Chapter 11
Urban Mobility Use Case

Urban mobility is concerned with the movement of people and goods in a city. Public transportation is a large and important part of it. Analyzing delays in public transportation significantly impacts urban mobility by enhancing efficiency, reliability, and passenger satisfaction. This analysis enables public transport authorities to optimize routes, schedules, and infrastructure investments, ultimately reducing congestion and improving travel times. Open data initiatives, together with the usage of data standards in the public transport domain, have enabled many cities to publish enough data for performing this analysis. This chapter studies the processing and analysis of public transport data.

In Sect. 11.1 we explain the various types of mobility data that can be available in a city. This includes public transport schedules, real-time vehicle positions, micromobility vehicle availability, and parking availability. We also present the standardized formats of these data. An important family of file formats is the Google Transit Feed Specification (GTFS), which consists of two parts, GTFS Schedule and GTFS Realtime, that concern, respectively, the scheduled and the real-time positions of public transport vehicles. Section 11.2 describe the data loading and analysis of GTFS Schedule using data from the city of Riga, Latvia. Section 11.3 shows the same methodology using data from the city of Genova, Italy. Section 11.4 explains how to load and query actual trip data from GTFS Realtime feeds. Section 11.5 explains how schedule and real-time data together can be used to analyze the actual speeds and delays of vehicles. This is a challenging task, since the two formats have significant differences. GTFS Schedule is a cyclic format where the schedule is defined as a set of rules for generating trips. In contrast, the GTFS Realtime is a flat format representing the actual positions of the vehicles. It is thus necessary to do a series of conversions to bring the two sources to a form that can be joined. The section illustrates the two formats and the joining pipeline in detail.

11.1 Urban Mobility Data and Applications

Urban mobility has been constantly evolving in the last years. New services and technologies are emerging, affecting the way people and goods move through cities. The main trends in this transformation involve replacing traditional fuel-powered vehicles with electric ones, shifting from privately-owned cars to shared mobility solutions, and promoting **multimodal mobility** that combine various modes of transport, such as buses, bicycles, and walking. A key driver behind these changes is the increasing use of data in urban transportation systems. As vehicles become equipped with sensors and connectivity features, vast amounts of data are being generated to track how people and goods move in cities. These data are crucial for making transportation more efficient, sustainable, and accessible. As cities strive to become more eco-friendly and resilient, the transportation sector is becoming more data-dependent and technologically advanced.

Public and private transportation providers are more willing than ever to share their data to help improving mobility services and the provision of real-time information to users. For example, public transit agencies share *static schedules* and *real-time vehicle positions* using standardized data formats such as the **Google Transit Feed Specification** (GTFS), which is composed of GTFS Schedule[1] for schedules and GTFS Realtime[2] for live data. This enables third-party applications, such as navigation and transport apps, to recommend optimal routes for users based on the most up-to-date information, including traffic conditions and service disruptions.

In addition to public transport, the **micromobility** sector has become an essential part of urban transportation. Micromobility is defined by the International Transport Forum as the use of vehicles not heavier than 350 kg and with a design speed not higher than 45 km/h which is broad enough to include human- and electric-powered vehicles. Services offering shared bikes and scooters use data formats like **General Bikeshare Feed Specification** (GBFS)[3] to provide real-time information about the availability and battery levels of shared vehicles. These data-driven applications are key to integrating micromobility services with public transport systems, enhancing the efficiency and convenience of multimodal travel across cities.

Another significant trend is the increasing availability of **open data** related to various aspects of urban mobility. Cities and transportation providers are making data on traffic flows, vehicle speeds, air quality, railway schedules, delays, and parking availability accessible to the public. This wealth of open data allows urban planners, researchers, and developers to study and optimize city-wide mobility systems. By analyzing traffic patterns and public

[1] https://developers.google.com/transit/gtfs

[2] https://developers.google.com/transit/gtfs-realtime

[3] https://github.com/MobilityData/gbfs

transport speeds, for example, it is possible to identify congestion hotspots and suggest improved routing strategies.

In response to the growing complexity of urban mobility, data-driven digital twins have emerged as powerful platforms for monitoring and predicting transportation dynamics. A **digital twin** is a virtual model of a physical object that spans the object's lifecycle and uses real-time data sent from sensors on the object to simulate the behavior and monitor operations. In the case of public transport, they combine data from various sources to create a comprehensive, real-time model of city transport systems. By integrating public transport, micromobility, and road traffic data, digital twins offer a cross-modal view of urban mobility, enabling authorities and businesses to make better informed decisions.

As urban mobility data continues to grow in volume and diversity, the potential for innovative applications is immense. Real-time route recommendations, traffic management tools, and sustainable urban planning solutions are just a few examples of how data are reshaping the future of urban transportation. This chapter studies urban mobility data, their sources, applications, and the impact it is having on the way we move within cities.

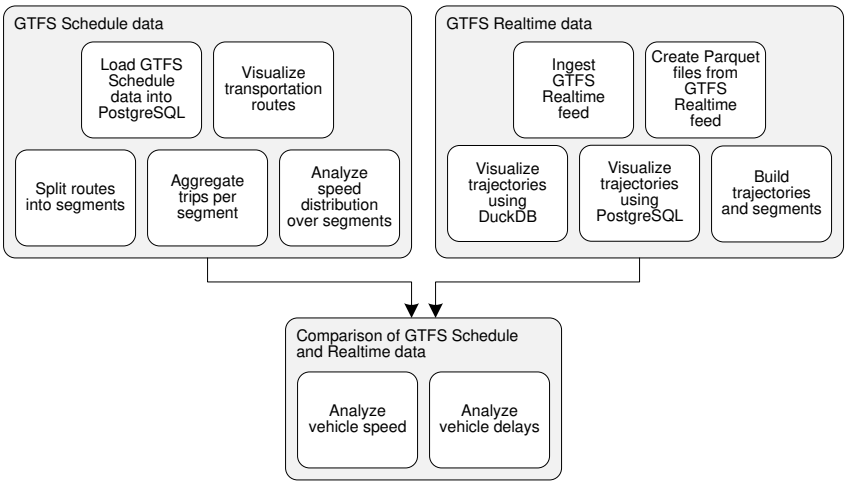


Fig. 11.1 Data pipeline for the urban mobility use case.

Figure 11.1 describes the use case that we study in this chapter. The figure shows three main analytics blocks. The first one, concerned with schedule data, depicts the loading and exploration of the GTFS Schedule data files and the speed analysis over segments of the public transport network. In this analysis, a segment is the part of transportation network between two consecutive stops in a route. The second block repeats a similar analysis, yet performed on the actual trip data coming from a GTFS Realtime endpoint.

Finally, in the third block we cross the two sources and analyze the speed and the delays of actual trips with respect to their schedule. The following sections will describe the tasks carried out in each of these three blocks.

Throughout this chapter we will use data from two research projects funded by the European Union, namely, EMERALDS[4] and MobiSpaces.[5] MobiSpaces envisions a set of toolboxes, suites, and tools that implement the concept of Mobility Data Spaces. EMERALDS aims at developing a Mobility Analytics as a Service (MaaS) toolset in the edge-fog-cloud continuum. In this book, we will mainly use the open data of *Rīgas satiksme*, the public transport company in Riga, Latvia,[6] which is a partner of the EMERALDS project. The use case serves as a logical structure to explore various tools and techniques that can enhance the understanding and processing of public transport data. To show the generality of our approach, we also show how it can be applied to analyze public transport data from the city of Genova in Italy,[7] which is a partner of the MobiSpaces project.

11.2 GTFS Schedule Data

The General Transit Feed Specification (GTFS Schedule) is a widely used data format for representing scheduled public transportation services. It is designed to share information about transit schedules, routes, stops, and other related data in a standardized format. This enables developers, service providers, and municipalities to integrate this information into various applications, such as trip planners, navigation systems, and research tools.

GTFS Schedule consists of a set of CSV files, representing various aspects of a transit system. These files are typically zipped together into a single archive and shared as part of a transit feed. We briefly describe next the content of the files in the GTFS Schedule specification:

- agency.txt: Contains information about the transit agencies operating the services. It includes agency name, URL, phone number, and language of communication.
- stops.txt: Contains stops and stations where vehicles pick up or drop off passengers. It has attributes such as stop identifier, stop name, geographic coordinates (latitude and longitude), and optional data, like wheelchair accessibility.

[4] https://emeralds-horizon.eu/

[5] https://mobispaces.eu/

[6] https://www.rigassatiksme.lv/en/about-us/publishable-information/open-data/

[7] https://www.amt.genova.it/amt/societa_trasparente/altri-contenuti/open-data/

- routes.txt: Describes the routes offered by the transit agency. It includes route identifier, route name, route description, and route type (e.g., bus, tram, subway).
- trips.txt: Describes the individual trips along the routes, representing one instance of a vehicle traveling along a route. Each trip is associated with a route identifier and includes a trip identifier, service days, and, optionally, a block identifier and shape identifier. The former identifies the block to which the trip belongs, where a block consists of a single or several sequential trips made using the same vehicle. The shape identifier identifies a geospatial shape describing the vehicle travel path for a trip.
- stop_times.txt: Provides the times when a vehicle is scheduled to arrive at and depart from each stop during a trip. It includes trip identifier, stop identifier, arrival time, departure time, and stop sequence, the latter indicating the order in which stops are visited.
- calendar.txt: Specifies the regular service dates for trips, indicating which days of the week each service runs. It includes service identifier, start and end dates, and a binary flag for each day of the week (e.g., 1 for active on Monday, 0 for no service).
- calendar_dates.txt: Provides exceptions to the regular service schedule, such as holidays or special events. It contains service identifier, date, and an exception type (e.g., service added or removed).
- shapes.txt (optional): Describes the path a vehicle takes along a route, which is useful for plotting the routes on a map. It includes shape identifier along with a sequence of latitude and longitude coordinates that define the shape of the route.
- frequencies.txt (optional): Specifies intervals at which trips operate, typically for services that do not have fixed times (e.g., "buses arrive every 10 minutes"). It defines the start and end times of the frequency-based service, along with the headway (the time between successive vehicles).
- transfers.txt (optional): Specifies rules for connecting routes at stops. It includes stop identifiers for transfer points and the type of transfer allowed (e.g., immediate transfer or a required wait).

