. It describes a road, traffic jam, or something else traffic related by giving a start and an end point. These points can be looked up in a TMC table, resulting in the exact location that is referenced.

Despite the simple nature of TMC, it has one major drawback. It is not possible to describe roads that are not in the TMC table, which limits the coverage to only those roads that are in the TMC table.

An example of TMC can be found in Figure 2.2a.

#### OpenLR

OpenLR is a location reference method for traffic data originally developed by TomTom before made available as an open source project [7]. The major advantage of OpenLR over TMC is that it does not need a table with indexed roads. Similar to TMC it describes a location using start and end points. However, OpenLR uses latitude and longitude for these points. Optionally it describes via-points, characteristics of the road between the points, and the length of the road between the points. Therefore it can describe every road in existence, including those not covered by TMC, which makes the coverage practically 100 percent.

The major advantage of OpenLR is also its major drawback. Due to its flexibility a location reference in OpenLR needs more bits than one in TMC, and translation from and to a record in OpenLR costs more computational power. In order to reduce the needed resources for handling OpenLR messages in our research, we limit the necessary data for determining the location of a message to the first location reference point. This is based on the fact that OpenLR messages describe road parts between road connections. Thus, if one would describe a route in OpenLR, they use a road segment completely from the beginning or not at all: it is not possible to start halfway.

**Example.** Figure 2.2b shows a traffic jam that can be described in OpenLR as follows. It starts at latitude and longitude coordinates (52.093, 5.174) to the west. It ends at latitude and longitude coordinates (52.087, 5.162) from the north. The entire jam is on a sliproad of a motorway and is 1950 meters long.

### 2.1.3 Web Map Service (WMS)

Web Map Service (WMS) is a protocol for creating a map with georeferenced data from a geographic information system (GIS) [8]. A WMS request consists of WMS parameters and and image parameters. WMS parameters include, for example, the WMS version, requested layers (roads, cities, rivers, etc.), and coordinates. Image parameters describe properties for the returned image, such as format (e.g. PNG, GIF or JPEG), width, and height. An example of a WMS request is shown in Listing 2.3. As can be obeserved, a WMS request is executed on top of HTTP. The $(x_{min}, y_{min})$ and $(x_{max}, y_{max})$ coordinates of the requested area's bounding box are included in the request. We refer to this bounding box as the WMS window that covers the requested area. In Listing 2.3 the coordinates are given in EPSG:900913, which is different to the more familiar WGS-84 latitude and longitude system. The supported coordinate systems may differ per WMS server.

Currently handling a WMS request is the only implemented method of the Connect 2 system.

```
GET /wms
    ?SERVICE=WMS
    &REQUEST=GetMap
    &VERSION=1.3.0
    &LAYERS=
    &STYLES=
    &FORMAT=image/png
    &TRANSPARENT=true
    &HEIGHT=256
    &WIDTH=256
    &REFRESHINTERVAL=60000
    &ANTICACHE=0.06480319829190706-1464250087472-472
    &DATAURL=http://example.com/lookup
    &URL=http://example.com/wms
    &LAYERCONTAINER=[object Object]
    &VISIBLEINLOCATIONSUMMARY=true
    &CRS=EPSG:900913
    &BBOX=313086.06785608194,6887893.4928338,469629.1017841229,7044436.52676184
HTTP/1.1
```

Listing 2.1: Example of WMS request.

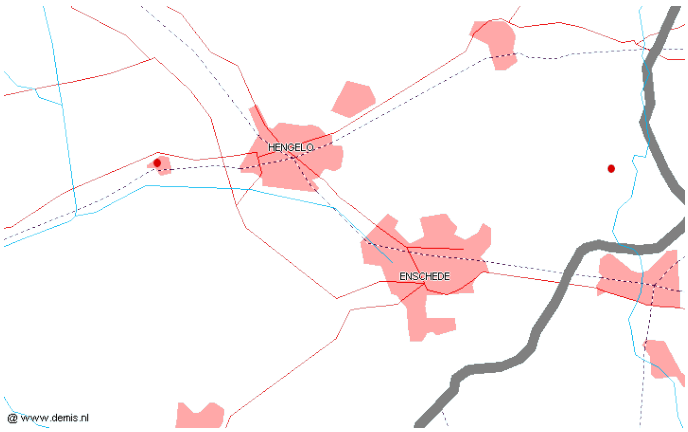



Figure 2.3: WMS response example.

## 2.2 Connect 2

Simacan is currently developing a new system, called Connect 2, that processes received road traffic data. The system is built around a Kafka system. Connect 2 is shown in Figures 2.4 and 2.5, splitting illustrating the architecture at the Kafka cluster.

### 2.2.1 From source to Kafka

The process of data from a source to the Kafka cluster in shown in Figure 2.4.

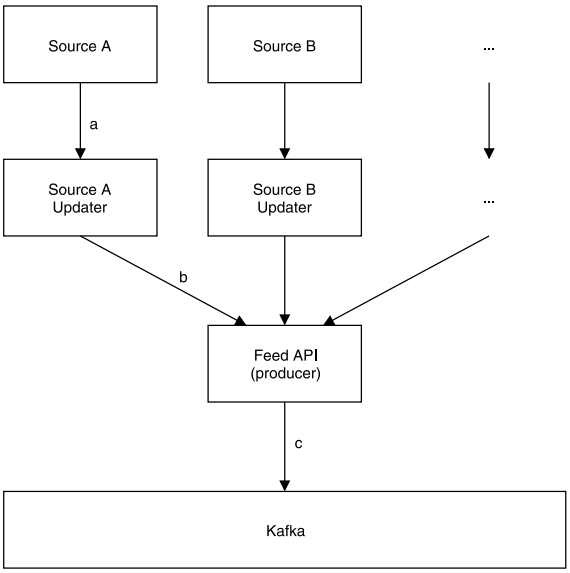



Figure 2.4: Schematic overview of Connect 2, from source to Kafka.

Data from a source is sent to the Updater service corresponding with the source (a). Each source has its own road segment set that could change from day to day. The size and characteristics of these data sets differ per source, e.g. for the Netherlands TomTom has about 200.000 segments, while Rijkswaterstaat has less than 25.000 measuring points. For each road segment, that can have a length between a few tens of meters and a few kilometers, multiple properties are monitored, such as velocity, incidents, or intensity of the segment's traffic. In general these properties are updated and published every minute, although this does not necessarily mean that Simacan retrieves a source's complete data set every minute. Fortunately several sources have their own optimizations when it comes to sending updates and as a result the worst case scenario of retrieving all data sets in their entirety is practically never approached. For example, TomTom only sends the current velocity of a segment if the current velocity differs from a predefined default velocity when no special circumstances are applied, which is defined as the freeflow velocity.

The Updater service receives data from the source, performs some error handling and translates the received data into Simacan's own standard format. This format is a JSON message, of which an example is shown in Listing 2.2. As shown, the complete message consists of the following fields.

- *id* The identifier of the message. Currently the encoded binary 64 string of the OpenLR data acts as the identifier.
- *location* The location reference in OpenLR encoded in a binary 64 string.
- *feed* The source of which the message originated.
- *pub_time_src* The time at which the message was published by the source.
- *received_time* The time at which the message was received by the updater service.
- *message* The message related to the location reference, including data of what the speed at the location is, the location's freeflow, the travel time over that segment, the quality of the provided travel time and speed as confidence, and if the road is blocked or not.

The JSON message (hereafter reffered to as message) is sent to the Feed API (b). This service is a producer in Kafka terms. It decides to which topic and partition a message should be sent (c). Since currently only one topic per source exists, the value of *feed* is used to determine the topic. Currently the message is sent to a single partition.

```
{
    "id":[100,"CwKUWCR5jRt8H/ubBbobEA==",],
    "location":[20000,"CwKUWCR5jRt8H/ubBbobEA==",],
    "feed":[50,"tomtom-hdflow-dev",],
    "pub_time_src":1456407690000,
    "received_time":1456407734036,
    "message":[650,
    {
        "speed":[10000,76],
        "freeflow":[10000,76],
        "traveltime":[30001,87],
        "confidence":[40000,67],
        "roadBlocked":false
    }]
}
```


Listing 2.2: Example of Simacan's JSON message

### 2.2.2 From Kafka to client

The process of data from the Kafka cluster to a client in shown in Figure 2.5.

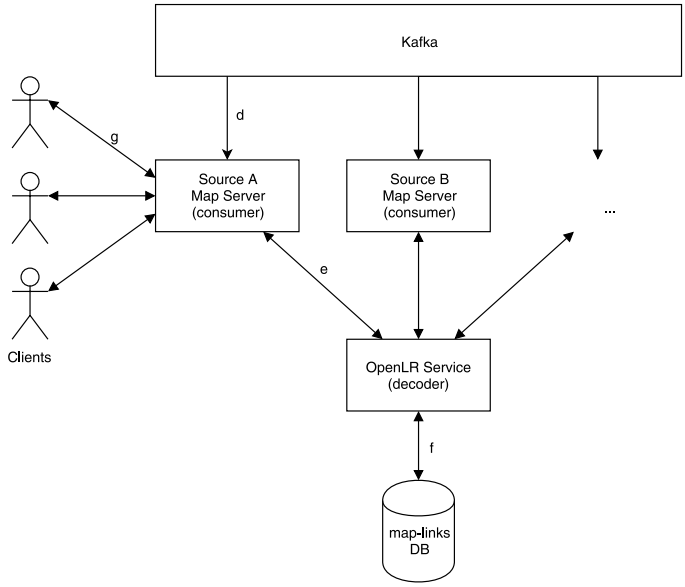



Figure 2.5: Schematic overview of Connect 2, from Kafka to client.

Data is continuously read from the Kafka brokers by a Map Server, which acts as a consumer in Kafka terms (d). The data that is received is immediately sent to the OpenLR Service (e). This service decodes the OpenLR data using the internal map, that is defined by map-links. The decoded data is visualized by the Map Server and kept in memory for two minutes. A client can sent a WMS request to the Map Server to retrieve the data. Since this data is the newest data available and generated not more than a couple of minutes before, Simican refers to this data as real-time data.

## 2.3 Use cases

Simacan has multiple use cases in which data currently is made available to its customers. As we shall see, these use cases can be divided into area or route based.

**Map visualization.** In this use case traffic data, such as travel times or traffic accidents, is visualized on a map. For this, a Web Map Service (WMS) request is used to define the area of which the data is required. This area is defined by a square window that depends on the zoom level and location a user has on a map. As described in 2.1.3, geographical coordinates from the

lower left corner and upper right are used to define this window. Currently only real-time data is visualized on a map, but also data from an earlier time (frame) can be visualized if the necessary methods for this are implemented on the consumer side.

**Estimated Time of Arrival.** When a route is defined, an estimated time of arrival (ETA) can be calculated. The difficulty in determining an ETA lies in the fact that the current situation on a part of the route may no longer apply by the time one reaches it. Therefore a combination of real-time data (data describing the current situation) and bought-in profile data (data that describes the usual traffic situation at a certain time based on historical data) is used when calculating the ETA.

**Time-distance diagrams.** In a time-distance diagram the traffic situation of a route is shown over a certain time. For example, on a route from A to B the average velocity of traffic is determined for every 1 kilometer for every 5 minutes. Each grid tile is then colored following a color scheme. The resulting image can be used by traffic experts to define the evolution of traffic jams or other traffic situations (Figure 2.6).

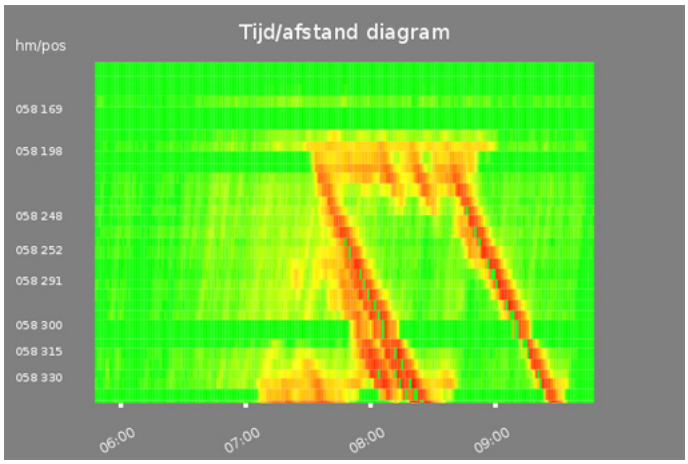



Figure 2.6: Example of a time-distance diagram that indicates two traffic jams [9].

**Events in area and/or on route.** Incoming events can be pushed to a client based on its route or area. These events can vary from traffic jam alerts to speed cams. Usually these messages come from different sources. Such event alerts are only useful when used real-time.

### 2.3.1 Conclusions

The use cases above show that there are multiple applications in which the collected traffic data is used. Each use case requests data from an area, a route, or a combination if a route is in multiple areas or multiple routes are in a single area. A possible future use case, in which data about a route and its surroundings is queried, also fits in this division as a combination of route and area based use case.

Although each use case also has a different time frame to request data from, this will not be considered in the remainder of this research. Historical or real-time are relative terms: all data is from the past and it is independent from the spatial partitioning from how far back the data needs to be retrieved.

Based on the available data that Simacan has provided and the current state of Connect 2 in which only WMS support is implemented, we focus in our experiments, as described in Chapter 6, on the area based use case of map visualization. The advantage of this use case is that it is not relevant to Simacan, but also to other map visualization services, in contrast to route based use cases that also require route and/or traffic data. For map visualization WMS requests are used to retrieve data that can be displayed on a map. Therefore our experiments in Chapter 6 will consist of an experiment on WMS requests.

# Chapter 3

# Spatial Partitioning in Publish-Subscribe Messaging Systems

In this chapter we discuss publish-subscribe messaging systems and what research has been done with regards to the application of spatial partitioning in such systems. Thereafter we present several existing spatial partitioning techniques. In these sections we provide an answer to how spatial partitioning can be applied in publish-subscribe messaging systems and which applicable methods already exist. The answer to these questions will be summarized at the end of this chapter.

## 3.1 Publish-subscribe messaging systems

When using multiple applications in a software environment, it may be necessary that applications have to communicate with each other. However, it is not practical for each application to have specified how to communicate with every other application. If that would be the case, many applications communicating with each other would end up in a so-called spaghetti architecture.

As a solution asynchronous communication consists of a loosely coupled environment in which applications send messages to an intermediary instead of directly to each other. One such environment is a publish-subscribe messaging system [10], in which applications that create information publish messages and applications that are interested in that type of information subscribe to it, hence the name publish-subscribe. These types of applications are referred to as publishers and subscribers respectively. Messages are published to an intermediary message broker, which forwards the messages to subscribers, based on their subscriptions.

Two important types of publish-subscribe messaging systems are topic-based (also referred to as channel-based) and content-based. In topic-based publish-subscribe, messages are published to topics. Subscribers have subscriptions to one or more topics, receiving messages published to topics to which they subscribe. In content-based publish-subscribe, subscribers only receive messages if they apply to properties as defined by the subscriber. Because messages have to be queried against each subscriber's properties, content-based costs more processing and resource usage than topic-based [11]. A hybrid of the two types is also possible; in such a system a subscriber only receives messages from topics it subscribes to *and* apply to properties as defined by the subscriber.

Publish-subscribe systems have two major advantages. As already mentioned, publish-subscribe is a loosely coupled environment. This means that publishers and subscribers are not aware of each other's existence and can function without each other's existence, unlike a tightly coupled environment such as a client-server architecture.

The second major advantage is that of scalability. Because the only communication publishers and subscribers have is with the intermediary broker, many publishers and subscribers can operate in the same environment. Communication can therefore be done in parallel and without updates about other nodes in the system, in contrast to systems in which a server needs to send updates about all clients to each client.

## 3.2 Spatial Publish Subscribe (SPS)

In [11] and [12] publish-subscribe for virtual environments (VEs) is discussed. In these environments multiple nodes move within a virtual space while sending and/or receiving updates about the space around them, i.e. nodes can be subscribers, publishers, or both at the same time. For example, nodes are users in an online game who send updates about their actions and receive updates about actions of other users within their area of interest. [11] refers to these operations as the *event-process-update* cycle: events are received by an intermediary that processes them and sends their results to nodes in the VE, whereafter new events are received and the cycle continues. This cycle can be executed in two ways: by spatial multicast or by spatial query. In spatial multicast messages are sent to nodes that are subscribed to an area within the VE. A node subscribes to an area of the VE when its area of interest overlaps with the area. In spatial multicast the VE is completely divided in multiple areas and therefore spatial multicast can be considered as the topic-based method. In spatial query nodes define properties which they query at times, e.g. the location of other nodes within its area of interest. Since these nodes only receive messages that apply to defined properties, this can be considered as the content-based method. A hybrid of the two approaches is possible as well, as presented by [13]. In this paper a model is presented for both spatial multicast (the spatial event model) and spatial query (the spatial subscription model). The spatial event model consists of the three most important aspects of a spatial event: who, when, and where. The spatial subscription model is used by subscribers to express their interest in spatial events, defined by i.a. a spatial predicate (within or distance). The hybrid between spatial multicast and spatial query is presented as a notification model that consists of a combination of the two other models.

Both spatial multicast and spatial query have their disadvantages. Implementation of spatial multicast faces the difficulty of finding the right area shape and size to divide the VE in. Furthermore, spatial multicast sends messages to all nodes which have their area of interest overlapping with an area, regardless of the message actually being in their area of interest. Nodes will therefore need to spend time and resources filtering these irrelevant messages.

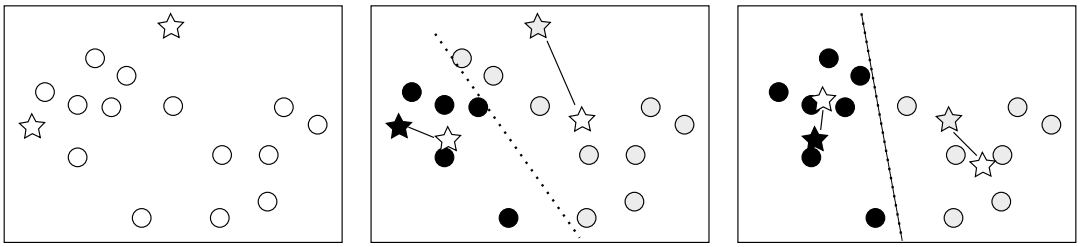
The major disadvantage of spatial query is that of its limited scalability. For $n$ nodes that are required to answer a query, querying may take $O(\log n)$ time [11], which limits the number of supportable nodes heavily. Furthermore, it may result in new nodes or updates not being queried or received fast enough.

In [11] and [12] it is argued that spatial multicast and spatial query do not satisfy basic requirements for VEs on their own, but they do represent needed aspects of a complete system. Spatial Publish Subscribe (SPS) is presented as such a system that tackles the limitations of spatial query and spatial multicast, but can als support both methods. SPS provides nodes the ability to have both a subscription and a publication area, which is considered a subscription or publication point when the area's size is 0. The intermediary message broker is referred to as an interest matcher, whose responsibilities are to record publication and subscription requests and to match published messages with subscriptions, sending the messages to the interested subscribers. Because of taking the area of interest into account in subscriptions, SPS is able to be more flexible and precise than spatial multicast. As for spatial query, a node in SPS does not need to query for updates as long as its subscription does not change.

## 3.3 Spatial partitioning methods

Although publish-subscribe systems such as SPS are designed for high scalability, supporting spatial publish-subscribe on a large-scale requires load distribution among the nodes. Since these nodes exist in a spatial environment, load distribution can be accomplished through partitioning of the spatial environment and its data, i.e. spatial partitioning. In SPS spatial partitioning is applied to the spatial environment in which the publishers and subscribers were located. However, it may not always be the case that publishers and subscribers have a spatial location themselves, and it may be that the data they send is spatial.

