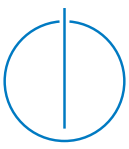
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Web Based Visualization and Analysis Platform for Sensor Data from Mobile Devices

Tim Pfeifle



DEPARTMENT OF INFORMATICS

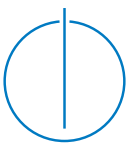
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Web Based Visualization and Analysis Platform for Sensor Data from Mobile Devices

Webbasierte Visualisierungs- und Analyseplattform für Sensordaten von Mobilen Endgeräten

Author:	Tim Pfeifle
Supervisor:	Prof. Dr.-Ing. Jörg Ott
Advisor:	M.Sc. Teemu Kärkkäinen
Submission Date:	16. May 2019



I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 16. May 2019 Tim Pfeifle

Abstract

With the number of sensing devices growing fast over the last years designing data pipelines to collect, store and process their data is of utter importance for many applications. The sensed data often arrives in bursts so the data ingestion has to be able to scale on demand. Due to its time series nature the data has the characteristic of piling up over time, which requires the data store and all other components to be highly scalable as well. Manually analyzing this data is infeasible. Because of the wide diversity of data formats and communication standards a flexible pipeline is needed. While there exist solutions for all steps of the pipeline integrating and maintaining them requires a lot of resources and is no option for small teams or for creating a proof of concept. In this thesis we provide a simple solution that is flexible enough to deal with the variety of use cases, integrates all steps from data collection to data exploration and is modular enough to allow the integration of external solutions. It enables the user to correlate different data by time and location for data exploration or data driven decision-making.

Contents

1 Introduction

The number of devices sensing their environment is growing fast, ranging from beacons to embedded devices such as the Raspberry Pi. Gartner forecasts that by 2020 more than 20 million IoT devices will be installed [1]. Those devices can be, e.g., in the case of a smart factory machines and production parts that send status updates continuously to controllers, or in the case of e-mobility providers like Uber cars equipped with GPS that send real-time updates of their position.

Problem Description

This continuous stream of data, puts a great strain on existing data pipelines. Exploring this large amount of data to gain usable insights is of great interest in commercial as well as in scientific applications.

Data Collection The devices communication protocols range from Bluetooth used for beacons to Wi-Fi. Depending on the network connectivity and the use case they either send the data in batches, consisting of a group of data entries, when they have network access, or produce a real-time stream of event data. While data uploads in batches require the data pipeline to handle large amounts of data at once, continuous data streams and the request for real-time output require low latency.

Data Type and Size The data that the sensors produce differs vastly between each device type. While a common data type in smart factories is *event data*, e.g., a machine sending an event every time it finishes a working step, Uber's cars produce *trajectory data* describing the movement of the car. And even data for the same data type can come in different formats, depending on how the respective sensor transmits it. The devices also sense their surroundings not only at one specific point in time, but instead produce a stream of data. This time series data has the nature of piling up over time and as we add more sensors we have to handle this large amount of data.

Data Processing Manually analyzing the diverse sensor data is infeasible. Because the sensing devices collect data in different formats we have to parse the data from

each specific format into a structured general form. To better capture the context of the sensors many devices additionally send location information like GPS coordinates. By adding this second dimension to the data we now have to analyze spatio-temporal data, which is difficult to visualize and to query. We therefore have to be able to write highly customizable transformation scripts to process it and explore it.

While the data pipeline itself is very similar for all kinds of sensors, the range of communication protocols, data types and data formats makes it difficult to abstract from them in a flexible way. It is difficult to integrate the different pipeline steps with each other. There do exist solutions like AWS IoT [2] that integrate all pipeline steps seamlessly, but most of them are not open-source, expensive and highly complex to set up and maintain.

Importance of the Problem

The increasing number of distributed sensors sending spatio-temporal data create opportunities for many mobile crowd sensing applications. Those range from monitoring traffic to predicting forest fires. Currently many steps of the sensor analysis process are done manually and separate for the different sensor types. Providing a pipeline that is flexible enough to integrate all those different use cases would highly reduce the effort in gaining knowledge from the sensor data. While it is reasonable for large companies to set up and configure a highly complex and scalable pipeline, there are many cases in which sensor data is produced by users with limited resources to analyze it. Providing an easy to use end-to-end solution for gaining knowledge from sensors would be incredibly valuable for small teams or to provide a proof of concept without investing a lot of time and resources upfront. Making the data visually explorable and exportable to share the gained knowledge with others in the form of research papers or in internal company meetings is of utter importance to enable teams with limited resources to step in the area of data driven decision-making.

Proposed Solution

The topic of this thesis is the design of a platform covering the complete sensor analysis pipeline. It includes an interface for the sensors to connect and upload the senor data to, a data store containing the data set, a processing unit to transform the data set and a user interface (UI) to visualize the spatio-temporal data. The services are accessible by an application programming interface (API) to easily integrate it with external solutions.

Sensor Interface We propose a single interface, decoupled from the sensors, to which the sensors can publish their data. It does not restrict the data type or data format used by the sensors and uses a communication protocol that is understandable by all devices.

Data Store To store the piling up time-series data we propose a scalable data store that integrates easily with the other pipeline components. It allows access to the data over an API and easily integrates with external services, like other visualization tools or processing units. It is optimized for a large number of concurrent inserts.

Processing Unit This unit allows the user to run generic transformation scripts with external dependencies. To cover all the different use cases it restricts him as little as possible. Therefore the user is personally responsible to optimize his data queries and transformations for good performance.

User Interface For data exploration purposes we propose to integrate a visualization user interface in the data pipeline that minimizes the effort to explore the transformed data.

Summary

The proposed solution will not and can not cover all possible-edge cases, but we want to keep the solution as modular as possible to make it easy to integrate existing solutions and to extend the platform further in later developments. The platform therefore has to provide flexible interfaces and should restrict the different use cases as little as possible. It is simple to setup the full pipeline for new sensors, from collecting the data to the final visualization of the data.

2 Background

Before we get into the design of the pipeline we give an overview of the kind of data common in mobile sensing applications, how it is collected as well as over existing data analytics solutions.

2.1 The Nature of Data

For the last decade most mobile crowd sensing (MCS) applications used specific wearable devices for e.g. embedded activity recognition or environmental monitoring. One prominent example of such a device is the "Mobile Sensing Platform" (MSP) [3] which contains on-body sensors like microphones and accelerometers. With the enormous distribution of smartphones and their network and compute capacities they are a scalable alternative and are used in many MCS applications. [4] They contain many embedded sensors like the following:

Smartphone Embedded Sensors

- Digital compass
- GPS
- Camera
- Microphone
- Gyroscope: direction & orientation
- Accelerometer
- Proximity sensors

With this sensor data we can perform classification of activities and contexts. Additional sensors for, e.g., temperature or air quality can be connected via Bluetooth. While smartphones are used for many applications other sources such as e.g. sensor networks, drones, smart home sensors or connected cars are also part of the wide range of devices used for mobile sensing. Sensing devices in general, whether smartphones or other devices, often contain multiple different sensors and have at least some form

of networking capabilities. The processing capabilities range from small embededd micro-controllers to octa-core processors like the Snapdragon 855 used in modern smartphones.

2.1.1 Spatio-Temporal Data

We refer to data with coordinates in time and space as spatio-temporal data. The major portion of data generated by mobile devices is of this nature. When analyzing this kind of data we have to extract the spatial relations, e.g., distance or direction, as well as the temporal relations, e.g., duration or occurrence time, from the raw data. Many sensing applications collect the data unbounded over time and it therefore does not have a fixed end time. While it has a defined sequential order, we often can not assume that it arrives in this specific order, because of, e.g., latencies in the network. The spatial aspect often reveals relationships based on proximity. Typical use cases for spatio-temporal data are traffic analysis and weather prediction. Because of the popularity of mobile phones and other GPS capable devices, spatio-temporal data mining and analysis have become increasingly important.

Auto-Correlation Unique to spatio-temporal data is the presence of dependencies between different measurements. Many other data mining methods assume independent and identically distributed (i.i.d) data [5]. In the case of spatio-temporal data, measurements are related in the context of space and time, which should not be ignored during data analysis [6]. Observations at close locations and timestamps are not independent but instead are correlated.

Data Types The data can be of different types, with the following most common ones:

1. *Event data*: discrete events with time and a point location (e.g. car accident events)
2. *Trajectory data*: the movement of an object is measured (e.g. route of a car)
3. *Raster data*: observations are measured for a fixed cell in a spatio-temporal grid (e.g. calls per cellular network cell).

Event data can be described by a list of (l_i, t_i) tuples with l_i representing the location and t_i representing the time point of an event. Additionally, each event can contain marked variables, providing additional information, e.g., speed and numberplate of a car. *Trajectory data* is often collected by mounting a sensor to a moving object and sending the location over time. It can also contain additional marked variables, but always needs to have a reference to the moving object. *Raster data* records measurements

in fixed time intervals and fixed locations. The locations and time intervals can be regularly or irregularly distributed. This data is common in weather sensing where fixed stations measure in fixed time intervals.

2.1.2 Paradigms and Scale

Sensors have different paradigms for how they sense their environment and depending on the scale of the application this information is combined. Based on the term edge computing, which refers to distributed computing on distributed device nodes, we will use the term *edge* to denote the set of distributed sensor devices.

Sensing Paradigms Continuously sensing and sending the sensory data called *continuous sensing* consumes power and therefore drains the battery. This is a big problem for sensing applications that run on smartphones, as battery power is limited on wearable devices. Alternatively one can collect and store the data on the device and upload it in batches (e.g. once a day) [4]. Especially if devices have limited connectivity it is a common approach to upload them in batches later when network access is available. This unavoidably leads to latencies between sensing and processing.

Sensing Scale The differentiation between applications designed for an individual called *personal sensing* and applications designed for multiple users called *group sensing* is common. An important benefit of using group sensing is the ability to validate the quality and correctness of the sensor data. On the other hand in the case of personal sensing one does not have to combine sensory data from multiple devices and it is therefore easier to move parts of the data pipeline towards the edge.

2.1.3 Security, Privacy and Trust

Many MCS applications deal with private and sensible data that has to be protected against malicious intend.

Security Because MCS allows potentially malicious mobile users to participate in the sensing we have to be prepared for attacks. Those range from *tampered data uploads* where a user uploads intentionally wrong data, to *Dos/DDoS Attacks* where an attacker tries to make the application unavailable to the users by e.g. intentionally flooding the application with traffic from many different sources [7]. If the devices are controlled and deployed by the platform owner those internal attacks are a less important issue, because we can assume them to be non-malicious. In those cases we have to focus on

external security threats. For example wireless channels make it easy to eavesdrop and monitor traffic, therefore connections should e.g. be encrypted.

Data Trust & Quality of Information We need to assess the *Quality of Information (QoI)* to deal with potentially unreliable devices. We also can not assume steady flow of information because the devices are not guaranteed to have consistent network access. So how do we deal with information from the same area and same time, but conflicting content, for example due to noise or network delay? Trust is the likelihood that a participant will submit reliable information based on his past behavior. Therefore it is mandatory to collect objective measurements (evidence) to determine the participants reliability. [8]

Different schemes were proposed on how to deal with false reporting. The simple scheme "voting-based truth discovery" for example assumes that the most often observed result is the truth. More advanced schemes use one of the following exemplary techniques to determine the trustworthiness of a result:

- Contextual information to infer the suitability of a participant to monitor a specific event
- Outliers detection and similarity
- Ground-truth data in form of a limited number of *mobile trusted participants* (MTPs) to "bootstrap" trust in the system

Privacy and Anonymization Techniques Anonymizing the data, either already on the sensing device or later in the data pipeline is essential for many applications. It is not limited to *deleting* information. Instead of deleting e.g. a name we can *replace* it with an id. The data is then only identifying if one knows the mapping between name and id, or if one is able to infer the name from the context. Sweeney e.g. demonstrated how easy it is to identify a person uniquely with their demographics [9]. She discovered, that 53% of the U.S. population were likely to be uniquely identified by only their residing city, their gender and their data of birth. In a different study researches collected anonymized mobility traces and were able to identify 95% of the individuals from only four locations [10]. To make this more difficult a common technique is to *generalize*. For instance the birth date of a person can be generalized by replacing it with the birth year. When dealing with large data sets we can also introduce random changes in the data with a zero-mean noise to every data point. This does not change the aggregate but the individual data entries are no longer identifying. Some schemes try to preserve privacy while aggregating the data using *homomorphic encryption* [11]. They therefore allow the service provider to aggregate the data without knowing its content, but while this preserves the privacy it makes it difficult to transform the data.

2.2 Existing Data Analytics Platforms

Other MCS platforms like *Hive* [12] and *Medusa* [13] focus on the data collection process close to the sensors and assume untrustworthy or unwilling users. Our platform's focus lies on the data processing and analysis after the edge. Therefore solutions closer related to our platform are Data Analytics Platforms. We will use the definition of Data Analytics Platforms as platforms that join different tools to provide a contextual analyzed data providing solution for all the steps in the OSEMN-Pipeline:

- **O**btaining data: Identify all of your available data sets and extract them into a usable format
- **S**crubbing / Cleaning data: Highly application dependent and should be highly customizable
- **E**xploring / Visualizing the data
- **M**odeling the data: Write data analysis scripts and create predictions
- I**N**terpreting data

They therefore have to provide means for data ingestion, data storage and data analysis, as well as some form of visualization solutions to interpret the data.

There are countless different Data Analytics Platforms so our overview in this chapter will only cover the most prominent and feature rich solutions. Our choice is based on Gartner's leaders in the "Magic Quadrant for Analytics and Business Intelligence Platfoms" 2019 [14].

Most of those solution focus on providing the user with real-time information for business decisions. They are designed for users with minimal programming skills and therefore have countless ways of visually exploring the data as well as visually creating database queries using query-builders. Their focus is on data visualization and exploration, but they often require other tools to handle more advanced data preparation and data modeling tasks, because they do not allow to write generic transformation scripts.

In the following we will adopt the term *augmented analytics* to denote the analytics paradigm that uses machine learning to discover data insights and helps in the data preparation.

Tableau [15] Tableau started to incorporate visualization features into databases to make them interactive and understandable. As a result it became one of the leaders in visualization techniques for data exploration. It offers an intuitive, interactive way for data exploration requiring no technical skills or coding. While providing unparalleled

visualization options it misses some data integration capabilities. For example, it does not support the querying of multiple fact tables in a single data source. Apparent from customer responses [14] it also has problems handling large data volumes.