The individual tables in GTFS Schedule encode rules that can be used to generate the scheduled trips. To generate these trips with their complete trajectories and timestamps, the tables need to be joined together and the exceptions get applied. Some tools and libraries already exist for performing this task. We will use the following two:

- gtfs-via-postgres:[8] An open-source tool built to simplify the process of importing, storing, and analyzing GTFS data into a PostgreSQL/PostGIS database. It is based on Node.js and can be installed via npm, the Node package manager.

[8] https://www.npmjs.com/package/gtfs-via-postgres/v/3.0.2

- gtfs_functions:[9] A Python package that provides functions for loading and manipulating GTFS feeds in Python. It can be installed via pip, the Python package manager.

11.2.1 Load GTFS Schedule Data into PostgreSQL

GTFS Schedule data come as a zip file, which must be downloaded to a local folder before it can be processed. In this example, we are using the GTFS Schedule data for Riga, which can be accessed through their open data portal.[10] The specific file we are working with is marsrutusaraksti08_2024.zip, which contains the itineraries scheduled for September 2024.

Our first task consists in loading the zip file into a PostGIS database. First, we create a database named UrbanMobility and the PostGIS and MobilityDB extensions for it. Then, we load the data using gtfs-via-postgres. If the application is not already installed, we run the following command:

```
npm install -g gtfs-via-postgres
```

We then unzip the downloaded file, for instance into a folder MarsrutuSaraksti08_2024. After this, the folder will contain the files agency.txt, calendar.txt, shapes.txt, stop_times.txt, calendar_dates.txt, routes.txt, stops.txt, and trips.txt. Within this folder we execute the following commands for loading the GTFS Schedule to the PostgreSQL database:

```
export PGUSER=postgresUser
export PGPASSWORD=password
export PGDATABASE=UrbanMobility
# ... export other connection parameters as needed ...
npm exec -- gtfs-to-sql --require-dependencies -- *.txt | psql -b
```

The resulting database should contain the tables and views depicted in Fig. 11.2. The tables simply reflect the content of the files, except for the frequencies table which is empty in this case. In addition, the command creates some useful views. We will use the following ones in our analysis:

- arrivals_departures applies stop_times to trips and service_days to provide all arrivals/departures at each stop, along with their absolute dates and times;
- connections applies stop_times to trips and service_days, just like arrivals_departures, but provide the departure (at stop A) and arrival (at stop B) pairs;
- shapes_aggregated aggregates individual shape points in shapes into a PostGIS line string.

[9] https://pypi.org/project/gtfs-functions/

[10] https://data.gov.lv/dati/lv/dataset/marsrutu-saraksti-rigas-satiksme-sabiedriskajam-transportam

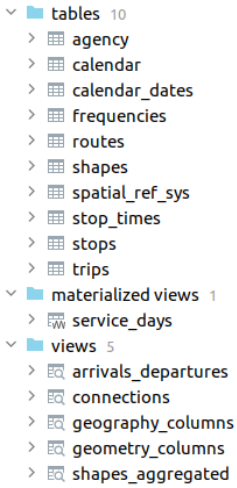


Fig. 11.2 Tables and views created by gtfs-via-postgres in a PostgreSQL database.

11.2.2 Visualize Transportation Routes

We first want to have an overview of the transit network. For this, we use the shapes_aggregated view. In what follows, in general we omit the script commands that set the database connections or import the necessary libraries. The reader can find the complete script in the companion GitHub site of this book. The following script creates an interactive map to visualize *individual* transit lines in the city. We use Folium to display the map and GeoPandas to handle geospatial data. The script for this is shown next.

```
# Library imports
import folium as fl
import geopandas as gpd

# Function that obtains the geometries of the transit lines from a PostgreSQL cursor
def map_individual_lines(cur):
    # SQL query to get the transit line shapes
    sql = 'SELECT * FROM shapes_aggregated;'
    shapes_gdf = gpd.GeoDataFrame.from_postgis(sql, cur.connection,
      geom_col='shape', crs='EPSG:4326')

    # Initialize the map centered on Riga
    map_indiv_lines = fl.Map(location=[56.937, 24.109], tiles='CartoDB positron',
      zoom_start=13, control_scale=True)

    # Add each route as a FeatureGroup
    for shape_id, shape_group in shapes_gdf.groupby('shape_id'):
      # Initially show all routes by default
      feature_group = fl.FeatureGroup(name=f"Route {shape_id}", show=True)
```

```
        # Add each LineString geometry as a PolyLine
        for geometry in shape_group.geometry:
            if geometry.geom_type == 'LineString':
                # Reverse coordinates to (latitude, longitude)
                coords = [(lat, lon) for lon, lat in geometry.coords]
                fl.PolyLine(locations=coords, color="blue", weight=2, opacity=0.8).
                    add_to(feature_group)

        # Add the feature group to the map
        feature_group.add_to(map_indiv_lines)

    # Add a layer control to toggle between various routes
    fl.LayerControl(collapsed=False).add_to(map_indiv_lines)

    return map_indiv_lines

individual_lines_map = map_individual_lines(cur)
display(individual_lines_map)
```

In the script above, each transit line, identified by its shape_id, is added to the map as a FeatureGroup. All lines are initially visible by default, which is achieved setting show=True. In the visualization, the user can toggle the visibility of individual lines through the LayerControl panel on the right, to visualize certain routes. The map_individual_lines function requires as a parameter a psycopg cursor (the variable cur in this case) connected to the PostGIS database. A snapshot of this visualization is shown in Fig. 11.3.

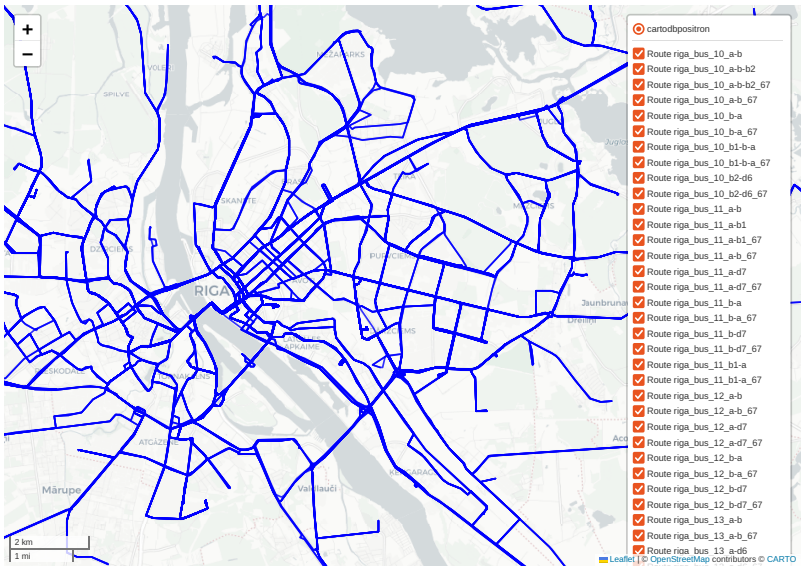


Fig. 11.3 Interactive visualization of the public transport routes in Riga, Latvia.

11.2.3 Split Routes into Segments

We now perform segment-based analysis over the schedule data. This analysis provides insights into the use of specific sections of the network and helps to identify bottlenecks, allowing transit authorities to target infrastructure improvements and adjust schedules for better efficiency. It also aids in evaluating route performance by highlighting congested or underutilized segments, optimizing vehicle allocation, and ensuring punctuality.

A *segment* represents the path between two consecutive stops. Splitting the network routes into segments requires joining most of the GTFS tables to identify consecutive stops, and then performing splitting the linestring objects into segments accordingly. This procedure is automated by the gtfs_functions library, as illustrated in the following script:

```
# Library imports
import gtfs_functions as gtfs
from sqlalchemy import create_engine

# Function that constructs the segments from the GTFS files
def load_gtfs_feed(file_path, start_date, end_date):
  # Load GTFS feed
  feed = gtfs.Feed(file_path, start_date=start_date, end_date=end_date)
  segments_gdf = feed.segments
  return segments_gdf

# Function that saves the segments into a PostGIS table
def save_to_postgis(gdf, table_name, db_host, db_port, db_user, db_pass, db_name):
  # Construct the connection URL using the provided parameters
  db_url = f"postgresql://{db_user}:{db_pass}@{db_host}:{db_port}/{db_name}"
  # Create the SQLAlchemy engine using the constructed URL
  engine = create_engine(db_url)
  # Save the GeoDataFrame to PostGIS
  gdf.to_postgis(table_name, engine, if_exists='replace')
  print(f"Data saved to {table_name} table in the {db_name} database.")

# Execute the functions
gtfs_file = "marsrutusaraksti08_2024.zip"
gtfs_segments = load_gtfs_feed(gtfs_file, '2024-09-01', '2024-09-30')
save_to_postgis(gtfs_segments, "segments", db_host, db_port, db_user, db_pass,
  db_name)
```

This script uses the gtfs_functions library to load a GTFS feed and save the processed data into a PostGIS-enabled PostgreSQL database. The function load_gtfs_feed receives as arguments a file path to the GTFS zip file, a start date and an end date. Then, it loads the GTFS data using the Feed function, returning the transit segments as a GeoDataFrame. The function save_to_postgis then takes the GeoDataFrame and saves it into a database. It constructs the database connection URL using the provided database parameters (host, port, user, password, and database name), creates a SQLAlchemy engine, and uses the to_postgis method to store the GeoDataFrame in the

specified database table. In the final three lines of the script, it defines the path to the file marsrutusaraksti08_2024.zip in the variable gtfs_file, and then calls the load_gtfs_feed function with this path and two dates, September 1st and September 30, 2024. The final line saves the data to a table named segments in the database. The segments table resulting of this script contains 6,932 segments. Figure 11.4 shows the route of Bus 43 Abrenes iela - Skulte and a segment of this route between two stops.

Fig. 11.4 The route (in red) of Bus 43 Abrenes iela - Skulte and a segment of this route (in blue) between stops (in green) Pagrieziens uz Piņķiem and Brīvkalni.

