Data Analysis Project
P3: Data Analysis and Visualization
Analysis of Speed and Traffic Congestion in Basel (Group 10)

Severin Memmishofer and Yash Trivedi

University of Basel
Databases (CS244) course
Autumn Semester 2022

1 Introduction

In this project, we aim to get insight into how the amount of vehicles influences the average speed at certain areas around the city. At which point does the traffic get too congested and vehicle speeds decrease? Which areas are most prone to speeding? What other factors and variables influence speeding?

2 Datasets

We will be using the following datasets for our project:

Verkehrszähldaten motorisierter Individualverkehr

- Source: https://data.bs.ch/explore/dataset/100006
- Size: 1.1 GB
- Format: JSON

Geschwindigkeitsmonitoring: Einzelmessungen ab 2021

- Source: https://data.bs.ch/explore/dataset/100097
- Size: 8.05 GB
- Format: JSON

Both datasets list data collected from the beginning of 2021 onwards, and were downloaded on the 15.10. (Newer data not included)

2.1 Verkehrszähldaten motorisierter Individualverkehr

This dataset consists of vehicle counts at different locations through Basel-city. It also differentiates between different types of vehicles like personal vehicles and trucks. The dataset counts from 2021 onwards.

2.2 Geschwindigkeitsmonitoring: Einzelmessungen ab 2021

This dataset consists of measurements of the speed of motor vehicles at certain locations in the city of Basel. It is measured by the cantonal police of Basel-city and serves to make decisions, where speed controls are necessary. The measurements begin in January of 2021.

3 Analysis Goals

Our goals are to analyze the traffic flow and connection between speeds and traffic congestion.
We also aim to answer some additional questions, if time allows, about possible causes / tendencies for speeding, like: Are people more likely to speed at certain times during the day (f.ex. when they get home from work)? Is speeding dependent on specific weekdays? Are people more likely to speed on roads with low traffic, or with more traffic (loosing their temper, trying to get home quickly)? What are speeding hotspots in the city?

3.1 Tools Used

Purpose	**Tool Name**	**Remarks**
Schema Creation	Draw.io	We used it to create the ER Diagrams
Documentation	Latex	All the Documentation was done using Latex
Data Integration	mySQL Server using Datagrip, Data cleaning: Python using Pandas library	We first cleaned our data in Python and later integrated it into our Database using mySQL
Data Analysis	mySQL using Datagrip	We mostly performed SQL queries to get the information we wanted for our Analysis. For detailed explanation look at section Analysis in this document. Our analysis.sql can be found on Gitlab.

Purpose	Tool Name	Remarks
Data Visualization	Python, PowerBi	Libraries used: sqlalchemy (connecting to Database), pandas, pymysql, matplotlib, seaborn, db_config, decimal, sklearn, folium (visualization of the map). We also used the help of ChatGPT (for researching purposes) for certain plots in our jupyter notebook.

4 Entity Relation Diagram per Data Source

4.1 Geschwindigkeitsmonitoring: Einzelmessungen ab 2021

See figure 1.

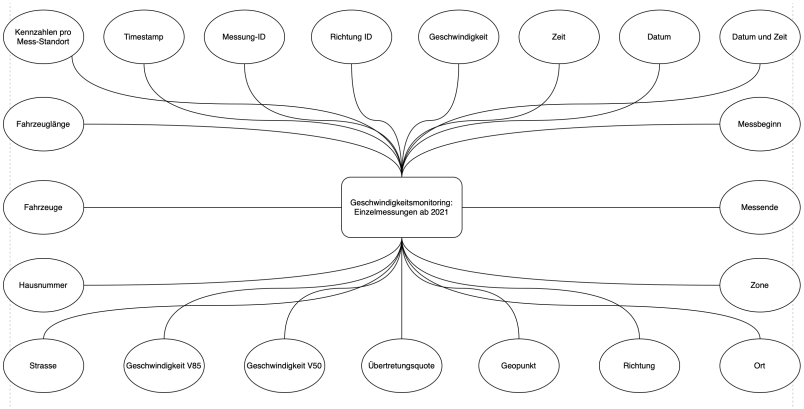

Fig. 1: Geschwindigkeitsmessungen

4.2 Verkehrszähldaten motorisierter Individualverkehr

See figure 2.

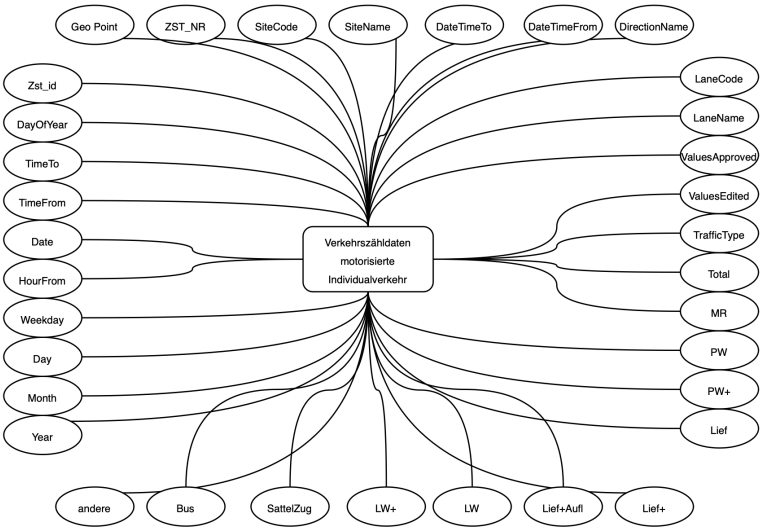

Fig. 2: Verkehrszählldaten

5 Integrated ER-Diagram

To make the image less cluttered, we will only include the attributes in our integrated ER-Diagram, which are relevant for analysis. The other attributes will still be present in our Database, we didn't drop the data. See figure 3.