Qlik [16] Qlik's data discovery product, Qlik Sense, is built around the API of the Qlik Analytics Platform. The platform is powered by an in-memory database engine. They have added many augmented features with its so called Cognitive Engine, which suggests most relevant insights, visualization types and much more. While this works pretty well for standardized sales-processes it is probably quite difficult to work on sensor data with uninformative data fields. Part of their platform is also Qlik GeoAnalytics, a tool for geo-spatial data.

Microsoft Power BI [17] Power BI is a powerful solution for data shaping and visual data discovery. Microsoft has started to integrate Power BI with many other Microsoft products. Therefore Power BI is able to use the augmented analytics capabilities of Azure Machine Learning such as text and sentiment analytics. It also integrates easily with other Microsoft tools like Excel. One should be cautious to not get vendor-locked-in by this approach.

2.3 Use Cases

The possible use cases for Mobile Crowd Sensing are limited mostly by the available sensors. With the availability of capable sensors, ranging from smartphones to IoT devices and connected cars, MCS's use cases now span many fields. There are countless applications as a recent survey paper shows [11], so we will present only a few to highlight some use cases in different fields.

Healthcare Biosensors that measure heart rate or blood pressure can be used to monitor diabetes or seizures. A popular example in the field of healthcare is the MCS application *TrackYourTinnitus* [18] that tries to reveal new medical aspects on tinnitus and its treatment. They collect large data sets consisting of the patients demographic and clinical characteristics as well as their response to treatments to suggest evidence-based treatments.

Environmental Physical, chemical and biological sensors help to collect raster data enabling, e.g., forest fire detection systems where smoke and temperature sensors allow monitoring of forest fire to react in time [19].

Smart City & Infrastructure In smart cities MCS applications are significant to monitor the services the city provides. Those applications monitor city noise [20], traffic congestion [21] and emergency incidents [22]. Structure health monitoring applications like *CrowdMonitor* [22] help to assess physical and digital activities of citizens during emergencies after natural disasters or terrorist attacks.

Social Sensing Social sensing applications collect data from personal activities. An example is the group-aware MCS system *MobiGroup* [23] that supports group activity organization by e.g. suggesting ongoing events based on the user's activity and interaction dynamics in a community.

2.4 Summary

Many MCS applications use smartphones because of their wide distribution and their available sensors as well as because of their compute and network capabilities. The major portion of the generated data is spatio-temporal, ranging in type from event data to trajectory data. Observations of close data entries are therefore correlated. The devices can either continuously sense their environment or upload their data in batches. Because the sensed data is often sensible, especially in the field of healthcare and social sensing, we have to protect it against external attacks and use anonymization techniques to protect the user's privacy. A common pipeline for end-to-end data analytics platforms is the OSEMN-pipeline (Section 2.2) that contains the whole process from ingesting the data to the visualization and interpretation.

3 Design

We will first give an overview of the platform in general and the platform's *Pipeline*. In the following sections we will focus on the different *Interfaces* and the *Data Storage*. The last section will then focus on the *Visualization* of the data.

Platform The platform receives spatio-temporal sensor data from different types of sensing devices and should allow the visualization and exploration of the data. The spatio-temporal data is, as described in the Background section, unbounded and might have large and inconsistent differences between the event time, the time at which the sensing took place, and the processing time, the time it arrives at the platform. This difference might be further increased by the varying network connectivity of the devices. Some of them therefore have to send their data in batches with a large difference in event and processing time. The received data then has to be parsed and transformed. Those steps have to be highly customizable for the different applications and should restrict the user of the platform as little as possible. Because of the spatio-temporal nature of the data we need to be able to visualize location data with e.g. maps. To provide those transformation and visualization services the platform also needs to store the data.

3.1 Pipeline

The pipeline processes the raw sensory data for analysis and visualization in the platform. The use cases differ quite drastically. Some are *compute focused* because they have computationally expensive transformations of the sensor data, others are *storage focused* because they produce large amounts of data and others are *integration focused* because they need to integrate easily with external services. Consequently, the pipeline has to be flexible enough to deal with this great variety. The main design principle of it is to enable an easy and flexible access to the collected data. As the data is already partly accumulated and filtered on the edge before arriving at this platform the pipeline represents only the last steps in a full MCS-Pipeline. We are therefore not dealing with the steps that highly depend on the network-architecture, communication-protocols and sensor-devices and will abstract from them in the following using a Pub/Sub interface.

Hence we will not compare the platform to existing MCS-Frameworks like Medusa [13] or Hive [12]. The pipeline more closely follows the steps of a Data Analysis Platform, specifically the OSEMN-Pipeline (Section 2.2):
We will *1) Obtain* the sensor data from the Pub/Sub interface and parse it into structured Data, which we then *2) Scrub* using custom processing scripts. Those results are then available for *3) Exploring* through visualization components. Analysis scripts can run on the processed data for *4) Modeling* and allow the user to easily *5) Interpret* the sensor data, as shown in Figure 3.1.

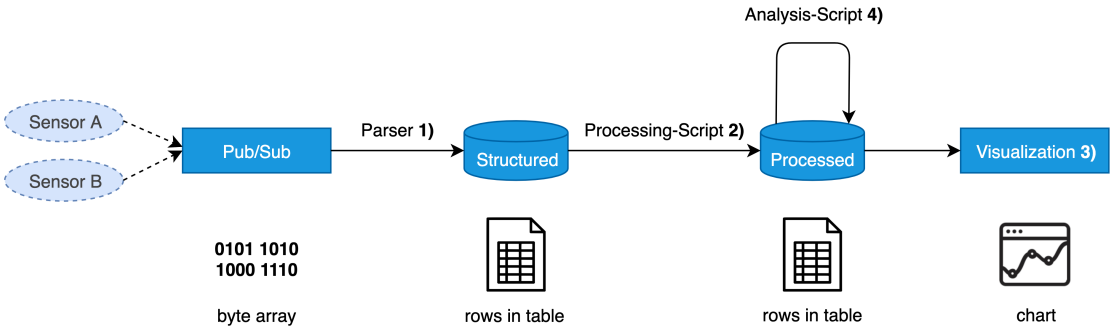
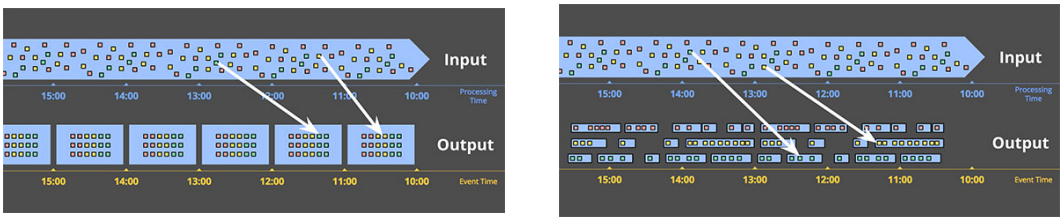


Figure 3.1: OSEMN-Pipeline: 1) Obtaining 2) Scrubbing 3) Exploring 4) Modeling

Stream Processing vs. Batch Processing We will follow Tyler Akidau [24] in using the terminology "streaming" only to describe the execution engines designed to deal with infinite data sets and use the specific terms "unbounded data" and "unbounded data processing" to relate to other concepts often also grouped under the "streaming umbrella". Unbounded data, e.g., simply relates to infinite data in contrast to finite data. It can be processed by all kinds of execution engines, but using the term "streaming" implies the usage of a streaming engine. "Unbounded data processing" can still be done with a batch engine.
As already described, the received unbounded sensor data might have large and inconsistent differences between the event time and the processing time. Because this difference is not static we can not easily map between event time and processing time. To process our data one usually applies windowing based on the processing time, but with this difference we can not assume to receive our events in the context in which they occurred (event time), because the difference would result in events ending up in windows that do not represent their event context. We can also not simply window by event time, because we do not know whether we already received all events for a specific window or if there is still one with a large difference in event and processing time waiting to be delivered. There is no way for us to guarantee this completeness [24].

Many processing engines therefore use some notion of completeness, e.g. by defining a maximum-difference.
Batch engines often use either fixed windows or sessions to process unbounded data. With fixed windows the engine splits the data by their processing time into windows with a fixed duration, as shown in Figure 3.2a. To have a higher probability of completeness the processing is often simply delayed. With sessions the fixed windows are subdivided into sessions that represent e.g. a period of activity on a web page. Those sessions can potentially span multiple windows which requires to stitch those back together, as can be seen in Figure 3.2b.



(a) Fixed windows

(b) Session windows

Figure 3.2: Batch processing approaches for unbounded data [24]

Because we mostly need to process our sensor data in batches and do not require real-time information we will focus on the simpler batch systems and will not go into details how streaming engines approach unbounded data. Because it is difficult to reason about time and to guarantee correctness streaming engines are often more complex to design.

As batch- and stream-processing pipelines are fundamentally different, changing the architecture to a stream-processing pipeline is not an easy task. One approach is to follow the Lambda-Architecture [25] as depicted in Figure 3.3 taking advantage of both our already designed batch processing, here called “batch layer” and stream processing, here called “speed layer”. The goal of this architecture is to use batch processing to provide accurate views of the batch data and to use the real-time information from the stream processing in parallel for online data. The results of both paths are then combined before presentation. An alternative approach is to replace the batch engine with a stream engine and only use this one path. As streaming solutions mature this approach is preferred as it simplifies the pipeline compared to the Lambda-Architecture.

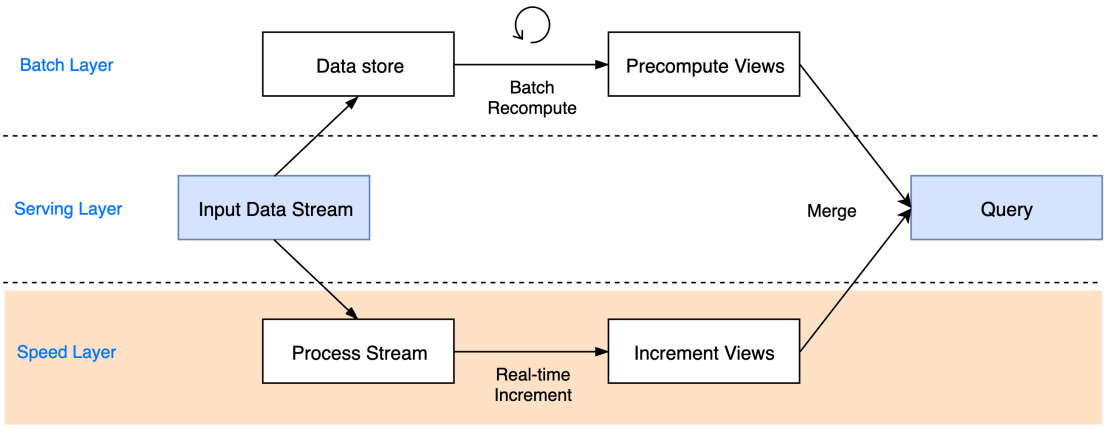


Figure 3.3: Lambda-Architecture: separate batch and speed layer which are merged when querying (adapted from [26])

3.2 Data Ingestion

The data is collected from sensors of varying types and our data ingestion has to be able to receive the data from them, working on an abstract of the different types. The number of sensors is not constant and might change so the receivers should be decoupled from the explicit sensors.

3.2.1 Sensor Interface

Xiao et al [27] write that "scale is the key to success of crowdsensing applications", because the users/sensor-devices are unreliable compared to classic sensor-sensing approaches and to get an acceptable accuracy therefore requires a larger user base. Handling that many data-streams requires that the different services in the pipeline are able to scale not only vertically but also horizontally. Decoupling the different services as well as reducing the state each of the services contains are important to scale seamlessly. It also means that we will have many devices with potentially all sorts of different communication protocols. The goal of the platform is to enable the use of all those devices as well as to reduce the complexity of adding new types.

Decoupling from Sensors The first interface (in the left of Figure 3.1) of the pipeline therefore decouples the platform from the sensors, which is a common approach in MCS-Frameworks as well as IoT-Platforms. Often the Publish-subscribe pattern [28] is used. It does not let so called publishers send their messages directly to the receivers,

here called subscribers, but instead categorizes the messages into different classes and lets the subscribers express their interest in receiving a specific class of messages. Consequently, the publishers and the subscribers do not have explicit knowledge of each other. In our use case the sensors and sensor-gateways act as publishers and the platform subscribes to the different classes of messages. This paradigm also enables other applications or services to subscribe to the message classes. For example might it be useful to keep the raw sensor-data to debug potential problems and to validate results which can simply be done by creating a subscriber that dumps all message classes into a data store. An alternative paradigm called *Message Queuing* receives the incoming messages and makes sure that each message is processed by exactly one consumer by deleting the message from the queue after consumption. It is allowed, e.g. in the case of network failures, to resend messages and therefore process them out of order.
Many technological solutions support both paradigms. For example a Message Queue can support multiple consumers by replicating the queue for each of them. We will follow the Pub-Sub paradigm and use the terminology Publisher and Subscriber instead of Producer and Consumer.

Data Format After decoupling the sensors we would like to abstract from the different communication protocols and their data structures. One can either require the data to be structured in some predefined way *on the sensor-devices* or structure the data *on the platform*. As seen in Figure 3.1, the advantage of already inferring the structure on the devices is that all applications that subscribe to the Pub/Sub-Service will receive messages following the same structure. On the other hand it requires the sensors to encode their messages following some predefined standard, e.g. as JavaScript Object Notation (JSON) strings. This makes it difficult to add existing sensors, because they might encode their messages already differently. If one structures the messages on the platform itself, then one is flexible to subscribe to all kinds of different sensors, as long as one makes sure to synchronize/configure a parser for each message data format. If need be we could still impose a single format by only having one parser and requiring all sensors to comply.

We will choose to structure the messages on the platform as the filtering already takes place on the edge and the goal is to work with arbitrary types of sensors. Therefore we accept all message-formats which we will abstract from with simple *byte arrays*.