11.2.4 Aggregate Trips per Segment

In this section, we calculate statistics at segment granularity. We focus on the density of trips per segment using the following query.

```
SELECT c.from_stop_id, c.from_stop_name, c.to_stop_id, c.to_stop_name, s.geometry,
  COUNT(trip_id) AS notrips, string_agg(DISTINCT c.route_short_name, ',')
  AS routes
FROM segments s, connections c
WHERE s.route_id = c.route_id AND s.direction_id = c.direction_id AND
  s.start_stop_id = c.from_stop_id AND s.end_stop_id = c.to_stop_id AND
  date BETWEEN '2024-09-01' AND '2024-09-30'
GROUP BY c.from_stop_id, c.from_stop_name, c.to_stop_id, c.to_stop_name,
  s.geometry;
```

The query performs a join between the segments table created in Sect. 11.2.3 and the connections view, which contains the segments of all scheduled trips during the month of September, 2024. The query computes the total number of scheduled trips that traverse each segment using the COUNT aggregate function. Figure 11.5 shows the result. We can see in column notrips the total number of trips that have traversed each segment during the period being considered and in column routes the list of routes that traverse each segment.

The query above is embedded in a function load_segments_from_postgis, which receives a database connection string conn and stores the result into a GeoDataFrame called segment_gdf. The following Python script shows the

from_stop_id text	from_stop_name text	to_stop_id text	to_stop_name text	geometry geometry	notrips bigint	routes text
0001	Latgales iela 450	0396	Rumbula	0102000020E610…	2223	15,18,31
0002	Latgales iela 450	0019	Apskates stacija	0102000020E610…	2262	15,18,31
0003	A.Čaka iela	0469	Tallinas iela	0102000020E610…	9459	18,23,35,47
0004	Dzeņu iela	0394	Rudens iela	0102000020E610…	1518	20,31,33
0004	Dzeņu iela	5455	Brāļu Kaudzīšu …	0102000020E610…	915	6
0005	Dzeņu iela	0488	Ulbrokas iela	0102000020E610…	4479	20,31,33,6
0006	Alfa	0221	Krustabaznīcas…	0102000020E610…	16860	12,16,21,…
0007	Alfa	0436	Sporta akadēm…	0102000020E610…	6651	31,34,4

Fig. 11.5 Number of trips per segment planned for September 2024.

results of this query using an interactive Folium map, which helps us to visualize the occupancy of the various segments of the network.

```
# Library imports
import branca.colormap as cm

# Function that visualizes the segments in a Folium map
def visualize_segments(segment_data, cutoff=10000):
  # Initialize a Folium map centered on Riga
  transport_map = fl.Map(location=[56.9496, 24.1052], tiles='CartoDB positron',
    zoom_start=12)

  # Define the colormap from white (low count) to red (high count)
  colormap = cm.LinearColormap(colors=['white', 'red'],
    vmin=0, vmax=cutoff, caption='Trip Count')
  colormap.add_to(transport_map)

  # Loop over each segment and add it to the map
  for _, segment in segment_data.iterrows():
    trip_count = min(segment['notrips'], cutoff) # Apply cutoff
    color = colormap(trip_count) # Set the color gradient

    # Compose a GeoJSON for the visualization
    geo_json = fl.GeoJson(
      data={
        "type": "Feature",
        "geometry": segment['geometry'].__geo_interface__,
        "properties": {
          "routes": segment['routes'],
          "from_stop_name": segment['from_stop_name'],
          "to_stop_name": segment['to_stop_name'],
          "notrips": segment['notrips'] }},
      style_function=lambda x, color=color: {
        'color': color, 'weight': 3, 'opacity': 0.7},
      tooltip=GeoJsonTooltip(
        fields=['routes', 'from_stop_name', 'to_stop_name', 'notrips'],
        aliases=['Routes', 'From Stop', 'To Stop', 'Trip Count'],
        localize=True ))
    geo_json.add_to(transport_map)
```

```
    return transport_map

# Display the map in the notebook
transport_map = visualize_segments(segment_gdf)
display(transport_map)
```

The function visualize_segments receives a GeoDataFrame, which contains the results of the previous SQL query, and a cutoff value to control the range of the color gradient, such that any trip count above the cutoff value is assigned the maximum intensity. The script begins by initializing a Folium map centered in Riga, using CartoDB Positron (light gray map) tiles. We then define a colormap, ranging from white for segments with lower trip counts to red for those with higher counts. This colormap is added to the map. For each segment, the trip count is computed and a color is applied according to the range defined in the colormap. The segment's geometry is converted into a GeoJSON format and added to the map with a custom style, which controls the color, line thickness, and opacity. Additionally, interactive tooltips are created to display relevant information, such as route names, stop names, and trip counts, when users hover over the segments. Finally, the map is rendered with all segments visualized according to their trip counts.

Figure 11.6 shows the result of the script above. In this map, segments with a higher number of trips are represented with a darker red color. Understanding the meaning of this visualization requires some knowledge of the city's transport system, but it offers a quick way to compare routes with high vehicle frequencies to those that are less frequently served. For example, by examining the tooltips on segments with a high trip count, it becomes clear that lines 1 and 15 are responsible for a significant portion of the trips. On another observation, the map highlights a distinct division created by the Daugava river, such that the eastern part of the city is served by more trips compared to the western part.

11.2.5 Analyze Speed Distribution over Segments

Generally speaking, in public transit companies there are three main departments that manage the flow of transport. The first is the *network department*, which is responsible for long-term planning, such as adding new routes or modifying existing ones. The second is the *planning department*, which designs the schedules taking into account several constraints like the availability of vehicles, driver shifts, passenger demand, and other operational factors. Finally, the *operations department* monitors the network in real-time and responds to any emerging issues or disruptions. This section focuses on how the planning department estimates the speed in the segments and incorporates this information into the schedules.

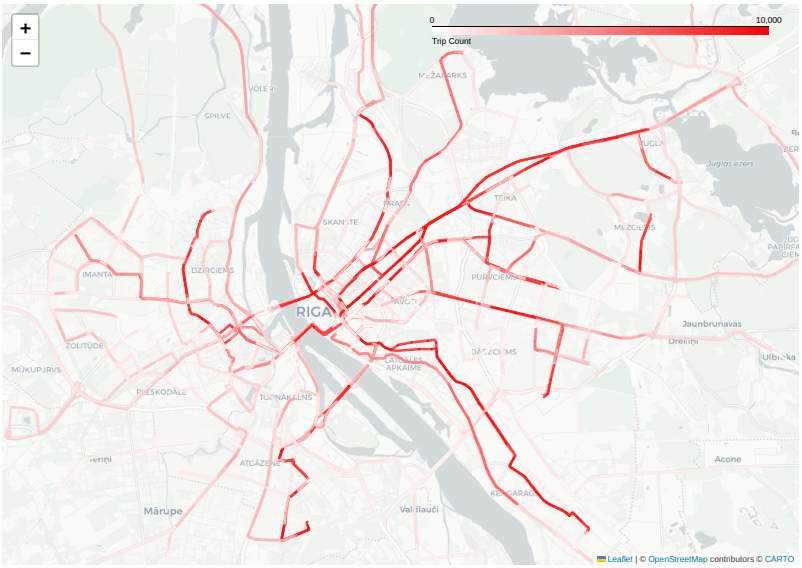


Fig. 11.6 Interactive visualization of the number of trips per segment in Riga showing the routes that cover a segment, the initial and end stops, and the total number of trips in the segment.

Based on historical data, the planning department estimates the average speed for each segment of the network. This speed may vary depending on the time of day, day of the week, or the season. The speed estimates are essential for predicting the arrival times of vehicles at each stop, and this information is encoded in the GTFS Schedule. In the analysis we present next, we aim at extracting the average speed data per segment, according to the planned schedule. The next query calculates the average speed (in kilometers per hour) per segment.

```
SELECT AVG(s.distance_m /
  EXTRACT(EPOCH FROM (c.t_arrival - c.t_departure)) * 3.6) AS avg_speed_kmh,
  c.from_stop_id, c.from_stop_name, c.to_stop_id, c.to_stop_name
FROM connections c, segments s
WHERE c.route_id = s.route_id AND c.direction_id = s.direction_id AND
  c.from_stop_id = s.start_stop_id AND c.to_stop_id = s.end_stop_id AND
  date BETWEEN '2024-09-01' AND '2024-09-30' AND
  EXTRACT(EPOCH FROM (c.t_arrival - c.t_departure)) > 0
GROUP BY c.from_stop_id, c.from_stop_name, c.to_stop_id, c.to_stop_name;
```

The calculation is based on the time difference between the vehicle's arrival at a stop and its departure from the previous stop, thus excluding the time where the bus stays at the stop. Again, the query uses the connections view, which contains information about the individual trip timings over segments, and the segments table, which contains details about the network segments and their lengths in meters. The speed is computed by dividing the segment

length (s.distance_m) by the travel time between two stops. This calculation is performed for each trip and segment in the schedule. We can visualize the estimated traffic speeds using a script similar to the one which includes the visualize_segments function shown in the previous section, and replacing the trip count aggregate with the average speed aggregate in the visualization.