We therefore investigate how spatial data that is processed by a publish-subscribe system can be partitioned. For this we present several spatial partitioning methods in this section. Similar to SPS we focus on partitioning methods that are able to divide a space completely without overlapping sections.



(a) (b) (c)

Figure 3.1: A space with two random added points as starting point of 2-means (a) and the first two repositions of these points (b and c). [14]

### 3.3.1 k-means

A partitioning method that is often referred to is k-means [14]. The idea of k-means is that data points within a multi-dimensional space are associated with the closest of the $k$ newly added points. This association is accomplished as follows. In a space consisting of a number of points, $k$ points are randomly added (Figure 3.1a). For every point in the space the nearest of these $k$ points in determined (Figure 3.1b). Then these $k$ points are recalculated based on the points that are closest to them: the $k$ points are placed in a spot where the distance between them and their associated points is the smallest (Figure 3.1b). Afterwards, for each point again the nearest of the $k$ points is determined, which are calculated once more (Figure 3.1c). This process is repeated until the $k$ points do not acquire a new position to be placed at or after a predefined number of iterations.

For the best possible outcome k-means is executed multiple times, each time with different starting locations of the $k$ points. This requirement is therefore one of the major drawbacks of k-means: it does not guarantee that the optimal result that can be found using this method is actually found in practice, since it requires infinite executions to cover all possible starting locations for $k$ random added points. Another disadvantage of k-means is that one cannot decide the location of the $k$ random added points.

### 3.3.2 kd-tree

One of the most known partitioning methods is that of kd-tree [15]. The name of this method refers to two main properties of this method: the result of executing the kd-tree algorithm can be drawn as a tree and the kd-tree algorithm works similar regardless of the amount of dimensions $k$. Although this thesis focuses only on spatial partitioning in two dimensions, it should be noted that despite kd-tree being able to work in higher dimensions, it only seems to work efficiently in the lower dimensions [16].

The idea of kd-tree is to split a space (node) that consists of data points into two subspaces (child nodes) in order to build a balancing tree in which every leaf node consists of a single data point. The general algorithm for this construction of a kd-tree for $n$ points in a $k$-dimensional space (Figure 3.2a) works as follows, iterating over the $k$ dimensions. In every dimension the median of the points in a node is determined and the node is split at the median (Figure 3.2b). Then for every child node the new median in the next dimension is determined and the node is split again (Figure 3.2c). This process is repeated until every node contains only a single point. Splitting at the median ensures that the resulting tree remains balanced.

We see that despite using both $k$ in their name, the $k$ in kd-tree refers to the number of dimensions, while the $k$ in k-means refers to the number of newly added points. Another difference between the two algorithms is that kd-tree guarantees that the optimal result that can be found using this method is actually found. However, in order to guarantee this, kd-tree does need to be executed for all possible sequences of the dimensions. For example, in two dimensions this sequence can be either starting with division over the x-axis followed by division over the y-axis (as depicted in Figure 3.2) or starting with division over the y-axis followed by the division over the x-axis.

### 3.3.3 Voronoi

In Section 3.2 Spatial Publish Subscribe for virtual environments (VEs) was discussed. In SPS Voronoi Self-Organizing partitioning (VSO) is used as a spatial partitioning method for the nodes

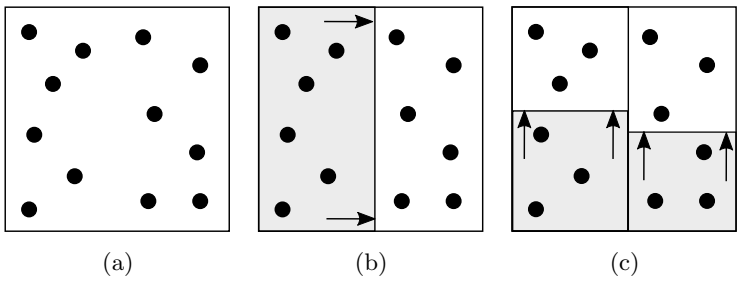



Figure 3.2: A two dimensional space (a) with the first two iterations of kd-tree (b and c). [16]

in VEs [11] and [12]. Due to the nodes being moveable in a VE, VSO is required to continuously adjust the cells of a Voronoi diagram in which the space of a VE is divided and in which the nodes move around.

A Voronoi diagram, shown in Figure 3.3, is the result of Voronoi, a spatial partitioning method that divides space according to the *nearest neighbour-rule*: each point, called a site, is associated with the region that is closer to it than to all other points in the space [17]. This means that a Voronoi diagram requires a pre-defined set of sites, in which each site has their own region without any other sites. The regions are separated by an edge that is exactly on the equal Euclidean distance between two sites. Region on the boundary of the diagram may have an arbitrary infinite size if the divided space has no boundaries by itself. A vertex defines the location where three or more edges meet, and thus the Euclidean distances are equal between the corresponding three or more sites. Now it can be observed that a vertex is the center of a circle that touches at least three sites, but does not enclose any site: otherwise a vertex would be located elsewhere. In summary, a Voronoi diagram with $n$ sites has $n$ regions that are separated by at most $\binom{n}{2}$ edges, although the actual number of edges would be much lower when the number of sites increases [17]. When the number of sites is high, most of them would have multiple regions between them that hide their separating edge. The number of vertices is at most in the same order of vertices, but can also be none if all sites are located in a straight line.

Although we focus on two dimensional Voronoi diagrams in this research, it should be noted that, similar to k-means and kd-tree, Voronoi diagrams can be applied to multi-dimensional space, which would result in multi-dimensional regions.

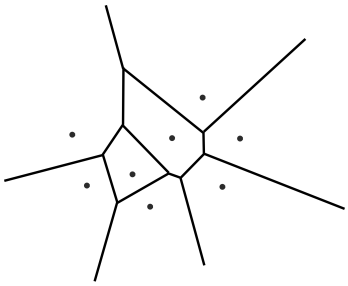


Figure 3.3: Example of a Voronoi diagram with 8 sites [17].

**Applications.** Voronoi diagrams are researched not only in the field of computer science, but also in the fields of applied natural sciences such as biology an astrophysics, and in mathematics. [17] argue that there are three main reasons for the popularity of Voronoi research. First, Voronoi can be used as a model for several natural processes, such as cell architecture in the field of biology. Second, Voronoi diagrams are used for solving a wide variety of computational problems, such as 3D computer animations Finally, for mathematics Voronoi diagram can be used to solve various geometric problems [18]. These problems include those of nearest neighbours and minimum spanning trees, which are often encountered in routing problems.

## 3.4 Conclusions

In Section 3.1 we discussed publish-subscribe messaging systems. The two important advantages of publish-subscribe are the loosely coupled architecture and, partly because of that, scalability of these type of systems. Important to note is that in general publish-subscribe is either topic-based or content-based.

We then presented a publish-subscribe mechanism for spatial virtual environments called Spatial Publish Subscribe (SPS) in Section 3.2. In SPS nodes can be both publisher and subscriber which communicate with an interest matcher. However, being able to support SPS on large-scale, load distribution is necessary. This can be achieved by dividing the nodes over multiple interest matchers. Since this partitioning takes place over nodes in a space, this is referred to as spatial partitioning.

Before we presented several spatial partitioning methods in Section 3.3, we argued that the way that spatial partitioning is used in SPS may not be applicable to other publish-subscribe systems, as it is not always the case that subscribers and publishers have a spatial location: it may also be in the data itself.

We presented several methods that can divide a space completely without overlapping such that data is not duplicated. This matches with a topic-based publish-subscribe system, in which messages are in only one topic. The methods we presented were k-means, kd-tree, and Voronoi. Based on our findings we decided to investigate the performance of applying Voronoi in a publish-subscribe messaging system. In contrast to k-means, Voronoi does guarantee that the most optimal result that it can find for a given set of points is returned. The main advantage of Voronoi compared to kd-tree is that Voronoi is widely used for solving geometric problems, including that of nearest neighbours and minimum spanning trees. These problems are often encountered in routing problems and thus are of great interest in the use cases of Simacan that cover road traffic. In Chapter 4 we present a model of Voronoi for its application in a publish-subscribe messaging model.

# Chapter 4

# Voronoi Model

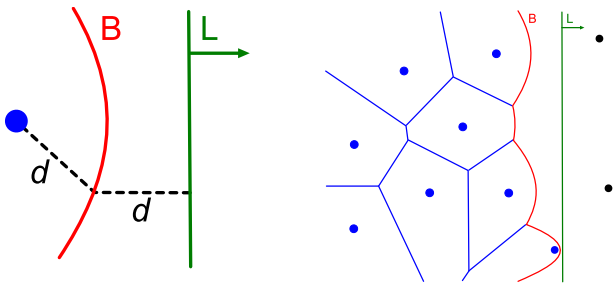
In Section 3.3.3 we presented Voronoi as a spatial partitioning method. In this chapter we describe how a Voronoi diagram can be constructed and how Voronoi can be applied as a spatial partitioning method in a publish-subscribe messaging system.

## 4.1 Construction of a Voronoi diagram

In this section we present three algorithms that are used for the construction of Voronoi diagrams. Recall from Section 3.3.3 that the construction of a Voronoi diagram requires a given set of points that are referred to as Voronoi sites.

### 4.1.1 Fortune's algorithm

Fortune's algorithm is a sweep line algorithm in which a straight sweep line $L$ sweeps over the Voronoi sites $S$ in one direction [19]. When a site $s \in S$ is passed by $L$, a beach line $B$ defines the line on which every point is equidistant to both L and the passed site, as shown in Figure 4.1a. Because $L$ is a straight line, $B$ results in a parabolic curve. Where beach lines of two sites meet, a Voronoi edge is created as the Euclidean distance between each site and this meeting point is equal (Figure 4.1b). Note that any sites that are not yet passed by $L$ do not affect points that are passed by $L$.



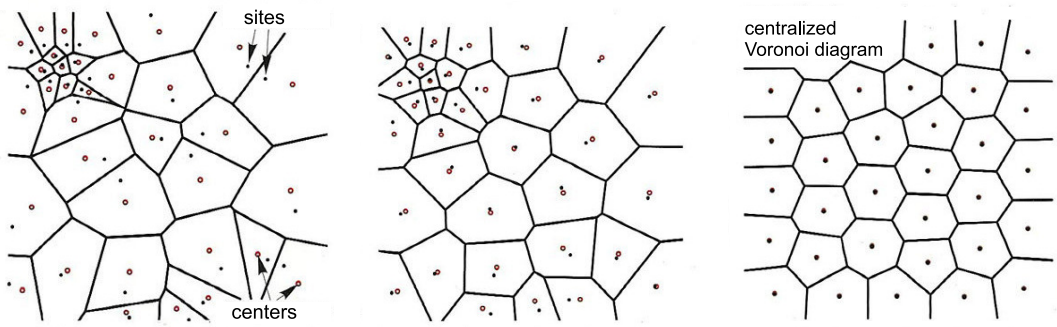

(a) (b)

Figure 4.1: Illustration of Fortune's algorithm for one site (a) and multiple sites (b) [20]. The beach line $B$ marks all points that have an equal distance $d$ to the Voronoi site and sweep line $L$.

### 4.1.2 Lloyd's algorithm

In Lloyd's algorithm a Voronoi diagram is created in a space such that each site is located at the center of its Voronoi cell [21]. The algorithm shows some similarities to k-means, which was described in Section 3.3.1. Lloyd's algorithm starts with $k$ sites in a space. For these $k$ sites a Voronoi diagram is constructed (Figure 4.2a). Then for each Voronoi cell the center is determined and all $k$ sites are moved to the center of their cell. Then a new Voronoi diagram is constructed using the new locations of the $k$ sites (Figure 4.2b), after which the center of each cell is determined,

and so on. This process is repeated until each site is located at the center of its Voronoi cell (Figure 4.2c).



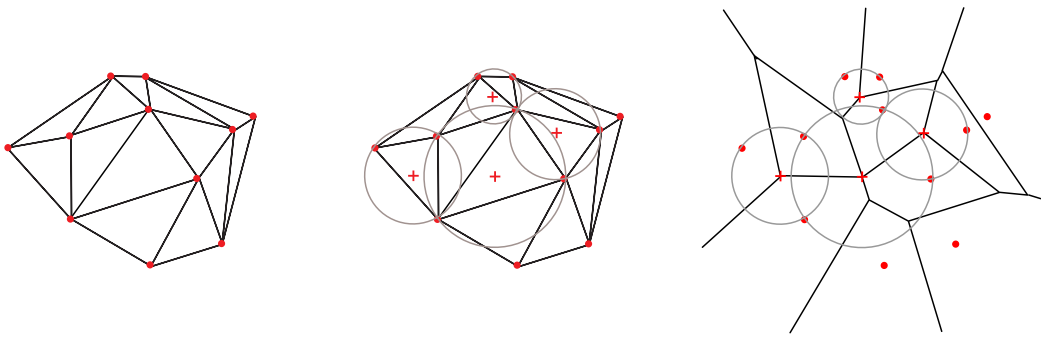

(a) (b) (c)

Figure 4.2: A two-dimensional space is divided in a Voronoi diagram (a). Each site is relocated to the center of its cell, resulting in a new Voronoi diagram (b). This is repeated until all sites are located on the center of their cell [22].

A clear difference to Fortune's algorithm is that Lloyd's algorithm can only be used if the positions of the Voronoi sites are not of any interest. This can be argued as either a drawback or an advantage, based on the requirements for Voronoi diagram. However, a clear drawback in comparison to Fortune's algorithm is that Lloyd's algorithm needs multiple iterations to achieve its purpose and each iteration a new Voronoi diagram is constructed.

### 4.1.3 Delaunay trangulation

A Voronoi diagram is the dual of a Delaunay triangulation [23]. Therefore, a Delaunay triangulation can be used to construct a Voronoi diagram. In geometry, a Delaunay triangulation is a type of triangulation, which is a planar object that is divided into triangles. A Delaunay triangulation contains a set of points $P$ such that no points exist within the circumcircle of any triangle in the object (Figure 4.3a). A Voronoi diagram of $P$ is constructed by determining the centers of the circumcircles of all triangles (Figure 4.3b). The centers are then connected with each other such that they include exactly one point $p \in P$ in each region they form. The connections between the centers are the edges of the Voronoi diagram, with the centers being the vertices (Figure 4.3c).



(a) (b) (c)

Figure 4.3: Construction of a Voronoi diagram from a Delaunay triangulation. Given a Delaunay triangulation (a) the centers of all circumcircles are determined (b). The centers are the vertices in the Voronoi diagram (c). Only four of the twelve circumcircles and centers are shown [23].

## 4.2 Voronoi spatial partitioning algorithm

Recall from Section 3.1 that a publisher sends data messages to a partition on a broker. The producer does not need to know the resulting Voronoi diagram, but only the locations of the sites.

For a data message it determines the nearest site by calculating the Euclidean distance between the message's location, which in our research is defined by the first location reference point in a message as discussed in Section 2.1.2, and each site. In terms of time complexity, this algorithm is $O(n)$. The message is then sent to the partition corresponding to the nearest site. Listing 4.1 shows our partitioning algorithm.

```
Set<Site> setOfSites;          // set of all sites in Voronoi diagram
Site location;                 // location from message
Site currentNN;                // current nearest neighbour

currentNN.Coordinates = (MAX, MAX);

foreach (Site s in setOfSites)
    if distance(location, s) < distance(location, currentNN)
        currentNN = s;
```

Listing 4.1: Pseudo code producer in $O(n)$

### Application of WMS requests

In Section 2.3.1 we decided to use the use case of WMS requests for our experiments to measure the performance of spatial partitioning methods. WMS requests can be handled by a subscriber in the Voronoi model as follows. Recall from Section 2.1.3 that a WMS request defines a square window that is defined by four pairs of latitude and longitude coordinates. In order to call the correct partitions, it is required to calculate which partitions overlap with the requested window. Due to the irregular shapes of the Voronoi cells, the only approach to do this is by spatial intersection. For this the window is compared to each Voronoi cell and returns true if it overlaps. Since all messages have to be retrieved from a partition, the shape and size of the overlapping is not required.

Figure 4.4 shows the process wherein overlap between the WMS request window and partitions is illustrated (Figure 4.4a), resulting in the partitions from which messages need to be retrieved (Figure 4.4b). The received messages are then filtered such that only the relevant messages remain, resulting in an image of the requested area (Figure 4.4c).

The process described shows that it is necessary for the subscriber to know the layout of the Voronoi diagram. During our experiments our subscriber used an open-source Java library based on Llyoyd's algorithm [24]. In this library a Voronoi diagram is computed using a Voronoi power diagram, as described in [25]. In a power diagram a Voronoi site has a weight that influences the size of its cell. By setting these weights to 0 for all sites, an ordinary Voronoi diagram is constructed since sites are not pulled towards the center of their cell. Thus only one iteration of Lloyd's algorithm is performed when constructing the Voronoi diagram.

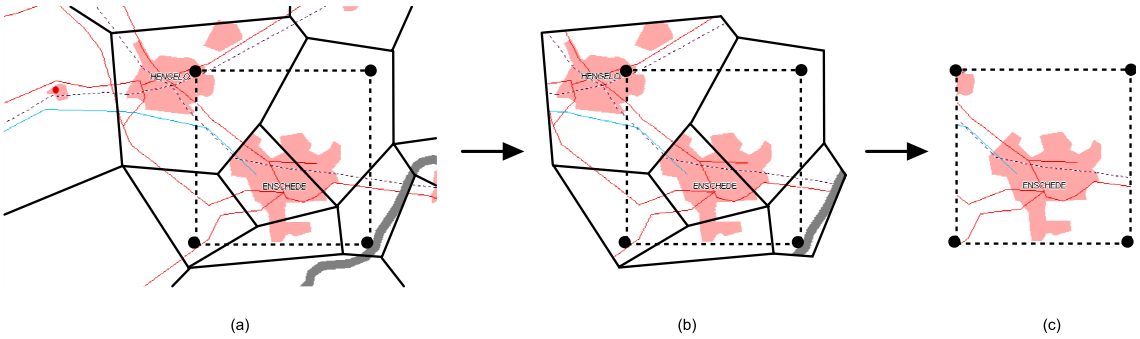



Figure 4.4: The process of selecting the required Voronoi partitions for a WMS request, resulting in an image of the requested area.

# Chapter 5

# Geohash Model

In this chapter we propose Geohash as a spatial partitioning method. Geohash is a technique that hashes a location's latitude and longitude coordinates into a single text string. This technique can be visualized as dividing the surface of the Earth into separate planes, i.e. spatial partitioning of Earth's surface. To our knowledge, despite this approach no research has been done on the application of Geohash as a spatial partitioning method. In this chapter we describe Geohash in detail, how it can be applied as a spatial partitioning method, and what its main differences are compared to Voronoi and other spatial partitioning methods.

## 5.1 Geohash

Geohash is a technique that allows a pair of geographical coordinates to be encoded into a single text string, called a geohash [26]. It was put in the public domain by Gustavo Niemeyer in February 2008 [27] [28]. The objective of Geohash is to have short strings of locations on Earth that can be used in URLs. This is accomplished by processing the latitude and longitude bitwise in base 32. Details of this process and the other way around are discussed in Sections 5.2 and 5.3. It is important to note that with *Geohash* we refer to the technique of encoding a coordinate pair into a single string, while *geohash* refers to this string that represents the encoding of a coordinate pair.