Parsing Messages Subscribers will subscribe to the different classes of messages and to determine the structure they are required to first parse the byte arrays. This parsing should be highly customizable for each type of sensor and therefore the subscriber will

Structure on Devices	Structure on Platform
Pro	Pro
The data arrives structured at the sensor-interface and therefore all subscribers receive already structured data which makes it easy for them to filter depending on the content of the messages	It is easy to add new sensors and existing sensors that might encode their messages already differently
Contra	Contra
Structuring on the devices makes it complicated to add new sensors as they first have to comply with the platform structure	The inferred structure on the platform has to be synchronized/configured to allow the message to be parsed

Table 3.1: Comparison of where to infer the structure of the sensor-data

inject a different parser for each type of sensor. In the *Evaluation* section we will give two different examples of such parsers (see Section 5)

3.3 Processing Scripts

These scripts transform the sensor-data and output the data models that will be visualized in the platform. This is Step 2 in Figure 3.1.

Flexibility The script-interface needs to generalize for many different use cases of the users and therefore must be highly expressive and flexible. The limitation in flexibility is a huge drawback of using existing solutions like *Tableau* which only enable the users to write data-store queries and not generic code. As this system is designed for non-malicious users with a background in Informatics we can give the user the flexibility to write generic code transforming the stored data. This freedom has many implications. Generic code is Turing complete which means that the scripts in theory are able to perform any calculation that a programmable computer could do. Turing completeness makes it impossible to compute whether a program would ever halt on a given input. Therefore the downside of writing these flexible scripts is that we have to assume that the user is writing computationally efficient scripts. What we can do is to isolate the different scripts from each other and give each one a limited amount of processing resources and time. This sandboxing makes sure that the scripts do not interfere with each other and provide an interface to manage the running scripts as well as to cancel them if need be.

The option to write generic code raises the obvious question in which programming language they have to be written. Our design allows any language, but each script language requires a script-interface in that language, containing data-storage connectors, as well as all the overhead that is required to run those scripts as processes in the backend. A python-script for example needs a python-runtime and a Java-script would instead need the JVM. Both would need custom python/JAVA data-storage connectors for reading the sensor data and writing the output.

External Dependencies In many programming languages it is common to specify external dependencies instead of packaging everything together into e.g. a binary. Those dependencies then have to be resolved using a dependency manager e.g. pip for python or npm for node.js. While this simplifies the development of processing scripts we have to provide a way to resolve those dependencies as well as to avoid conflicts between them. If script A e.g. requires version <1.0 of dependency X and script B requires version >1.0 of the same dependency we need a way to serve the specified versions to both of them. One common way is to run the scripts in separate environments. In those environments the respective dependencies are resolved and only available inside them, making sure that different version requirements do not conflict.

Communication The scripts will need to access the collected sensor data to scrub and prepare it for exploration. Apart from this access the required communication with the scripts is for debugging and execution purposes. The most straightforward approach therefore is to create a process in the backend application every time we need to run a script. This makes the communication simple as we do not need anything apart from the already existing inter-process-communication of child to parent processes. Unfortunately this approach is not scalable: If the spawned processes consume large parts of the existing resources those resources are not available to serve for, e.g., web requests of the backend. The alternative *service oriented architecture* [29] approach is scalable and would allow the processes to be run on a remote machine as well as to communicate with the backend using TCP sockets. Both approaches enable us to monitor the execution of the scripts. To access the sensor data (*Structured Data* in Figure 3.1 after being parsed) from the scripts and to also not restrict the scripts in any way we propose to pass a database-connection as an interface to the script as well as a method to write the resulting processed data to our *Processed Data Store*.

Scheduling To automate the execution of the scripts we want to be able to schedule them. Because the scripts can be dependent on the results of other scripts we also have to make sure to execute them in a specific order. In the case of manual runs, the user is responsible to execute them in the required order. The script dependencies can be depicted using a directed acyclic graph (DAG) where children depend on the execution of their parent. Apart from that the scripts can be run in arbitrary order which allows us to run scripts on the same level in the tree in parallel. The actual execution order is typically determined by running a topological search on the DAG.

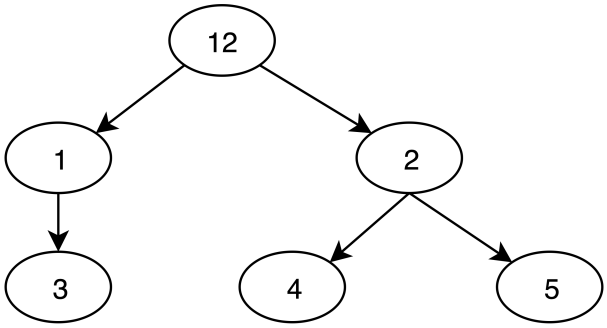


Figure 3.4: Execution tree of scripts: Children depend on the results of their parents, each node represents a script

A special case worth mentioning is that if we schedule e.g. one job to be run daily and one to be run weekly, then once a week we will run both jobs which begs the question of which job to be run first or whether to combine both execution trees into one. This could prevent running scripts more often than required. If e.g. scripts in both jobs depend on the same script we should not execute it twice but instead should merge their execution trees.
One can also define the dependencies of the scripts using a priority queue of script-lists. Each element (here a list of scripts) in the queue represents a priority and they will be executed in ascending order one after the other. In contrast to Figure 3.4 with a priority queue we can not simply run script 3 before script 2 completes, if script 12 and 1 already finished. A priority queue is simply not as expressive as a tree structure, but therefore also simpler to implement.

$$[12] \rightarrow [1, 2] \rightarrow [3, 4, 5]$$

Figure 3.5: Priority queue: each number represents a script and each group represents a priority-group

On the lower level of scheduling the processes we let the processor handle everything. One could for example run scripts only if there is an idle process available instead of running scripts independent of that. This would restrict the user from starting numerous scripts at the same time, between which the CPU would have to split its computation time and therefore would result in many long-running scripts. Instead, we could make sure that they all run one after the other.

Debugging One important interface, especially during the development of the scripts, is the debug interface. Depending on the use case this can range from simply returning the exit code of the script, over returning the output from the stderr and stdout streams, to being able to halt the execution of the script and to access the state of the script at specific break-points. To find out whether a script is working as designed we would like to run the scripts in a context matching the production-context as close as possible. This makes it a lot easier to reproduce problems. But running the script directly in the production-context can be risky and can result in crashing the production system. Nonetheless, some problems are very hard to find and therefore the use of so called *Remote Debugging* can help in those cases. During debugging it is common to make rapid changes to the source code to simplify the code and to find the root of the problem. The process of deploying those changed scripts should therefore also be as fast as possible. Running software locally reduces this latency enormously and makes it easier to integrate interactive debuggers. Being able to run the processing scripts locally would therefore enable the users to use the feature-rich debuggers of their IDEs. To debug problems after our data and scripts already have changed it would make sense to keep a log of important events and states.

Monitoring Because we allow generic scripts, monitoring their execution is important to detect non-terminating processes and other anomalies. Common tools to monitor processes are Linux command-line tools like *htop, iotop*. They display information as CPU/Memory-usage, disk I/O stats or how long the process is already running. One can compare those values with previous runs of the same process to detect anomalies automatically or simply display them to give the user the ability to make informed decisions whether or not to terminate a process. It can also help to detect if a process is generally not running as expected, e.g., if the I/O statistics do not match the expectations or the execution time is too long/short.

Terminating If the user decides to terminate a script we have to inform the process using the so called *kill* command. This commands triggers the operating system to send a signal to the program and depending on the implementation of the program it

does or does not listen to it. Listening to those signals allows the program to gracefully terminate by, e.g., saving not saved states. A common approach, especially in the world of data-stores and databases, is to use transactions which are single units of work that must be either completed in its entirety or to have no effect at all. This "all-or-nothing" approach results in a consistent state of work. In the case of the termination of a program, the program has to *rollback* all steps of the transaction it could not finish before being terminated. To terminate programs that do not listen to those signals or do not terminate after receiving them there exists the *SIGKILL* signal which triggers the operating system to kill the process without giving it the choice to shutdown gracefully.

3.3.1 Data Store Interface

To access the stored data we need to write queries against the data store. The query language has to be powerful and expressive in our case. Adding an Object-Relational Mapping (ORM) on top would abstract from the data store and would make it easier to replace it if the requirements change, but would also require the users to learn the ORM-language.
Because we designed a batch-processing pipeline the scripts will be run in specific intervals/after specific triggers and store their results again in the data store. The data store interface allows the user therefore to access all of the structured data and transform it. To limit the side effects between the scripts, the structured-data is *read only* and all transformations have to be stored in a data-store which is only editable from the script which created it.

3.4 Data Storage

In the following we will relate to the sensor data in the received byte array format as **Raw Data**, the structured data after running through the parser as **Structured Data** and the data after being processed by custom scripts as **Processed Data** as seen in Figure 3.6.
The main questions for each of those data sets before deciding which data store to use are the following:

- What **kind** of data are we expecting?
- What **amounts** of data are we expecting?
- What **operations** do we need to run on the data?

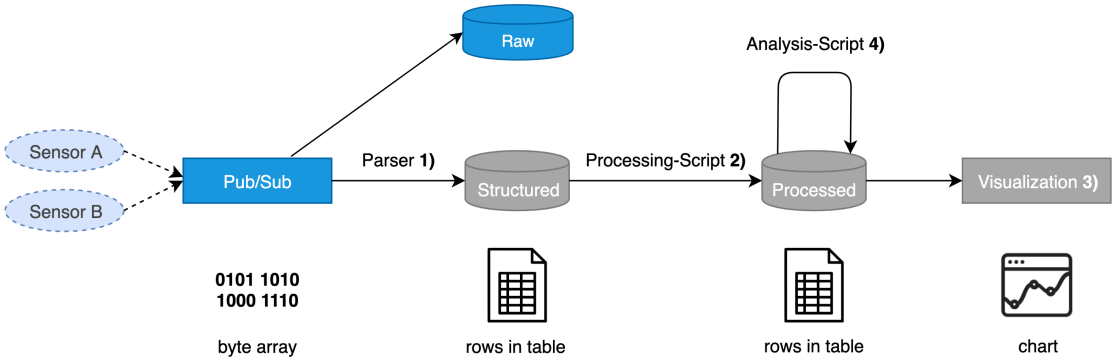


Figure 3.6: Optional data store for the raw sensor data before parsing

Raw Data Store We assume only limited network access for the sensors and therefore will only transfer their data in batches. Each data entry is assumed to be less than 10KB, but the size highly depends on the different sensors. Filtering the data will be done on the edge reducing the network and storage requirements. The data will be received over the Pub/Sub-Interface described in *Data store interface*. Storing the raw sensor data instead of only passing it on to the next step in the pipeline increases the used storage, but enables debugging and is fault-tolerant to faulty parsers. In our design we will give the option to store the raw sensor data, but we will run this optional step not as part of the main pipeline. By adding a new subscriber to the sensor data we can run this parallel to our main pipeline and therefore do not increase the latency of the main path (see Figure 3.6). The raw data store mostly inserts time series data and is only read in sequential order. It should be able to deal with large amounts of data and should be optimized for fast INSERTs. Our operations will be approximately one READ and WRITE operation per data point, therefore a space-efficient data store should be a good fit.

Structured Data Store The data will still be time series because it is only parsed but not yet transformed into data models. We will therefore have potentially large amounts of data and our operations are mostly INSERT and simple READ operations. Because this is our main storage of the sensor data, we potentially want to query this store also with external tools (e.g. Tableau).

Processed Data Store As the output from the processing scripts this data set is assumed to be by far the smallest, we will have much more READ than WRITE operations and should optimize for that. To explore the data we want to have a large set of available operators to query and transform it.

3.4.1 Distributed Storage

When the data sets become to large for a single machine we have to distribute the database. This brings many challenges with it including the *CAP theorem* which states that distributed systems can only provide two of the following three guarantees:

- Consistency
- Availability
- Partition tolerance

Therefore a common approach for large systems, called *Polyglot Persistence*, is to use multiple databases and to leverage each specialty for the respective use case increasing the overall performance over using a "one type fits all". We assume that the data fits onto a single machine and therefore can guarantee all three CAP properties.

3.4.2 Database Comparison

Time series data represents how data changes over time and therefore keeps not only the last state, but also all of the intermediate states. To analyze e.g. how the temperature in a specific area changed we require to keep all those intermediate states as well. For storing the sensor data we therefore need to perform many INSERT operations instead of many UPDATE operations. This leads to an ever-increasing amount of stored data. In the following we will compare the pros and cons of using a time series database instead of a classic relational database for this type of data. Figure 3.2 summarizes the following comparison.

Time Series Databases (TSDB) vs. Relational Databases (RDB) Generally speaking, RDBs are designed to store relational data and TSDBs to index data primarily by its timestamps. As time series data is piling up, the data pages of RDBs do not fit into memory anymore and they have to swap the pages between memory and disk. Especially if the data indices (B-Trees [30]) do not fit into memory anymore this becomes very costly. We need to use efficiencies that are specific for time series data to handle those steadily increasing data sets. Because time series workloads are mostly append operations we can use specific index-structures to improve the database performance when scaling-up. TimescaleDB [31] e.g. splits the data according to the time and primary key dimension building distinct tables called chunks. With appropriate sized chunks we can then fit the latest chunk and its respective B-Tree completely in memory as seen in Figure 3.7. For append operations we therefore avoid the swap-to-disk problem while maintaining support for multiple indices.

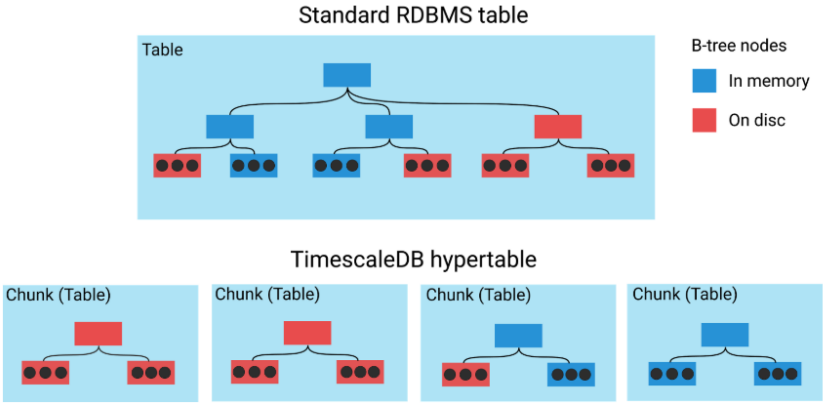


Figure 3.7: TimescaleDB : Split each table into chunks and store them in an internal table. The chunk can then fit into memory, which avoids the swap-to-disk problem [32]

Operation Properties Both ACID and BASE relate to a set of properties of database operations. ACID stands for ***A**tomicity, **C**onsistency, **I**solation and **D**urability* and operations that fulfill all those criteria are called transactions. Fulfilling these criteria guarantees validity for database operations, even in the case of system failures, but also restrict the database and reduces availability and performance in large systems. This is why databases like InfluxDB relax these criteria and only guarantee ***B**asic **A**vailability, **S**oft-state and **E**ventual consistency*. Results to queries therefore might be only approximately correct. Many NoSQL databases give guarantees in the continuum between ACID and BASE. The closer to BASE the easier it is to scale a database, but also the weaker are the guarantees. TSDBs can be based on SQL/NoSQL and allow to join time series data based on overlapping areas of time. TSDBs based on SQL, like TimescaleDB are basically a RDB with additional features specific for handling time series data. Other TSDBs like InfluxDB deviate further away by relaxing the ACID consistency constrains to BASE constrains. Because we assume that the database should not be distributed and we do not need to weaken the database guarantees to increase scalability.