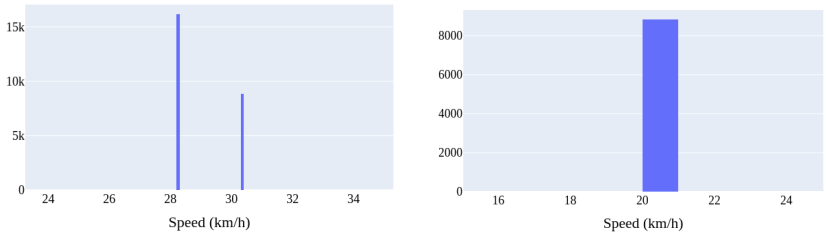
We can compute the speed for individual trips across a segment as follows.

```
SELECT (s.distance_m /
  EXTRACT(EPOCH FROM (c.t_arrival - c.t_departure)) * 3.6) AS speed_kmh,
  c.from_stop_id, c.from_stop_name, c.to_stop_id, c.to_stop_name
FROM connections c, segments s
WHERE c.route_id = s.route_id AND c.direction_id = s.direction_id AND
  c.from_stop_id = s.start_stop_id AND c.to_stop_id = s.end_stop_id AND
  date BETWEEN '2024-09-01' AND '2024-09-30' AND
  EXTRACT(EPOCH FROM (c.t_arrival - c.t_departure)) > 0;
```

The query above allows us to understand the distribution of the estimated speeds, which could benefit applications such as routing and travel-time estimation. The Python script below, builds an interactive web application using the Dash tool introduced in Chap. 9, to visualize the speed distribution over segments. The visualization allows us to select a specific segment from a dropdown menu and view a histogram that displays the distribution of vehicle speeds on that segment. The data comes from the SQL query and include information about the speed of vehicles across various segments between two stops. Figure 11.7 shows the result of this script for two selected segments. Histograms displaying only 1–2 bins indicate that the transport company uses a fixed average speed for a segment, without varying it based on factors such as time of day or day of the week.

```
# Library imports
import psycopg as pg
import dash
from dash import dcc, html
from dash.dependencies import Input, Output
import plotly.express as px
import dash_bootstrap_components as dbc

# Function that fetches the speed values per segment from PostgreSQL
def fetch_segment_speeds():
    conn = pg.connect(...)
    query = """
      # Here goes the speed computation SQL query shown above;
    """
    df = pd.read_sql_query(query, conn)
    conn.close()
```

```
    # Add a column that concatenates 'from_stop_name' and 'to_stop_name'
    df['segment'] = df['from_stop_name'] + " -> " + df['to_stop_name']

    return df

# Initialize Dash visualization
app = dash.Dash(__name__, external_stylesheets=[dbc.themes.BOOTSTRAP])

# Load data
speed_df = fetch_segment_speeds()

# Layout of the Dash visualization
app.layout = dbc.Container([
  # Dropdown for selecting segments
  dcc.Dropdown(
    id="segment-dropdown",
    options=[{"label": seg, "value": seg} for seg in sorted(speed_df['segment'].unique())],
    value=speed_df['segment'].unique()[0], # Default value
    clearable=False,
    style={"width": "80%"}
  ),

  # Graph to display speed distribution
  dcc.Graph(id="speed-distribution-plot") ])

# Callback function to update the graph based on selected segment
@app.callback(
  Output("speed-distribution-plot", "figure"),
  Input("segment-dropdown", "value"))
def update_graph(selected_segment):
  # Filter data for the selected segment
  segment_df = speed_df[speed_df['segment'] == selected_segment]

  # Create histogram for the speed distribution
  fig = px.histogram(segment_df, x="speed_kmh", nbins=30,
  title=f"Speed Distribution for {selected_segment}",
  labels={"speed_kmh": "Speed (km/h)"})

  # Ensure the bar does not stretch too wide by setting a reasonable x-axis range
  if not segment_df['speed_kmh'].empty:
    speed_min = segment_df['speed_kmh'].min() - 5 # Padding for better visibility
    speed_max = segment_df['speed_kmh'].max() + 5
    fig.update_xaxes(range=[speed_min, speed_max])

  fig.update_layout(xaxis_title="Speed (km/h)", yaxis_title="Frequency")
  return fig

# Run the visualization
if __name__ == "__main__":
  app.run_server(debug=True, port=8050) # If taken, set another port
```



(a) Alfa – Krustabaznīcas iela **(b)** Bebru iela – Jūrmalas gatve

Fig. 11.7 Estimated speed distribution across two segments in Riga.

11.3 Analysis of GTFS Schedule Data for Genova

We repeat the analysis above for the city of Genova, Italy, using the GTFS Schedule files that are available for downloading.[11] We will not give details since the idea here is to show that the procedure explained in the previous sections can be easily reproduced for other cities. The corresponding scripts are available on the companion GitHub repository. Figure 11.8 shows the interactive view of the transport routes in Genova. This figure is analogous to Fig. 11.3 for Riga.

Figure 11.9 shows the interactive visualization of the number of trips per segment in Genova, which is analogous to Fig. 11.6 for Riga. In this case, we can see that the darker tones of red concentrate along the sea and in two branches perpendicular to the sea shore, which reveals a very different situation compared to Riga.

We conclude the section showing, in Fig. 11.10, the speed distribution for two segments on the city of Genova. In this case, with respect to Fig. 11.7, we observe a more detailed distribution, indicating that the transport company maintains fine-grained statistics that account for various operational factors.

11.4 GTFS Realtime Data

GTFS Realtime is an extension of GTFS Schedule developed to provide real-time updates on public transit operations. While GTFS Schedule provides static data such as routes, stops, and scheduled trip times, GTFS Realtime provides live information about changes in service, vehicle positions, and trip statuses. These real-time data enable applications to obtain the current state

[11] https://www.amt.genova.it/amt/societa_trasparente/altri-contenuti/open-data

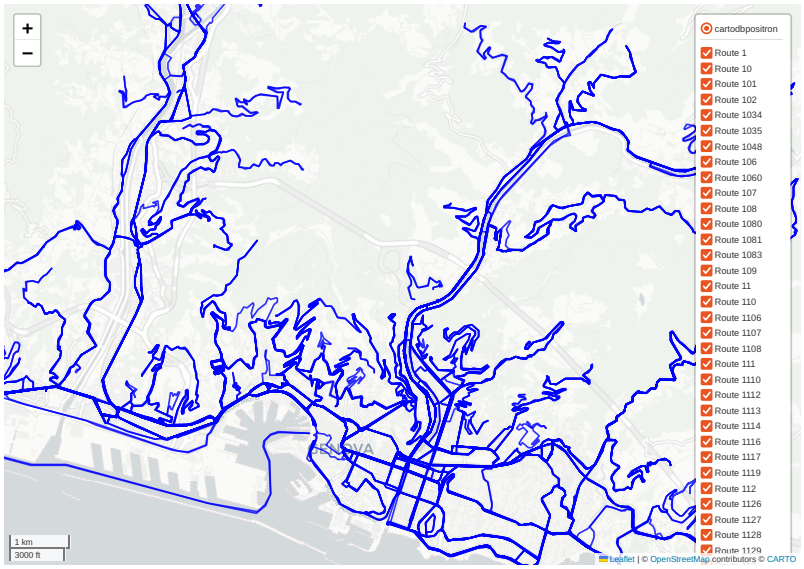


Fig. 11.8 Interactive visualization of the public transport routes in Genova, Italy.

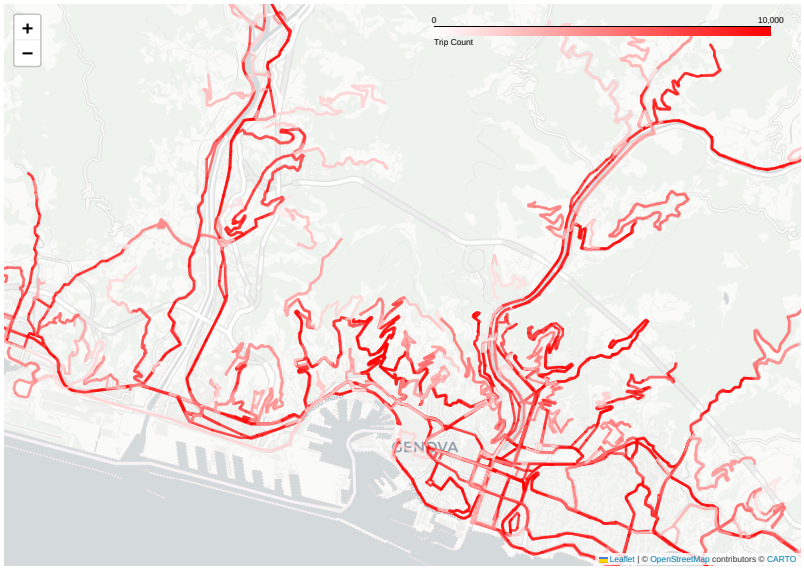
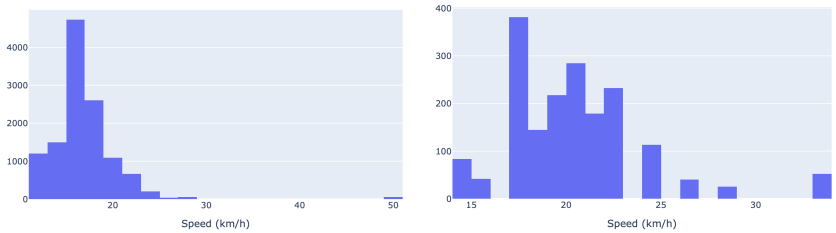


Fig. 11.9 Visualization of the number of trips per segment in Genova.

of a transit network in a machine-readable format, offering up-to-date transit recommendations and alerts.



(a) Avio1 – Pacinotti **(b)** Apparizione – Monte Fasce

Fig. 11.10 Estimated speed distribution across two segments in Genova.

The GTFS Realtime data are encoded in protocol buffers (commonly known as protobuf), a compact and efficient binary format designed by Google. A GTFS Realtime feed may contain three core types of updates:

- *Vehicle positions*, which provide live locations of transit vehicles, such as buses or trams. These data are essential for displaying vehicle locations on maps or performing speed analysis.
- *Trip updates*, which reflect changes to scheduled trips, such as delays, cancellations, or changes in route. These data keep passengers informed of deviations from planned services.
- *Service alerts*, which broadcast service-related information, such as disruptions, accidents, or changes in routes due to road closures. These alerts allow users to make informed decisions based on real-time conditions.

Applications ingesting GTFS Realtime feeds, such as navigation apps or transit management platforms, can provide users with real-time trip planning, accurate arrival predictions, and up-to-date service information. Unlike static GTFS data, which rarely changes, GTFS Realtime is designed to be updated frequently, with new information often being published every ten to thirty seconds, depending on the transit agency. This real-time feed format has been widely adopted by transit agencies around the world and is supported by a variety of transit data platforms and applications.

11.4.1 Ingest GTFS Realtime Feed

This section presents a Python script that collects and stores real-time vehicle position data from the Rīga Satiksme GTFS Realtime endpoint.[12]

```
# Library imports
import requests
from google.transit import gtfs_realtime_pb2
```

[12] https://saraksti.rigassatiksme.lv/gtfs_realtime.pb

```
GTFS_REALTIME_URL = "https://saraksti.rigassatiksme.lv/gtfs_realtime.pb"

# Function that gets a parsed GTFS Realtime protobuf feed from a URL
def fetch_gtfs_realtime_data(url):
  feed = gtfs_realtime_pb2.FeedMessage()
  try:
    response = requests.get(url, verify=False)
    # Check for HTTP errors
    response.raise_for_status()

    # Parse the protobuf data
    feed.ParseFromString(response.content)

  except requests.exceptions.RequestException as e:
    print(f"Error fetching GTFS Realtime data: {e}")
    return None
  return feed
```

Function fetch_gtfs_realtime_data retrieves the GTFS Realtime feed from the Riga endpoint using the requests library. Notice that the SSL verification is disabled, which is generally a risky and unadvisable practice. We use it for simplicity in this example, to work around the https certificate. In a deployment scenario, a more secure solution should be devised, for instance, downloading and manually trusting the certificate. The response, which is encoded in protobuf, is parsed using Google's gtfs_realtime_pb2 library. If the feed is successfully fetched, it returns a FeedMessage object containing the real-time data.