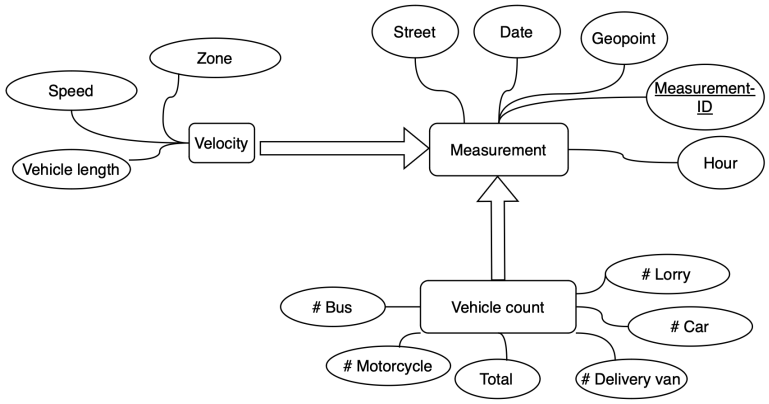

Fig. 3: Integrated ER-Diagram

6 Relational Schema

6.1 Logical relational schema

Measurement(Measurement-ID, Hour, Date, Street, Geopoint)
Velocity(Measurement-ID, Speed, Zone, Vehicle length)
Vehicle count(Measurement-ID, Total, # Delivery van, # Motorcycle, # Car, # Lorry, # Bus)

7 Database Setup Guide

7.1 Step 1: Clean the datasets

Make sure that there exists a directory named 'DataSources' in the same directory, where all the python files are located.

Make sure that the datasets exist in the 'DataSources' directory in a .csv format, and that they are named correctly; e.g. '100006.csv' and '100097.csv'. (As they are downloaded from the source; 'data.bs.ch').

Now you are ready to run the 'datacleaning.py' code; which calls the other two python files and performs the respective data cleaning steps of the data sources.

If everything worked correctly, there should now exist a folder in the same directory, named 'cleanedFiles'.

7.2 Step 2: Create tables in DataGrip

Make sure that MySQL and DataGrip (or any of your desired database tools) are configured and installed correctly. Start a mySQL server inside DataGrip (or connect to it, if it's already running) Open and run 'createTables.sql' script in DataGrip.

7.3 Step 3: Import data into mySQL via the terminal

First we need to set up the mySQL commandline interface. *Note: This step may work differently, depending on your system configuration and the OS used.* For us it worked like this (for MacOS):

```
cd /usr/local/mysql-8.2.0-macos13-arm64/bin
```

Modify the following file: `open -e ~/.zshrc` and insert the following line:

```
export PATH="/usr/local/mysql-8.2.0-macos13-arm64/bin:$PATH"
```

Now you should be able to directly execute SQL commands like this:

```
mysql --version
```

If this command works, and gives you the SQL version, you should be ready to go.

Log into mySQL using the following command and enter your password:

```
mysql --local-infile=1 -u root -p
```

Note: we used the root user/account; change this depending on the account used

Enable local infile data loading via the following command, if it isn't already:

```
set global local_infile=true;
```

to check, if it worked correctly:

```
show global variables like 'local_infile';
```

This should now show the 'local_infile' variable as ON.

Now, we can finally load the Data into the database, using the sql script 'loadDataInfile.sql'. Run this in the mySQL command line interface:

```
source 'PathToSQLFile'
```

Note: adjust the file path depending on your local system and storage location; enter the absolute path to the location of the 'loadDataInfile.sql'file

Now, after this is done correctly, you should be able to take a look at the tables inside DataGrip.

7.4 Step 4: Create the Integrated Database on mySQL

Run the 'integrated.sql' script inside DataGrip. This should load the data into the integrated SQL table, and performs some additional steps, like creating coherent artificial primary keys.

8 Data Access

The SQL Dump can be accessed via the following access link for SWITCHfilesender:

https://filesender.switch.ch/filesender2/?s=download&token=d88d091d-c4ea-4b69-906d-75bd5ed60063

9 Revision

As a small revision, compared to the integrated ER diagram in the previous milestone, we added attribute 'TIME' to the 'Velocity'-table, as we discussed with the tutors. This way, we don't lose any granularity, and if we might need the exact time of a Velocity Measurement, we already have it in the integrated Database. This can be seen in the following CREATE TABLE SQL statement:

```
CREATE TABLE Velocity (
    Measurement_ID INT PRIMARY KEY,
    Time TIME,
    Speed INT,
    Zone INT,
    Vehicle_length DECIMAL(3, 1),
    FOREIGN KEY (Measurement_ID) REFERENCES Measurement(Measurement_ID)
);
```

10 Analysis

In the following subsections, we will explain our SQL queries used for the analysis. Since the whole document is quite long, we will explain the steps on some example queries; the whole document can be found at the end or on github.