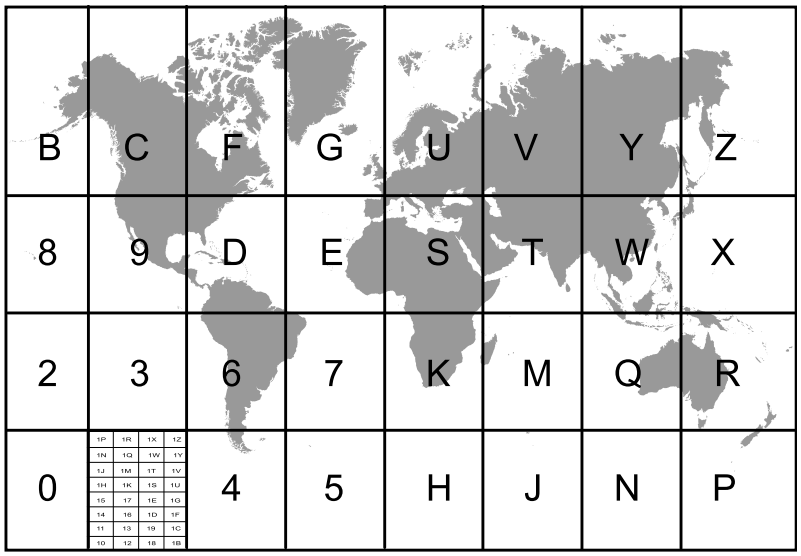



Figure 5.1: Geohash example

Geohash can be visualized as a division of Earth into 32 planes, each of which can be divided again into 32 planes, and so on. Each plane is associated with a unique geohash and each division results in an additional character for its string. Figure 5.1 shows this division, where a second division is shown in the plane associated with geohash *1*. We refer to these divisions as Geohash levels by defining level $x$ the division that results in geohashes of length $x$. In short, we use

Geohash-$x$ to define the level that consists of geohashes of length $x$. Thus, Figure 5.1 shows Geohash-1 and a part of Geohash-2. We define an implementation with geohashes of a shorter length as a lower Geohash level. For example, the highest Geohash level that consists of a geohash that covers the Netherlands in its entirety, describes therefore the longest geohash that completely covers the Netherlands (namely u1). Lower Geohash levels, and therefore shorter geohashes, would have the same results in our experiments since they consist of the same amount of geohashes when it comes to covering the Netherlands (namely one), while higher Geohash levels, and therefore longer geohashes, covers only parts of the Netherlands.

Although there is no official maximum (one can make a string as precise as they desire), in practice, including the official website geohash.org [29], usually a maximum of 12 characters is used, which results in an area of 3,7 by 1,9 cm around the equator, see Table 5.1.

| Length of geohash | Width | Height |
|---|---|---|
| 1 | 5.009,4 km | 4.992,6 km |
| 2 | 1.252,3 km | 624,1 km |
| 3 | 156,5 km | 156,0 km |
| 4 | 39,1 km | 19,5 km |
| 5 | 4,9 km | 4,9 km |
| 6 | 1,2 km | 609,4 m |
| 7 | 152,9 m | 152,4 m |
| 8 | 38,2 m | 19,0 m |
| 9 | 4,8 m | 4,8 m |
| 10 | 1,2 m | 59,5 cm |
| 11 | 14,9 cm | 14,9 cm |
| 12 | 3,7 cm | 1,9 cm |

Table 5.1: Approximate area size of a plane around the equator corresponding to a geohash of a certain character length [30].

This approach of Geohash can be used to define bounding boxes around spatial objects, a technique that can be easily applied to current systems without any modifications to their internal structure, since its resulting hash can be used as a key in a key-value system. As a result Geohash supports efficient updating, because no costly data structures such as trees or Voronoi diagrams are used.

Another advantage of Geohash emerge from the fact that spatial partitioning methods as Voronoi need (pre-defined) data points to divide an area, and those data points can be argued over. Geohash does not require any data apart from the area, i.e. Geohash is independent of any data set. This has as advantage that an area can be divided before any data is known.

In addition, because of Geohash hashing technique, it can be used as a simple and highly efficient filtering technique based on the prefixes in the hashes. When prefixes of hashes are common, the locations that those hashes represent are geographically close to each other as well. However, it is not guaranteed that locations that are close to each other result share a common prefix in their hashes. This drawback of Geohash can be illustrated by the following problems [31].

- **Edge-Case Problem.** Locations that are located on opposite sides of a covered area are close to each other geographically, but do not have a common prefix. It should be noted that this problem only appears in case of an area that is continuously at the edges. E.g. points in the far west and the far east when covering earth, (Figure 5.2).

- **Z-Order Problem.** Due to the Z-order traversal in which Geohashes are stored, points can be stored close to each other while not being close to each other geographically, or vice versa. E.g. areas 7 and 8 in Figure 5.3 are stored close to each other despite being geographically far from each other, while areas 7 and E are close to each other geographically but not in storage.

It should be noted however that both limitations do not play a major part in our research. The Z-Order Problem describes how the Z-order traversal in which geohashes are stored can lead to geohashes being stored close to each other, while they are geographically not close to each other at all. This problem can be neglected, because in a publish-subscribe messaging system messages

are usually stored on receiving time and not in another, location dependent index. The Edge-Case Problem may still exist in the design of our experiments, but does not play a role in our research due of the scope of our experiments being limited to the Netherlands, thus the covered area is not continuously at the edges.

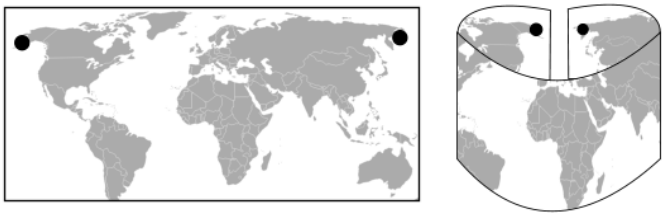


Figure 5.2: Visualization of the Edge-Case Problem of Geohash.

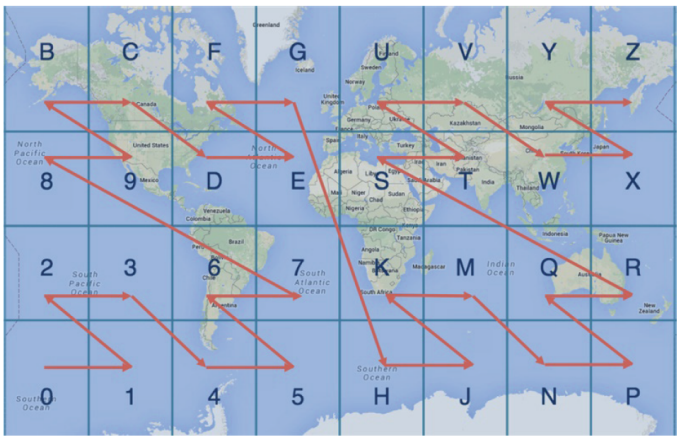



Figure 5.3: Visualization of the Z-Order Problem of Geohash [31].

## 5.2 Encoding algorithm

The Geohash encoding algorithm constructs a geohash from a pair of latitude and longitude coordinates. The base 32 character mapping that is used for encoding and decoding is shown in Table 5.2.

| Decimal | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Base 32 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | b | c | d | e | f | g |
| Decimal | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| Base 32 | h | j | k | m | n | p | q | r | s | t | u | v | w | x | y | z |

Table 5.2: Base 32 character mapping, used for encoding and decoding of geohashes.

The idea behind the encoding algorithm is that the longitude and latitude coordinates are used alternately to define the bits that make up a character, starting with the longitude coordinate. Since base 32 is used, five bits are required for each character. During execution of the algorithm the interval in which the coordinates is located is reduced. The more iterations are executed, the more precise the interval is, resulting more bits and thus more characters (i.e. a longer string).

Beforehand we set the intervals on (-90,90) for latitude and (-180,180) for longitude, as these are the maximum values for latitude and longitude respectively. Furthermore, we keep track of the current character bits $ch$ and the current bit $b$, both starting with 0. For every five bits a character can be appended to the initially empty geohash. This number of the current bit is also used to determine the second value for the OR execution in step 2 of the algorithm as described below: for this an array $bits = 16, 8, 4, 2, 1$ is defined, where $bits[b]$ will be the second value for the OR execution.

The algorithm works as follows. Remember that we start with the longitude coordinate. When all steps are completed, the next iteration will be executed with the latitude coordinate, and so on.

1. The first step is to determine the middle value of the current interval.
2. If the current coordinate value is higher than the middle, a bitwise inclusive OR is executed between the current character bits $ch$ and $bits[b]$, where $bits = 16, 8, 4, 2, 1$. The result of this execution is stored in $ch$.
3. The current interval is changed to the half in which the coordinate value is included.
4. If we have not yet done five iterations of this algorithm, i.e. $b < 4$, the value of $b$ will be increased by 1. Otherwise the value of $ch$ defines a base 32 character, which is added at the end of the current geohash. Both the values of $b$ and $ch$ are set to 0 afterwards.

The used implementation of this algorithm in Java can be found in the *encode()* method in Appendix A.

**Example.** Suppose we have the following longitude/latitude coordinates: $lon = 6.851656$ and $lat = 52.244048$. As described above, we set the character bits $ch$ and current bit $b$ to 0: $ch = b = 0$. The current coordinate intervals are $lonInt = (-180, 180)$ and $latInt = (-90, 90)$ for longitude and latitude respectively. We now show the first two iterations of the algorithm.

Iteration 0

1. Define middle $mid$ of current interval $lonInt = (-180, 180)$: $mid = 0$.
2. It holds that $lon > mid$ since $6.851656 > 0$, thus $ch|bits[b] = 0|16 = 16$.
3. Since $lon > mid$, we change the current interval to the higher half: $lonInt = (0, 180)$
4. We increase the current bit: $b = 1$.

Iteration 1

1. Define middle $mid$ of current interval $latInt = (-90, 90)$: $mid = 0$.
2. It holds that $lat > mid$ since $52.244048 > 0$, thus $ch|bits[b] = 16|8 = 24$.
3. Since $lat > mid$, we change the current interval to the higher half: $latInt = (0, 90)$
4. We increase the current bit: $b = 2$.

We leave it up to the reader to show that the first character will be $u$, and the geohash can be calculated to $u1kc5ytpxhmj$.

## 5.3 Decoding algorithm

Similar to the enconding algorithm of Geohash as described in Section 5.2, base 32 is used for decoding geohashes into latitude and longitude coordinates. The base 32 character that is used can be found in Table 5.2.

The idea behind the decoding algorithm is to do the opposite as the encoding algorithm, that is using the bits that define each character to specify the coordinates. Beforehand we set the intervals on (-90,90) for latitude and (-180,180) for longitude, as these are the maximum values for latitude and longitude respectively. The first step of the algorithm is to rewrite eacg character of the given geohash into five bits following the base 32 character map. Then, assign to each bit a number, starting with 0 and increasing by 1 for each next bit. All even numbered bits, including the first numbered with 0, belong to the longitude coordinate. All remaining odd numbered bits belong to the latitude coordinate. Then, the following steps are executed for the longitude and latitude coordinate separately.

1. Take the first bit. If low (0), set the lower half of the current interval as the new interval. If high (1), set the higher half of the interval as the new interval. E.g. if 1 is the first latitude bit, (0,90) is the next interval.

2. Repeat for every bit.

The implementation of this algorithm can be found in the *decode()* method in Appendix A.

**Example.** Suppose we have the following geohash: $u1kc5ytpxhmj$. As described above, we set the current coordinate intervals to $lonInt = (-180, 180)$ and $latInt = (-90, 90)$ for longitude and latitude respectively. Then we rewrite the geohash using Table 5.2 and number the resulting bits. The resulting bit sequence and numbering can be found in Table 5.3.

| Base 32 | u | | | | | 1 | | | | | k | | | | | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Decimal | 26 | | | | | 1 | | | | | 18 | | | | | ... |
| Bits | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | ... |
| Bit # | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | ... |

Table 5.3: Decoding of first three characters of the geohash $u1kc5ytpxhmj$.

Recall that the even numbered bits belong to the longitude coordinate and all odd numbered bits belong to the latitude coordinate. We now show the first five iterations for each coordinate.

Longitude bits: 10000, which starting interval $lonInt = (-180, 180)$.

1. The first bit is 1, so we change the interval to the higher half of the current interval: $lonInt = (0, 180)$.

2. The next bit is 0, so now we change the interval to the lower half of the current interval: $lonInt = (0, 90)$.

3. Again, the next bit is 0, thus: $lonInt = (0, 45)$.

4. Again, the next bit is 0, thus: $lonInt = (0, 22.5)$.

5. Again, the next bit is 0, thus: $lonInt = (0, 11.25)$.

Latitude bits: 11001, which starting interval $latInt = (-90, 90)$.

1. The first bit is 1, so we change the interval to the higher half of the current interval: $latInt = (0, 90)$.

2. The next bit is 1 again, thus: $latInt = (45, 90)$.

3. The next bit is 0, so now we change the interval to the lower half of the current interval:: $latInt = (45, 67.5)$.

4. The next bit is 0 again, thus: $latInt = (45, 56.25)$.

5. The next bit is a 1, thus: $latInt = (50.625, 56.25)$.

We leave it up to the reader to show that the geographical coordinates of the geohash will be 6.851656 longitude and 52.244048 latitude.

## 5.4 Geohash applied as a spatial partitioning method

In this section we describe how Geohash is applied as a spatial partitioning method. In addition, we propose a method that merges the created partitions based on the data distribution within the space.

### 5.4.1 Spatial partitioning method application

The first step of applying Geohash as a spatial partitioning method is to define the geohashes that cover the area, e.g. by calculating the geohashes of the areas outer borders and filling up the geohashes inbetween. Depending on the exact calculation to define the area's geohashes, duplicate geohashes may result. Therefore, the resulting list of geohashes should be filtered such that only unique geohashes remain with a given length. This amount of geohashes will be equal to the amount of partitions and each geohash is associated with exactly one partition. This association is stored in a list. Now the publisher and subscriber determine partitions as follows. They calculate a geohash from the given coordinates using the encoding algorithm described in Section 5.2. Then the resulting geohash is looked up in the list of association between geohashes and partitions to retrieve the associated partition.

#### Application of WMS requests

In Section 2.3.1 we decided to use the use case of WMS requests for our experiments to measure the performance of spatial partitioning methods. WMS requests can be handled in the Geohash model as follows. Recall from Section 2.1.3 that a WMS request defines a square window that can be defined by four pairs of latitude and longitude coordinates. In order to call the correct partitions, it is required to calculate which partitions (or geohashes) overlap with the requested window. Because both the window and the visualization of the partitions are rectangular shapes, there are two approaches by which the correct partitions can be determined. Both approaches make use of the structured way geohashes are created and do not require an expensive spatial intersect as used by Voronoi.

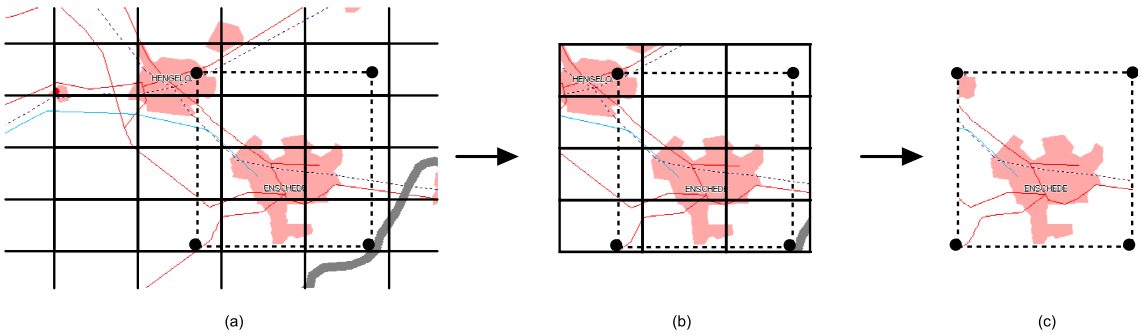



Figure 5.4: The process of selecting the required Geohash partitions for a WMS request, resulting in an image of the requested area.

The first approach is to use a matrix structure. Using this approach one could start in any corner of the WMS window and walk through the matrix to the point diagonal of it via the two neighbouring points. For example, in our implementation we started at the left upper corner of the WMS window and walked down and right through the matrix until we found the bottom left corner and upper right corner respectively. Then we walked right and down respectively until we found the bottom right corner. Afterwards, the geohashes in between are added, e.g. by a floodfill algorithm.

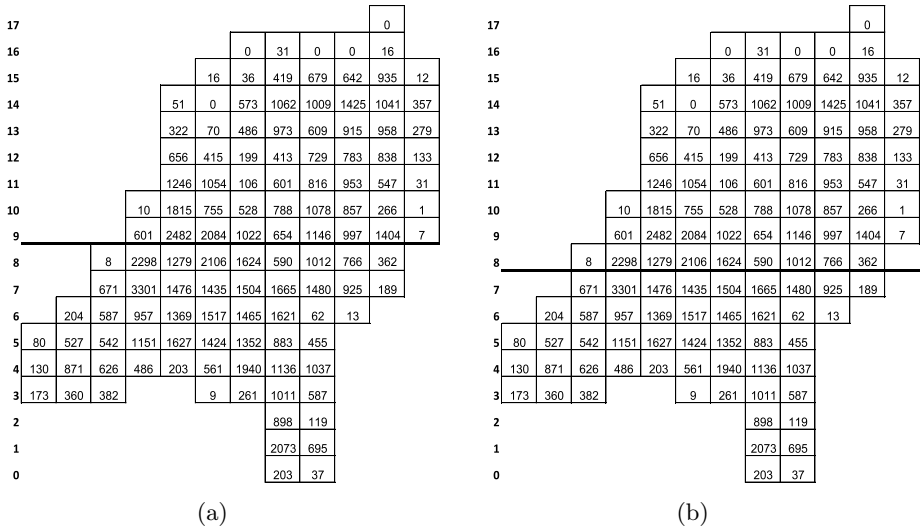
The second approach is to calculate the geohashes of each corner and the geohashes in between using the structure of geohashes. This works similar to the approach above, but it calculates geohashes on-the-fly instead of using a memory costly structure. For example, one could start with the upper left corner and work towards the upper right corner until it finds the geohash of the given pair of latitude and longitude coordinates. It then does the same for the row below, until it finds the geohashes of the bottom corners.

Figure 5.4 shows the process wherein overlap between the WMS request window and partitions is illustrated (Figure 5.4a), resulting in the partitions from which messages need to be retrieved (Figure 5.4b). The received messages are then filtered such that only the relevant messages remain, resulting in an image of the requested area (Figure 5.4c).

### 5.4.2 Merge algorithm

As mentioned in Section 5.1, Geohash divides a given space without any knowledge about the data. This can be an advantage, since there is no need for constructing a tree or Voronoi diagram. However, it may be more efficient if it is known how the data is actually distributed over the space, e.g. for load balancing. In Section 3.3.2 we described kd-tree as an often used method to partition space based on data points. Based on kd-tree we propose a weighted kd-tree method for Geohash that associate multiple geohashes with one partition instead of one geohash per partition.

In this method every plane that is a result of Geohash is given a weight, based on the amount of data points. Then a kd-tree is constructed in which the weights are considered for where to put the median. The advantage of this method is that the partitions are not only balanced because of the kd-tree, but also remain rectangular shapes when drawn on a map, i.e. the advantages for handling WMS requests remain.




(a) (b)

Figure 5.5: First iteration of weighted kd-tree algorithm for Geohash-4 in which the median initially placed above $y = 8$ (a), but moved downwards to $y = 7$ where it remains (b).