The next step consists in extracting the feed content, which is done below.

```
# Function that extracts the vehicle positions from a feed
def extract_vehicle_positions(feed):
  vehicle_positions = []
  for entity in feed.entity:
    if entity.HasField('vehicle'):
      vehicle = entity.vehicle
      vehicle_positions.append({
        "id": entity.id,
        "trip_id": vehicle.trip.trip_id,
        "schedule_relationship": vehicle.trip.schedule_relationship,
        "latitude": vehicle.position.latitude,
        "longitude": vehicle.position.longitude,
        "bearing": vehicle.position.bearing,
        "speed": vehicle.position.speed * 3.6, # Speed in km/h
        "current_status": vehicle.current_status,
        "timestamp": vehicle.timestamp,
        "stop_id": vehicle.stop_id,
        "vehicle_id": vehicle.vehicle.id,
        "vehicle_label": vehicle.vehicle.label,
        "vehicle_license_plate": vehicle.vehicle.license_plate
      })
  return vehicle_positions
```

Note that this feed contains two types of messages, namely, vehicle_position and trip_update. In this analysis, we focus on the vehicle_position messages. The extract_vehicle_positions function given below iterates over the entity fields within the feed. The condition HasField('vehicle') selects only the vehicle_position messages from the feed. Each entity contains information about an individual vehicle, including its trip_id, position (latitude, longitude, speed), and vehicle-specific details such as the vehicle_id, label, and license_plate. The function constructs and returns a list with these data.

We are now ready to ingest the data and store them for further processing.

11.4.2 Create Parquet Files from GTFS Realtime Feed

A function call retrieving a GTFS Realtime feed delivers a snapshot of the vehicle positions at the time of the call. For analytical purposes, such as speed analysis, we would need to ingest the feed multiple times during a certain time interval. The ingestion rate is defined according to the application requirements, for example, every ten to thirty seconds. Collecting vehicle position data over an extended period enables constructing trajectories of vehicles and performing trajectory-level analysis. The script below read the feeds and store them into a pandas DataFrame.

```
# Function that collects a sequence of timestamped positions from a real-time feed
def collect_vehicle_positions(duration_minutes, interval_seconds):
  collected_data = []
  end_time = time.time() + duration_minutes * 60 # Convert minutes to seconds

  while time.time() < end_time:
    # Fetch the GTFS Realtime data
    feed = fetch_gtfs_realtime_data(GTFS_REALTIME_URL)

    # If feed is fetched, extract vehicle positions
    if feed:
      vehicle_positions = extract_vehicle_positions(feed)
      if vehicle_positions:
        collected_data.extend(vehicle_positions) # Add the new data to the list
    else:
      print("Failed to fetch the GTFS Realtime feed.")

    # Wait for the specified interval before making the next request
    time.sleep(interval_seconds)

  # Convert the collected data into a DataFrame
  df = pd.DataFrame(collected_data)
  return df
```

The function extract_vehicle_positions introduced in the previous section obtains the positions of vehicles at a given instant. Function collect_vehicle_positions collects a sequence of these feeds by continuously fetch-

ing the GTFS Realtime data at regular intervals (for instance, every ten seconds) for a specified duration (say, one hour). Each time a vehicle position is extracted, it is appended to a list. After the collection period is over, the collected data are converted into a pandas DataFrame and returned. Now, we need to store these data permanently. In this section, we present a solution based on Parquet file storage.

The Parquet file format is a columnar storage format optimized for use in big data processing environments.[13] It was developed as part of the Apache Hadoop ecosystem and is now widely used for storing large datasets in an efficient way. Organizing data into columns allows obtaining efficient throughput when reading data and particularly when performing data aggregation tasks. Also, the encoding scheme makes this solution very efficient when we have datasets with many columns ("wide" tables) but where we only need a small subset of them. High compression rates are obtained with columnar formats, thus reducing the space needed to store large datasets. Parquet is also compatible with various data processing frameworks, including Apache Spark, and Python's pandas and PyArrow libraries, making it a popular choice for data-intensive applications.

The Parquet file format offers a range of compression options to optimize storage and performance for large datasets. By default, Parquet applies column-wise compression, meaning each column is compressed independently, which significantly enhances the ability to compress data with similar types and values. Several compression algorithms are available in Parquet, each with different trade-offs between compression speed and size. Usual algorithms include (1) Snappy, which provides fast compression and decompression with moderate compression ratios, making it ideal for real-time analytics; (2) GZIP, which offers higher compression ratios but at the cost of slower read and write performance (3) Brotli, which is highly effective for compressing large text or string-based columns; and (4) ZSTD (Zstandard), which offers a balance between high compression ratios and speed, particularly effective for large-scale data processing. The flexibility in choosing compression algorithms allows users to tailor the storage format to specific performance or storage constraints, depending on the workload and system architecture. By default, PyArrow uses Snappy compression when writing Parquet files if no compression option is explicitly set.

The function below stores the vehicle positions in Parquet format.

```
# Library imports
import pyarrow.parquet as pq
import pyarrow as pa

# Function that stores a DataFrame containing vehicle positions in a Parquet file
def save_to_parquet(df, file_name):
    if not df.empty:
        table = pa.Table.from_pandas(df)
```

[13] https://parquet.apache.org

```
        pq.write_table(table, file_name)
        print(f"Data successfully saved to {file_name}")
    else:
        print("No data to save.")
```

The function above is called by a script whose main body is shown below. This script sets the parameters for data collection and storage and then stores the data in a Parquet file.

```
# Main script
if __name__ == "__main__":
    # Parameters
    DURATION_MINUTES = 60 # Collect data during one hour
    INTERVAL_SECONDS = 10 # Call the endpoint every ten seconds

    # Collects vehicle positions over the specified duration
    print(f"Starting data collection for {DURATION_MINUTES} minutes,
      querying every {INTERVAL_SECONDS} seconds...")
    vehicle_positions_df = collect_vehicle_positions(DURATION_MINUTES,
      INTERVAL_SECONDS)

    # Save the data to a Parquet file
    output_file = "riga_vehicle_positions.parquet"
    save_to_parquet(vehicle_positions_df, output_file)
```

The function save_to_parquet takes two arguments: a pandas DataFrame vehicle_positions_df containing the data to be saved and a string stating the name of the output file, namely riga_vehicle_positions.parquet. The function converts the DataFrame into a PyArrow in-memory table, which has the same columnar structure of a Parquet file. This conversion is necessary because the Parquet format relies on the PyArrow library for handling columnar data. The PyArrow table is then written to disk as a Parquet file, which will contain the vehicle positions ingested into the DataFrame during the specified period. To allow the reader to reproduce the results in this chapter, we provide the file that we have obtained in the companion GitHub repository. Obviously, running this script to get the current feeds would produce different results from the ones shown here.

11.4.3 Visualize Trajectories using DuckDB

Modern query engines have evolved to handle large-scale data stored in columnar formats like Parquet. This offers flexibility and performance for data science workflows. In this section, we show how we can query the data stored in the Parquet file produced in the previous section using DuckDB,[14] an in-memory database optimized for interactive queries on local datasets.

[14] https://duckdb.org/

The result is then displayed using Python libraries. In the next section, we show a similar solution using PostgreSQL.

The Python script below uses a DuckDB query that retrieves the real-time trajectory of Tram 1 in the city of Riga.

```
# Library imports
import duckdb
import pandas as pd
import plotly.express as px

# Path to the Parquet file
parquet_file = "riga_vehicle_positions.parquet"

# Function that extracts the trajectories of Tram 1 from the Parquet file
def query_tram1_trajectory(parquet_file):

    # Connect to DuckDB in-memory database
    con = duckdb.connect()

    query = f"""
    SELECT *
    FROM parquet_scan('{parquet_file}')
    WHERE trip_id LIKE '%TRAM1-%'
    ORDER BY timestamp
    """

    # Store the result of the query in a DataFrame
    df = con.execute(query).fetchdf()

    # Close the connection
    con.close()

    return df
```

The script imports DuckDB as a Python library. The query_tram1_trajectory function first establishes a connection with DuckDB using duckdb.connect. In the SQL query, the parquet_scan function reads data directly from the Parquet file, without converting it to another format. The rest of the query filters the data to extract only the records where the trip_id matches Tram 1 and sorts the result chronologically by the timestamp. The result of the query is returned as a pandas DataFrame using fetchdf. After executing the query, the DuckDB connection is closed, freeing any associated resources.