10.1 Find close measuring stations

We first updated our table, with the location info as *POINT* datatype in SQL, since we decided to use String for the geopoint information when originally integrating the data. For this we added a new Column, and converted the geopoint string back to a POINT datatype.

```
ALTER TABLE measurement
ADD COLUMN Point POINT;

UPDATE measurement
SET Point = ST_PointFromText(CONCAT('POINT(', SUBSTRING_INDEX(Geopoint, ', ', -1),
' ', SUBSTRING_INDEX(Geopoint, ', ', 1), ')'))
WHERE Geopoint IS NOT NULL AND Geopoint != '';
```

In the next step, we created a set of unique geopoints (which correspond to measuring stations) for each dataset from our integrated schema, namely the vehiclecount and velocity datasets.

```
CREATE TEMPORARY TABLE temp_distinct_geopoint_vc AS
SELECT DISTINCT m.Point, m.Geopoint
FROM measurement m
INNER JOIN vehicle_count vc ON m.Measurement_ID = vc.Measurement_ID;

CREATE TEMPORARY TABLE temp_distinct_geopoint_vel AS
SELECT DISTINCT m.Point, m.Geopoint
FROM measurement m
INNER JOIN velocity vc ON m.Measurement_ID = vc.Measurement_ID;
```

Subsequently, we wanted to find out, whether there were some measuring stations of speed and vehicle counts, which were close to each other. For this we used the *ST_Distance_Sphere* function from mySQL, which returns the minimum spherical distance between Point arguments on a sphere, in meters.

```
CREATE TABLE close_points_50
SELECT
    vc.Point AS VC_GeoPoint,
    vc.Geopoint AS VC_GeoPoint_var,
    vel.Point AS VEL_GeoPoint,
    vel.Geopoint AS VEL_GeoPoint_var
FROM
    temp_distinct_geopoint_vc vc,
    temp_distinct_geopoint_vel vel
WHERE
    ST_Distance_Sphere(vc.Point, vel.Point) <= 50;
```

We also created similar tables for 100 and 200 metres. We thus ended up with tables, which contain the measuring stations of speed and vehiclecounts, which were within a distance of 50, 100 or 200 metres of each other. In a first step, we decided to use the 100m measurement table, which contained 15 Pairs of measuring locations. We thought, 200m was too much, and 15 was a good number to work with, while the 50m distance only amounted to 8 Pairs. After that, we created new, smaller tables, which only contain the data from the stations in this 'close' set, to only work with the relevant data, which also significantly speeds up the remaining querys, and the rest is not relevant for this part of our analysis anyways.

```
CREATE TABLE close_measurements AS
SELECT m.*
FROM measurement m
INNER JOIN close_points_100 cp
ON m.Geopoint = cp.VC_GeoPoint_var OR m.Geopoint = cp.VEL_GeoPoint_var;
```

```
CREATE TABLE new_vehicle_count AS
SELECT vc.*
FROM vehicle_count vc
INNER JOIN close_measurements cm
ON vc.Measurement_ID = cm.Measurement_ID;

CREATE TABLE new_velocity AS
SELECT vel.*
FROM velocity vel
INNER JOIN close_measurements cm
ON vel.Measurement_ID = cm.Measurement_ID;
```

Our initial plan was, to check the dataset, if there were measurements of both speed *and* vehicle counts, happening on the same date at the same location. This would provide the most accurate data. For this, we first joined the new tables with the close measurements again, to get the Date.

```
CREATE TABLE vc_date AS
SELECT vc.*, cm.Date, cm.Hour, cm.Street, cm. Geopoint, cm.Point
FROM new_vehicle_count vc
INNER JOIN close_measurements cm
ON vc.Measurement_ID = cm.Measurement_ID;

CREATE TABLE vel_date AS
SELECT vel.*, cm.Date, cm.Hour, cm.Street, cm. Geopoint, cm.Point
FROM new_velocity vel
INNER JOIN close_measurements cm
ON vel.Measurement_ID = cm.Measurement_ID;
```

However, before inspecting the date, we thought it would be wise to inspect the remaining geopoints on a map to analyse, if they really correspond to the same street, and if the data is comparable. For this we analyzed the data in PowerBI, where we created two maps, one showing the vehicle count measuring stations and the other one showing the speed measuring stations in the close_points_100 table. There we also realised, that the velocity dataset had a total of 15 points showing, while the vehiclecount dataset only had 12, so we had to have some duplicates.