**Example.** Suppose we use Geohash-4 as spatial partitioning method for the Netherlands and has as result that messages are distributed over the partitions as shown in Figure 5.5. Starting with division over the y-axis for our first iteration, we determine that the median is $\lfloor \frac{17+0}{2} \rfloor = 8$. Thus, we place the median after (above) $y = 8$ (Figure 5.5a). We now calculate the sum of partition weights at either side of the median: 52325 for the lower half and 38931 for the upper half. Since the weight of the lower half is greater than the upper half ($52325 > 38931$), we investigate the possibility to move the median backwards (downwards). Thus, the median is now after (above) $y = 7$ and the calculated sums are 42280 and 48976. If the difference between the sums is smaller, we fullfil the possibility to move the median. Since $48976 - 42280 < 52325 - 38931$, we define $y = 7$ as our new median. Now we again check if we want to move the median backwards to $y = 6$ through the same steps. In this case the difference is larger, so we complete this iteration by stating that the partitions should be divided by $y = 7$ as shown in Figure 5.5b. Now for both halfs the algorithm can be executed again separately, similar to the regular kd-tree algorithm. Note that a median can either move forward or backwards during an iteration, but not both ways.

The formalisation of this algorithm can be found in Listing 5.1.

```
function kdtree (listOfPoints pointList, int depth)
{
    // Select axis based on depth so that axis cycles through all valid values
    var int axis := depth mod k;

    // Sort point list and choose median as pivot element
    select median by axis from pointList;

    // Create node and construct subtrees
    var tree_node node;
    if (values of pointList.UntilMedian == values of pointList.AfterMedian)
        // nothing
    else if (diff := values of pointList.UntilMedian > values of pointList.
    AfterMedian)
        median = backwards(pointList, median, diff);
    else if (diff := values of pointList.UntilMedian < values of pointList.
    AfterMedian)
        median = forwards(pointList, median, diff);
    node.leftChild := kdtree(pointList.UntilMedian, depth+1);
    node.rightChild := kdtree(pointList.AfterMedian, depth+1);
    return node;
}

function backwards (listOfPoints pointList, point median, int diff) {
    select point before median from pointList as new_median;
    if ((new_diff := values of pointList.UntilNewMedian - values of pointList.
    AfterNewMedian) < diff)
        backwards(pointList, new_median, new_diff);
    return median;
}

function forwards (listOfPoints pointList, point median) {
    select point after median from pointList as new_median;
    if ((new_diff := values of pointList.AfterNewMedian - values of pointList.
    UntilNewMedian) < diff)
        forwards(pointList, new_median, new_diff);
    return median;
}
```

Listing 5.1: Algorithm for constructing a weighted kd-tree

# Chapter 6

# Experiments

In this chapter we present our experiments on the application of spatial partitioning methods in publish-subscribe messaging systems and discuss their results with regards to a system without a spatial partitioning method applied. The experiments are executed with multiple implementations of the Voronoi and Geohash models that were presented in Chapters 4 and 5. In the following sections we first present the setup used for the experiments. Thereafter, we evaluate the experiments on partition sizes, which was indicated as a supporting factor for load balancing improvements in Section 2.1.1. In the next section we evaluate our experiments on WMS requests, which was identified as a suitable use case in Section 2.3.1. We conclude this chapter with a summary of our findings.

## 6.1 Experimental setup

Figure 6.1 shows a schematic view of our experimental setup, which is based on the current setup of Simacan's Connect 2 as described in Section 2.2. It is shown that a Kafka producer retrieves input data and forwards this data to a Kafka cluster. A Kafka consumer reads the data that is stored within the Kafka cluster. The retrieved data is then analyzed and processed for the results sections of this chapter. The location of the data on the cluster is defined by a partitioning method in the partitioner, which is used by the producer and consumer to determine where data needs to be stored or can be retrieved from. Each instance is discussed in more detail in the remainder of this section.

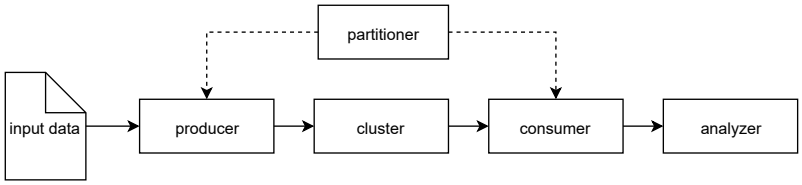



Figure 6.1: Schematic overview of the experimental setup.

**Input data.** As input data a set of TomTom's HD Flow data was used. This data set consists of 91.256 messages that describe situations on traffic roads in the Netherlands during one minute. An example of these messages is shown in Listing 2.2. TomTom updates the situations each minute, without changing the location and amount of messages within the data set. Each message is around 300 bytes, resulting in a data set of 27 megabytes per minute.

**Partitioner.** The partitioner is an implementation of a partitioning method and it defines the amount and distribution of necessary partitions. In our experiments the partitioning methods were only implemented for covering the Netherlands, since the input data consists only of messages within the Netherlands. The partitioner is used by the producer and consumer to determine which partition to send messages to or receive messages from.

**Producer.** The producer reads the input data one message each. The location in each message,

which is encoded in OpenLR, is decoded from binary 64 to retrieve its geographical coordinates in WGS-84. These coordinates are used by the partitioner to determine a partition as destination to which the producer sends the message to.

**Cluster.** A Kafka cluster was set up similar to the current test setup at Simacan, thus with three brokers and two replicas in total. A ZooKeeper instance was used as controller.

**Consumer.** During the experiments a Kafka consumer pulls messages from the Kafka cluster. The consumer pulls all messages from all partitions within a specified topic. When finished the consumer sends all received messages to an analyzer, after which it terminates.

For the experiments on WMS requests as described in Section 6.4, a set of 2.162.784 WMS requests is used. This set contains requests that are sent to Simacan's map visualization systems during 18 days by customers and developers of Simacan, and all requests contain a squared window. A specialized consumer processes each request by determining from which partitions messages need to be retrieved. This is determined by overlapping the window of the WMS request with the visualized shapes of the partitions, as described in Sections 4.2 and 5.4.1.

**Analyzer.** All data received by the consumer are analyzed and when required processed further, for example to an image.

**Machine architecture.** The experiments were performed in an Oracle VM VirtualBox, which was assigned 4 GB of RAM and two CPUs based on the host's Intel(R) Core(TM) i7-3630QM 2.40GHz processor. The virtual machine had Ubuntu 14.04 LTS installed. The experiments were performed on Apache Kafka 0.9.0.0, which at the start of this thesis was the latest stable release of Apache Kafka, and Java 8 (1.8.0).

## 6.2 Model implementations

During our experiments we use multiple implementations of the Voronoi and Geohash models that are described in Chapters 4 and 5. In this section the different implementations are presented. All implementations are used in each experiment, unless stated otherwise, and all are evaluated to the existing method that does not apply any partitioning, called the baseline method. First we shortly describe this baseline method, whereafter the implementations of the Voronoi and Geohash models are presented.

### 6.2.1 Baseline method

Recall that our experimental setup is based on Simacan's Connect 2 as described in Section 2.2. In that system, no partitioning is applied yet. This results in all data being stored on a single partition and thus is retrieved from the same single partition. Since this is the current situation and the other implementations are compared to this method, we refer to this method as the baseline method.

### 6.2.2 Voronoi

The different implementations of the Voronoi model are due to the different choices of Voronoi point sets. By using multiple different sets of Voronoi points a general behaviour of Voronoi may result, which allows us to make a general comparison between Voronoi, Geohash, and the baseline method.

#### Cities based on population

Cities with a large population often cover a larger geographical area than cities with a smaller population. It is expected from larger areas to contain a larger amount of roads, and a larger amount of main roads that are amongst the most frequently used roads. When an area consists of more roads and road traffic, one could expect more messages about the situation on these roads, which are preferably stored together to have as less partitions accessed as possible. Therefore, this set contains cities of the municipalities that are the largest in the Netherlands in terms of

(a) (b) (c)

Figure 6.2: Visualization of Voronoi partitioning for all cities (a), interchanges A-only (b), and interchanges N-included (c).

population. Based on the fact that Geohash-4 weighted kd-tree results in 78 and 96 partitions, we selected the capital cities of the top 100 largest municipalities in order to give an appropriate comparison with Geohash(-4). The complete list of these cities and their properties can be found in Tabels C.1 and C.2, and a visualization of the resulting partitions is shown in Figure 6.2a. The different implementations using cities as Voronoi points are established through the sequence and condition in which the cities are added.

We distinguish the following implementations with cities as Voronoi points.

- *PopDesc (Population Descending).* The cities are added in descending order based on their population, i.e. starting with Amsterdam and Rotterdam.
- *PopAsc (Population Ascending).* The cities are added in ascending order based on their population, i.e. starting with Medemblik and Berkelland.

**Traffic road interchanges**

Long traffic routes, such as generated by route planners, often consist of a part from the point of departure to a highway, a part on a highway, and a final part from a highway to the destination. The part of the route that runs over highways is usually defined from interchange to interchange. Regular traffic jams often are just before or just after an interchange [32]. Thus, one can expect a high amount of traffic messages around these interchanges, where logically one would be interested in possible jams before *and* after the interchange.

This set therefore consists of official traffic road interchanges of the Dutch highways. The complete list of these interchanges and their properties can be found in Tables C.3, C.4, and C.5.

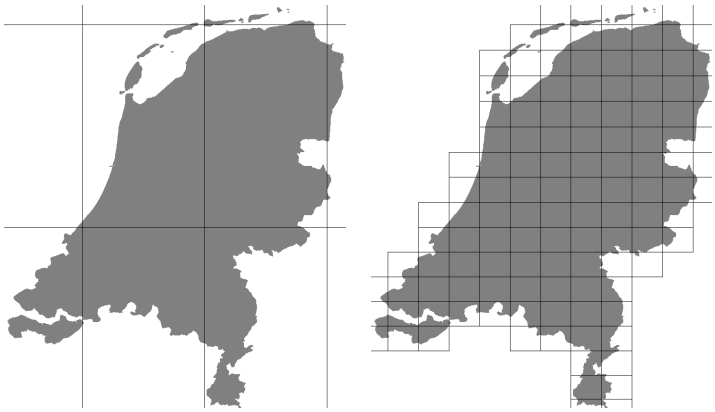
We distinguish the following implementations with interchanges as Voronoi points.

- *A-only.* This set contains only official interchanges between at least A-roads, which are roads that are managed by the national government. A visualization of the resulting partitions is shown in Figure 6.2b.
- *N-included.* This set contains both interchanges between A-roads and N-roads, the latter being roads that are managed by a regional government. Similar to the A-only set, N-included does only contain official interchanges as defined by the national government. A visualization of the resulting partitions is shown in Figure 6.2c.

### 6.2.3 Geohash

The different implementations of the Geohash model are due to the different Geohash levels, i.e. the length of the Geohash strings (geohashes). In short, we use Geohash-$x$ to refer to the division that results in geohashes of length $x$, as explained in detail in Section 5.1.

In our implementations each geohash is associated with a partition. Thus a Geohash implementation with $y$ geohashes that together cover an area completely would result in $y$ partitions. The



(a) (b)

Figure 6.3: Visualization of Geohash partitioning for Geohash-3 (a) and Geohash-4 (b).

Geohash implementations used in our experiments are those which result in multiple partitions. Because Geohash-2 and lower only result in a single partition, they are omitted. However, we do use both Geohash-3 (Figure 6.3a) and Geohash-4 (Figure 6.3b), which result in 9 and 121 geohashes respectively for the Netherlands. The geohashes corresponding to these partitions can be found in Appendix B. Geohash-5 and higher are omitted in this research, due to both time constrictions for defining the correct set of geohashes (which will be over 3000) and since Simacan is looking for a relatively small amount of partitions as improvement for their current system: their current system process the Netherlands in a single partition with enough efficiency for their desires. We admit that a modification such as weighted kd-tree would reduce the number of partitions, and therefore may be a sustainable improvement for the future. We therefore highly recommend any research to this implementation as future work in Section 7.1.

The weighted kd-tree is implemented for both Geohash-3 and Geohash-4 using the algorithm described in Section 5.4.2. In these implementations adjacent geohashes are associated to one partition based on their weight, which in our research is defined as the amount of messages. We distinguish starting the iterations of weighted kd-tree over the y-axis (vertical division) and over the x-axis (horizontal division).

To summarize, we have the following implementations of the Geohash model.

- *Geohash-3 (unmodified).* The regular Geohash-3 implementation consisting of 9 partitions.
- *Geohash-3 (y).* The application of weighted kd-tree to Geohash-3, starting with division over the y-axis (vertical division).
- *Geohash-3 (x).* The application of weighted kd-tree to Geohash-3, starting with division over the x-axis (horizontal division).
- *Geohash-4 (unmodified).* The regular Geohash-4 implementation consisting of 121 partitions.
- *Geohash-4 (y).* The application of weighted kd-tree to Geohash-4, starting with division over the y-axis (vertical division).
- *Geohash-4 (x).* The application of weighted kd-tree to Geohash-4, starting with division over the x-axis (horizontal division).

## 6.3 Experiments on partition size

In this section we discuss our experiments regarding the size of the partitions in Kafka. As indicated in Section 2.1.1 the size of the partitions, or rather the size uniformity, could improve load balancing. In our experiments we investigate how the uniformity in partition size holds in the different implementations of the Geohash and Voronoi models. Recall that the baseline method results in one partition that contains all 91.256 messages. It could be argued that this means that it holds optimal uniformity, but as dividing work load requires multiple partitions by definition, we do not

compare the different implementations of the Geohash and Voronoi models directly to the baseline method. We do however refer to the baseline method to illustrate the differences in partition sizes between the baseline method and the different Geohash and Voronoi implementations.

In the following sections the results of the experiments are presented per model. We present our results in the following ways.

**Bar graphs.** In bar graphs an equal distribution is indicated by bars of the same height, i.e. the smaller the height differences the more equal the distribution. Of course a lower bar indicates a lower amount of messages on a partition, i.e. a smaller partition size. We use bar graphs to illustrate the distribution at the maximum amount of partitions for each implementation.

**Box plot graphs.** In a Tukey box plot graph, as described in [33], a lower position of a box indicates smaller partition sizes. In addition, a smaller box plot represents a more equal distribution; a completely equal distribution should result in a box plot where the median, first quartile, third quartile, and whiskers are on the same height. When a box has not those properties, it is investigated whether the data is skewed to a particular direction. Data is skewed when the distance between each whisker and the mean is not equivalent, or if the distance between each quartile and the mean is not equivalent. We use box plot graphs to illustrate the distribution progress for an implementation when increasing the amount of partitions.

**Line graphs.** For the differences to the average partition size line graphs are used. A lower line indicates a lower difference to the average, and thus a more equal distribution. We use line graphs to illustrate the average difference between the actual amount of messages per partition and the average amount of messages per partition.

We conclude this section with a comparison between the results of all models.

### 6.3.1 Results Voronoi model

In this section we present the results of the experiments on partition size using the implementations of the Voronoi model as they are defined in Section 6.2.2. First, we present the results of implementations related to the cities based on population. After that, we present the results of implementations related to the road traffic interchanges.

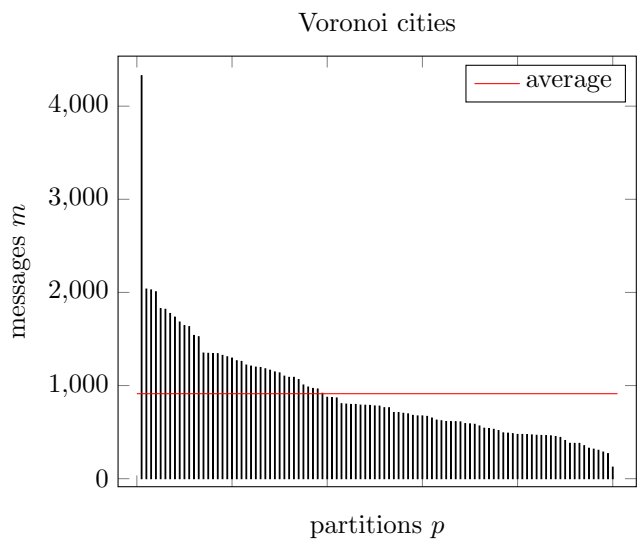



Figure 6.4: Distribution of the messages over the partitions using Voronoi for the top 100 cities in the Netherlands with the largest population.

### Cities

Figure 6.4 shows how the messages are distributed over the partitions using the top 100 cities in the Netherlands with highest population using the Voronoi model. It can be seen that the majority of the partitions has a message amount below average, as indicated by the red line ——. The largest partition, which corresponds with Groningen, consists of 4327 messages, while the smallest partition, corresponding with Pijnacker, consists of only 124 messages - a difference of 4203 messages. Even more noteworthy is the difference of the largest partition with the second largest partition, which corresponds with Leeuwarden: 4327 to 2036 - that is less than half the highest partition. Recall that the baseline method results in one partition containing all 91.256 messages - 21 times more than the largest partition of the Voronoi cities implementations.

Figure 6.5 shows the distribution of the messages over the partitions for the different implementations of cities in the Voronoi model. It should be noted that the box plot for 100 partitions, which is indicated in red, is the same for both PopDesc and PopAsc, since both box plots refer to exactly the same set of cities.

The figure shows that for the distribution using PopDesc the boxes and the difference between their whiskers initially shrink when increasing the number of partitions. In addition, the position of the boxes show a exponential decay. Throughout the increase of the number of partitions the amount of outliers stays the same in general. For $p = 50, 80, 90, 100$ the distribution consists of an extreme outlier. Note that the position of the upper whisker changes constantly between each measure point. Also, observe that the data is skewed to the larger partitions from $p \geq 40$ onwards.

For the distribution in PopAsc it is shown that the boxes and the difference between their whiskers shrink when the number of partitions increases, as well as the positions of the boxes showing an exponential decay. In addition, the data is skewed to the larger partitions These are all similar to the PopDesc variant. However, the amount of outliers is in general lower than PopDesc, although in all cases the highest outlier is an extreme outlier.

In Figure 6.13 the average absolute difference in amount of messages per partition to the average amount can be found. Looking only to the Voronoi cities implementations, it can be seen that the difference to the average decreases when increasing the number of partitions. For the PopDesc implementation it can be seen that the difference starts very high, with a difference that is equal to the average amount of messages per partition. From $p = 30$ onwards the two implementations show a similar pattern close to each other, with PopDesc having a lower difference from $p = 40$ onwards.

Figure 6.14 shows the relative difference to the average amount of messages per partition. The similar behaviour of PopDesc and PopAsc can again be observed from $p = 30$, with PopDesc having a lower relative difference than PopAsc from $p = 40$ onwards. The observation of PopDesc having a difference equal to the average amount of messages per partition can be made here again at $p = 20$, where this difference is 100.96% on average.

### Interchanges

Figure 6.6 shows how the messages are distributed over the partitions using the interchanges in the Voronoi model. The implementation using only interchanges between A-roads (A-only) is shown on the left; the implementations using all official interchanges between A- and/or N-roads (N-included) is shown on the right. The A-only implementation consists of 80 partitions and the N-included adds another 14. It can be seen that in both implementations the majority of the partitions has a message amount below average, as indicated by the red line ——.

The A-only graph shows a very high peak on the left side, indicating a very large partition in comparison to the remaing partitions. In fact, the largest partition consists of 7300 messages, while the second largest only contains 3837 messages - just over half that of the largest partition. The high amount of messages can be explained that the corresponding interchange for this partition is Julianaplein in Groningen, in the northeast of the Netherlands. There are less A-only interchanges in this area of the Netherlands than in, for example, the highly populated western area.

The N-included graph shows a high peak on the left as well, bar it not so extreme as for A-only. The largest partition of N-included contains 3434 messages against 2640 messages in the second largest partition. In this case the largest partition corresponds with the interchange Kooimeer, a newly added interchange in the N-included set.