3.4.3 Choosing the Time Series Database

Two of the most widely used and most different TSDBs are TimescaleDB and InfluxDB which we will compare in the following. Figure 3.3 summarizes the comparison.

Data Model TimescaleDB follows the *wide-table model*, where the user stores meta-data in separate tables and relates to it using foreign keys. It models data in rows consisting

Time Series DB	Relational DB
• Mostly INSERT, rare UPDATE operations • Optimized for data organized by time • Optimized for a large number of small records	• Typically with schema and conforming to ACID model → strong consistency guarantees • Uses SQL • Does not easily scale horizontally

Table 3.2: Comparison of different database types

of a time field and an arbitrary number of other fields. Those fixed models enable it to validate input against constraints of uniqueness or not-null values. As with other schema-based models we have to choose this schema beforehand. Similar to other SQL databases it supports composite indices.
InfluxDB uses a *narrow-table model* where each data entry consists of a time value and a list of tags and fields. The tags represent the metadata and the fields contain the actual content. Indices are automatically created on all the tags and not on the fields, which limits its flexibility. It also does not provide data validation, because it does not know the "valid" schema.

Query Language TSDBs often introduce new query languages like Flux [33] with the goal in mind to make it easier for developers to work with time series data. While this is a reasonable idea one can perform, at least in theory, better query optimization with SQL and it requires the developers to learn a new query language.

Reliability InfluxDB implemented most fault-tolerance mechanisms from replication over restore-mechanisms whereas TimescaleDB relies for that on PostgreSQL. In the past InfluxDB instances have often been unreliable and lost or corrupted data.

Performance InfluxDB's storage engine is like many other NoSQL databases very similar to log structured merge (LSM) trees [34]. To circumvent the "swap-to-disk" problem of classic B-Trees which perform updates "in-place" on the data pages, LSM trees instead queue up several updates into the write ahead log (WAL) segments and write them as one batch to disk. Therefore, writes are first performed in memory and only merged back in disk when the queue segment is full. This leads to faster sequential writes. The downside of LSM trees is that they do not have a global index

.	InfluxDB	TimescaleDB
Data scheme	No	Yes
Query-Language	Flux or SQL-Like	SQL
Consistency	Raft	ACID
Scaling	Horizontal	Master-slave
Other	Integrated TICK stack	Directly tied to PostgreSQL

Table 3.3: Comparison of time series databases

to give a sorted order over all keys. Therefore, looking up data entries for a specific key can take longer and read performance suffers. TimescaleDB uses B-Trees but by splitting them into chunks circumvents the "swap-to-disk" problem as well. InfluxDB's performance leads for low cardinality INSERTs but drops for larger cardinality (» 2000 rows per second), which results in TimeScaleDB outperforming it for larger cardinality [35]. Many non-production scale applications do not need to perform more than 2000 INSERTs per second and for them both databases perform similar.

Integration Important for the choice of the database is also how easy it is to integrate the data store into other parts of our pipeline. If we want to replace e.g. our visualization Step with a Data-Analysis Platform like Tableau we need data store integrators. All platforms introduced in Section 2.2 have connectors for PostgreSQL and while InfluxDB is less popular many platforms support it as well by now.

3.4.4 Data Queries

"One could argue that the choice of query language is the most important criteria for choosing a database" [36]. While clearly the performance of a database is important we should also make sure that the method to access the data is compatible with the needs of the application.

Limiting Data Entries

Requesting data from the processed data store over a long time range can result in large amounts of data. Sometimes we do not require to get every single data entry because we e.g. can not visualize them in higher resolution than our monitor outputs, or we want to reduce the latency when querying the data. A straightforward approach is to set a limit to the data entries we return. But which data entries do we return if our data set exceeds that limit? Given a table of sales it would make sense to return only every n-th entry and to set its profit to the *sum* of all n entries. If we are interested

in the response times of emergency agencies we would probably prefer to *average* the response time. So we would like to declare which data field we want to consolidate by and using which aggregation function (e.g. sum, average, min, max). It is important to transparently show which aggregation function is used, because they produce vastly different outputs.

3.5 Visualization

To easily analyze and visualize the data we require a user interface that connects visualization components with the processed data and lets the user explore the data sets.

Storing the Query Results for Visualization On the client-side in the browser we have multiple different options for caching the query results. In many cases we want to share the queried data for multiple visualizations and because the processed data is not changing frequently we should think about ways to store the query results on the client. This reduces latency and saves bandwidth as seen in Figure 3.8.

- *No caching*: This requires the application to request the data for all changes in the query resulting in bigger latency till the data can be displayed. This approach is still common in many applications but the loading times are a drawback for the usability especially if dealing with real-time data and frequent data changes.
- Caching *on the server*: If many clients often request the same data and the latency between the server and client is negligible, in contrast to the time the query takes to finish, this improves the usability a lot through shorter loading times.
- Caching *on the client*: Recently the trend is to push the data model to the clients as well, by using a data store on the client side that can answer cached queries nearly without latency.

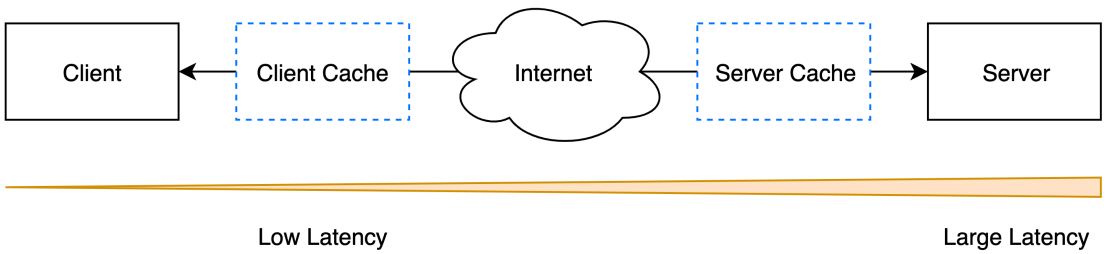


Figure 3.8: Caching: the closer to the client the results are cached, the smaller is the latency for requesting them

3.5.1 Map Visualization

To visualize the spatial component of our data we need a server providing map data. A common source for the map data is the open data source OpenStreetMap [37]. This data is then served with a map server.

Tiles Because serving a map as one large image takes to much bandwidth one instead splits the map into a pyramid of images for different zoom levels, so called tiles. Each covers a fixed geographic area and is assigned with a scale. When zooming in we will request the next scale layer of tiles in the pyramid as visualized in Figure 3.9. One can either serve those tiles rendered or as vectors. Vectors have the advantage of reducing the size by more than 50% and allow to change the map style without redownloading them again. But raster tiles can still make sense if the client has limited resources to render them or if the latency of rendering them on each client is an issue.
Depending on the map server in use one can request the tiles e.g. following the popular Tile Map Service (TMS) specification or the more complex Web Map Service (WMS) standard which is often used for enterprise applications.

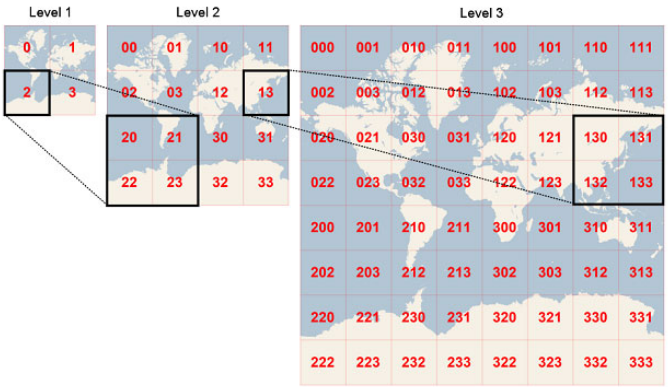


Figure 3.9: Pyramid for loading the tiles depending on the zoom-level [38]

Map Styles When using vector tiles it is common to style them before rendering. Styling can range from deciding what of the data to draw, over the order in which to draw it, to the style in which to draw it. A style can e.g. reduce a vector tile to a visualization of the highways or into a rendering of hiking tracks. Many visualization frameworks follow the common Mapbox GL style specification which defines a style through a custom JSON-file.

Map Projections To transform the latitudes and longitudes of locations of a sphere into points on a plane one necessarily has to introduce some form of distortion. The "best" projection always depends on the use case which determines what form of distortion is least problematic. Common for web maps is the spherical *Web Mercator* projection which preserves shape and angles. Almost all map APIs use this projection, making it easy to exchange tiles between them. It is based on the standard *Mercator* projection, but uses the spherical transformation for all scales from small to large. While this reduces the computational complexity and preserves angles the disadvantage is that the areas close to the poles are greatly exaggerated as seen in Figure 3.10.

Figure 3.10: Left: Mercator projection, Right: True size of each country [39]

3.5.2 Visualization Components

For different tasks we need different types of visualizations. Because there are countless types of charts we will here list only the most common ones and for which cases they are particularly useful.

Bar Chart This chart type helps to visualize *differences between categories*, e.g. sales per market or rainfall per region. We group the data entries by their categories and then aggregate the entries with an aggregate function like sum or average.

Line Chart To connect distinct data points the line charts represents them as a continuous evolution. This visualization helps to display *value changes relative to each other*. Especially for the temporal component in the spatio-temporal data this is useful to visualize changes over time, e.g. the speed of a car or the temperature of a sensor.

Scatter Plot This chart is useful to detect outliers and to get an overview of the distribution of the data. It is very similar to the line chart, but because it does not connect the plotted data points it is often used to investigate the relationship between

two variables. By displaying the density distributions as histograms on each axis we can even easier get a feeling for the type of *data distribution*.

Map Maps are the most common visualization type for the *spatial component* in the spatio-temporal data. Depending on how we plot our data on the map chart we can highlight different aspects. E.g. by connecting points we can visualize trajectories or by plotting each location with less than full opacity we can create heat-maps to visualize the density distribution.

3.6 Summary

We designed a pipeline that obtains the sensor data from a Pub/Sub interface, parses it into a structured format, transforms it with processing scripts and then visualizes them in a UI containing charts. The Pub/Sub interface decouples from the sensors and collects the data in arbitrary formats. For each format we can define a parser to parse it into the required structure. The transformations on the data are then executed by the processing scripts, to which we provide only a data store interface and apart from that allow them to be completely generic. We schedule their execution, create for each a separate environment including its required dependencies and provide an API to monitor and terminate them. We compared different data stores and their guarantees as well as their performance for time series data. For the visualization we proposed a set of common chart types and how to provide a map visualization for the spatial data.

4 Implementation

As outlined in the Design our data analysis solution consists of 4 steps (see Figure 3.1):

1. The Pub/Sub-Interface to collect the data
2. The Parser to structure the data
3. The Processing scripts to transform it
4. The Visualization to display the results

To allow the user to execute generic parsers and processing scripts (simply called scripts in Figure 4.1) we have implemented interfaces for them. The provided services are accessible through a web interface and a REST-API. Supported are the services through our infrastructure that consists of the mentioned *Pub/Sub-Service*, a *Data Store* and a *Map-Tiles Server* (see Figure 4.1).

Before getting into their specifics we will highlight our data interface paradigm and introduce Typescript, the programming language used for most of the implementation.

Data Interface Paradigm The data between all our services is passed in tables. The services access the input data through a table and output it again in a table. The only exception are the interfaces outside of the pipeline, namely the parser that accesses the input data from the Pub/Sub-Service and the visualization that outputs the data to the web application (seen in Figure 3.1). Depending on the data store those tables can be simple PostgreSQL tables or TimescaleDB tables, a superset of PostgreSQL tables optimized for time series data, but the interface language for all tables is PostgreSQL. Because of the wide availability of integrations for Relational Databases this makes it easy to integrate external services in this pipeline.