We first deleted the Points at 'Lindenhofstrasse', as they had no counterpart in the vehiclecount dataset. (Not shown in the map, already deleted beforehand)

```
DELETE FROM vel_date
WHERE vel_date.Street = 'Lindenhofstrasse';
```

As a next step, we found out that the two geopoints in the velocity dataset at 'Gundelingerstrasse' were very close together, in fact so close, that they can be considered the same. We suspect that the measuring station was set up two

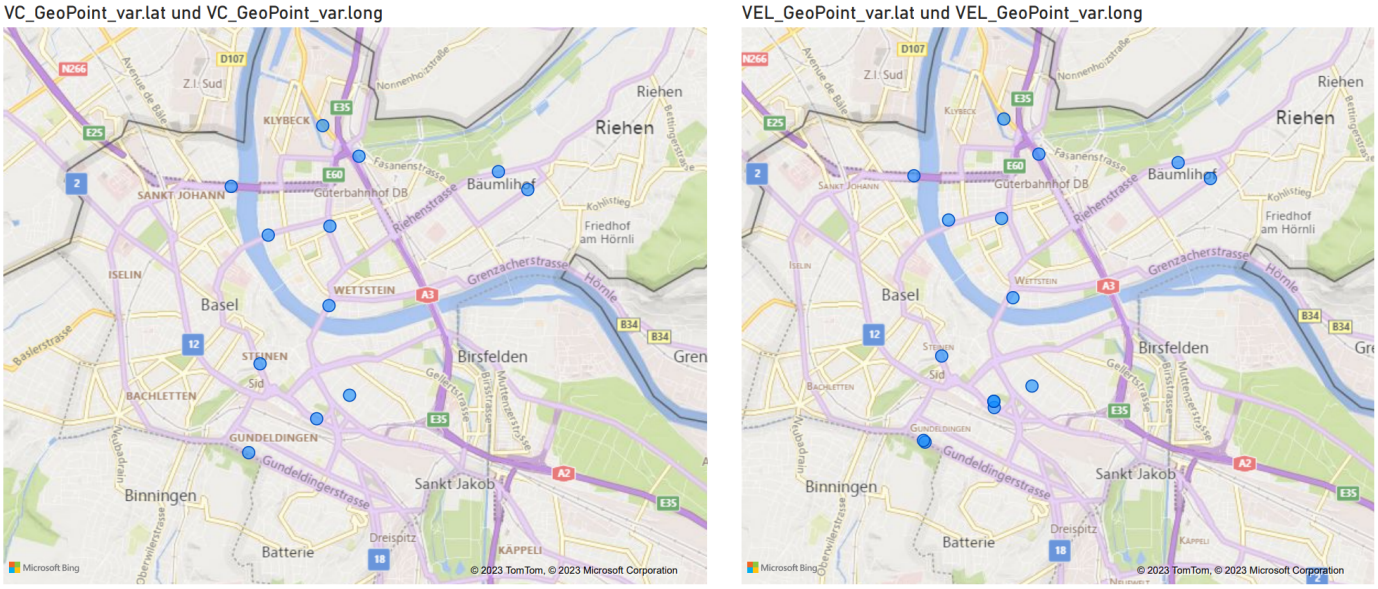

Fig. 4: close_points_100

different times but with a slightly different geopoint, this may also be due to GPS inaccuracies. So we updated one geopoint to the other, as they are functionally the same.

```
SET @target_point = (SELECT DISTINCT v.Point FROM vel_date v
WHERE v.Geopoint = '47.54284527862306, 7.584511581055648');
SET @target_geopoint = (SELECT DISTINCT v.Geopoint FROM vel_date v
WHERE v.Geopoint = '47.54284527862306, 7.584511581055648');
SELECT @target_point;
SELECT @target_geopoint;

UPDATE vel_date AS vel
SET
    vel.Point = (@target_point),
    vel.Geopoint = (@target_geopoint)
WHERE
    vel.Geopoint = '47.54301974645416, 7.584266731369311';
```

There were also several cases, where we found out after further inspection of the map, the measuring stations for speed and vehicle count were close geographically, but they were set up on a different street. So we had to ditch these results as well. Here you can see such an example:

```
DELETE FROM vc_date
```

```
WHERE vc_date.Geopoint = '47.5487720164889, 7.600745516688138';
DELETE FROM vel_date
WHERE vel_date.Geopoint = '47.54874481353451, 7.601200321293068';
```

We did this for several more measuring stations. The whole code can be found on Gitlab to reduce clutter. After all these steps, we finally had 7 pairs of stations remaining. When comparing the final set to the 'close_points_50' set, we found out that they were largely identical, except 1 station in the 8 results of the 50m dataset, where they were still on a different street. So we did some unnecessary work here, but also on a bright side, we found out some inconsistencies, which we might not have found otherwise.

Now we grouped the remaining data by weekday, to later take the average per weekday and hour. Again, we will only show the code for monday, as it can be done analogously for wednesday, friday, saturday and sunday. Note that the index for the weekday does begin with 1 on sunday, which is a bit counter-intuitive. This is for the vehicle-count dataset.

```
CREATE VIEW mondays_only AS
SELECT *
FROM vc_date vc
WHERE DAYOFWEEK(vc.Date) = 2;
```

Now we averaged over the Total amount of vehicles counted, and grouped it by Hour to find out the hours, where there was the most congestion.

```
CREATE VIEW res_monday_vc_avg AS
SELECT hour AS Hour, AVG(total) AS AverageTotalPerHour
FROM mondays_only
GROUP BY Hour
ORDER BY AverageTotalPerHour DESC;
```

We then did the same for the speed dataset. Again we grouped per hour, which had the highest average speed. And again see gitlab for the full code.

```
CREATE VIEW mondays_only_speed AS
SELECT *
FROM vel_date vel
WHERE DAYOFWEEK(vel.Date) = 2;

CREATE VIEW res_monday_vel_avg AS
SELECT hour AS Hour, AVG(Speed) AS AverageSpeedPerHour
FROM mondays_only_speed
GROUP BY Hour
ORDER BY AverageSpeedPerHour DESC;
```