Recall that the baseline method results in one partition containing all 91.256 messages - 12 times more than the largest partition of the Voronoi interchanges A-only implementation. For the

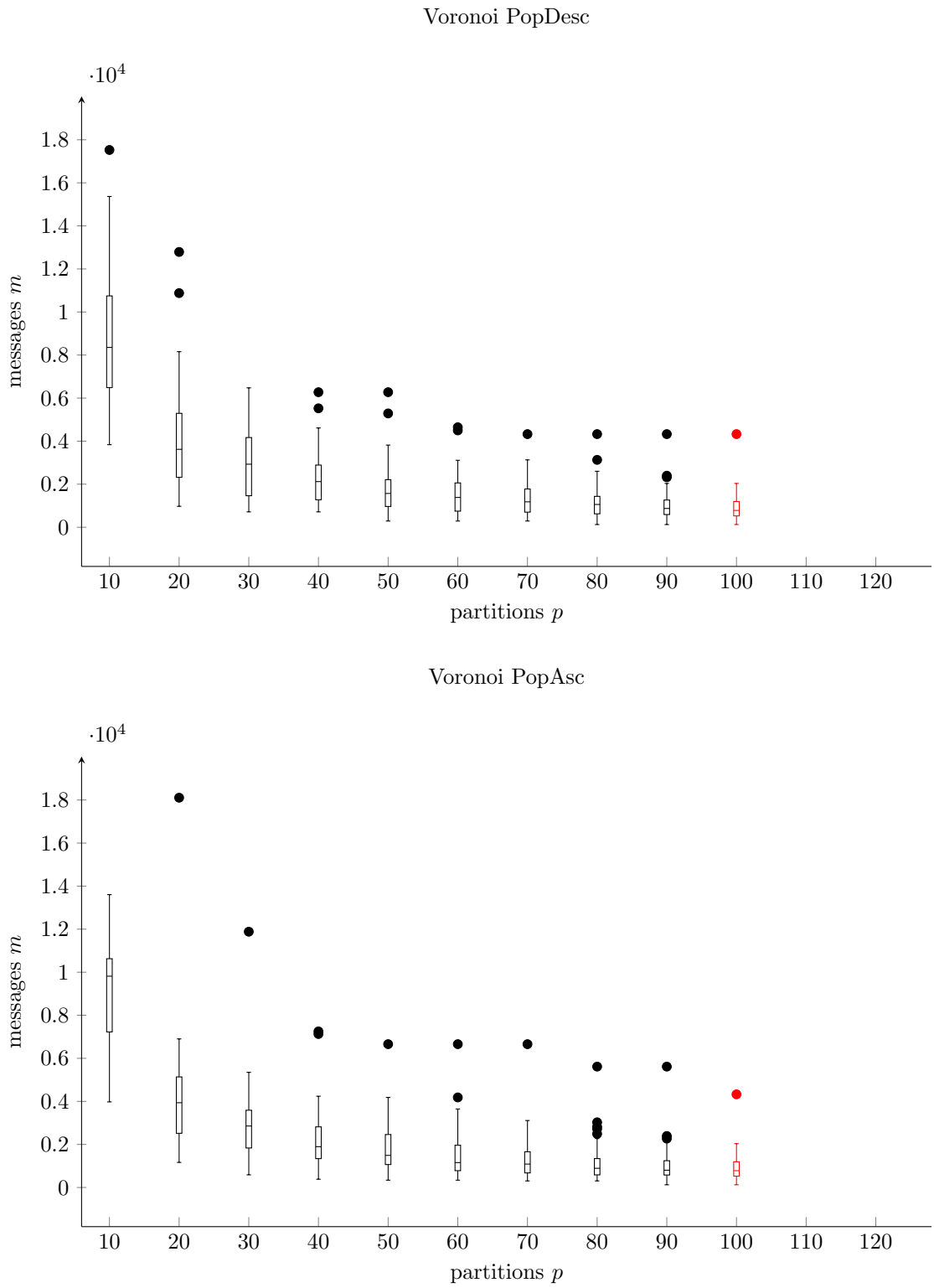



Figure 6.5: Boxplots of the message distribution over the partitions using Voronoi PopDesc (top) and PopAsc (bottom).

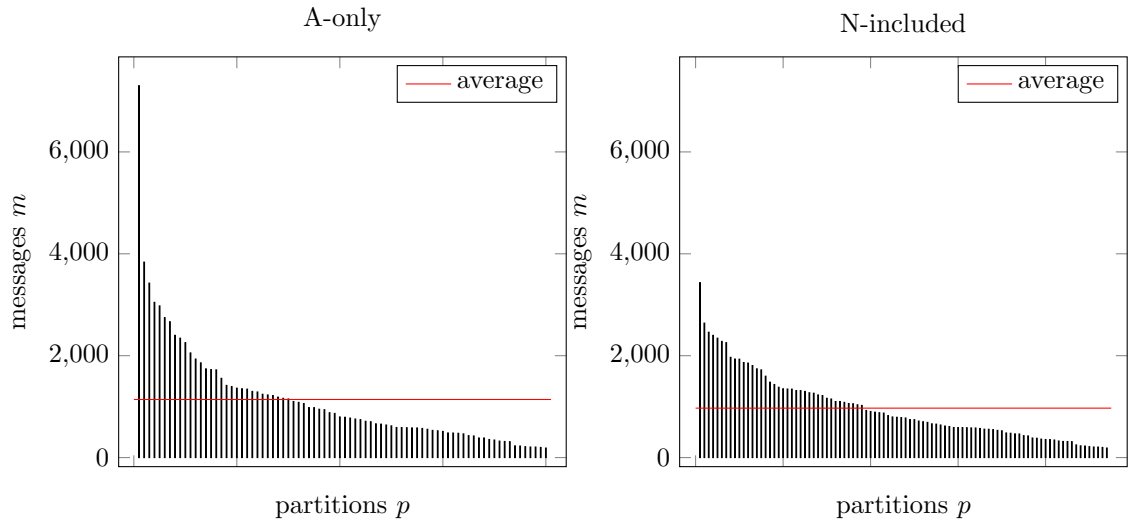



Figure 6.6: Distribution of the messages over the partitions using Voronoi interchanges A-only (left) and N-included (right).

N-included implementation the baseline method has a partition more than 26 times the size of its own largest partition.

Figure 6.7 shows the distribution of the messages over the partitions for the different interchange implementations in the Voronoi model. The implementation using only interchanges between A-roads (A-only) is shown on the left in red; the implementations using all official interchanges between A- and/or N-roads (N-included) is shown on the right in blue.

The figure shows that there is not much difference between the positions of the boxes and their whiskers, quartiles, and means. The major difference can be seen in the outliers. The A-only implementation has no less than seven outliers, the highest of them is an extreme outlier. The N-included has just two outliers which are both weak outliers. The data is skewed to large partitions in both implementations.

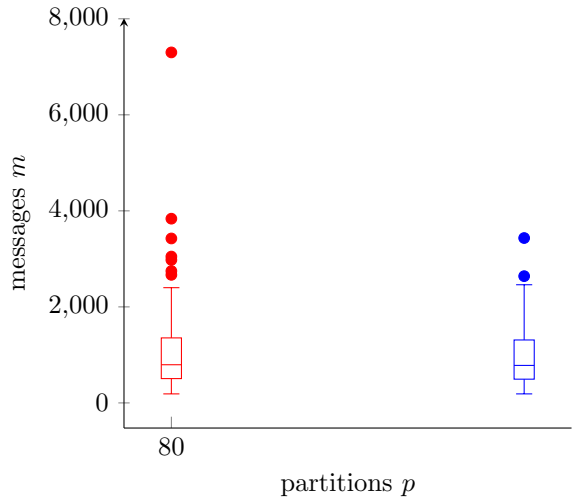



Figure 6.7: Distribution of the messages over the partitions using Voronoi interchanges A-only (left) and N-included (right).

In Figure 6.13 the average absolute difference in amount of messages per partition to the average amount can be found. Looking at the Voronoi interchanges implementation, it can be seen that the difference to the average decreases when increasing the number of partitions. This is similar to the other Voronoi implementations. However, the difference is higher for the interchanges implementation.

Figure 6.14 shows the relative difference to the average amount of messages per partition. Again

it shows that the difference for the Voronoi interchanges implementation is much higher than the cities implementations, more than 10% for A-only.

### 6.3.2 Results Geohash model

In this section we present the results of the experiments on partition size using the implementations of the Geohash model as they are defined in Section 6.2.3. First, we present the results of implementations related to Geohash-3. After that, we present the results of implementations related to Geohash-4.

#### Geohash-3

Figure 6.8 shows the amount of messages for each of the 9 partitions that belong to Geohash-3. The red line —— indicates the average amount of messages per partition (10.140 messages). Although barely visible, the smallest partition has no messages at all (the two to its left contain 8 and 47 messages). To illustrate the difference, the partition with the highest number of messages consists of 26.267 messages. Recall that the baseline method consists of a single partition containing all 91.256 messages, which concludes that the largest partition of Geohash-3 contains less than a third that amount.

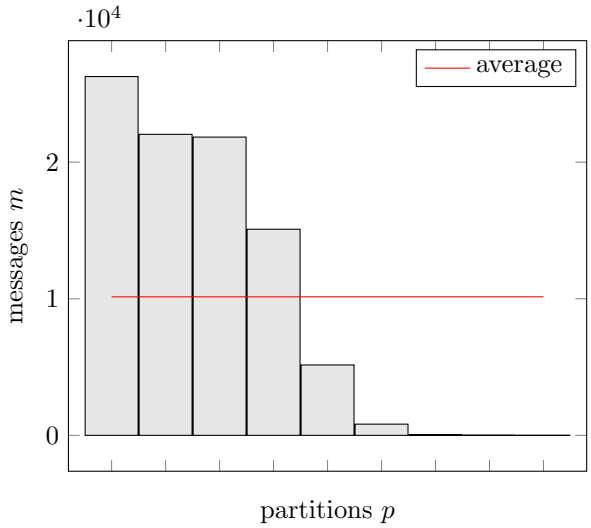



Figure 6.8: Distribution of the messages over the partition using Geohash-3, combined with the average amount of messages per partition.

Figure 6.9 shows the distribution of the messages over the partitions during iteration $i$ of Geohash-3 weighted kd-tree. The top four graphs show the iterations starting with dividing over the y-axis (vertical division), and the bottom four graphs show the iterations starting with dividing over the x-axis (horizontal division). As it is shown, for both implementations there was a maximum of four iterations. The graphs show that after two iterations the messages are distributed exactly the same, regardless of the axis on which the implementation started with dividing. Iteration 4 starting with dividing over the x-axis results in the same distribution as unmodified Geohash-3. It is also shown that after two iterations not much improvement is made: the partition with the highest number of messages barely decreases while new partitions consist of a very low amount of messages, increasing the difference between the largest and smallest partition.

In addition to Figure 6.9 the distribution of the messages is presented in Figure 6.10 using Tukey boxplots for $p \geq 5$ (iterations with a lower amount of partitions are omitted due too small sample size [34]). As described above not much improvement is made after the second iteration, which is the last iteration with less than five partitions. The observation that the addition of smaller partitions has barely impact on the larger partitions, made in the discussion of Figure 6.9, can also be seen in the box plots, which indicate skewed data to the larger partitions when the number of partitions increases.

Although Figures 6.9 and 6.10 show the progress of the distribution of the amount of messages when the number of partitions is increased, Geohash-3 remains restricted by the number of parti-

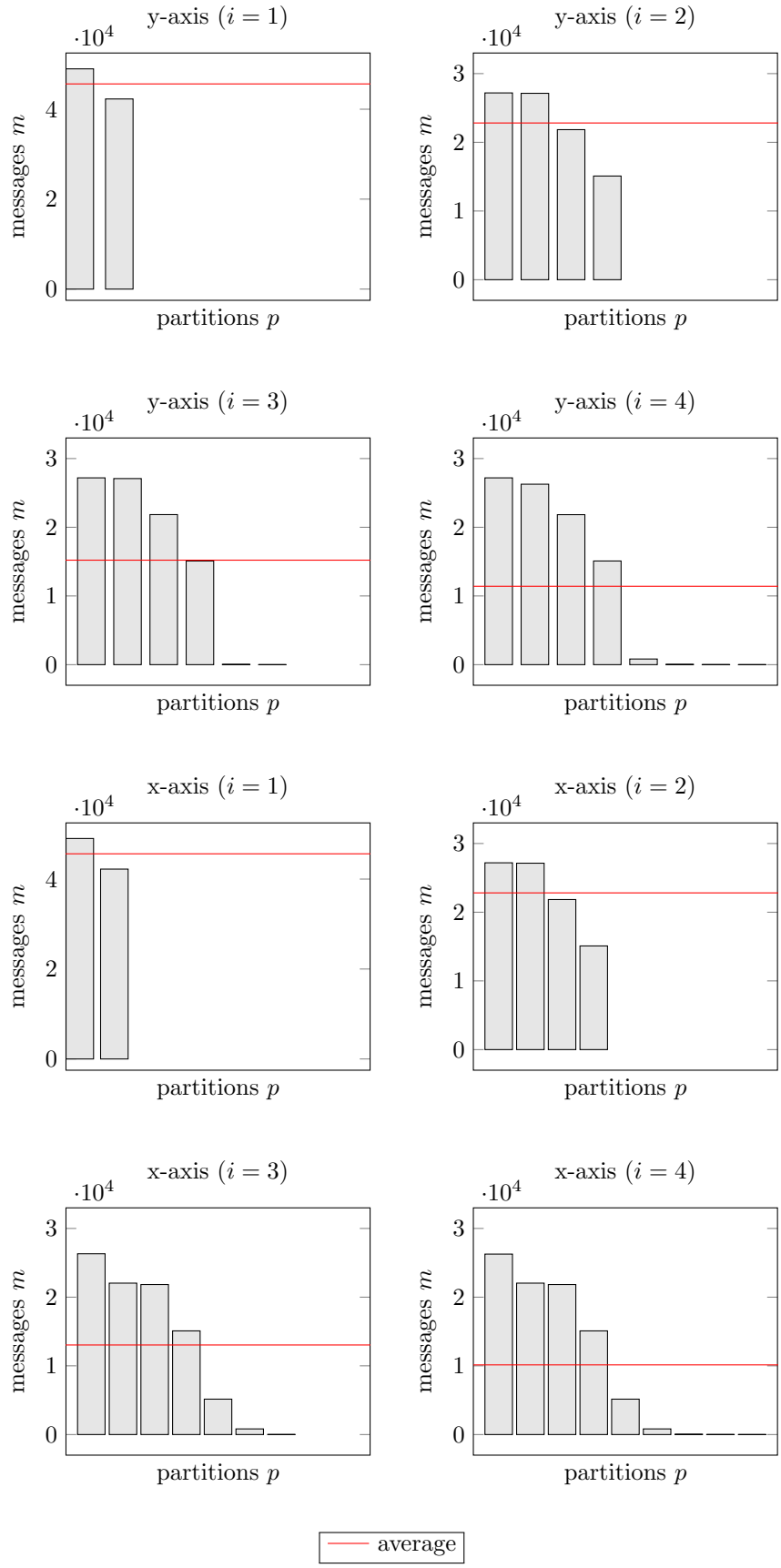



Figure 6.9: Barplots of Geohash-3 weighted kd-tree starting over y-axis (top four) and x-axis (bottom four), indicated by the number of total iterations $i$.

tions it has at maximum. Since the results of the experiments performed with Geohash-3 were not in the same range as Geohash-4 and Voronoi because of its limited number of partitions, we omit Geohash-3 in the results for the remainder of this thesis.

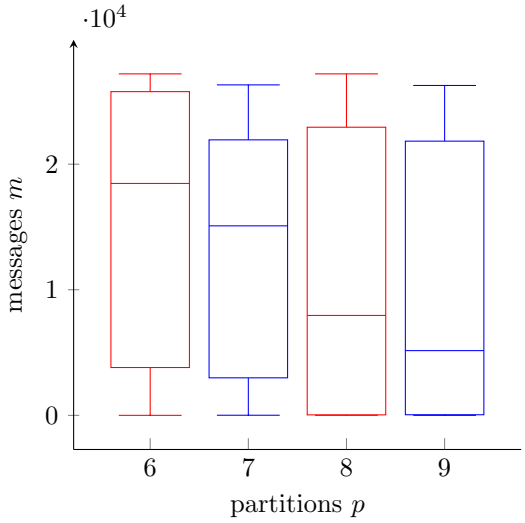



Figure 6.10: Distribution of the messages over the partitions using Geohash-3 weighted kd-tree with starting dividing over y-axis (red) and x-axis (blue).

#### Geohash-4

Figure 6.11 shows the amount of messages per partition when using Geohash-4. As can be seen most partitions have a lower than average (754 messages) amount of messages stored on them, as indicated by the red line ——. In total five partitions do not have any messages stored on them. In the figure a high peak can be observed on the left side. The corresponding partition has almost a third more messages than the next partition: 3301 against 2482. This peak can also be observed as most extreme outlier in the corresponding boxplot, which is showed in both graphs in red in Figure 6.12. The partition corresponding to this peak covers large parts of Rotterdam, the second largest city in the Netherlands, which explains why it is among the largest partitions. Recall that the baseline method has all 91.256 messages on its single partition - more than 27 times more than this peak.

Figure 6.12 shows the distribution of the messages over the partitions using Geohash-4 weighted kd-tree. As can be seen the boxes indicate an exponential decay, especially in the first iterations. This can be explained to the algorithm itself that aims to divide each current partition into half. From the figure it can be concluded that after five to six iterations (around 60 partitions) the distribution does not improve necessarily. It could be argued that it gets even worse with the increase of outliers. Note that the data is a little bit skewed to the larger partitions when the number of partitions increases.

In Figure 6.13 the average absolute difference in amount of messages per partition to the average amount can be found. In the figure an exponential decay in differences can be observed for both implementations of Geohash-4. Geohash-4 (y) has a lower difference than Geohash-4 (x) from $p = 30$ onwards. It can also be observed that the difference increases towards the last iterations. This is also indicated by the higher difference for unmodified Geohash-4.

Figure 6.14 shows the relative difference to the average amount of messages per partition. An increasing line for both implementations can be seen, with again a lower difference for Geohash-4 (y). Both implementations end around the same percentage as the unmodified Geohash-4.

### 6.3.3 Discussion

As described in the introduction of this section the goal of our experiments on partition size is to investigate the uniformity of the distribution of the messages of the partitions, i.e. the uniformity of the partition sizes. A uniform distribution has several advantages, such as load balancing an predictability. Recall that the baseline method consists of a single partition containing all 91.256

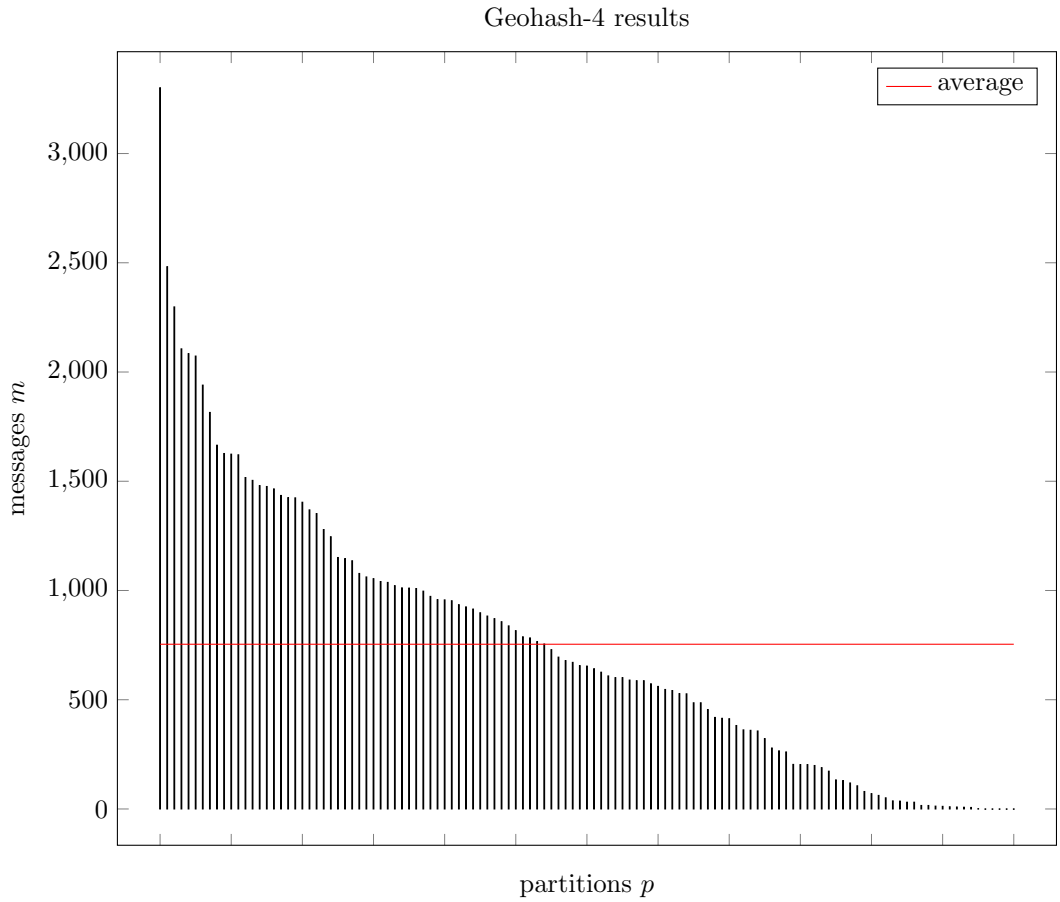



Figure 6.11: Geohash-4 results per partition, combined with the average per partition.

messages. Since a single partition is always uniform with itself, one could argue that the baseline method has an optimal uniformity. However, by definition division of work load requires multiple instances. Therefore, we do not consider the baseline method better because it has a higher uniformity than partitioning methods by default. We consider a partitioning method with a high uniformity on partition size better in terms of dividing the work load, although it should be noted that such a method may have drawbacks in other areas to reach a high uniform distribution.