Typescript We programmed the Web-UI as well as the REST-API using the JavaScript superset Typescript. It adds planned features from future JavaScript editions to current JavaScript engines. Because of such features Typescript is known as an object oriented programming language combined with many functional programming features like lambda syntax, generators and iterators and the spread operator. But more importantly it adds a typing system, enabling static code analysis with the added annotations and

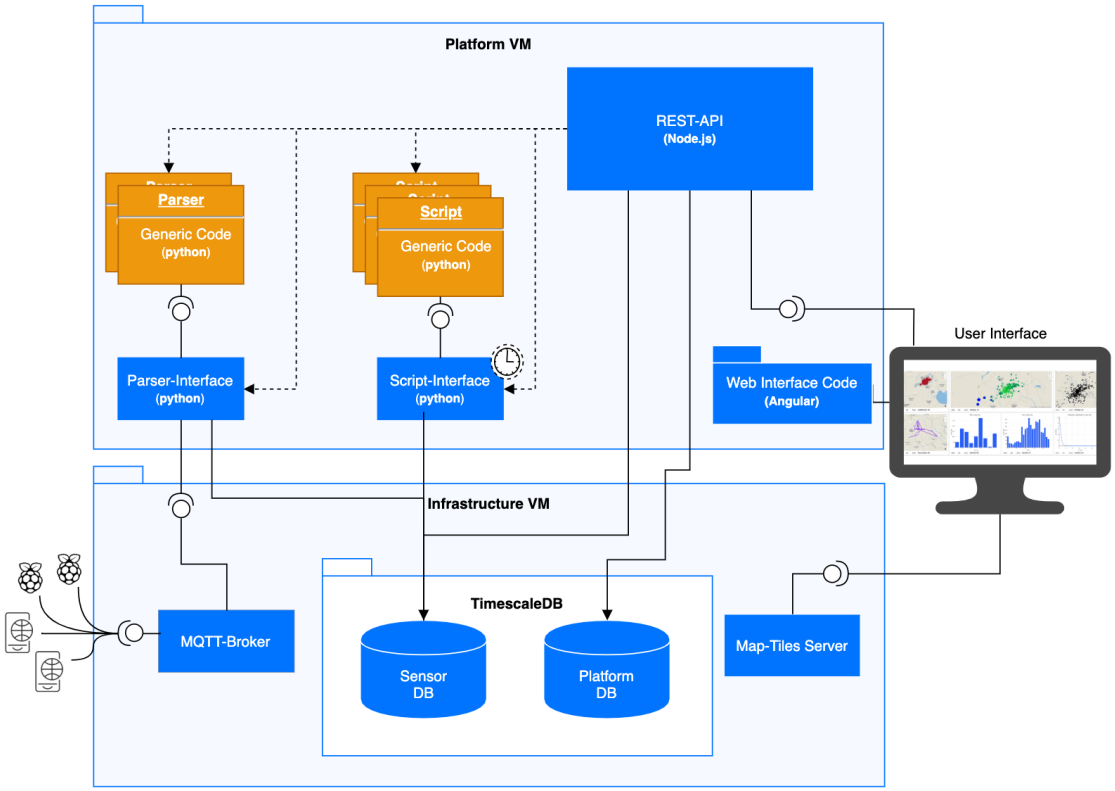


Figure 4.1: Components overview of the platform

Typescript's type inference abilities. This reduces bugs due to type mismatches during deployment and can increase development speed due to the autocompletion features IDEs provide based on the added types. Typescript's descriptiveness also improves readability and code maintainability. Because Typescript compiles to JavaScript, the "lingua franca" of the web, the produced code is executable by the Chrome-V8 JavaScript [40] engine and we can use much of the available tooling. It also allows our application to run on all browsers that support a recent JavaScript standard.

4.1 Data Ingestion

The sensor devices publish their data in an arbitrary encoding to our *Pub/Sub-Service*, where it is parsed by the user defined *Parsers* and stored in the "SensorDB".

4.1.1 Pub/Sub-Service

We run the MQTT-Broker *mosquitto* [41] behind a firewall. Therefore the broker itself does not handle the authentication of clients. Those are assumed to be already authorized if they are able to access the broker. We use the Quality-of-Service (QoS) level 2 to guarantee that the subscribers receive the messages each exactly once.

4.1.2 Data Format

We do not restrict the data format in which the sensors publish and only require that they publish to an MQTT-topic in the form `/<device_id>/<sensor_id>`. The published data can be any byte array. Two very common data formats for IoT and sensor data are protocol buffers and JSON strings. Protocol buffers provide structure while minimizing the format-overhead, but also require the sensor and server to have a *synchronized* protocol buffer message type in form of a compiled .proto-File. Because of this required compilation step the resulting messages are not human-readible. The alternative JSON-Format has a larger format overhead, but does not require the sensor and server to be synchronized. This makes sense if the data-structure often changes, especially if sensors regularly add fields that do not need to be known for the filtering step. In the *Evaluation* section we provide examples using both formats.

4.2 REST-API

This is the central piece of the platform and manages the parsers, scripts and visualizations. It provides an interface for those services as a REST API.

4.2.1 Framework

With the asynchronous event driven JavaScript runtime Node.js [42] it is easy to build scalable network applications. Node.js does not use threads but instead a so called event-loop runtime construct and callbacks. We therefore do not have to worry about deadlocks because Node.js does not have the concept of locks. We can still fork child processes to perform other tasks of e.g. the processing-unit, in parallel. Specifically we use the Node.js web application framework Nest.js [43], which is scalable and loosely coupled and most importantly provides many features to help structure the application and to provide HTTP utilities and middlewares.

Object Relation Mapper (ORM) For the platform itself we use a simple PostgreSQL database (Platform DB in Figure 4.1) and the object relation mapper (ORM) *TypeORM* [44] on top of it. TypeORM is highly influenced by Doctrine [45] and similarly stores each entity in a file, specifying e.g. relations simply by annotations. We use an ORM here, because only the platform developer has to learn the ORM's syntax and it simplifies the queries. We use the DataMapper pattern [46], because it allows great flexibility between the domain and the database.

Automatic Documentation We use Swagger UI [47] to visualize our API's resources. It is automatically generated from our module's controllers and is therefore not prone to be out of date. The REST-API serves it per default on port 3001.

Logging We log all errors to a file called `error.txt` to track possible errors during production and store actions and important events in a file called `log.txt` to support the debugging of the production system and to keep track of the actions users take.

Configuration To easily deploy the API on different platforms, e.g. switching between the production and the development environment, one only has to set the environment variable `NODE_ENV` to "production" or "development". It is possible to add other environments by adding a new `<environment_name>.env` file to the folder `environments`. There one can specify all the variables to fit the requirements and then activate it by setting the variable `NODE_ENV` to `<environment_name>`. Subsequently, when building the API the specified configuration will be used.

4.2.2 Parser

The parsers can be uploaded over the RESTful API and subscribe to the Pub/Sub-Service. They have to consist of a `main.py` file that implements three required methods (`init`, `on_message`, `clear`), and have to specify a title and a topic to subscribe to.

Once uploaded the parsers will always be started with the REST-API. Because they should not override their past state, every time they get restarted, they have to implement separate `init` and `on_message` methods. The `init` method is only run directly after the parser is uploaded and should be used to instantiate the table it needs or other setup it requires. The `on_message` method is called every time data is published to the specified topic and the `clear` method can be manually triggered with the UI-button "Clear data". This is useful in the case one wants to delete all the previously collected data, but one can use this method also for arbitrary other tasks. The passed parameters of the methods (seen in Figure 4.2) are the following:

- `parser_db_connection`: A `psycopg2` connection to the parser database.
- `parser_db_cursor`: A `psycopg2` cursor for the parser database connection. It can be used to write the parsed data to the database.
- `msg`: The published message object. Depending on the publisher this can be encoded as e.g. protocol buffer or a JSON-string

```
def init(parser_db_connection, parser_db_cursor):
[...]

def on_message(msg, parser_db_connection, parser_db_cursor):
[...]

def clear(parser_db_connection, parser_db_cursor):
[...]
```

Figure 4.2: Signatures of the methods that need to be implemented in the main.py file of the parser

Handling Parser Dependencies Optionally one can provide other python files to e.g. specify a protocol buffer object required by the parser. We also allow the user to upload a list of packages to be installed by the python package manager *pip*. They will then be installed for each parser in a separate conda-environment. By default, we install the conda packages `pip` and `psycopg2` and the pip packages `python-dotenv`, `paho-mqtt` and `protobuf`.

4.2.3 Processing-Unit

Processing scripts transform the parsed data for later visualization and exploration and are called from the REST-API application. The platform includes a script-interface that connects to the database and then calls the processing scripts, passing them the database connection similar to the parser. While the platform itself allows generic scripts we will in the following focus only on python-scripts. Those processing scripts have to implement a method with the following signature, which will be called by the script-interface:

```
def run(cursor, create_table, writer)
```

The passed parameters are the following:

- cursor: A `psycopg2` cursor for the sensor database connection. This can be used to query for the structured sensor data or the output of other scripts.
- create_table: This function has to be called with the sql-string of the required table fields (e.g. `id serial PRIMARY KEY, timestamp bigint`) to create the output table.
- writer: This function is used to write the transformed results back to the database in batches. Its parameters are:
    1. The table fields as list of strings
    2. A list of a list of values

The user can create new processing-units by uploading the scripts through the Web-Interface. The Web-UI and the API then enable the user to run those scripts, to terminate them or to monitor the output. To support other programming languages one needs to implement a matching script- and parser-interface.

Handling Script Dependencies Most scripts have external dependencies. Those dependencies can be conflicting between different scripts for example if two scripts require other versions of a package. Therefore we decided to run each script in a virtual environment and let it pass a list of required packages. We allow the user to upload a list of packages to be installed by the python package manager *pip*. They will then be installed for each script in a separate conda-environment. By default, we install the conda packages `pip` and `psycopg2` and the pip package `python-dotenv`.

Debugging The code of the script-interface is open-source with the rest of the platform and can be run locally, allowing the user to use his favorite debugging tools. To debug errors that only arise while deployed to the production system, the platform displays the error-output from the scripts in the Web-UI.

Scheduling Scripts The user can schedule the execution of scripts by defining cron [48] jobs. By passing a priority queue of the scripts the user can specify in which order scripts should be run. If a script fails, then all scripts that follow in the priority queue will not be run. The time-intervals have to be specified following the syntax of cronjobs (reference). The REST-API passes those cronjobs then to the *crontab* command. They will therefore run independently of the REST-API on the platform server.

4.3 Visualization

To provide a simple visualization that is open-source, free to use and contains the required charts for spatio-temporal data we have implemented a visualization UI. This gives the user a dashboard for each project in which he can create sheets, containing the charts. The UI also provides interactive access to manage the parsers and the scripts. Before going into their details we will present the framework we used and the programming paradigms that are common for them all.

Angular Angular [49] is a platform to build Typescript-based client applications. Specifically for interactive single page applications (SPAs) Angular brings many useful features. Those range from utilities for client side routing, over dependency injection to keep the code lean, to templates that combine HTML with Angular markup to modify the HTML elements before being displayed. This binding between the Angular markup and HTML elements is at the core of the Angular architecture and helps to circumvent many pitfalls of highly interactive applications like unsynchronized Views and Models. It is easy to integrate the following paradigms into Angular and to write clean and modular code.

4.3.1 Applied Programming Paradigms

Model to View Transformation Before we can display, e.g., our Model from the database, we have to transform it to the form we want to display it in. Because we want an interactive UI we have moved the Model to View transformation to the client. Angular provides many useful features for this transformation, most importantly data bindings and change detection. Data bindings define the communication between the Model in Angular and the View of the Document Object Model (DOM). The DOM is a programming interface representing the HTML document as a tree where each node relates to an object in that document. Change detection detects when the Model has changed since it was rendered the last time and updates the View. Because these

DOM updates are expensive Angular provides many techniques to trigger them only if needed.

State Management We use a centralized store architecture, similar to an in-memory database. This store is used for the application state and for client side caching. Instead of storing our data in each Angular-Component we move it all into this store. Conceptually we therefore also move the“Model to View Model Mapping” from the backend to the client. Instead of storing a representation of the data in each chart that needs it we store it in the store service and inject it into all the charts that need the data. If a component wants to modify the data it sends an “Action object" to the store service. The components are therefore not tightly coupled. The store service then uses the “Action objects" information to create an updated version of the state and broadcasts it to all interested components. This makes sure that all View Models are in sync with the data model. In our implementation we use reactive programming and the reactive programming, state management library *ngxs* [50] to handle the state.

Reactive Programming In reactive programming everything from a variable to a http-request can be a data stream. The library ngxs provides observables and operators to transform, filter and select data from the event streams. It combines the Observer and Iterator pattern [51] and blends them with functional programming. This makes it easy to handle asynchronous streams of data. Because our interactive UI creates many events that we have to handle in an asynchronous way we can simplify our code by using the reactive programming paradigm and the ngxs operators.

Client Side Routing We dynamically rewrite our current page instead of loading new pages from the server. Such applications are called Single Page Applications (SPAs) and are common for dashboards and interactive web applications. The server serves for all URLs the same `index.html` that bootstraps the application. The routing is fully done on the client side.

4.3.2 Web-UI

The Web-UI allows the user to interactively access information from the REST-API and execute commands. There he can create projects, each encapsulating its own scripts, parser and dashboard. In each dashboard he can create the visualization sheets containing the charts (see Figure 4.3 for structure).

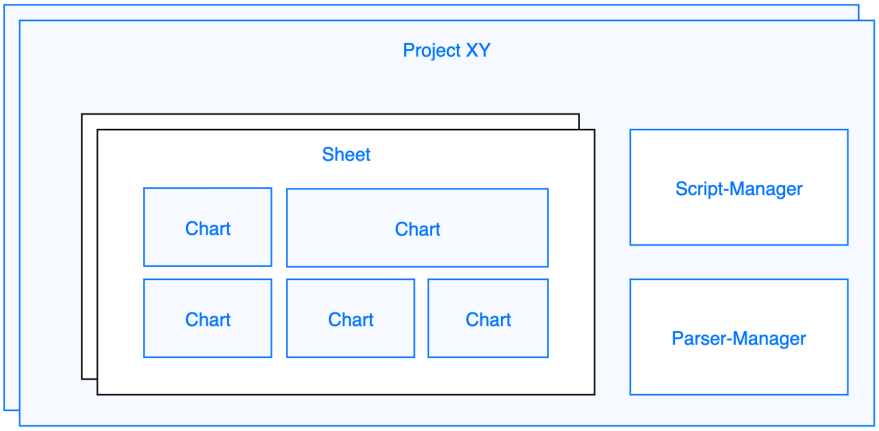


Figure 4.3: Structure of the projects

Dashboards They allow the visualization and exploration of the processed data. Each project's dashboard can have multiple sheets to separate different topics in each project. Each sheet can contain multiple charts. Charts have a specific type which can range from a classic Line-Chart to charts for spatial data like a Heatmap-Chart.

Chart Editor To select what data to visualize we can use the chart editor. We have to specify from which processed table we want to display data. We can not display data from multiple tables. Merging multiple data sets has to be part of the data transformation step. Each chart type then has a list of input fields. For example a Line-Chart needs to know which data to plot on the x- and the y-Axis and a Bar-Chart needs to know which fields to cluster by. These fields can than be selected in the chart editor from the list of available fields in the selected data-store table. The selections are then stored in the Platform DB.