Now, since we only had averages over all measuring stations, we also wanted to take a look at a single measuring station, to observe, if that makes a difference. For this we wanted to find the measuring station, which had the most measurements, to get as much data as possible.

```
SELECT COUNT(DISTINCT mo.Date)
FROM mondays_only mo
GROUP BY mo.Geopoint
ORDER BY mo.Geopoint;

SELECT COUNT(DISTINCT mo.Date)
FROM mondays_only_speed mo
GROUP BY mo.Geopoint
ORDER BY mo.Geopoint;

SELECT DISTINCT mo.Geopoint
FROM mondays_only_speed mo
ORDER BY mo.Geopoint;
```

Doing this for all our 5 analyzed weekdays revealed, that the speed measuring station with Geopoint '47.54284527862306, 7.584511581055648', located at 'Gundelingerstrasse' had the most different speed measurings of all points, across all weekdays. In general the speed measurments were much fewer than the vehicle count measurings, which made them the limiting factor. We then took a look at the data at this station, similar to before:

```
CREATE VIEW res_monday_vc_spec AS
SELECT hour AS Hour, AVG(total) AS AverageTotalPerHour
FROM mondays_only
WHERE mondays_only.Geopoint = '47.54273560620849, 7.584995701861084'
GROUP BY Hour
ORDER BY AverageTotalPerHour DESC;

CREATE VIEW res_monday_vel_spec AS
SELECT hour AS Hour, AVG(Speed) AS AverageSpeedPerHour
FROM mondays_only_speed
WHERE mondays_only_speed.Geopoint = '47.54284527862306, 7.584511581055648'
GROUP BY Hour
ORDER BY AverageSpeedPerHour DESC;
```

And as always, the procedure was very similar for all the other weekdays and the full code can be found on Gitlab. This marks the end of our SQL queries, and we subsequently went to the visualization part of the project.

11 Visualization

For our data visualization we used PowerBI for the map in the analysis part. But for the graphs, we decided to use Python in DataSpell, as we personally found Python more intuitive and we were more efficient with our time. We used sqlalchemy and ipython-sql packages to connect to the database and import all the data. When imported, we used pandas dataframes to handle the data

and save it as small csv files, as this made our work with the result tables much more faster. Finally for making the plots, we used seaborn and matplotlib.

"Seaborn is a Python data visualization library based on matplotlib" We also used folium to create some more maps. The steps for connecting to the database are explained inside the jupyter notebook. Finally we can take a look at the

graphs and our results. For better analysation of the correlation between speed and vehicle counts, we also divided the average total per hour by 8, because else the scaling was not very easy to see, since the average speed was mostly around 45 km/h and the vehicle counts were sometimes in the 400+ range. (Vehicles per hour). In our first plot, we took a look at the views that we created at the end of our SQL Analysis part, and in particular the average data. See figure 5.