With the resulting DataFrame, we can produce visualizations and perform further analyses. We show below how to visualize using the Plotly library, the trajectory of Tram 1 together with its speed along the way.

```
# Function that visualizes the trajectory of Tram 1 from a DataFrame using Plotly
def visualize_tram1_trajectory(df):
    if df.empty:
        print("No data available for Tram 1.")
        return
```

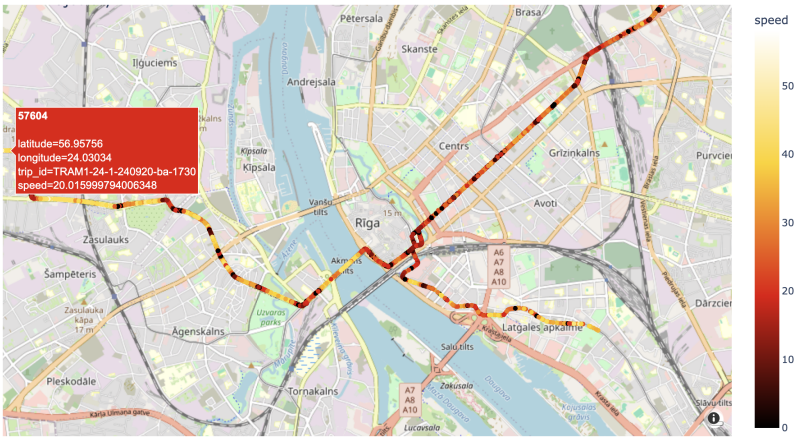


Fig. 11.11 Visualization of a real-time trajectory and speed of Tram 1 in Riga.

```
    fig = px.scatter_mapbox(df, lat="latitude", lon="longitude",
        hover_name="vehicle_id", hover_data=["trip_id", "speed"],
        color="speed", color_continuous_scale="Hot",
        title="Trajectory of Tram 1", zoom=12)

    fig.update_layout(mapbox_style="open-street-map", mapbox_zoom=12,
        mapbox_center={"lat": 56.9496, "lon": 24.1052})
    fig.update_layout(margin={"r":0,"t":0,"l":0,"b":0})

    fig.show()

# Main script
if __name__ == "__main__":
    # Load and query the trajectory of Tram 1
    tram1_df = query_tram1_trajectory(parquet_file)

    if not tram1_df.empty:
        # Visualize the trajectory of Tram 1
        visualize_tram1_trajectory(tram1_df)
    else:
        print("No data found for Tram 1.")
```

We do not get into the code details to avoid repetition of concepts. We can see that the code includes the possibility of hovering over the trajectory interactively to obtain details. The resulting visualization is shown in Fig. 11.11. On the right-hand side we can see the speed scale. We can see that the speed is relatively low in the central part of the city and it gets higher as the tram moves away from the city center. We can also see the red text block, as a result of hovering over the figure.

11.4.4 Visualize Trajectories using PostgreSQL

In this section, we show an alternative strategy to the one in the previous section, where we load the vehicle position data from the Parquet file and store them in a PostgreSQL database for visualization and further analysis. We can use DuckDB to transform the Parquet file into a CSV file as follows.

```
COPY (SELECT * FROM read_parquet("riga_vehicle_positions.parquet") )
TO '/home/user/data/riga_vehicle_positions.csv' (HEADER, DELIMITER ',');
```

We can now create the vehicle_positions table as follows.

```
CREATE TABLE vehicle_positions(id text, trip_id text, schedule_relationship text,
  latitude float, longitude float, bearing float, speed float, current_status text,
  timestamp bigint, stop_id text, vehicle_id text, vehicle_label text,
  vehicle_license_plate text);
COPY vehicle_positions(id, trip_id, schedule_relationship, latitude, longitude, bearing,
  speed, current_status, timestamp, stop_id, vehicle_id, vehicle_label,
  vehicle_license_plate)
FROM '/home/user/data/riga_vehicle_positions.csv' DELIMITER ',' CSV HEADER;
```

Next, we create an interactive web application using the Dash framework to visualize vehicle positions from the table in the PostgreSQL database.

```
# Library imports
import psycopg as pg
import dash
from dash import dcc, html
from dash.dependencies import Input, Output
import plotly.express as px
import dash_bootstrap_components as dbc

# Function that connects to the PostgreSQL database
def connect_to_postgres():
    try:
        conn = pg.connect(...)
        cur = conn.cursor()
        return conn, cur
    except Exception as e:
        print(f"Error connecting to PostgreSQL: {e}")
        return None, None

# Function that fetches available trip_ids from table vehicle_positions
def fetch_trip_ids():
  conn, cur = connect_to_postgres()
  if conn is None or cur is None:
    return []

  trip_id_query = "SELECT DISTINCT trip_id FROM vehicle_positions;"
  cur.execute(trip_id_query)
  trip_ids = [row[0] for row in cur.fetchall()]
  cur.close()
  conn.close()
  return trip_ids
```

```
# Function that fetches the vehicle positions for a specific trip_id
def fetch_vehicle_positions(trip_id):
  conn, cur = connect_to_postgres()
  if conn is None or cur is None:
    return pd.DataFrame()

  query = """
  SELECT latitude, longitude, vehicle_id, speed, timestamp
  FROM vehicle_positions
  WHERE trip_id = %s
  ORDER BY timestamp;
  """

  cur.execute(query, (trip_id,))
  vehicle_positions = pd.DataFrame(cur.fetchall(), columns=
    ['latitude', 'longitude', 'vehicle_id', 'speed', 'timestamp'])
  cur.close()
  conn.close()

  return vehicle_positions

# Dash visualization layout
app = dash.Dash(__name__, external_stylesheets=[dbc.themes.BOOTSTRAP])

# Get initial trip_ids to populate the dropdown
trip_ids = fetch_trip_ids()

# Set default trip_id (first one in the list)
initial_trip_id = trip_ids[0] if trip_ids else None

app.layout = dbc.Container([
  html.H1("Visualization of the Vehicle Positions"),

  # Dropdown for selecting trip_id
  dcc.Dropdown(
    id="trip-dropdown",
    options=[{'label': trip_id, 'value': trip_id} for trip_id in trip_ids],
    value=initial_trip_id, # Default to the first trip_id
    clearable=False, style={"width": "80%"}),

  # Graph to display vehicle positions
  dcc.Graph(id="vehicle-map")
])

# Callback function to update map based on selected trip_id
@app.callback(
  Output("vehicle-map", "figure"),
  Input("trip-dropdown", "value"))
def update_map(trip_id):
  if trip_id is None:
    return px.scatter_mapbox() # Empty map if no trip_id is selected
```

```
    # Fetch vehicle positions for the selected trip_id
    vehicle_positions_df = fetch_vehicle_positions(trip_id)

    if vehicle_positions_df.empty:
        return px.scatter_mapbox() # Return empty map if no data


    # Create a map visualization using Plotly
    fig = px.scatter_mapbox(vehicle_positions_df,
        lat="latitude", lon="longitude", hover_name="vehicle_id",
        hover_data=["speed", "timestamp"],
        title=f"Vehicle positions for {trip_id}", zoom=12)

    fig.update_traces(marker=dict(size=10, color='red')) # Increase the size and set
        color to red

    fig.update_layout(mapbox_style="open-street-map",
        mapbox_center={"lat": vehicle_positions_df["latitude"].mean(),
        "lon": vehicle_positions_df["longitude"].mean()},
        margin={"r":0,"t":0,"l":0,"b":0})

    return fig

# Main function to run the Dash visualization
if __name__ == "__main__":
    app.run_server(debug=True, port=8050)
```

The first function in the script is, as usual, connect_to_postgres, which sets the connection to the database, and after that, the function fetch_trip_ids is called to read the identifiers of the trips from the vehicle_positions table. Then, the function fetch_vehicle_positions takes a trip identifier as a parameter to run a query over table vehicle_positions and select a particular trip. It provides an interface where the first trip_id is selected by default, allowing users to visualize vehicle positions on a map generated by Plotly with OpenStreetMap tiles. Finally, update_map(trip_id) builds the map, which is shown in Fig. 11.12. We can see that users can select a specific trip_id from a dropdown menu to display the corresponding vehicle position points on a map. The map shows the vehicle's latitude and longitude, and additional data such as speed and timestamp are available via tooltips.

As shown in Fig. 11.13, the vehicle positions are not precisely aligned with the road network. If required for more accurate analysis, the map-matching techniques discussed in Sect. 9.7 can be employed to correct these GPS discrepancies. However, in this section, as highly accurate coordinates are not essential for our analysis, we will apply a simpler snapping approach instead of full map-matching, as will be demonstrated later.

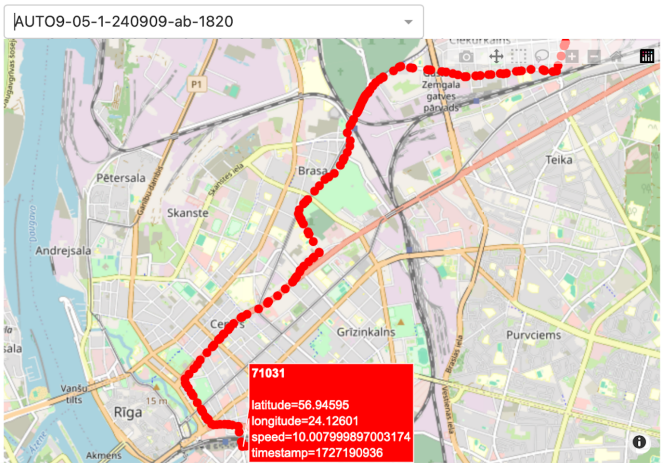


Fig. 11.12 Visualization of the GTFS Realtime vehicle positions of a trip in Riga.

Fig. 11.13 Detailed view of the trip in Fig. 11.12 showing small errors in the vehicle positions, making them to appear outside the road network.

11.4.5 Build Trajectories and Segments

Before moving into the analysis between GTFS Schedule and GTFS Realtime data, we need to perform some preparation tasks. We first add the attribute geom to the vehicle_positions table and populate it by constructing a point geometry using the ST_Point function.

```
ALTER TABLE vehicle_positions ADD COLUMN geom geometry(Point, 4326);
UPDATE vehicle_positions SET geom = ST_Point(longitude, latitude, 4326);
```

We now create and populate a table called actual_trips that will contain the spatiotemporal trajectories of the vehicle trips as follows:

```
CREATE TABLE actual_trips(trip_id, trip) AS
WITH positions(trip_id, geom, t) AS (
  -- We use DISTINCT since we observed duplicate tuples
  SELECT DISTINCT trip_id, ST_Transform(geom, 3059), to_timestamp(timestamp)
  FROM vehicle_positions )
SELECT trip_id, tgeompointSeq(array_agg(tgeompoint(geom, t) ORDER BY t)
  FILTER (WHERE geom IS NOT NULL))
FROM positions
GROUP BY trip_id;
```

Table positions converts the geometries into the local coordinate system with SRID 3059 for Riga using ST_Transform and converts the timestamps from Unix epoch to PostgreSQL timestamps with time zone with function to_timestamp. The query then populates the actual_trips table by grouping the tuples in the positions table that belong to the same trip as explained in Chap. 5: the vehicle positions are grouped by trip_id, and the spatiotemporal points are aggregated into a sequence using the aggregate function array_agg and the constructor tgeompointSeq. This results in almost 900 trips. Notice that some of these trips are not complete. Since the data collection was performed during one hour, only a portion of some trips were captured during this interval. Finally, we add a column with the geospatial trajectory of the trips to facilitate visualization as follows:

```
ALTER TABLE actual_trips ADD COLUMN trajectory geometry;
UPDATE actual_trips SET trajectory = trajectory(trip);
```

The visualization of these trajectories in QGIS is shown in Fig. 11.14. Even if the trajectories look accurate, a closer look would reveal that they are not map-matched and thus, there are small errors that result in portions of the trajectories falling outside the road limits.