Charts They contain the graphs visualizing the data. We read the selected fields to display from the platform's database as specified using the chart editor or the REST-API. The component then independently selects the data from the store. Currently supported charting libraries are *plotly* [52] for simple graphs and *mapboxGL* [53] for maps. Each chart is associated with a chart-type and each chart-type with a charting-library. Adding new charting libraries does not break existing charts. Currently supported chart-types are the following:

- Line-Chart + Bar-Chart
- Route-Map: Displaying the trajectory a list of coordinates represent

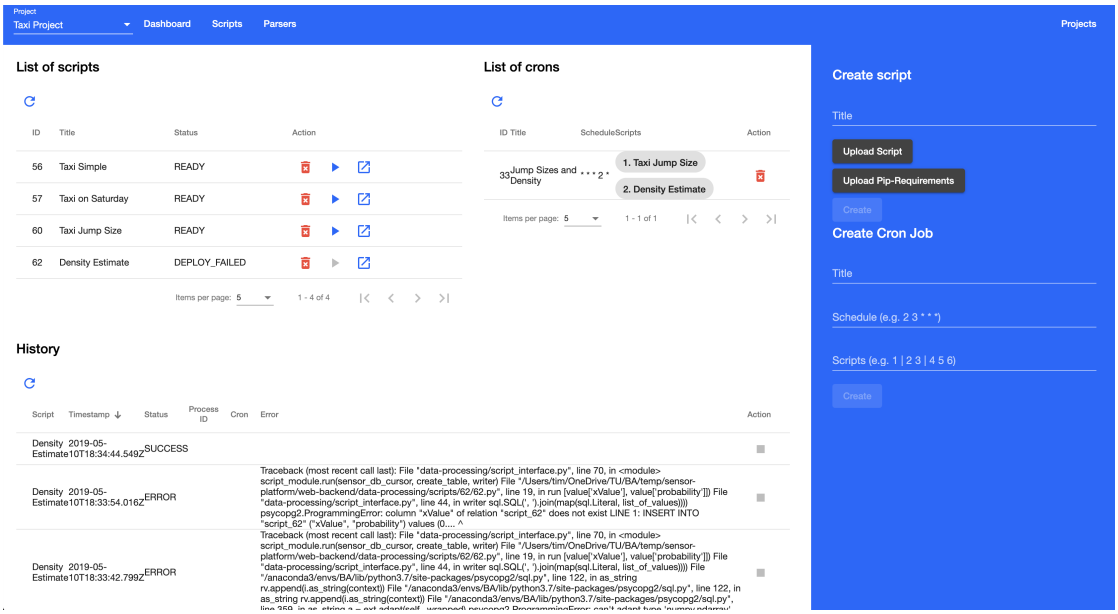


Figure 4.4: User interface to create scripts and cronjobs and display the status of script executions

- Heat-Map
- Density-Chart

Examples for each type are provided in the *Evaluation* Section.

Script-Manager This provides a UI to execute, terminate, delete and create scripts, as well as a history of the script-runs. Creating cronjobs is also possible from this component, as shown in Figure 4.4.

Parser-Manager This provides a UI to create, delete and list existing parsers. It also provides for each parser a button to call its "clear" method that depending on the parser can be used to e.g. clear the already collected data. Deleting a parser does not remove the collected data (no cascading).

4.4 Infrastructure

We run the platform on two virtual machines (VMs). On the *Infrastructure*-VM we run the databases, the tile-server and the Pub/Sub-Service and on the *Platform*-VM we run the backend (Parser, Scripts, REST-API) and the frontend (Visualizations, UI) as depicted in the component diagram (Figure 4.1).

4.4.1 Data Storage

We chose TimescaleDB because it provides a pure SQL interface, the stability of PostgreSQL and outperforms in many benchmarks other common TSDBs like InfluxDB. We use it for all our data stores, from the structured to the processed data. We have split the data store in a database containing all the platform related data (Platform DB), e.g., projects and list of charts and a database to store the structured and processed data (Sensor DB). We decided against using two separate databases for structured and processed data to simplify the data access for the user. Because by using the same data store we allow the scripts to access results from other scripts, which is useful for scripts that depend on each other. The structured and processed tables are transformed to TimescaleDB tables. Those tables will automatically be split into chunks by TimescaleDB with their own index trees to circumvent the "swap-to-disk" problem of large index trees.

4.4.2 Tile Server

We are running an openmaptiles-server [54] using docker [55] on the infrastructure server and serve the "whole planet" data set from openstreetmap [37] with the style klokantech-basic. The docker container is configured with the restart policy "always" to automatically restart after a server restart.

4.5 Code Design

The code is open-source and publicly available[1]. As this thesis will serve as the documentation of the current state of the software we will discuss the code structure and code design here as well.

[1] https://github.com/tpfeifle/STAV

REST-API The Node.js API is structured into modules representing functional units. We implemented the following utility modules that are used in multiple other modules:

- Config: Loads the configuration as specified in the *NODE_ENV* environment variable (default: development)
- Logger: Logs errors and user actions to the log.txt and error.txt files.

The following modules expose functionality for the REST-API.

- Chart: CRUD[2] operations for charts
- Parser Manager: CRUD[2] operations for the parser and starting of the parser with the system
- Project: CRUD[2] operations for projects
- Script Manager: CRUD[2] operations, starting/stopping of scripts
- Sensor Table: Querying of the sensor data store
- Sheet: CRUD[2] operations for sheets

Each module consists of a *module declaration*, a *controller*, a *service* and optional *entity definitions*. The module declaration defines the dependencies to other modules. The controller defines the routes of the REST-API and extracts the HTTP parameters and data. It then calls the module's service which contains the database queries e.g. calls to manage the parsers and scripts. The uploaded parser are stored each in a separate directory under `/parser/<parser\_id>` and the scripts under `/data-processing/<script\_id>`. In their parent directory lies the respective interface which executes them. The project dependencies are handled with the Node Package Manager (npm) and we adhere to the style guidelines defined in the `tslint.json`.

User Interface The components are structured in hierarchical order, e.g., the project contains the dashboard, parser and scripts. Each component contains its Model View transformation code in its Typescript file, its styling in the Sassy CSS (SCSS) file and its layout in its HTML file. Apart from those we also have guard files that are executed before the client router navigates to the page and make sure that e.g. the required data is already loaded and else waits till it is loaded. The shared folder contains the state management of ngxs. There we have the Model Views that represent the types we store in our client-side store and we have the store itself. The store is our global state container and we have split it into separate functional units. Each of them has a set of actions to mutate the store. All the aforementioned components select the subset of

[2] **C**reate, **R**ead, **U**pdate, **D**elete

the store they need and are injected the updated content after each update. Actions describe the action to take and contain meta-data which the store uses to execute those actions. Similar to the REST-API we use npm to handle the dependencies and adhere to the style guidelines defined in the `tslint.json`.

4.6 Deployment

The platform services and the infrastructure services are deployed on separate virtual machines with the specifications of Table 4.1.

	Infrastructure-VM	Platform-VM
CPU	1 core	1 core
Memory	8GB	6 GB
Storage	200GB	20GB

Table 4.1: Specifications of the deployed virtual machines, that run on a Xeon E5-2630v4@2.2GHz with 768GB RAM

For deployment we have to compile the Typescript code. For the REST-API we can do this by executing the command `npm run start` which starts the Typescript-Compiler configured for the REST-API and serves the result with Node.js. We compile and minify the Web-UI with the Angular command line interface (CLI) `ng build --prod` to minimize loading times and save bandwidth. The result can be served using a web server.

4.7 Summary

We have implemented a solution consisting of a *Parser* to ingest sensor data from the Pub/Sub-Service, a *Processing-Unit* to execute generic scripts for data transformation and a *Visualization* interface for data exploration. We use data store tables as the common interface between the different pipeline steps and provide a REST-API for all of the mentioned services, as well as to query data from the data store. The user interface is build with the framework Angular and the REST-API with the Node.js framework Nest.js. For the data store we use a time series database build on PostgreSQL, called TimescaleDB.

5 Evaluation

Based on the problem statement set out to address in the Introduction, we will evaluate the platform with two different use cases, representative for real-world applications.

Representative Use Cases The following use cases make use of all the pipeline steps, from ingesting the sensor data to the final data visualization. We test the platform first on artificial data with a use case we call *Workplace*. In our second use case *Taxi Mobility* we evaluate our platform's map visualization and analysis capabilities by visualizing trajectories of taxis in Beijing. The workplace data set is with 4,000 data entries comparatively small compared to the taxi data set with ~790,000 data entries (see Table 5.1). In the first use case we highlight specifically the data ingestion and in the second use case the data processing.

5.1 Workplace

We simulate devices sensing their surrounding temperature, noise volume and CO_2 level. They send the data in batches in intervals of 40 seconds to the Pub/Sub-Service of our platform. In this artificial use case we assume that those devices are placed at different workplaces ranging from a production hall with many loud machines to an office at a university. We want to measure the relation between the hour of day and the sound volume for each location as well as patterns in the sensor data that might help us to discover if, e.g., someone opened a window or a machine stopped working.

[1] assuming the following data type sizes: integer: 4 Byte, datetime: 8 Byte, float: 4 Byte, string: 4 Byte per character (max. 5 characters)

	Workplace Sensors	Taxi Mobility
Size per data entry [1]	32 Byte	20 Byte
Number of data entries	4,000	787,754

Table 5.1: Comparison of the data-sets used in the two use cases

Data Format The devices sense with specific sensors the temperature, CO_2 level and noise volume. We identify the different workplaces by attaching a user name. To keep track of the changes over time we add a timestamp. Here is an example data entry:

```
temperature: 24.26634795194493
co2_level: 456.46546478476034
noise_volume: 50.95365834607114
name: "Franz"
timestamp: 1557461540
```

We minimize the required bandwidth by using protocol buffers to serialize the sensed data to byte arrays. Therefore we have to specify how our information is structured by defining a protocol buffer message type, as shown in Figure 5.1.

```
syntax = "proto3";
package sensing;

message WorkplaceSensor {
        double temperature = 1;
        double co2 = 2;
        double noise_volume = 3;
        string name = 4;
        int32 timestamp = 5;
}
```

Figure 5.1: workplace.proto: Example of a protocol buffer message type for the workplace sensor

We compile this message type to get a python file to serialize and deserialize our data entries. For comparison, we also created a JSON formatted string from the same data resulting in the following:

```
[{"timestamp": 1557461540, "temperature": 24.26634795194493, "co2":
    ↪ 456.46546478476034, "noise": 50.95365834607114, "name": "Franz"}]
```

By sending the data as a protocol buffer instead of as a JSON string we reduced the size of our data entries from an average 133.4 Bytes (JSON) to 40 Bytes (protobuf) by 70%. After serializing the data each device publishes the information to our Pub/Sub-Service with a topic in the format */<device_id>/workplace_sensor*.

Parser To receive and parse the data we need to upload a parser to the platform. By specifying our subscription topic as `/+/workplace_sensor`, where "+" is a MQTT

wildcard matching all strings at its level, we are able to subscribe to all devices of type "workplace_sensor". The parser for this use case is defined in the *main.py* listed in Figure 5.2. The init method creates the table to store the information in and is executed after uploading the parser. The on_message method uses the compiled message type as defined above to extract the data. We then anonymize it by replacing the user name with the respective sha-224 hash and then write it to the structured data store. We also need to upload the compiled message type for the parser. The provided clear method allows the user to clear the whole table if needed. This is useful to reset the parsed data if our sensors produced data that we are no longer interested in.

To evaluate the performance of our data ingestion we ingested 4,000 data entries encoded with protobuf and measured the execution time over 5 separate runs. We used a single parser without threading and published the data entries one after the other similar to the case of batch uploads for devices with limited network connectivity. As can be seen in Table 5.2, it took on average 13.39 seconds from publishing the data till all of it was parsed and stored in the data store.

Processing Scripts After arbitrary transformation code we then write the result back to the data store. Because we focus in this use case on the data ingestion we only copied the data from the structured data store to the processed data store. For the full data set this took on average over 5 runs 0.24 seconds (see Table 5.2). In the next use case we focus on the data processing and provide an example script (see Figure 5.5).

Visualization To plot the processed data we use the chart editor to select for each chart from which script result we want to display which fields (as shown on the right side in Figure 5.3). Depending on the chart type the required parameters change. For a line chart we can select which fields to display on the x- and y-axis. The same is true for the density chart. For the bar chart we can select the field over which we want to cluster and which entry we want to sum up (y-axis).
We visualized the temperature, CO_2 levels and noise volume for the two different work places (represented by the field *name* in our data set). In Figure 5.3 we visualize the data for the workplace *office* and can see two spikes in the noise volume (the third chart): One at 5:30 AM and one around 6:30 PM. In the first chart we see that the temperature rose during the day but had a drop around 2 PM, which might be due to e.g. somebody opening a window. In the second chart we see the distribution of CO_2 levels and the temperature and could investigate whether or not they are correlated.

```
from .workplace_sensor_pb2 import WorkplaceSensor
import hashlib
import datetime as dt

def init(parser_db_connection, parser_db_cursor):
        parser_db_cursor.execute("DROP␣TABLE␣IF␣EXISTS␣workplace_sensors;"
        "CREATE␣TABLE␣workplace_sensors␣"
        "(id␣serial,␣timestamp␣TIMESTAMP␣PRIMARY␣KEY,␣temperature␣float,␣"
        "co2_level␣float,␣noise_volume␣float,␣user_hash␣varchar(100));"
        "SELECT␣create_hypertable('workplace_sensors',␣'timestamp');")
        parser_db_connection.commit()

def on_message(msg, parser_db_connection, parser_db_cursor):
        workplace_sensor_data = WorkplaceSensor()
        workplace_sensor_data.ParseFromString(msg.payload)
        user_hash = int(hashlib.sha224(bytes(workplace_sensor_data.name, "
            ↪ utf-8")).hexdigest(), 16)
        parser_db_cursor.execute('INSERT␣INTO␣workplace_sensors␣'
                '(timestamp,␣temperature,␣co2_level,␣noise_volume,␣
                    ↪ user_hash)␣'
                'VALUES␣(%s,␣%s,␣%s,␣%s,␣%s)',
                (dt.datetime.utcfromtimestamp(workplace_sensor_data.
                    ↪ timestamp).strftime('%Y-%m-%dT%H:%M:%S.%f'),
                workplace_sensor_data.temperature,
                workplace_sensor_data.co2_level,
                workplace_sensor_data.noise_volume,
                user_hash))
        parser_db_connection.commit()

def clear(parser_db_connection, parser_db_cursor):
        parser_db_cursor.execute('DELETE␣FROM␣workplace_sensors')
        parser_db_connection.commit()
```

Figure 5.2: main.py - Parser for the workplace sensors

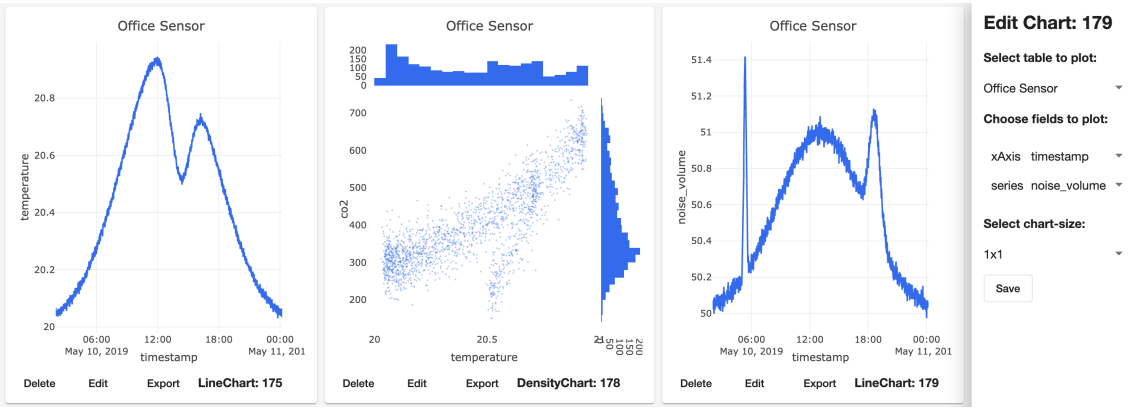


Figure 5.3: Visualization charts for the *office sensor* in the workplace data set

In Figure 5.4 we see the exemplary sensor output from a workplace at a *production line*. The temperature and CO_2 level distribution show that they are in this interval not correlated and from the third chart we see regular jumps in the noise volume. Those might be a machine producing parts in a fixed interval. We could then see from this sensor output, that it ran from 5 AM till 5 PM, but stopped between 12 PM and 2 PM. By exploring this data further one might be able to predict shift changes or machine malfunctions. To explore the data in more details e.g. at a specific time interval, the components allow us to zoom in. They can also be directly exported as images for use in papers or reports.