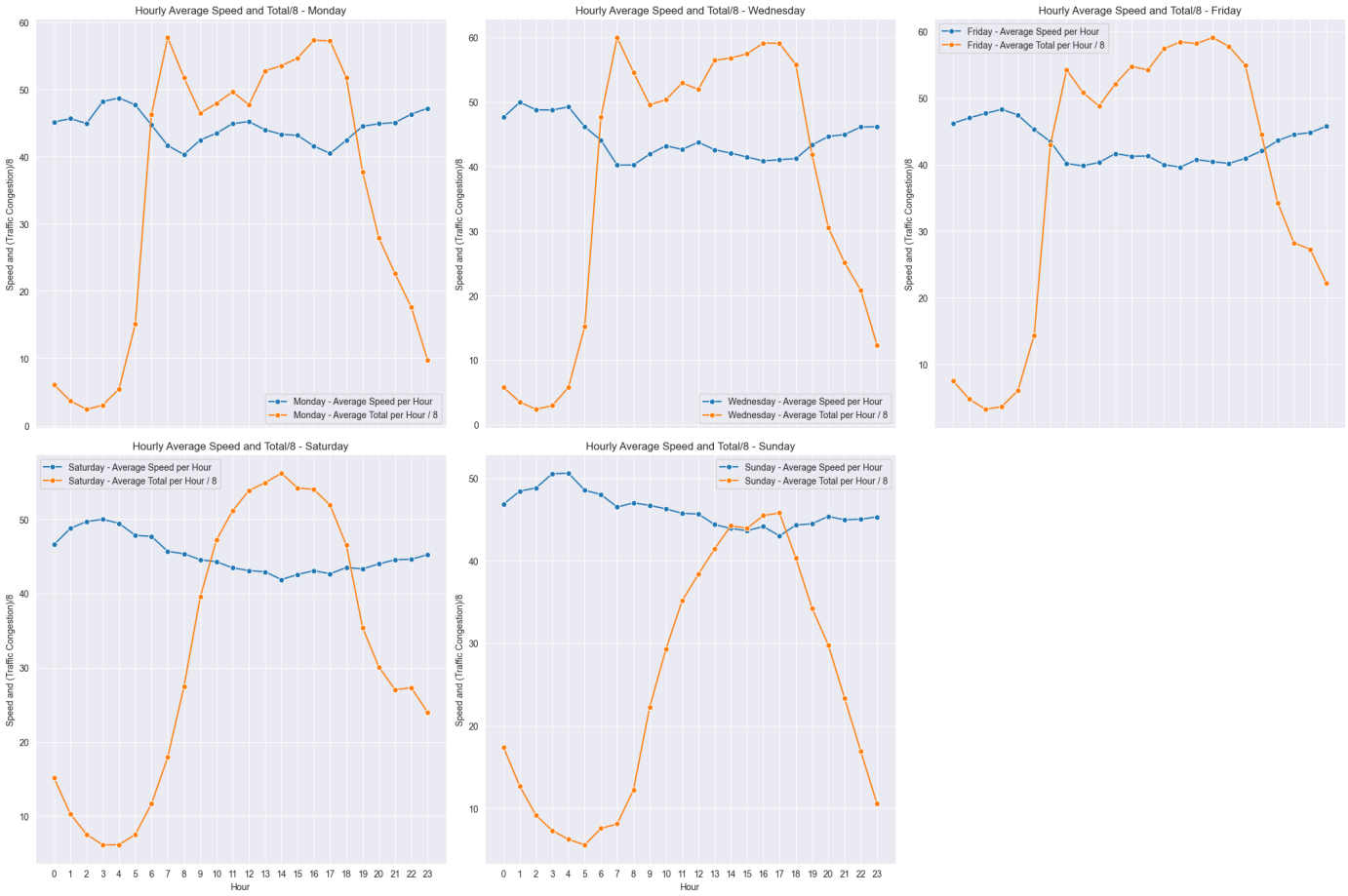

Fig. 5: average across measuring stations and weekdays

Now, we also made the same graphics for a specific measuring station, namely the one at *Gundelingerstrasse*, as explained in the analysis part. See figure 6. Note that here, the scaling was not /8, but /4, as the average traffic count numbers from this station were lower, as it is a one-way street. We will also upload the images in full resolution to Gitlab.

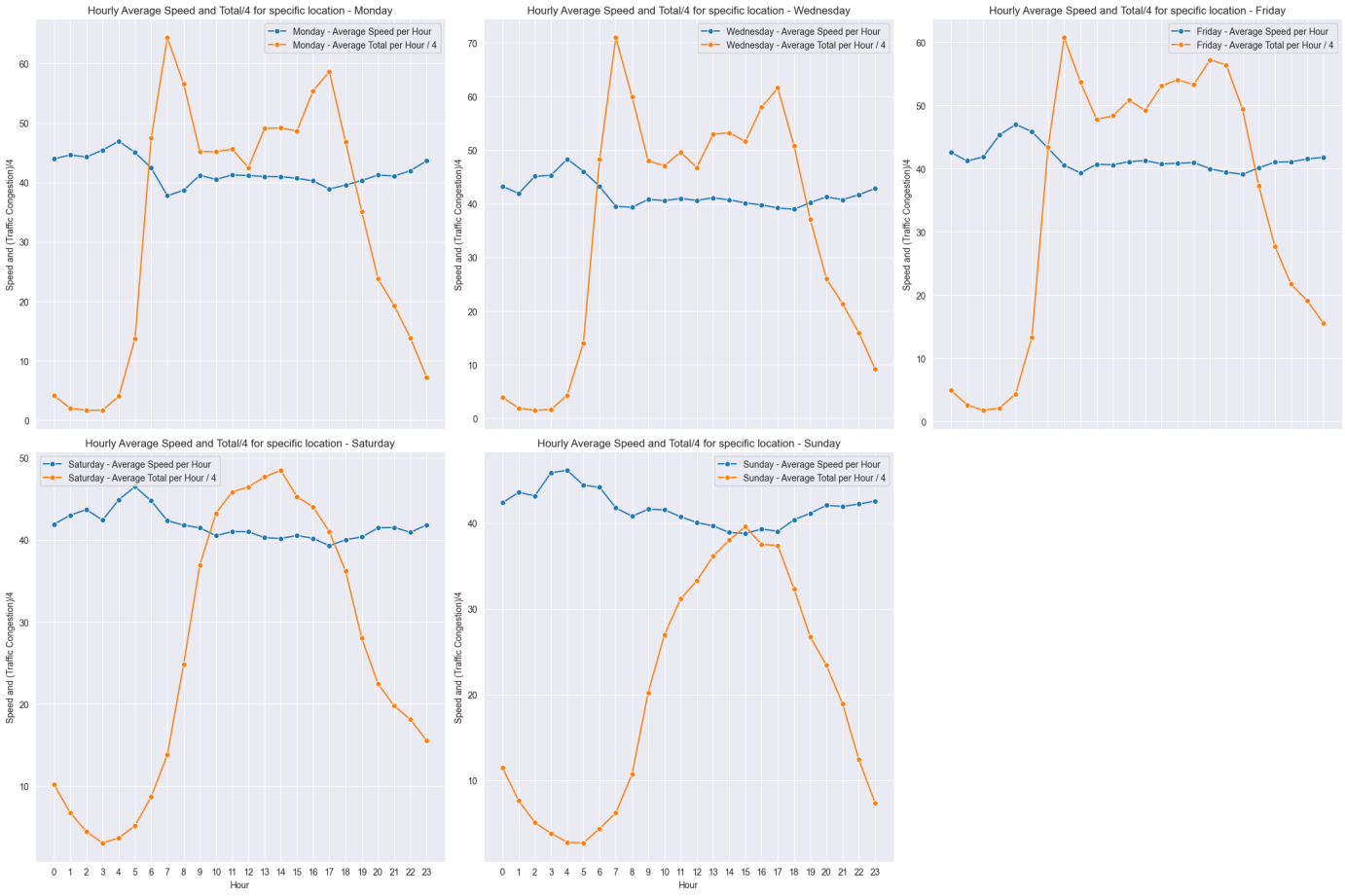

Fig. 6: measuring station at 'Gundelingerstrasse'