In Sections 6.3.1 and 6.3.2 the results of the different implementations of the Geohash and Voronoi models were presented and discussed. Those results were presented in bar graphs, box plots, and line graphs. Recall that in bar graphs an equal distribution is indicated by bars of the same height, i.e. the smaller the height differences the more equal the distribution. Recall that in box plot graphs a smaller box plot represents a more equal distribution; a completely equal distribution should result in a box plot where the median, first quartile, third quartile, and whiskers are on the same height. Recall that line graphs are used to show the average difference between each partition size and the average partition size of all partitions. A lower line represents a lower difference and thus a more equal distribution.

For the Voronoi cities implementation PopDesc an initial decrease in differences between the largest and smallest partitions can be seen as well. For this implementation the differences steady from $p = 60$. However, the box plots still change after that point onwards with a lowering upper whisker and the addition of a weak outlier. For the Voronoi cities implementation PopAsc the box plot is less consistent, although it has a decrease in differences between the largest and smallest partitions while the position of the box plots lowers as well.

In Figure 6.9 we presented the progress of Geohash-3 weighted kd-tree as the number of partitions increases. In the graphs it can be seen that the differences between the largest and smallest partitions increase or stay the same with each iteration, except between $i = 3$ and $i = 4$ when starting dividing over the x-axis, when it decreases from 26.306 to 26.267 messages difference. This is probably due to the low number of partitions. Because of this restriction and the indication that

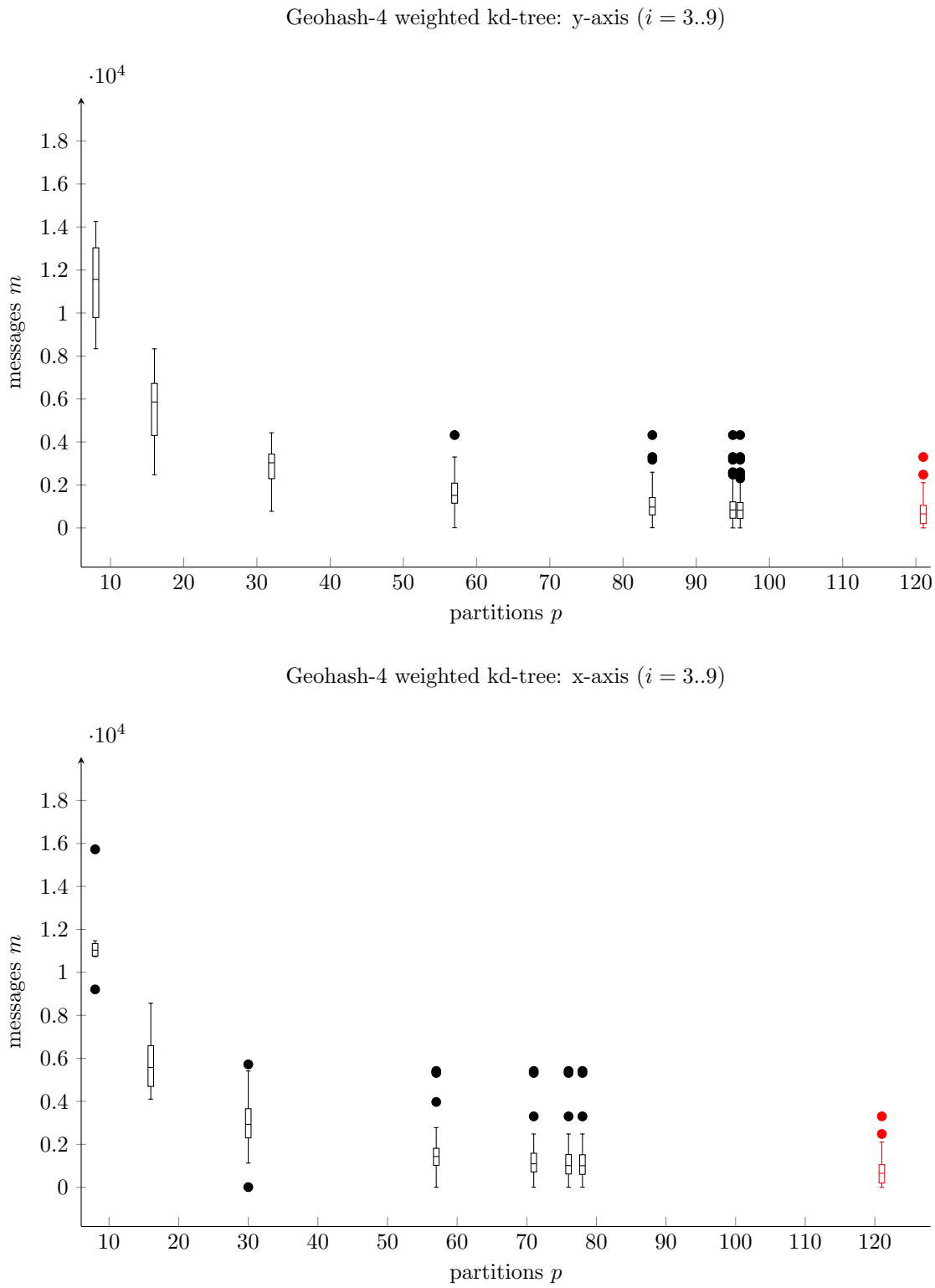



Figure 6.12: Boxplots of the message distribution over the partitions using Geohash-4 weighted kd-tree starting over y-axis (top) and x-axis (bottom). In red the boxplot of unmodified Geohash-4 is shown for comparison.

it has results that are not in the same range as Geohash-4 and Voronoi, we have decided to omit Geohash-3 in the remainder of this thesis.

When look to the differences between the largest and smallest partitions of the different Geohash-4 implementations, we see a decrease in the first iterations, whereafter the differences are not changes from $i = 6$. This can also be observed in the box plots corresponding to these implementations: from $i = 6$ the box plots do not change that much.

In Figures 6.13 and 6.14 the average difference to the average amount of messages per partition is shown for both Voronoi and Geohash. It clearly shows that Geohash has lower differences that Voronoi until $p = 60$. For Voronoi the relative difference is more or less stable from $p = 40$ onwards, while Geohash shows the stability in the absolute differences from the same amount of partitions.

All considered we argue that a larger number of partitions has a lower difference between the largest and smallest partition than a smaller number of partitions. In addition, the absolute difference to the average amount of messages per partition generally decreases when increasing the number of partitions. The relative difference differs per method, where Voronoi starts high before stabilizing and Geohash has a steady increasing progress.

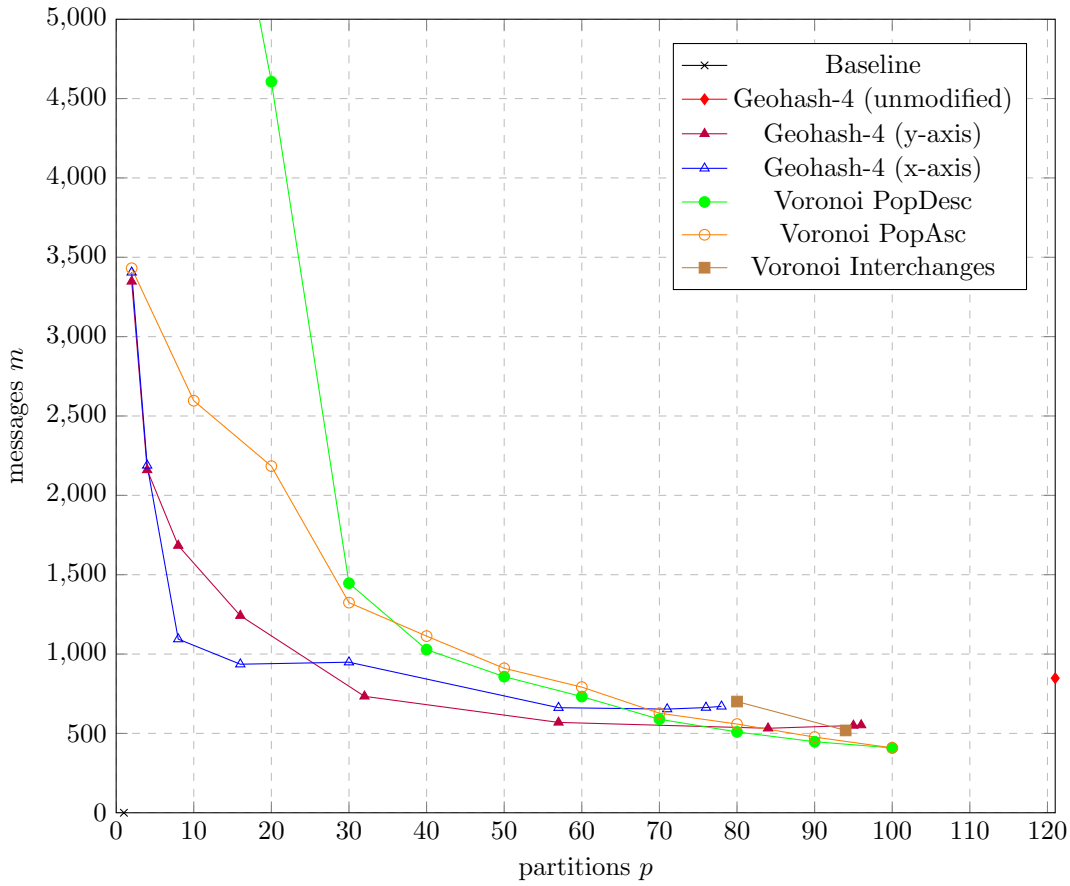



Figure 6.13: Average difference to the average amount of messages per partition.

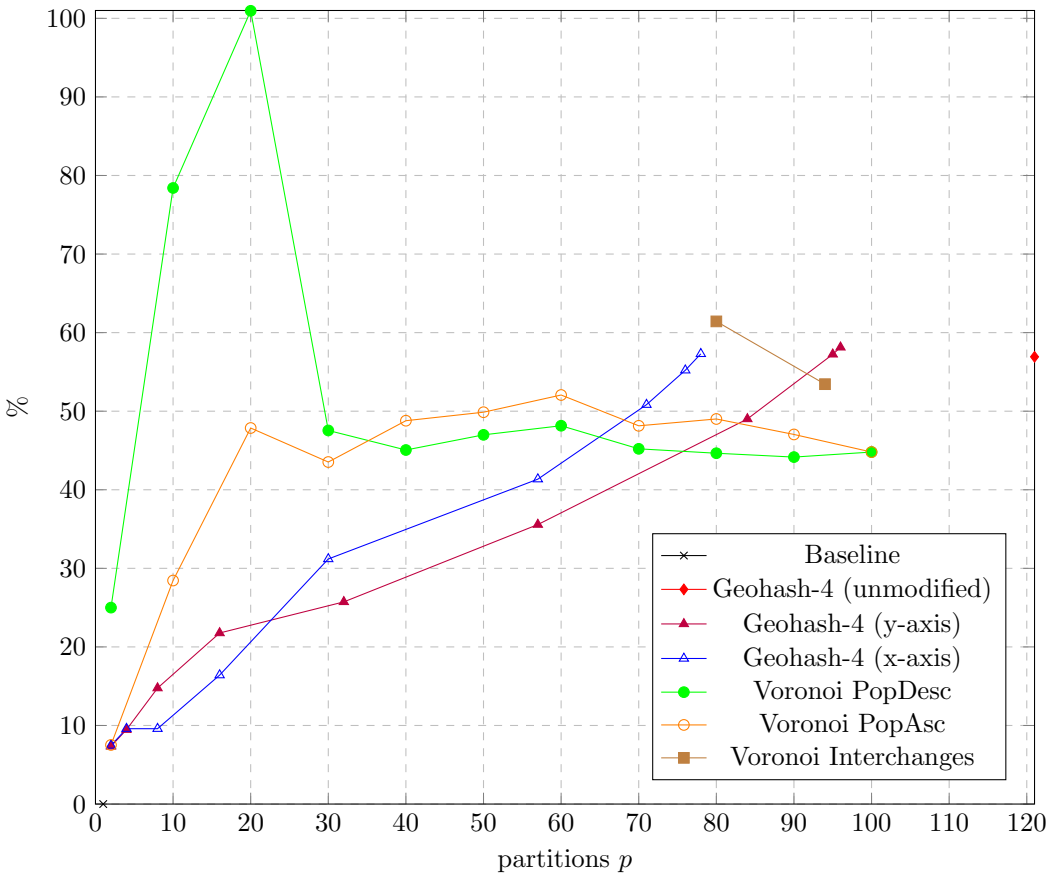



Figure 6.14: Relative average difference to the average amount of messages per partition.

## 6.4 Experiments on WMS requests

In this section we discuss our experiments regarding Web Map Service (WMS) requests. WMS requests were identified as a suitable use case in Section 2.3.1, in which they are used for data visualization on a map. Recall that WMS requests define a square window for the desired area and that this window is intersected with the partition area to retrieve the corresponding data. In our experiments we investigate how the different implementations of the Voronoi and Geohash models perform when WMS requests are used to retrieve data and how their performances compare to the baseline method. In the first section we discuss the results regarding the number of messages sent, which indicates the amount of data is sent. Thereafter, we investigate the performance of the methods regarding how many messages are actually useful, i.e. fall within the WMS window.

### 6.4.1 Messages per WMS request

In this experiment we investigate the data traffic between the Kafka cluster and consumer by determining the amount of messages sent per WMS request. The average amount of messages retrieved from the cluster per WMS request is shown in Figure 6.15. For comparison we added a line for the average amount of messages stored per partition. This is the ideal partitioning method that requires to retrieve messages from only one partition at most. Since all implementations sent more messages per WMS request than stored per partition (on average), it can be seen that on average a WMS request accesses more than one partition to retrieve messages from, up to five partitions per request on average for the highest number of partitions.

In addition the results of the baseline method are added for comparison between the baseline and the implementations of the Voronoi and Geohash models. Note that the baseline method sends less messages than the ideal partitioning. This is due to WMS requests covering an area that falls outside the area covered by the partitions, thus no messages are sent for these requests.

The figure shows a clear exponential decay for the Voronoi cities implementations. From $p = 70$ onwards, both cities implementations show nearly equivalent results. Note that the sets of both implementations become more equivalent when the sets become larger, i.e. when the number of partitions increases.

The Voronoi interchanges implementations show very similar results compared to their cities counterparts when using the the same number of partitions.

Geohash-4 shows a similar behaviour of exponential decay compared to Voronoi. The Geohash-4 unmodified implementation has the lowest amount of messages sent on average, albeit it is not much improvement from either implementation due to the nature of exponential decay.

When comparing all Voronoi and Geohash implementations we can conclude the following. Both implementations show that increasing the amount of partitions lowers the amount of messages sent. For a low number of partitions ($p \leq 30$) Geohash-4 results in the lowest amount of messages sent. When the number of partitions increases, the differences between Geohash-4 and Voronoi implementations are neglectable.

### 6.4.2 Relevant messages per WMS request

In this experiment we investigate the amount of relevant messages that are received per WMS request. By relevant message we mean a message that falls within the requested WMS window and thus are relevant for the map visualization. As shown in Figures 4.4 and 5.4 during the process of retrieving messages for a WMS request, all messages of the partitions the WMS request window overlaps are retrieved. The relevant messages are only those which are used in the generated map image, as displayed in (c) in both images.. Other messages that are received due to being in a partition that intersects with the WMS window, are referred to as junk messages. This filtering is performed at the subscriber's side after it has received all messages from the broker. Thus, such a filtering method is indifferent for Voronoi, Geohash, or any other spatial partitioning method.

The results of this experiment for all implementations can be found in Figure 6.16. The figure shows that each implementation has a higher amount of relevant messages in comparison to the currently used baseline method, which results in 2,1% relevant messages.

It can be observed that the Voronoi implementations result in a higher percentage as the number of partitions grows, with a rapid increase before it flattens out. However, Geohash-4 implementations seem to reach a maximum between 9 and 10% before it reaches the maximum amount of partitions of their unmodified implementation, which has a lower percentage of relevant

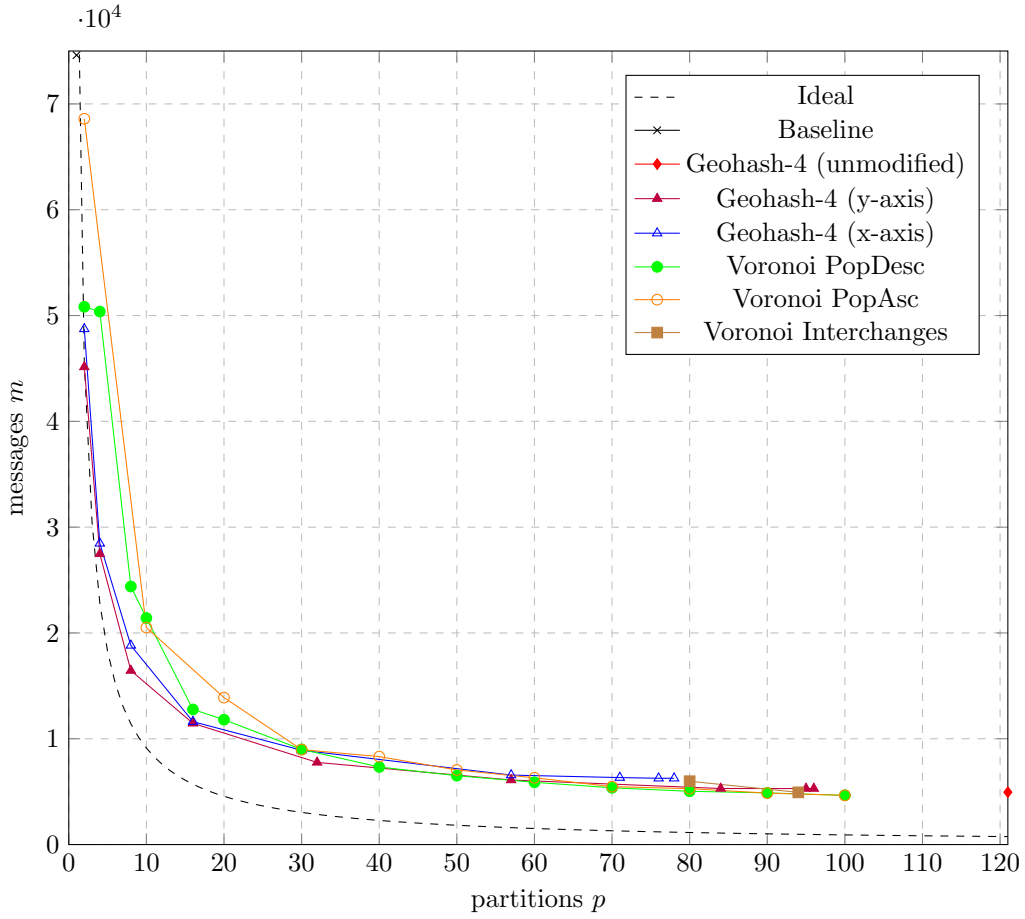



Figure 6.15: Average amount of messages received per WMS request for the implementations.

messages. Also noteworthy is that the first iteration of the Voronoi cities implementations is very close to the baseline method, despite containing two partitions instead of one. The Voronoi interchanges implementation achieves even higher percentage of relevant messages.