As we mentioned, one of our goals is to evaluate the vehicle speeds and delays along the network segments. To achieve this, we need to relate the actual trip trajectories with the predefined stops and segments from the schedule data. However, note that the trajectories stored in table actual_trips do not contain explicit information about the stops or segments. Therefore, to analyze speed and delays we need to join the data obtained from GTFS Realtime with the data in the GTFS Schedule. We explain this next.

Normally, the GTFS Realtime and GTFS Schedule datasets contain a trip_id attribute, which is used to link the real-time data with the scheduled trip information. In theory, this trip_id should provide a straightforward way to join these two datasets. However, in practice, many cities' open data portals contain GTFS Realtime feeds where the trip identifiers are either missing or do not match the corresponding identifiers in the GTFS Schedule data and therefore the two datasets cannot be directly joined based on the trip_id. There is no standardized method for solving this issue and the approach may vary depending on the city and the data quality. In the case of

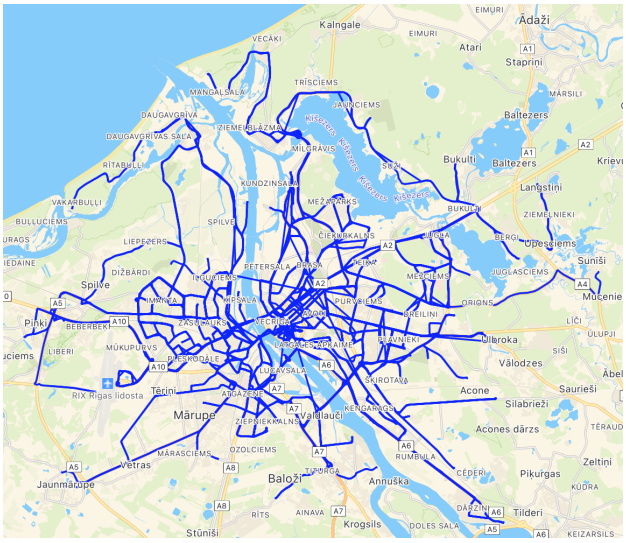


Fig. 11.14 Visualization of the real-time public transport trajectories in Riga.

the Rīgas Satiksme data used in this chapter, we observed that the trip_id in the GTFS Schedule and GTFS Realtime datasets only differ in the portion of their values that encodes the date. For example, the trip_id=TRAM1-12-1-240905-ab-1740 in the schedule data corresponds to TRAM1-12-1-240920-ab-1740 in the realtime data. We can see that the only difference is the part corresponding to the date (240905 versus 240920), although both identifiers represent the same trip.

The solution of the mismatch problem explained above and the join between the schedule and realtime datasets are shown in the following query.

```
CREATE TABLE trip_stops AS
WITH Temp AS (
  SELECT a.trip_id AS actual_trip_id, ad.trip_id AS schedule_trip_id, shape_id,
    s.stop_id, s.stop_name, ad.stop_sequence, s.stop_loc::geometry,
    t_arrival AS schedule_time, nearestApproachInstant(a.trip,
    ST_Transform(s.stop_loc::geometry, 3059)) AS stop_instant
  FROM actual_trips a, arrivals_departures ad, stops s
  WHERE ad.date = '2024-09-24'::timestamp AND ad.stop_id = s.stop_id AND
    regexp_replace(a.trip_id, '([^-]+-[^-]+-[^-]+)-[0-9]+(-.*)', '\1\2') =
    regexp_replace(ad.trip_id, '([^-]+-[^-]+-[^-]+)-[0-9]+(-.*)', '\1\2') AND
    nearestApproachDistance(a.trip, ST_Transform(s.stop_loc::geometry, 3059)) < 10)
SELECT actual_trip_id, schedule_trip_id, shape_id, stop_id, stop_name, stop_sequence,
  stop_loc, schedule_time, getTimestamp(stop_instant) AS actual_time,
  ST_Transform(getValue(stop_instant), 4326) AS trip_geom
FROM Temp;
```

The SQL script above creates a new table trip_stops, which samples actual trips at the stop locations defined in the GTFS Schedule. The process begins

computing the table Temp, which selects data from three sources: actual_trips, which contains the real-time trajectories of the trips, the arrivals_departures view, which is derived from the GTFS Schedule and contains all scheduled stops for each trip, and table stops, which contains the geometries of the stops. In the WHERE clause we used the PostgreSQL regexp_replace function which masks the date portion of the trip_id in both datasets, enabling us to match the remaining parts of the identifier to link the trips in the schedule data with those in the real-time data. The query retrieves also the shape_id attribute from the arrivals_departures view, since it is the one that will allows us to identify all the trips that pass through the same segment, this attribute is also in the Segments table.

The query above also uses two important MobilityDB functions, nearestApproachInstant and nearestApproachDistance, to estimate the position and the time at which the vehicle in the actual trip came closest to a scheduled stop. These functions allow us to match the actual trip trajectory with the scheduled stop locations. The condition in the nearestApproachDistance function applies a cut-off threshold of ten meters, ensuring that only stops that the trip actually traversed are considered in the final result. This is crucial because some actual trips may be incomplete as explained above due to limited data collection interval, meaning that not all trips in the data visited every scheduled stop. Without this threshold, the script might join incomplete trips with all their scheduled stops leading to inaccurate or misleading results. By sampling only the stops that the trip closely passed by, we can estimate the time when each vehicle reached the stop, since this information is not delivered by the GTFS Realtime feed, which is crucial for delay analysis.

In the final step, data from the Temp table is selected and inserted into the trip_stops table. The table includes identifiers of scheduled and actual trips, the shape_id of the trips, the stop information (that is, identifier, name and stop_sequence), and the scheduled time of arrival (schedule_time) for each stop. The stop_sequence allows us to detect the portion of the trajectory captured by the feed. Additionally, the script records the actual time when the vehicle reached its nearest approach to the stop (actual_time) and the corresponding geographical location (trip_geom) in longitude and latitude using SRID 4326. This table provides a detailed view of the differences between scheduled and actual trip data, allowing analysis of delays, early arrivals, or deviations from the planned route. Figure 11.15 shows the result of this query, where the red diamonds visualize the stop_loc attribute and the white dots visualize the trip_geom attribute. We can see that all points reflect the condition of being at less than ten meters from each other.

We are now ready to build the segments between two consecutive stops. The following SQL query creates a table called trip_segments that creates such segments by joining consecutive trip stops.

```
CREATE TABLE trip_segments AS
SELECT actual_trip_id, schedule_trip_id, shape_id, stop_id AS end_stop_id,
  schedule_time AS end_time_schedule, actual_time AS end_time_actual,
```

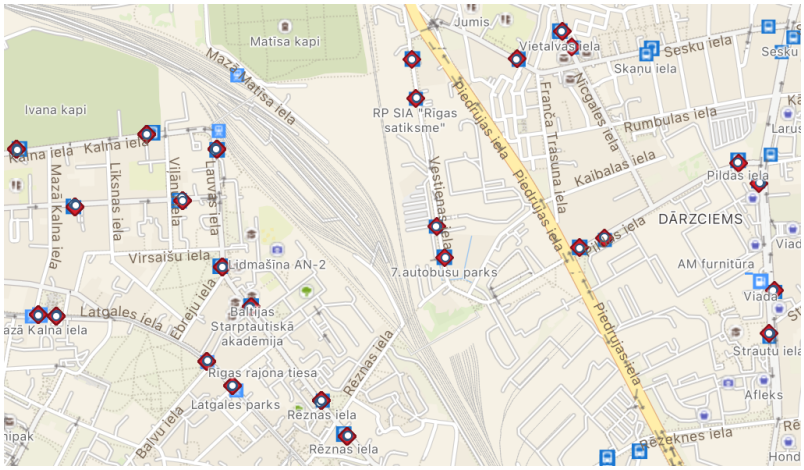


Fig. 11.15 Actual trip trajectories (white dots) sampled at their scheduled stops (red diamonds).

```
  -- Use LAG to get the previous stop's information
  LAG(stop_id) OVER (PARTITION BY actual_trip_id ORDER BY stop_sequence)
    AS start_stop_id,
  LAG(schedule_time) OVER (PARTITION BY actual_trip_id ORDER BY
    stop_sequence) AS start_time_schedule,
  LAG(actual_time) OVER (PARTITION BY actual_trip_id ORDER BY stop_sequence)
    AS start_time_actual
FROM trip_stops;
```

The query uses the LAG window function for joining a stop with its preceding one in the same trip_id tuple. Each tuple in the resulting trip_segments table contains two consecutive stops (start_stop_id and end_stop_id), and the scheduled and actual times for the start and end stops of each segment (e.g., end_time_schedule and end_time_actual).

The trip_segments table above, together with the segments table from the GTFS Schedule, which contains in particular the distance between two consecutive stops, are used to produce segment-wise statistics. This is shown in the next section.

11.5 Comparison of GTFS Schedule and Realtime Data

We now have the machinery required to perform analysis of the public transport performance. From the GTFS Schedule we have obtained the planned itineraries and from GTFS Realtime feeds we obtained the actual arrivals and departures occurring at each stop. We will now enhance these data storing them in a mobility database to identify delays and their causes at three

different granularities: trajectories, segments, and stops. This allows us to study, for instance, the performance of the whole trips, the speed at each segment (also comparing it against the scheduled average speed), and the delay at each stop (as a whole and per line). In this section, we show how to analyze speeds and delays using the data produced in the previous section.