So, what information can we gain from these graphs? In the introduction part, we mentioned that our hope was to find a connection between the number of vehicles and the average speeds. If we take a closer look at the graphs, we can observe that the peaks of the orange curve (average total number of cars per hour) coincide with the lowest speeds pretty well. (blue curves; average speed per hour). This is true for both the average across all stations and also the one

at Gundelingerstrasse. Also, the highest average speeds correlate even better to the lowest number of vehicles (dips in orange curve, peaks of blue curve). So we can summarize for this part, that a lower number of vehicles lead to an increase in the average speed of the vehicles. This also corresponds to the intuition most people would have had, before taking a look at the data.

Now that we answered our main question, we also researched some other questions we found interesting. Firstly we plotted the average total vehicle count per hour of the weekdays we took a look at, over each other, to compare the traffic congestion depending on the day and find a pattern. See Figure 7.
As we can nicely see here, the peak congestions among weekdays (Monday, Wednesday, Friday) are in the morning at 7:00 and in the evening at 16:00 and 17:00. So the typical working schedule for most people in switzerland. However on the weekend people tend to sleep longer in the morning and thus the peaks only occur later in the day. At 14:00 on Saturday and a little longer on Sunday at 16:00 and 17:00. This also correspond with the stereotypical schedule, that a lot of people might go shopping on saturday and the peak at Sunday could be people returning home from visiting, etc. And also on the weekend the traffic in the night is higher than during the week. And in the late evening at 22:00 we see Friday and Saturday with the highest congestion, which is also as expected, as people can sleep longer the next day and tend to stay up longer.

Another question, which interested us, was where the worst speeding offences took place. For this we plotted the 5 worst speeding offences on a map using the *folium* library. See figure 8. (Map shows only 4 markers, because 2 of them took place at the same location).

To also further illustrate the measured speed and the violation compared to the zone, we plotted these four locations along with the zone and the measured top speed in figure 9.