In comparison to Geohash-4, the Voronoi implementations score higher percentages overall. Their results are more predictable and do not contain a limit in halfway. They also pass the 10% relevant message, up to five times the efficiency of the baseline method for the interchange implementation.

## 6.5 Conclusions

In Chapter 2 we discussed a case study concerning Simacan's Connect 2 with multiple use cases for spatial data. During this case study we found that partitioning may improve load balancing when the partition sizes are uniform. The corresponding experiment in which we evaluated the performance of the individual implementations with regards to the uniformity of the partition sizes, can be found in Section 6.3. As mentioned, a single partition such as defined by the baseline method would be the optimal situation regarding uniformity (since a single partition is uniform with itself). However, since dividing work load requires multiple instance to divide work load over, other partitioning methods are very much relevant and thus compared to the optimum of uniformity of the partition sizes. When comparing the individual implementations of Voronoi and Geohash to this optimum, Geohash-4 seemed to achieve the best results with lower partition sizes with lower variation in size and with smaller differences to the average partition size than Voronoi at the same number of partitions (for $p \leq 30$). It should be noted that Geohash-3 was limited by the maximum amount of partitions in comparison to the other implementations and was therefore omitted in any other experiments.

From the presented use cases in Section 2.3 we decided to use WMS requests for our experiments, since the map visualization use case is not only useful to Simacan and their customers, but also

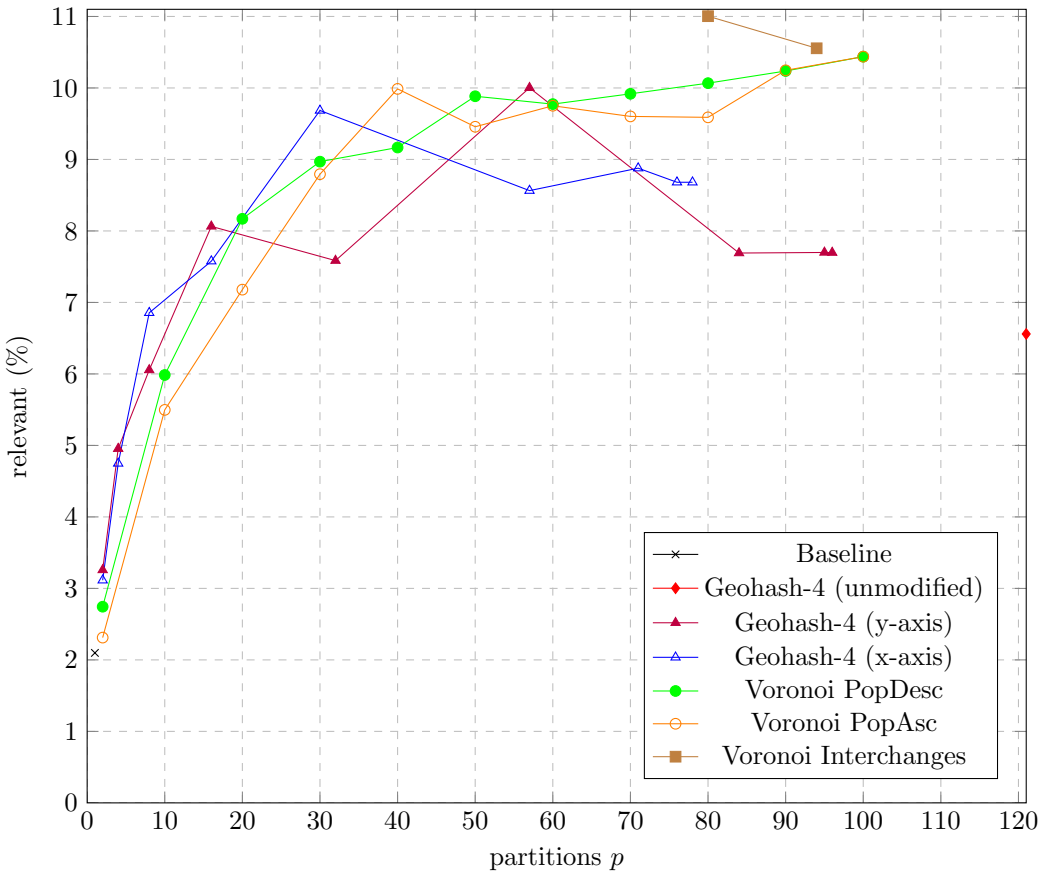



Figure 6.16: Percentage relevant messages per WMS request for the different implementations.

to other map visualization services. In Section 6.4 we evaluated the performance of the individual implementations of Voronoi and Geohash with regards to WMS requests. During the experiments, regarding both those on average amount of sent messages and percentage useful messages sent, all implementations showed better results than the baseline method. We also can conclude from these results that the input data consisted of WMS requests that did not cover the Netherlands as a whole, since not all 91.256 messages were sent for each WMS request. When comparing the implementations to each other, Geohash-4 and Voronoi show a similar decreasing line for the average amount of messages sent per WMS request from around 30 partitions onwards. During the lower partition amounts Geohash-4 sends less messages. In comparison, the baseline method sent 74.606 messages on average.

Geohash-4 and Voronoi showed very different lines for the results of the experiments on useful messages. Geohash-4 has again better results in the lower partition amounts with higher percentages of useful messages. However, it reaches a limit before declining again, with a limit that is lower than Voronoi achieves in the long run. Voronoi cities can achieve almost five times the efficiency of the baseline method (10,4% to 2,1%), while Voronoi interchanges A-only even has 11,1% useful messages.

# Chapter 7

# Conclusions & Future Work

During our research we were able to find an answer to all the research questions that were stated in Section 1.4. We first summarize the answer to each subquestion before answering the main research question.

*How can spatial partitioning be applied in publish-subscribe messaging systems?*

As described in Chapter 3, research has been done to spatial partitioning of the publishers and subscribers, based on their locations. However, we found that the spatial data itself that is sent or received by publishers and subscribers can also be partitioned on an intermediary broker. This can be done by defining a spatial partitioning method that is used by the publishers and subscribers to determine the partition on a broker to which data is sent or from which data is retrieved. Thus, the partition method has to be known by both the publisher and the subscriber.

*Which spatial partitioning methods exist that can be applied in publish-subscribe messaging systems?*

In Chapter 3 we presented several spatial partitioning methods, namely kd-tree, k-means, and Voronoi. We argued that of these three Voronoi was the best applicable for our research. A model of Voronoi that was used in our experiments was defined in Chapter 4. In Chapter 5 we proposed a new spatial partitioning method based on Geohash that can be applied in publish-subscribe messaging systems

*What is a suitable use case to evaluate applicable methods and how do these methods perform in this use case in comparison to a system without spatial partitioning?*

In Chapter 2 we presented a case study concerning Simacan's Connect 2 with multiple use cases for spatial data. From the presented use cases we decided to use WMS requests for one of our experiments, since the map visualization use case is relevant to Simacan, their customers, and other map visualization services.

The Voronoi and Geohash methods were evaluated in the corresponding experiment described in Section 6.4. In this experiment their performances in a publish-subscribe messaging system with regards to WMS requests were compared to a baseline method which did not apply any spatial partitioning in the system.

In this experiment Geohash-4 and Voronoi showed a similar decrease for the average amount of messages sent per WMS request while the number of partitions increased. For the lower partition amounts Geohash-4 had better results with lower amounts of messages sent. From halfway the maximum amount of partitions that Geohash-4 can have, Voronoi and Geohash-4 have comparable results. For the percentage relevant messages Geohash-4 shows better results for the lower partition amounts as well, but hits a limit before declining. Voronoi achieves higher efficiency in terms of relevant messages without a limit, having a percentage relevant messages that is five times that of the baseline method. However, Voronoi needs more partitions than Geohash-4 to achieve this percentage.

Both Voronoi and Geohash performed better than the baseline method of a system without

spatial partitioning: in all cases the evaluated methods retrieved lower amounts of messages and higher relative amounts of relevant messages.

We can now provide an answer to the main research question.

*How can spatial partitioning methods help to scale up publish-subscribe messaging systems for processing big spatial data?*

In this thesis we have shown that spatial partitioning can help the upscaling of publish-subscribe messaging systems for processing big spatial data. By applying an existing (Voronoi) and a proposed (Geohash) spatial partitioning method in a publish-subscribe messaging system we showed that these methods can help upscaling of publish-subscribe systems. This is accomplished by applying a spatial partitioning method to the publisher and subscriber of a publish-subscribe system, which allows them to determine the partition to which data needs to be sent to or retrieved from. Our experiments showed improvements in the areas of load balancing, transferred messages, and the amount of relevant messages when using spatial partitioning methods.

During our research we proposed a new spatial partitioning method based on Geohash. Our experiments show that Geohash was able to achieve better results than Voronoi in general. Only when the amount of partitions is more than half of what Geohash can define, Voronoi showed slightly better results.

## 7.1 Future Work

With this research we only have set a small step in the area of applying spatial partitioning in publish-subscribe messaging systems. During this research we have identified several directions for future work.

#### Other spatial partitioning methods

In Section 3.3 we described multiple spatial partitioning methods that can be applied to publish-subscribe messaging systems. Of these methods we decided to investigate the performance of only Voronoi in this research, but for future research the remaining methods are excellent candidates to evaluate as well to produce a complete overview of the performance and trade-offs of the different methods.

#### Experiments on routes

In Section 2.3 we presented several use cases, of which we decided to evaluate our methods on WMS requests for map visualization. However, another important aspect of Connect 2 is that information can be used for route based use cases. Therefore, for further research we highly recommend to consider use cases based on routes, since they may have other requirements.

#### Geohash variants

During our experiments Geohash-4 with weighted kd-tree seemed the best method overall. However, even within this Geohash variant other directions can be taken. In our Geohash model we implemented weighted kd-tree as improvement, but for future work other techniques could be considered, e.g. considering all geohashes as nodes in a graph and finding subgraphs of the same weight. In addition, Geohash-5 or higher can be considered, depending on the area, for a more complete picture how Geohash (weighted kd-tree) behaves as a spatial partitioning method.

#### Processing time

During the execution of our experiments we noticed differences in processing times between the different amounts of partitions. Especially when a low amount of partitions were used, the processing time was generally higher. Because our setup was not controlled enough for time related experiments, we could not measure these times with enough confidence. However, we believe that processing time should be considered when choosing a partitioning method. We recommend such

an experiment to be performed on a controlled environment, such as a cloud infrastructure, and multiple times to even out any mismeasurements.

### Multi-dimensional Geohash

In this thesis we presented Geohash as a spatial partitioning method for two-dimensional space. However, other mentioned spatial partitioning methods are also usable in multi-dimensional space. As future work we propose to investigate if Geohash can be used for partitioning multi-dimensional space, and if so, how it performs compared to existing methods.

# Appendices

# Appendix A

# Implementation Geohash

```java
package com.simacan.kafka;

import org.apache.commons.lang.ArrayUtils;

/**
 * Encode geographical coordinates to Geohash.
 */
public class GeohashEncoder {

  /**
   * The Constant char map BASE_32.
   */
  private final static char[] BASE_32 = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'j', 'k', 'm', 'n', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z' };

  /** The bits. */
  private static final int[] bits = { 16, 8, 4, 2, 1 };

  /**
   * Encodes the given latitude and longitude into a geohash code with given
   * precision.
   *
   * @param latitude
   *            the latitude
   * @param longitude
   *            the longitude
   * @return The generated geohash from the given latitude and longitude.
   */
  public String encode(final int precision, final double latitude, final double longitude) {
    final StringBuilder geohash = new StringBuilder();
    boolean even = true;
    int bit = 0;
    int ch = 0;

    final double[] latitudeInterval = { -90.0, 90.0 };
    final double[] longitudeInterval = { -180.0, 180.0 };

    while (geohash.length() < precision) {
      double mid = 0.0;
      if (even) {
        mid = (longitudeInterval[0] + longitudeInterval[1]) / 2D;
        if (longitude > mid) {
          ch |= bits[bit];
          longitudeInterval[0] = mid;
        } else {
          longitudeInterval[1] = mid;
        }

      } else {
        mid = (latitudeInterval[0] + latitudeInterval[1]) / 2D;
        if (latitude > mid) {
          ch |= bits[bit];
```

```
52          latitudeInterval[0] = mid;
          } else {
54          latitudeInterval[1] = mid;
          }
56        }

58        even = !even;

60        if (bit < 4) {
            bit++;
62        } else {
            geohash.append(BASE_32[ch]);
64          bit = 0;
            ch = 0;
66        }
      }
68
      return geohash.toString();
70  }

72  /**
     *
74   * @param geohash
     * @return latitude, longitude
76   */
    public double[] decode(String geohash) {
78    boolean even = true;
      double[] latitude = { -90.0, 90.0 };
80    double[] longitude = { -180.0, 180.0 };
      double latitudeError = 90.0;
82    double longitudeError = 180.0;

84    for (int i = 0; i < geohash.length(); i++) {
        char c = geohash.charAt(i);
86      int cd = ArrayUtils.indexOf(BASE_32, c);
        for (int j = 0; j < 5; j++) {
88        int mask = bits[j];
          if (even) {
90          longitudeError /= 2.0;
            longitude = refine_interval(longitude, cd, mask);
92        } else {
            latitudeError /= 2.0;
94          latitude = refine_interval(latitude, cd, mask);
          }
96        even = !even;
        }
98    }
      double lat = Math.abs((latitude[0] + latitude[1]) / 2.0);
100   double lon = Math.abs((longitude[0] + longitude[1]) / 2.0);

102   return new double[] { lat, lon };
    }
104
    /**
106  * Added for decoding
     *
108  * @param interval
     * @param cd
110  * @param mask
     * @return
112  */
    private double[] refine_interval(double[] interval, int cd, int mask) {
114   int bitwise = cd & mask;
      double[] result = interval;
116   if (bitwise == 0)
        result[0] = (interval[0] + interval[1]) / 2;
118   else
        result[1] = (interval[0] + interval[1]) / 2;
120   return result;
    }
122
}
```

Listing A.1: Implementation of Geohash encoding and decoding algorithm, based on [35].

# Appendix B

# Resulting partitions of Geohash

## B.1 Geohash-3

| | | | |
|---|---|---|---|
| | u1e | u1s | |
| u16 | u17 | u1k | u1m |
| u14 | u15 | u1h | |

Table B.1: Mapping of the 9 partitions created by the unmodified Geohash-3 partitioning method when applied to the Netherlands

## B.2 Geohash-4

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | u1sc | |
| | | | | | | u1eb | u1s0 | u1s2 | u1s8 | us1b | |
| | | | | | u17x | u17z | u1kp | u1kr | u1kx | u1kz | u1mp |
| | | | | u17q | u17w | u17y | u1kn | u1kq | u1kw | u1ky | u1mn |
| | | | | u17m | u17t | u17v | u1kj | u1km | u1kt | u1kv | u1mj |
| | | | | u17k | u17s | u17u | u1kh | u1kk | u1ks | u1ku | u1mh |
| | | | | u177 | u17e | u17g | u1k5 | u1k7 | u1ke | u1kg | u1m5 |
| | | | u174 | u176 | u17d | u17f | u1k4 | u1k6 | u1kd | u1kf | u1m4 |
| | | | u171 | u173 | u179 | u17c | u1k1 | u1k3 | u1k9 | u1kc | u1m1 |
| | | u16b | u170 | u172 | u178 | u17b | u1k0 | u1k2 | u1k8 | u1kb | |
| | | u14z | u15p | u15r | u15x | u15z | u1hp | u1hr | u1hx | u1hz | |
| | u14w | u14y | u15n | u15q | u15w | u15y | u1hn | u1hq | u1hw | | |
| u14m | u14t | u14v | u15j | u15m | u15t | u15v | u1hj | u1hm | | | |
| u14k | u14s | u14u | u15h | u15k | u15s | u15u | u1hh | u1hk | | | |
| u147 | u14e | u14g | | | u15e | u15g | u1h5 | u1h7 | | | |
| | | | | | | | u1h4 | u1h6 | | | |
| | | | | | | | u1h1 | u1h3 | | | |
| | | | | | | | u1h0 | u1h2 | | | |

Table B.2: Mapping of the 121 partitions created by the unmodified Geohash-4 partitioning method when applied to the Netherlands.