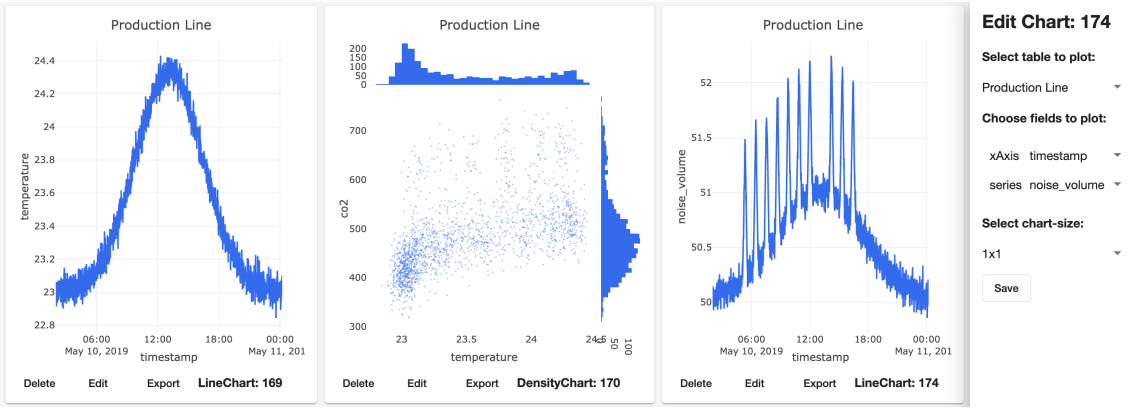


Figure 5.4: Visualization charts for the *production sensor* in the workplace data set

5.2 Taxi Mobility

In this use case we evaluate our data pipeline on the T-Drive trajectory data sample [56] prominently used in [57] and [58]. The data sample contains GPS trajectories of taxis in Beijing from February 2 to February 8, 2008. We use a subset of 500 taxis totaling in 787,754 data points, which is therefore more than two orders of magnitude larger than in the previous use case. With this data set we evaluate the platform's visualization capabilities for spatio-temporal data. While we highlighted the data ingestion in the last use case we here highlight the processing scripts and how to use external dependencies.

Data Format The data set consists of spatio-temporal data where each data entry contains the date time without time zone, the taxi id and the spatial coordinate in form of longitude and latitude. The data entries have the format `(taxi_id,timestamp ↪ ,longitude,latitude)` and the fields are separated by commas, as shown in the example below:

```
1,2008-02-02 15:36:08,116.51172,39.92123
1,2008-02-02 15:46:08,116.51135,39.93883
1,2008-02-02 15:46:08,116.51135,39.93883
...
```

We ingest the data directly to the data store and skip the Pub/Sub-Service for this use case. This took us 11 minutes with a single process and without threading.

Processing Scripts The visualizations do not allow us to combine results from different scripts in a single chart. Therefore we have to do the data modeling completely in the processing step. To discover mobility patterns in the data we run the following scripts on the same parsed data set:

1. Calculating the jump sizes
2. Estimating the probability density of the jump sizes
3. Calculating the center of mass and gyration radius

The jump sizes are defined as the distances between consecutive trajectory positions. We calculate them in kilometers and additionally extract the hour from the timestamp to later group the data points by it.

To estimate the probability density of the jump sizes we use the kernel density estimate from the external library *scipy*, as shown in Figure 5.5. We install scipy by adding the line `scipy=1.2.1` to our pip requirements and upload this file together with

```
import numpy as np
from scipy.stats.kde import gaussian_kde

def run(cursor, create_table, writer):
        create_table('id␣serial␣PRIMARY␣KEY,␣xvalue␣float,␣probability␣
            ↪ float')
        cursor.execute('SELECT␣jump_size␣FROM␣script_1')
        records = cursor.fetchall()

        jumps = np.array(list(map(lambda record: record[0], records)))
        kernel_density_estimate = gaussian_kde(jumps)
        x = np.linspace(0, np.max(jumps), 100)
        pdf_histogram = list(map(lambda xValue: {'xValue': xValue, '
            ↪ probability': kernel_density_estimate(xValue)[0]}, x))
        writer(['xvalue', 'probability'], pdf_histogram)
```

Figure 5.5: Estimating probability density of jump sizes using external libraries

our script. Because we do not want to calculate the jump sizes again, we instead use the result from our previous script (called here script_1, in general it is script_<script_id>) for our SQL query. To automatically execute the scripts in the required order we create a cronjob and assign the jump size script a higher priority in our priority queue (see Figure 4.4).

With our third script we calculate the center of mass for each taxi as well as its gyration radius. The gyration radius represents the linear size occupied by each taxi's trajectory up to the time t [59] and the center of mass is simply the average location of the taxi.

Visualization As a first step we visualize the data points. As can be seen in Figure 5.6 we plotted the trajectories, each colored by taxi. Because this plot is quite noisy and to further explore the data we decided to plot the data points, colored again by taxi, without the connecting paths between them.

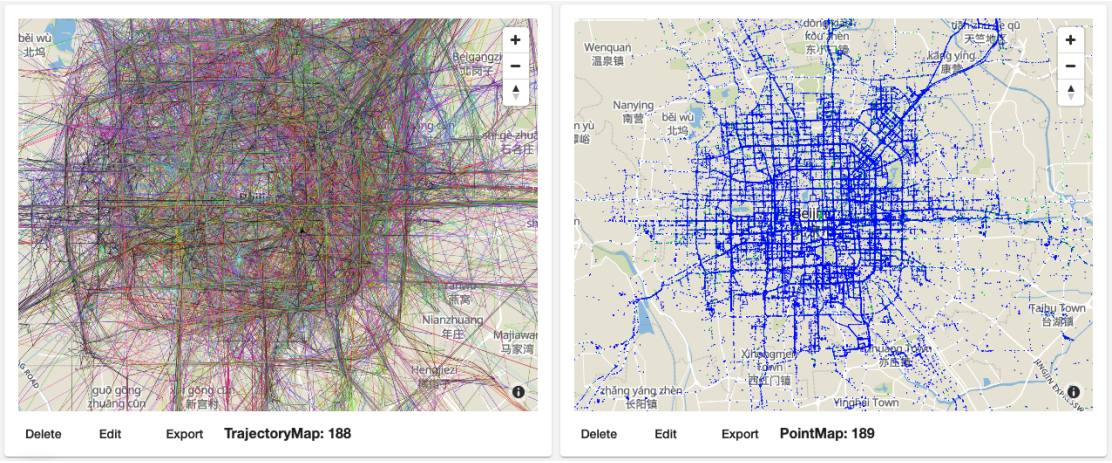


Figure 5.6: Left: Taxi trajectory map, Right: Taxi point map

This gives us a good overview over the data set. To answer the question during which times the taxis are driving the most we added the plots of Figure 5.7. In the right chart we use the Bar-Chart to group the extracted jump sizes by hour and sum them up. We can tell that the hours when the taxis traveled the most total distance during this week were 12 PM and from 5 PM to 6 PM. In combination with the Heat-Map on the left we can get a fast overview which regions head a high taxi density during a specific hour. A limitation is that we are not able to animate the Heat-Map over time with a time slider, but can only visualize it for specific hours.

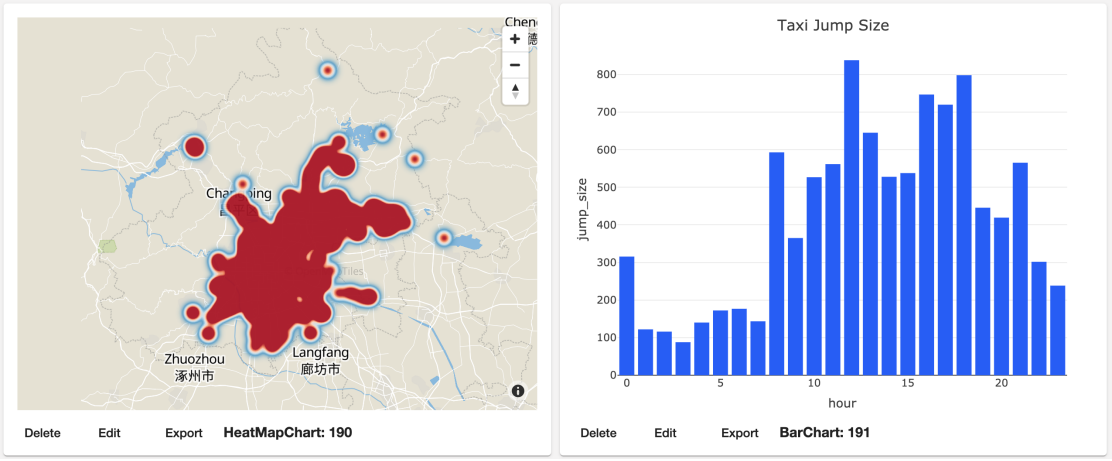


Figure 5.7: Left: Taxi heat map, Right: Summed jump sizes by hour of day

As a next step we grouped the jump sizes by taxi instead of by hour to determine which taxis drove the longest total distances (seen on the left side in Figure 5.8). This is in our case clearly taxi number 313. Following the paper [59] we used the right chart to display the estimated probability density of the jump sizes (see script in Figure 5.5).

We can see a large number of small jump sizes and a small number of larger jump sizes, which confirms the findings of [59] that most people travel only over short distances and only a few regularly move farther away.

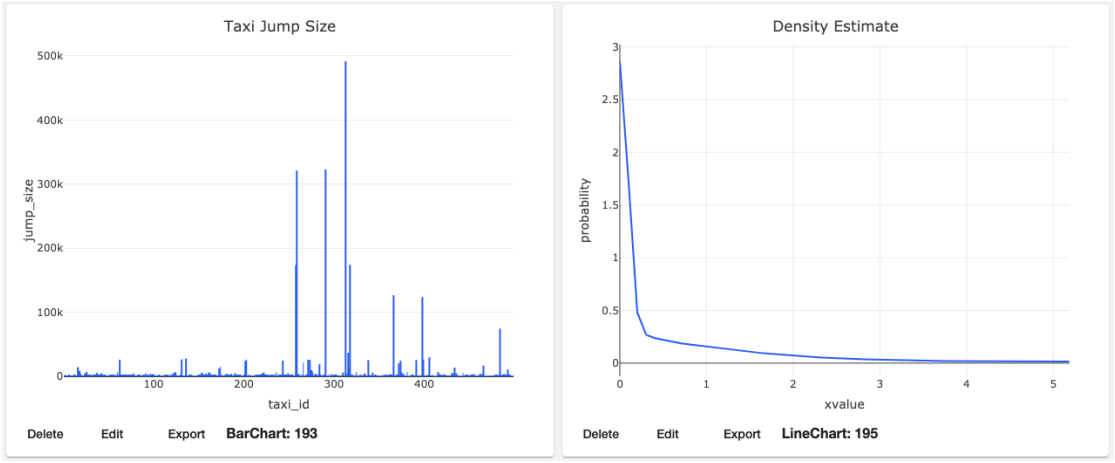


Figure 5.8: Left: Summed jump sizes by taxi, Right: Pdf estimate for jump sizes

For our last visualization we plotted the calculated center of mass locations and their radius of gyration for each taxi (see Figure 5.9). The size and color of each circle is determined by its gyration radius. By clicking on a point we can display the associated taxi id. This chart shows that taxis traveling to the regions in the north-east and south-west of Beijing traveled more often for larger distances outside of their center of mass then taxis in the center of Beijing, because their circles are larger and darker. Therefore those taxis are probably not regularly driving only in those regions, but instead had customers that wanted to travel so far out of Beijing. In the north-east where many customers want to travel to is the airport of Beijing, which corresponds to the pattern we discovered.