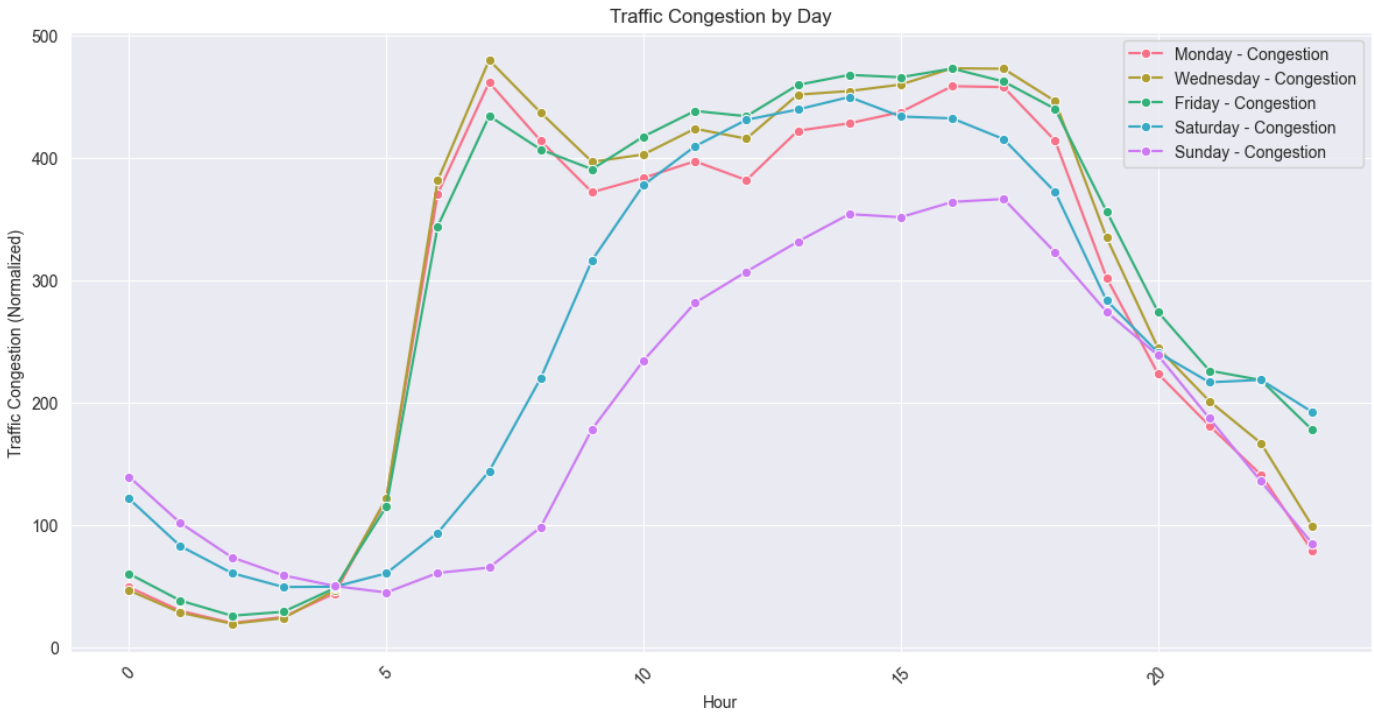

Fig. 7: traffic congestion compared on weekdays

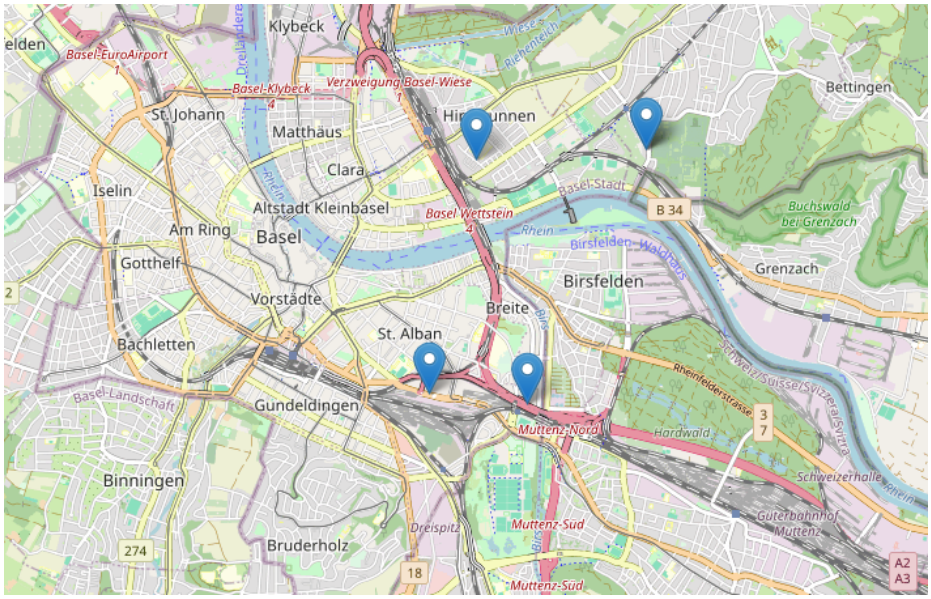

Fig. 8: traffic congestion compared on weekdays

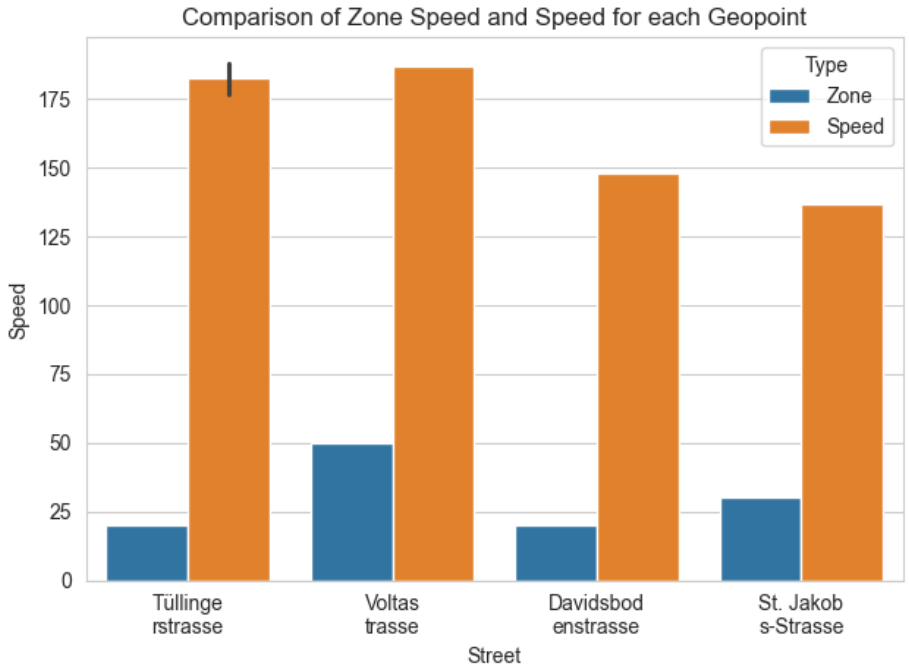

Fig. 9: traffic congestion compared on weekdays

12 Conclusion, Reflection and Limitations

In conclusion, our results showed that average speed stands in an inverse relation with the vehicle count. So, if there is more traffic, the average vehicle speed decreases and vice versa. This result also corresponds with the intuition most people would have had, if we posed the question to them beforehand, but we were happy to back our hypothesis up with data.

Overall we were quite satisfied with the results, we had, however we are aware of the limitations in our analysis, such as taking the average over all measuring points and also having a limited set of close measuring stations. Our analysis could benefit from more data, especially measuring stations which measure both the speed and also the vehicle count at the same time, which we found quite rare in the dataset.