# Appendix C

# Voronoi sites

The following pages consist of the Voronoi sites chosen for the Voronoi cities and interchanges implementations.

| # PopDesc | # PopAsc | City | Pop. | Coordinates |
|---|---|---|---|---|
| 1 | 100 | Amsterdam | 840486 | 52.366667, 4.900000 |
| 2 | 99 | Rotterdam | 632388 | 51.916667, 4.500000 |
| 3 | 98 | Den Haag | 521107 | 52.095556, 4.316389 |
| 4 | 97 | Utrecht | 340153 | 52.088889, 5.115556 |
| 5 | 96 | Eindhoven | 225060 | 51.435556, 5.480556 |
| 6 | 95 | Tilburg | 213032 | 51.557222, 5.091111 |
| 7 | 94 | Groningen | 199860 | 53.216667, 6.566667 |
| 8 | 93 | Almere | 199479 | 52.376667, 5.216111 |
| 9 | 92 | Breda | 182394 | 51.587500, 4.775000 |
| 10 | 91 | Nijmegen | 172463 | 51.833333, 5.866667 |
| 11 | 90 | Apeldoorn | 159475 | 52.210000, 5.966111 |
| 12 | 89 | Haarlem | 158395 | 52.380278, 4.640556 |
| 13 | 88 | Enschede | 157848 | 52.219444, 6.896389 |
| 14 | 87 | Arnhem | 154815 | 51.983333, 5.916667 |
| 15 | 86 | Amersfoort | 153913 | 52.156389, 5.389722 |
| 16 | 85 | Zaandam | 152973 | 52.441667, 4.825556 |
| 17 | 84 | Den Bosch | 151948 | 51.710069, 5.311267 |
| 18 | 83 | Hoofddorp | 145103 | 52.303333, 4.668889 |
| 19 | 82 | Zwolle | 125241 | 52.516667, 6.083333 |
| 20 | 81 | Zoetermeer | 124437 | 52.059444, 4.495833 |
| 21 | 80 | Leiden | 122956 | 52.166667, 4.483333 |
| 22 | 79 | Maastricht | 122017 | 50.850000, 5.683333 |
| 23 | 78 | Dordrecht | 118442 | 51.816667, 4.666667 |
| 24 | 77 | Ede | 112872 | 52.050000, 5.666667 |
| 25 | 76 | Alphen a/d Rijn | 108142 | 52.133333, 4.666667 |
| 26 | 75 | Leeuwarden | 107959 | 53.200000, 5.783333 |
| 27 | 74 | Alkmaar | 107906 | 52.633333, 4.750000 |
| 28 | 73 | Emmen | 107614 | 52.783333, 6.900000 |
| 29 | 72 | Naaldwijk | 105202 | 51.993056, 4.205000 |
| 30 | 71 | Delft | 100774 | 52.016667, 4.366667 |
| 31 | 70 | Venlo | 100533 | 51.366667, 6.166667 |
| 32 | 69 | Deventer | 99214 | 52.250000, 6.150000 |
| 33 | 68 | Sittard | 93442 | 51.000000, 5.866667 |
| 34 | 67 | Oss | 90325 | 51.766667, 5.516667 |
| 35 | 66 | Helmond | 90219 | 51.483333, 5.650000 |
| 36 | 65 | Amstelveen | 88792 | 52.300000, 4.850000 |
| 37 | 64 | Hilversum | 88194 | 52.233333, 5.166667 |
| 38 | 63 | Heerlen | 87531 | 50.883333, 5.983333 |
| 39 | 62 | Spijkenisse | 85368 | 51.850000, 4.333333 |
| 40 | 61 | Sneek | 84036 | 53.033333, 5.666667 |
| 41 | 60 | Hengelo | 81126 | 52.266667, 6.800000 |
| 42 | 59 | Purmerend | 79912 | 52.500000, 4.950000 |
| 43 | 58 | Schiedam | 77238 | 51.916667, 4.400000 |
| 44 | 57 | Roosendaal | 77035 | 51.533333, 4.450000 |
| 45 | 56 | Lelystad | 76726 | 52.500000, 5.483333 |
| 46 | 55 | Leidschendam | 74311 | 52.083333, 4.400000 |
| 47 | 54 | Almelo | 72357 | 52.350000, 6.666667 |
| 48 | 53 | Hoorn | 72356 | 52.650000, 5.066667 |
| 49 | 52 | Vlaardingen | 71881 | 51.900000, 4.350000 |
| 50 | 51 | Gouda | 71557 | 52.016667, 4.716667 |

Table C.1: Voronoi Cities (1-50). The population is based on May 2016 [36].

| # PopDesc | # PopAsc | City | Pop. | Coordinates |
|---|---|---|---|---|
| 51 | 50 | IJmuiden | 67457 | 52.458611, 4.619444 |
| 52 | 49 | Assen | 67325 | 53.000000, 6.566667 |
| 53 | 48 | Capelle a/d IJssel | 66487 | 51.933333, 4.583333 |
| 54 | 47 | Bergen op Zoom | 66157 | 51.500000, 4.300000 |
| 55 | 46 | Katwijk | 64235 | 52.200000, 4.416667 |
| 56 | 45 | Maarssen | 64138 | 52.135144, 5.041297 |
| 57 | 44 | Veenendaal | 63953 | 52.016667, 5.550000 |
| 58 | 43 | Zeist | 62456 | 52.083333, 5.233333 |
| 59 | 42 | Nieuwegein | 61785 | 52.033333, 5.083333 |
| 60 | 41 | Hardenberg | 59898 | 52.566667, 6.616667 |
| 61 | 40 | Bergschenhoek | 59401 | 51.980000, 4.500000 |
| 62 | 39 | Roermond | 57197 | 51.200000, 5.983333 |
| 63 | 38 | Doetinchem | 56899 | 51.966667, 6.300000 |
| 64 | 37 | Bussum | 56816 | 52.283333, 5.166667 |
| 65 | 36 | Den Helder | 56416 | 52.933333, 4.750000 |
| 66 | 35 | Barneveld | 55702 | 52.133333, 5.583333 |
| 67 | 34 | Drachten | 55446 | 53.100000, 6.100000 |
| 68 | 33 | Hoogeveen | 55302 | 52.733333, 6.483333 |
| 69 | 32 | Krimpenerwaard | 54799 | 51.983333, 4.783333 |
| 70 | 31 | Terneuzen | 54724 | 51.333333, 3.833333 |
| 71 | 30 | Heerhugowaard | 54628 | 52.666667, 4.833333 |
| 72 | 29 | Oosterhout | 54156 | 51.633333, 4.866667 |
| 73 | 28 | Kampen | 52168 | 52.550000, 5.916667 |
| 74 | 27 | Pijnacker | 52150 | 52.016667, 4.433333 |
| 75 | 26 | Woerden | 51357 | 52.083333, 4.883333 |
| 76 | 25 | Joure | 51308 | 52.966667, 5.783333 |
| 77 | 24 | Heerenveen | 50165 | 52.966667, 5.916667 |
| 78 | 23 | Rijswijk | 49842 | 52.033333, 4.316667 |
| 79 | 22 | Weert | 49642 | 51.250000, 5.700000 |
| 80 | 21 | Houten | 48894 | 52.033333, 5.166667 |
| 81 | 20 | Doorn | 48721 | 52.033333, 5.350000 |
| 82 | 19 | Middelharnis | 48479 | 51.750000, 4.166667 |
| 83 | 18 | Middelburg | 47992 | 51.500000, 3.616667 |
| 84 | 17 | Barendrecht | 47876 | 51.850000, 4.533333 |
| 85 | 16 | Anna Paulowna | 47433 | 52.870000, 4.870000 |
| 86 | 15 | Waalwijk | 47143 | 51.683333, 5.066667 |
| 87 | 14 | Overbetuwe | 47139 | 51.916667, 5.850000 |
| 88 | 13 | Zutphen | 46990 | 52.133333, 6.200000 |
| 89 | 12 | Emmeloord | 46625 | 52.709722, 5.750833 |
| 90 | 11 | Schagen | 46201 | 52.783333, 4.800000 |
| 91 | 10 | Harderwijk | 46091 | 52.350000, 5.616667 |
| 92 | 9 | Lingewaard | 46079 | 51.883333, 5.900000 |
| 93 | 8 | Kerkrade | 45920 | 50.866667, 6.066667 |
| 94 | 7 | Soest | 45541 | 52.183333, 5.300000 |
| 95 | 6 | Ridderkerk | 45147 | 51.866667, 4.600000 |
| 96 | 5 | Zwijndrecht | 44437 | 51.816667, 4.650000 |
| 97 | 4 | Veldhoven | 44417 | 51.416667, 5.400000 |
| 98 | 3 | Vlissingen | 44404 | 51.450000, 3.566667 |
| 99 | 2 | Berkelland | 44318 | 52.116667, 6.533333 |
| 100 | 1 | Medemblik | 43854 | 52.683333, 5.033333 |

Table C.2: Voronoi Cities (51-100). The population is based on May 2016 [36].

| Interchange | Roads | Coordinates |
|---|---|---|
| Almere | A6 / A27 | 52.381547, 5.299386 |
| Amstel | A2 / A10 / S110 | 52.327786, 4.913978 |
| Azelo | A1 / A35 | 52.311078, 6.697289 |
| Badhoevedorp | A4 / A9 | 52.328611, 4.795278 |
| Bankhoef | A50 / A326 | 51.814839, 5.679367 |
| Batadorp | A2 / A58 | 51.482559, 5.403042 |
| Beekbergen | A1 / A50 | 52.187615, 6.022911 |
| Benelux | A4 / A15 | 51.873108, 4.386789 |
| Beverwijk | A9 / A22 | 52.485122, 4.691533 |
| Buren | A1 / A35 | 52.285514, 6.743228 |
| Burgerveen | A4 / A44 / N207 | 52.233756, 4.648872 |
| Coenplein | A5 / A8 / A10 | 52.423611, 4.877222 |
| De Baars | A58 / A65 | 51.552178, 5.124839 |
| De Hoek | A4 / A5 / N201 | 52.290417, 4.729958 |
| De Hogt | A2 / A67 | 51.404836, 5.433322 |
| De Nieuwe Meer | A4 / A10 | 52.337836, 4.840539 |
| De Poel | A58 / A256 | 51.482811, 3.870119 |
| De Stok | A17 / A58 | 51.529992, 4.430964 |
| Deil | A2 / A15 | 51.857053, 5.228306 |
| Diemen | A1 / A9 | 52.333333, 5.003611 |
| Eemnes | A1 / A27 | 52.231975, 5.238842 |
| Ekkerswijer | A2 / A50 | 51.491703, 5.431014 |
| Empel | A2 / A59 | 51.723908, 5.324628 |
| Everdingen | A2 / A27 | 51.976210, 5.101776 |
| Ewijk | A50 / A73 / N322 | 51.858161, 5.711317 |
| Galder | A16 / A58 | 51.526028, 4.744469 |
| Gorinchem | A15 / A27 | 51.846667, 4.950278 |
| Gouwe | A12 / A20 | 52.022222, 4.654167 |
| Grijsoord | A12 / A50 | 52.025278, 5.818889 |
| Hattemerbroek | A28 / A50 / N50 | 52.486325, 6.023761 |
| Heerenveen | A7 / A32 | 52.975000, 5.916111 |
| Hellegatsplein | A29 / N59 / A59 | 51.701944, 4.391389 |
| Het Vonderen | A2 / A73 | 51.121944, 5.881944 |
| Hintham | A2 / A59 | 51.698631, 5.352144 |
| Hoevelaken | A1 / A28 | 52.175614, 5.427589 |
| Holendrecht | A2 / A9 | 52.289163, 4.944878 |
| Hoogeveen | A28 / A37 / N48 | 52.707778, 6.458056 |
| Hooipolder | A27 / A59 | 51.687264, 4.883839 |
| Joure | A6 / A7 / Geert Knolweg | 52.957778, 5.815833 |
| Julianaplein | A7 / A28 / N7 / Emmaviaduct | 53.203056, 6.564444 |
| Kerensheide | A2 / A76 | 50.967400, 5.787194 |
| Kethelplein | A4 / A20 | 51.925556, 4.365556 |
| Klaverpolder | A16 / A17 | 51.700014, 4.649522 |
| Kleinpolderplein | A13 / A20 | 51.930877, 4.438734 |
| Kruisdonk | A2 / A79 | 50.870761, 5.730056 |
| Kunderberg | A76 / A79 | 50.876111, 5.955000 |
| Lankhorst | A28 / A32 | 52.666667, 6.197778 |
| Leenderheide | A2 / A67 / Leenderweg | 51.404953, 5.502608 |
| Lunetten | A12 / A27 | 52.055000, 5.144444 |
| Maanderbroek | A12 / A30 | 52.025942, 5.611631 |

Table C.3: Voronoi Interchanges A-only (1-50).

| Interchange | Roads | Coordinates |
|---|---|---|
| Markiezaat | A4 / A58 | 51.431419, 4.277367 |
| Muiderberg | A1 / A6 | 52.315194, 5.104178 |
| Noordhoek | A17 / A59 | 51.636413, 4.545035 |
| Oud-Dijk | A12 / A18 | 51.914167, 6.140833 |
| Oudenrijn | A2 / A12 | 52.065578, 5.070190 |
| Paalgraven | A50 / A59 | 51.737083, 5.578933 |
| Princeville | A16 / A58 | 51.581450, 4.722858 |
| Prins Clausplein | A4 / A12 | 52.062180, 4.375742 |
| Raasdorp | A5 / A9 | 52.356944, 4.740278 |
| Ressen | A15 / A325 | 51.896706, 5.866994 |
| Ridderkerk | A15 / A16 / A38 | 51.866667, 4.583333 |
| Rijkevoort | A73 / A77 | 51.666944, 5.901667 |
| Rijnsweerd | A27 / A28 | 52.091944, 5.161111 |
| Rottepolderplein | A9 / A200 / N205 | 52.378333, 4.708333 |
| Sabina | A29 / A59 | 51.660022, 4.419144 |
| Sint-Annabosch | A27 / A58 | 51.550800, 4.830592 |
| Terbregseplein | A16 / A20 | 51.950719, 4.529514 |
| Tiglia | A73 / A74 | 51.344544, 6.158122 |
| Vaanplein | A15 / A29 / Vaanweg | 51.865450, 4.515492 |
| Valburg | A15 / A50 | 51.908167, 5.767839 |
| Velperbroek | A12 / A348 / N325 | 51.981944, 5.978611 |
| Velsen | A9 / A22 | 52.440833, 4.668333 |
| Vught | A2 / A65 | 51.668978, 5.310944 |
| Waterberg | A12 / A50 | 52.016111, 5.922778 |
| Watergraafsmeer | A1 / A10 | 52.351069, 4.963750 |
| Ypenburg | A4 / A13 | 52.044045, 4.349899 |
| Zaandam | A7 / A8 | 52.455661, 4.841536 |
| Zaarderheiken | A67 / A73 | 51.393611, 6.128056 |
| Zonzeel | A16 / A59 | 51.645281, 4.692056 |
| Zoomland | A4 / A58 / Wouwsestraatweg | 51.498747, 4.315464 |

Table C.4: Voronoi Interchanges A-only (50-80).

| Interchange | Roads | Coordinates |
|---|---|---|
| Assen | A28 / N33 | 52.971206, 6.543861 |
| Bodegraven | A4 / N11 | 52.067694, 4.746083 |
| Emmeloord | A6 / N50 | 52.692778, 5.774722 |
| Euvelgunne | A7 / N46 | 53.217500, 6.616111 |
| Gooimeer | A6 / N702 | 52.342221, 5.174489 |
| Holsloot | A37 / N34 | 52.727778, 6.788889 |
| Kooimeer | A9 / N9 / N242 | 52.614801, 4.740880 |
| Neerbosch | A73 / rw 783 Neerbosscheweg | 51.842222, 5.779722 |
| Stelleplas | A58 / N62/Sloeweg | 51.482239, 3.653683 |
| Ten Esschen | A76 / N281 | 50.905556, 5.924722 |
| Werpsterhoek | N31 / N32 | 53.158783, 5.791669 |
| Westerlee | A20 / N213 / N223 | 51.972077, 4.216457 |
| Zuidbroek | A7 / N33 | 53.170547, 6.893922 |
| Zurich | A7 / N31 | 53.096853, 5.382228 |

Table C.5: Additional interchanges for Voronoi Interchanges N-included.

# Bibliography


[1] Volker Gaede and Oliver Günther. Multidimensional access methods. ACM Computing Surveys (CSUR), 30(2):170–231, 1998.

[2] Ahmed Eldawy and Mohamed F Mokbel. The era of big spatial data. In Data Engineering Workshops (ICDEW), 2015 31st IEEE International Conference on, pages 42–49. IEEE, 2015.

[3] A. Aji, F. Wang, H. Vo, H. Lee, Q. Liu, X. Zhang, and J. Saltz. Hadoop gis: A high performance spatial data warehousing system over mapreduce. Proceedings of the VLDB Endowment, 6(11):1009–1020, 2013.

[4] Apache Kafka. http://kafka.apache.org.

[5] Apache ZooKeeper. https://zookeeper.apache.org/.

[6] Jeroen Soesbergen. Introduction to top location referencing methods. https://www.simacan.com/en/2015/10/02/location-referencing-introduction/, October 2015. Accessed: 11-02-2016.

[7] OpenLR. http://www.openlr.org/.

[8] Web Map Service. http://www.opengeospatial.org/standards/wms.

[9] Simacan Traffic Center. https://www.simacan.com/nl/producten/traffic-center/. Accessed: 19-04-2016.

[10] Michael P. Papazoglou. Web Services & SOA: Principles and Technology. Pearson Education, 2012.

[11] Shun-Yun Hu. Spatial publish subscribe. In Proc. of IEEE Virtual Reality (IEEE VR) workshop, Massively Multiuser Virtual Environment (MMVE'09), 2009.

[12] Shun-Yun Hu and Kuan-Ta Chen. Vso: Self-organizing spatial publish subscribe. In Self-Adaptive and Self-Organizing Systems (SASO), 2011 Fifth IEEE International Conference on, pages 21–30. IEEE, 2011.

[13] Xiaoyan Chen, Ying Chen, and Fangyan Rao. An efficient spatial publish/subscribe system for intelligent location-based services. In Proceedings of the 2nd international workshop on Distributed event-based systems, pages 1–6. ACM, 2003.

[14] Anil K Jain. Data clustering: 50 years beyond k-means. Pattern recognition letters, 31(8):651–666, 2010.

[15] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. Communications of the ACM, 18(9):509–517, 1975.

[16] B. Mcfee and G. Lanckriet. Large-scale music similarity search with spatial trees. Proceedings of the 12th International Society for Music Information Retrieval Conference, ISMIR 2011, pages 55–60, 2011.

[17] Franz Aurenhammer. Voronoi diagrams—a survey of a fundamental geometric data structure. ACM Computing Surveys (CSUR), 23(3):345–405, 1991.

[18] Franz Aurenhammer and Rolf Klein. Voronoi diagrams. Handbook of computational geometry, 5:201–290, 2000.

[19] Steven Fortune. A sweepline algorithm for Voronoi diagrams. Algorithmica, 2(1-4):153–174, 1987.

[20] Ivan Kuckir. Fortune's algorithm and implementation. http://blog.ivank.net/fortunes-algorithm-and-implementation.html. Accessed: 21-07-2016.

[21] Stuart Lloyd. Least squares quantization in pcm. IEEE transactions on information theory, 28(2):129–137, 1982.

[22] Qiang Du, Vance Faber, and Max Gunzburger. Centroidal voronoi tessellations: applications and algorithms. SIAM review, 41(4):637–676, 1999.

[23] Steven Fortune. Voronoi diagrams and delaunay triangulations. Computing in Euclidean geometry, 1:193–233, 1992.

[24] Java Power Voronoi Diagram. https://github.com/ArlindNocaj/power-voronoi-diagram. Accessed: 25-05-2016.

[25] Arlind Nocaj and Ulrik Brandes. Computing voronoi treemaps: Faster, simpler, and resolution-independent. In Computer Graphics Forum, volume 31, pages 855–864. Wiley Online Library, 2012.

[26] Kisung Lee, Raghu K. Ganti, Mudhakar Srivatsa, and Ling Liu. Efficient spatial query processing for big data. In Proceedings of the 22Nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL '14, pages 469–472, New York, NY, USA, 2014. ACM.

[27] Gustavo Niemeyer. geohash.org: Short links for referencing a position. http://forums.groundspeak.com/GC/index.php?showtopic=186412, February 2008. Accessed: 04-03-2016.

[28] Geohash. https://en.wikipedia.org/wiki/Geohash. Accessed: 04-03-2016.

[29] Geohash - geohash.org. http://geohash.org/. Accessed: 04-03-2016.

[30] Geohash grid aggregation. https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations-bucket-geohashgrid-aggregation.html. Accessed: 27-06-2016.

[31] LeHong Van and Atsuhiro Takasu. An efficient distributed index for geospatial databases. In Database and Expert Systems Applications, volume 9261 of Lecture Notes in Computer Science, pages 28–42. Springer International Publishing, 2015.

[32] Rijkswaterstaat. Publieksrapportage Rijkswegennet 3e periode 2015, January 2016.

[33] Michael Frigge, David C Hoaglin, and Boris Iglewicz. Some implementations of the boxplot. The American Statistician, 43(1):50–54, 1989.

[34] Martin Krzywinski and Naomi Altman. Points of significance: visualizing samples with box plots. Nature methods, 11(2):119–120, 2014.

[35] An easy-to-implement library for the GeoHash algorithm. https://github.com/astrapi69/jgeohash. Accessed: 28-04-2016.

[36] Cbs statline - bevolkingsontwikkeling; regio per maand. http://statline.cbs.nl/Statweb/publication/?DM=SLNL&PA=37230ned&D1=17&D2=57-594&D3=l&LA=EN&HDR=G2&STB=T,G1&CHARTTYPE=1&VW=T, June 2016. Accessed: 04-07-2016.