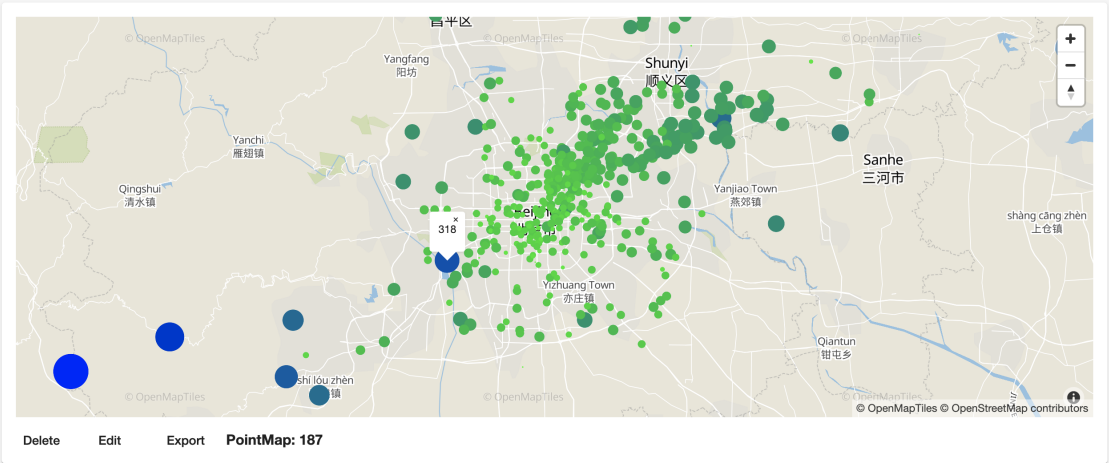


Figure 5.9: Gyration radius and center of mass for each taxi

5.3 Quantitative Evaluation

For the quantitative evaluation we ran the mentioned scripts and in some cases changed the input size to better compare their performance (details in Figure 5.2). The performance evaluations were run on a MacBook Pro 2016 for better control of the execution context. We did not parallelize the parser, scripts or database and did run all of them on a single CPU. We averaged each execution time over 5 runs and reported the sample standard deviation (denoted as s).

Speed of Inserts During the evaluation we discovered that the performance for large cardinality of inserts was far below the reported performance benchmarks of TimescaleDB. This was due to executing the inserts one at a time instead of using batch inserts, therefore requiring a full client to database round trip for every insert. After using the batch insert functionality of the psycopg2 module the insertion time for the Taxi Mobility data dropped by 96% from 11 minutes to 26.62 seconds. Because the parser inserts each data point on its own the performance is far below this rate. This is the reason it took more than 13 seconds to parse the 4,000 published data points. In general the bottleneck for the scripts is the insert performance of the database. While we were able to insert more than 20,000 rows per second we have not yet evaluated if the performance could be further increased with multiple database clients inserting in parallel.

Speed of Transformations For the calculation of the jump sizes we transform the around 790,000 data points in less than 3 seconds and even for the more complex calculation of the gyration radii we take less than 30 seconds (see Figure 5.2). The data querying scales to large data sets very well and at least in our use cases was not the limiting factor. Because the data ingestion and data processing are the main limiting factors the "PDF Jump Size" script outperformed the "Workplace" processing script even though it queries and processes more than 100x times the amount of data.

	Workplace	Jump Size	PDF Jump Size	Gyration Radius
Data points [2]				
Input	4,000	787,754	787,254	787,754
Output	4,000	787,254	100	500
Durations [3]				
Parsing	13.39 (s=0.133)	26.62 (s=5.017) [4]	26.62 (s=5.017) [4]	26.62 (s=5.017) [4]
Processing	5.45 (s=0.179)	33.96 (s=0.963)	2.83 (s=0.106)	29.02 (s=0.675)

Table 5.2: Comparison of the different scripts regarding execution time and amount of processed data points.

5.4 Summary

The presented solution integrates all steps from the OSEMN-Pipeline, from obtaining the data to interpreting the data, in one platform. While there are more powerful solutions for the different steps in the pipeline, e.g., Apache Airflow for the transformations, Kafka for data ingestion or Tableau for visualization, this solution makes it really easy for the users to perform highly custom data modeling and easily access the sensor-data as well as quickly create visualizations. It is still modular enough to replace every part by one of the more powerful solutions with the disadvantage of increasing the complexity. This becomes increasingly useful as one has more complex requirements, but always adds a lot of complexity on the platform side as well as for the user who writes the transformation-scripts to integrate them with their API. Further work could focus on improving the performance of batch inserts into the data store, adding a stream processing engine or more visualization features.

[3] number of data points in rows
[4] in seconds
[5] the time it took to write the entries from disk to the database

List of Figures

List of Tables

Bibliography

[1] I. Gartner. *Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016*. 2017. URL: https://www.gartner.com/en/newsroom/press-releases/2017-02-07-gartner-says-8-billion-connected-things-will-be-in-use-in-2017-up-31-percent-from-2016 (visited on 05/08/2019).

[2] A. W. Services. *AWS IoT*. URL: https://aws.amazon.com/de/iot/ (visited on 05/13/2019).

[3] T. Choudhury, G. Borriello, S. Consolvo, D. Haehnel, B. Harrison, B. Hemingway, J. Hightower, K. Koscher, A. LaMarca, J. A. Landay, et al. "The mobile sensing platform: An embedded activity recognition system." In: *IEEE Pervasive Computing* 7.2 (2008), pp. 32–41.

[4] N. D. Lane, E. Miluzzo, H. Lu, D. Peebles, T. Choudhury, and A. T. Campbell. "A survey of mobile phone sensing." In: *IEEE Communications magazine* 48.9 (2010).

[5] G. Atluri, A. Karpatne, and V. Kumar. "Spatio-temporal data mining: A survey of problems and methods." In: *ACM Computing Surveys (CSUR)* 51.4 (2018), p. 83.

[6] A. Eklund, T. E. Nichols, and H. Knutsson. "Cluster failure: Why fMRI inferences for spatial extent have inflated false-positive rates." In: *Proceedings of the national academy of sciences* 113.28 (2016), pp. 7900–7905.

[7] W. Feng, Z. Yan, H. Zhang, K. Zeng, Y. Xiao, and Y. T. Hou. "A Survey on Security, Privacy, and Trust in Mobile Crowdsourcing." In: *IEEE Internet of Things Journal* 5.4 (2018), pp. 2971–2992.

[8] F. Restuccia, N. Ghosh, S. Bhattacharjee, S. K. Das, and T. Melodia. "Quality of information in mobile crowdsensing: Survey and research challenges." In: *ACM Transactions on Sensor Networks (TOSN)* 13.4 (2017), p. 34.

[9] L. Sweeney. "Simple demographics often identify people uniquely." In: *Health (San Francisco)* 671 (2000), pp. 1–34.

[10] Y.-A. De Montjoye, C. A. Hidalgo, M. Verleysen, and V. D. Blondel. "Unique in the crowd: The privacy bounds of human mobility." In: *Scientific reports* 3 (2013), p. 1376.

[11] K. Abualsaud, T. M. Elfouly, T. Khattab, E. Yaacoub, L. S. Ismail, M. H. Ahmed, and M. Guizani. "A Survey on Mobile Crowd-Sensing and Its Applications in the IoT Era." In: *IEEE Access* 7 (2019), pp. 3855–3881.

[12] D. V. Pavlov. "Hive: An extensible and scalable framework for mobile crowdsourcing." In: *Diss. Imperial College London* (2013).

[13] M.-R. Ra, B. Liu, T. F. La Porta, and R. Govindan. "Medusa: A programming framework for crowd-sensing applications." In: *Proceedings of the 10th international conference on Mobile systems, applications, and services*. ACM. 2012, pp. 337–350.

[14] C. Howson, R. L. Sallam, J. L. Richardson, J. Tapadinhas, C. J. Idoine, and A. Woodward. "Magic quadrant for analytics and business intelligence platforms." In: *Retrieved Aug* 16 (2018), p. 2018.

[15] T. Software. *Tableau*. URL: `https://www.tableau.com/` (visited on 05/08/2019).

[16] Qlik. *Qlik*. URL: `https://www.qlik.com/` (visited on 05/08/2019).

[17] M. Corporation. *Power BI*. URL: `https://powerbi.microsoft.com/` (visited on 05/08/2019).

[18] W. Schlee, R. C. Pryss, T. Probst, J. Schobel, A. Bachmeier, M. Reichert, and B. Langguth. "Measuring the moment-to-moment variability of tinnitus: the TrackYourTinnitus smart phone app." In: *Frontiers in aging neuroscience* 8 (2016), p. 294.

[19] A. K. Sharma, M. F. R. Ansari, M. F. Siddiqui, and M. A. Baig. "IOT enabled forest fire detection and online monitoring system." In: *Int J Curr Trends Eng Res (IJCTER)* 3.5 (2017), pp. 50–54.

[20] N. Maisonneuve, M. Stevens, and B. Ochab. "Participatory noise pollution monitoring using mobile phones." In: *Information Polity* 15.1, 2 (2010), pp. 51–71.

[21] B. Pan, Y. Zheng, D. Wilkie, and C. Shahabi. "Crowd sensing of traffic anomalies based on human mobility and social media." In: *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM. 2013, pp. 344–353.

[22] T. Ludwig, C. Reuter, T. Siebigteroth, and V. Pipek. "CrowdMonitor: Mobile crowd sensing for assessing physical and digital activities of citizens during emergencies." In: *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. ACM. 2015, pp. 4083–4092.

[23] B. Guo, Z. Yu, L. Chen, X. Zhou, and X. Ma. "MobiGroup: Enabling lifecycle support to social activity organization and suggestion with mobile crowd sensing." In: *IEEE Transactions on Human-Machine Systems* 46.3 (2016), pp. 390–402.

[24] T. Akidau. "The world beyond batch: Streaming 101." In: *oreilly. com* 20 (2016).

[25] N. Marz and J. Warren. *Big Data: Principles and best practices of scalable real-time data systems*. New York; Manning Publications Co., 2015.

[26] M. Hausenblas. *Lambda architecture*. 2014. URL: https://berlinbuzzwords.de/sites/berlinbuzzwords.de/files/media/documents/michael_hausenblas_-_lambda_architecture.pdf (visited on 05/08/2019).

[27] Y. Xiao, P. Simoens, P. Pillai, K. Ha, and M. Satyanarayanan. "Lowering the barriers to large-scale mobile crowdsensing." In: *Proceedings of the 14th Workshop on Mobile Computing Systems and Applications*. ACM. 2013, p. 9.

[28] K. Birman and T. Joseph. *Exploiting virtual synchrony in distributed systems*. Vol. 21. 5. ACM, 1987.

[29] D. Krafzig, K. Banke, and D. Slama. *Enterprise SOA: service-oriented architecture best practices*. Prentice Hall Professional, 2005.

[30] R. Bayer and E. McCreight. "Organization and maintenance of large ordered indexes." In: *Software pioneers*. Springer, 2002, pp. 245–262.

[31] M. A. Mike Freedman. *TimescaleDB*. https://github.com/timescale/timescaledb. 2019.

[32] I. Timescale. *Time-series data: Why (and how) to use a relational database instead of NoSQL*. 2017. URL: https://blog.timescale.com/time-series-data-why-and-how-to-use-a-relational-database-instead-of-nosql-d0cd6975e87c/ (visited on 05/08/2019).

[33] Influxdata. *Flux - Influx data language*. URL: https://github.com/influxdata/flux (visited on 05/08/2019).

[34] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. "The log-structured merge-tree (LSM-tree)." In: *Acta Informatica* 33.4 (1996), pp. 351–385.

[35] I. Timescale. *TimescaleDB vs. InfluxDB: Purpose built differently for time-series data*. 2018. URL: https://blog.timescale.com/timescaledb-vs-influxdb-for-time-series-data-timescale-influx-sql-nosql-36489299877/ (visited on 05/13/2019).

[36] M. Arye. *SQL vs. Flux: Choosing the right query language for time-series data*. 2018. URL: https://blog.timescale.com/sql-vs-flux-influxdb-query-language-time-series-database-290977a01a8a/ (visited on 05/12/2019).

[37] *OpenStreetMap*. URL: https://www.openstreetmap.org/ (visited on 05/08/2019).

[38] M. Corporation. *Bing Maps Tile System*. 2018. URL: https://docs.microsoft.com/en-us/bingmaps/articles/bing-maps-tile-system (visited on 05/08/2019).

[39] J. Nowosad. *A relation between the Mercator projection and the true size of each country*. 2018. URL: https://upload.wikimedia.org/wikipedia/commons/e/ee/Worlds_animate.gif (visited on 05/13/2019).

[40] T. C. Project. *Chrome V8*. URL: https://v8.dev/ (visited on 05/08/2019).

[41] E. Foundation. *Eclipse Mosquitto: An open source MQTT broker*. URL: https://mosquitto.org/ (visited on 05/08/2019).

[42] *Node.js*. URL: https://nodejs.org/ (visited on 05/08/2019).

[43] *Nest.js*. URL: https://nestjs.com/ (visited on 05/08/2019).

[44] *typeORM*. URL: https://typeorm.io (visited on 05/08/2019).

[45] *Doctrine Project*. URL: https://www.doctrine-project.org/ (visited on 05/08/2019).

[46] M. Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[47] S. Software. *Swagger*. URL: https://swagger.io/tools/swagger-ui/ (visited on 05/08/2019).

[48] *SQL vs. Flux: Choosing the right query language for time-series data*. 2008. URL: http://pubs.opengroup.org/onlinepubs/9699919799/utilities/crontab.html (visited on 05/13/2019).

[49] Google. *Angular*. URL: https://angular.io (visited on 05/08/2019).

[50] *ngxs*. URL: https://ngxs.gitbook.io/ngxs/ (visited on 05/08/2019).

[51] E. Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[52] Plotly. *Plotly*. URL: https://plot.ly/ (visited on 05/08/2019).

[53] Mapbox. *Mapbox GL JS*. URL: https://docs.mapbox.com/mapbox-gl-js/api/ (visited on 05/08/2019).

[54] MapTiler. *OpenMapTiles Map Server*. URL: https://openmaptiles.com/server/ (visited on 05/08/2019).

[55] I. Docker. *Docker*. URL: https://www.docker.com/ (visited on 05/08/2019).

[56] Y. Zheng. *T-Drive trajectory data sample*. T-Drive sample dataset. Aug. 2011.

[57] J. Yuan, Y. Zheng, X. Xie, and G. Sun. "Driving with knowledge from the physical world." In: *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2011, pp. 316–324.

[58] J. Yuan, Y. Zheng, C. Zhang, W. Xie, X. Xie, G. Sun, and Y. Huang. "T-drive: driving directions based on taxi trajectories." In: *Proceedings of the 18th SIGSPATIAL International conference on advances in geographic information systems*. ACM. 2010, pp. 99–108.

[59] M. C. Gonzalez, C. A. Hidalgo, and A.-L. Barabasi. "Understanding individual human mobility patterns." In: *nature* 453.7196 (2008), p. 779.