11.5.1 Analyze Vehicle Speed

After the preparation of trip segments in the previous section, ce can compute segment-wise statistics using actual trip trajectories. The query below calculates the average speed for each segment of a trip.

```
SELECT AVG(s.distance_m /
  EXTRACT(EPOCH FROM (t.end_time_actual - t.start_time_actual)) * 3.6) AS
  speed_kmh, s.geometry, t.start_stop_id, t.end_stop_id
FROM trip_segments t, segments s
WHERE t.start_stop_id = s.start_stop_id AND t.end_stop_id = s.end_stop_id AND
  t.start_time_actual IS NOT NULL AND t.shape_id = s.shape_id
  EXTRACT(EPOCH FROM (t.end_time_actual - t.start_time_actual)) > 0
GROUP BY s.geometry, t.start_stop_id, t.end_stop_id;
```

The query joins the trip_segments table with the segments table, matching the start and end stop identifiers to retrieve the corresponding segment's geometry and distance. The speed is calculated by dividing the segment's distance (distance_m) by the actual time taken to travel between the two stops (end_time_actual - start_time_actual), and the result is multiplied by 3.6 to convert the speed from meters per second to kilometers per hour. The query ensures that only valid segments with non-null actual times and positive travel times are included. Finally, the result is grouped by segment, producing an average speed for each segment, which can be used to analyze the vehicle's performance on different parts of the network. The result of this query is visualized in Fig. 11.15. The main portions of the script are shown below, the full script can be found on the companion GitHub repository.

```
# Library imports
import ...

# SQL query to get the segments, the average speed, and the geometries
sql_query = """
SELECT AVG(s.distance_m /
  EXTRACT(EPOCH FROM(t.end_time_actual - t.start_time_actual)) * 3.6) AS
  speed_kmh, s.geometry, t.start_stop_id, t.end_stop_id
FROM trip_segments t, segments s
...
GROUP BY s.geometry, t.start_stop_id, t.end_stop_id;
"""

# Function that fetches the average speed per segment from PostgreSQL
def fetch_segment_speed_data():
```

```
    conn = pg.connect(...)
    gdf = gpd.GeoDataFrame.from_postgis(sql_query, conn, geom_col='geometry')
    conn.close()
    return gdf

# Function that creates a color map from white (low speed) to red (high speed)
def create_colormap(min_speed, max_speed):
    return cm.LinearColormap(['red', 'white'], vmin=min_speed, vmax=max_speed)
# Function that visualizes the average speed per segment on a Folium map
def visualize_speed_map(gdf, cutoff=35):
    # Create a Folium map centered around the area
    transport_map = fl.Map(location=[56.9496, 24.1052], zoom_start=12)
    # Create a colormap for speed values
    min_speed = gdf['speed_kmh'].min()
    # Apply cutoff to the maximum speed
    max_speed = min(gdf['speed_kmh'].max(), cutoff)
    colormap = create_colormap(min_speed, max_speed)
    # Add each segment to the map
    for _, segment in gdf.iterrows():
        # Apply cutoff to the speed
        speed_kmh = min(segment['speed_kmh'], cutoff)
        # Get color based on speed
        color = colormap(speed_kmh)
        # Convert the geometry to GeoJSON with associated properties
        geo_json = GeoJson(
            data={
                "type": "Feature",
                "geometry": segment['geometry'].__geo_interface__,
                "properties": {
                    "speed_kmh": round(segment['speed_kmh'], 2),
                    "from_stop_id": segment['start_stop_id'],
                    "to_stop_id": segment['end_stop_id']
                }
            },
            style_function=lambda x, color=color: {
                'color': color,
                'weight': 3,
                'opacity': 0.7
            },
            tooltip=GeoJsonTooltip(
                fields=['speed_kmh', 'from_stop_id', 'to_stop_id'],
                aliases=['Speed (km/h)', 'From Stop', 'To Stop'],
                localize=True
            )
        )
        geo_json.add_to(transport_map)
    # Add the colormap to the map
    colormap.add_to(transport_map)
    return transport_map
```

```
# Main script
if __name__ == "__main__":
    # Fetch the segment speed data
    segment_speed_gdf = fetch_segment_speed_data()
    # Visualize the average speed per segment on a Folium map
    speed_map = visualize_speed_map(segment_speed_gdf)
    # display the map
    display(speed_map)
```

The scheme of the script is as usual: a function fetch_segment_speed_data to fetch each segment and the speed data, a function create_colormap to create a colormap where limits for the color gradient are defined, and a function visualize_speed_map to visualize the map which iterates over the segments and converts the geometries and their properties to a GeoJSON structure.

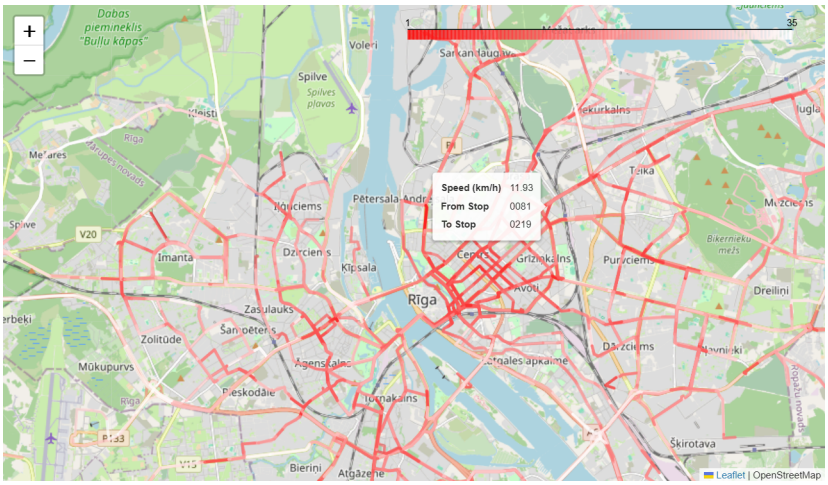


Fig. 11.16 Average speed of public transport vehicles over the network segments.

The query below computes a table with the actual and schedule times per segment, the delay per segment (measured as the difference between the former two values), and all the information of the segments.

```
CREATE TABLE trips_join_segments AS
SELECT t.actual_trip_id, t.schedule_trip_id, t.shape_id, s.distance_m,
  t.end_time_actual, t.start_time_actual,
  EXTRACT(EPOCH FROM (t.end_time_actual - t.start_time_actual)) AS
  elapsed_time_actual,
  EXTRACT(EPOCH FROM (t.end_time_schedule - t.start_time_schedule)) AS
  elapsed_time_schedule, t.end_time_schedule, t.start_time_schedule,
  s.geometry AS seg_geo, t.start_stop_id, t.end_stop_id, s.stop_sequence
FROM trip_segments t, segments s
WHERE t.start_stop_id = s.start_stop_id AND t.end_stop_id = s.end_stop_id AND
  t.start_time_actual IS NOT NULL AND t.shape_id = s.shape_id AND
  EXTRACT(EPOCH FROM (t.end_time_actual - t.start_time_actual)) > 0;
```

With this information we can compute, for example, how many trips have traversed each segment during the time period measured, as follows.

```
SELECT start_stop_id, end_stop_id, COUNT(*) AS no_trips_seg
FROM trips_join_segments
GROUP BY start_stop_id, end_stop_id
ORDER BY no_trips_seg DESC;
```

As a more involved example, we want to compute the number of vehicles per segment for intervals of 20 minutes. This can help us later, to see if there is a correlation between this level of segment occupancy and the delays in each network segment. The query that computes such aggregation is:

```
WITH bounds(start_time, end_time) AS (
  SELECT timestamp '2024-09-24 16:00:00', timestamp '2024-09-24 17:00:00'),
time_bins(bin) AS (
  SELECT span(h, h + interval '20 minutes')
  FROM bounds, generate_series(start_time, end_time, interval '20 minutes') AS h)
SELECT start_stop_id, end_stop_id, bin, seg_geo, COUNT(*) AS no_trips_seg_20min
FROM trips_join_segments t, time_bins b
GROUP BY start_stop_id, end_stop_id, bin, seg_geo
ORDER BY no_trips_seg_20min DESC, start_stop_id, end_stop_id, bin;
```

The bounds table defines the time interval when the GTFS Realtime feed was obtained. Table time_bins generates the intervals from the previous table, and the aggregation is computed in the main query. We included the geometry of the segment in the query result so we can display the result in QGIS as Fig. 11.17 shows. In the figure we can see that the green lines indicating a low time level of transit along the segment occur mainly in the outskirts of the city, while blue and red lines occur in the center.

11.5.2 Analyze Vehicle Delay

The analysis done in the previous sections allowed us to detect discrepancies between scheduled and actual trips, identifying where and how frequently delays occur. We now analyze public transport delay using GTFS Schedule and GTFS Realtime data together.

A preliminary analysis could be performed using the trip_stops table created in Sect. 11.4.5. To compute and visualize the delay at each stop, we first add a delay column to it as follows:

```
ALTER TABLE trip_stops ADD COLUMN delay numeric;
UPDATE trip_stops
SET delay = EXTRACT(EPOCH FROM actual_time - schedule_time) / 60;
```

The column contains the time difference between the scheduled and actual arrival time at each stop. After this, we can simply compute the average delay (for all lines) as follows:

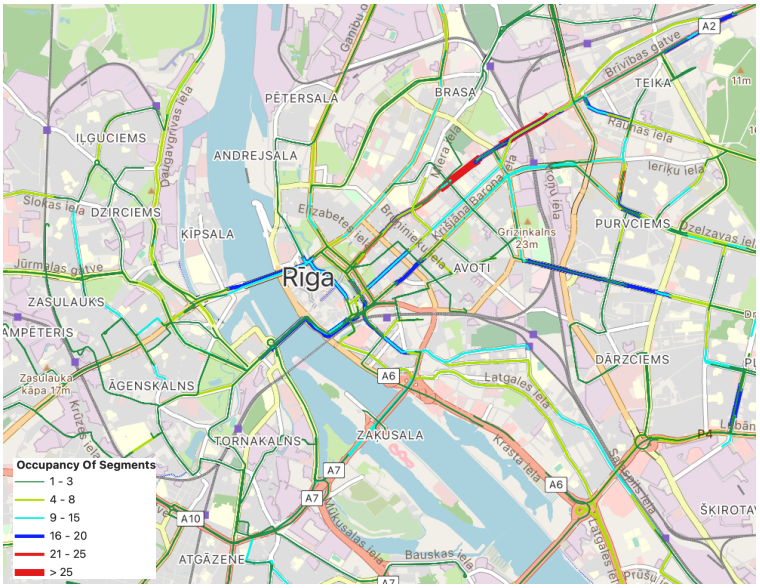


Fig. 11.17 Number of vehicles per segment in an interval of 20 minutes. Thick red and blue lines indicate the most busy segments, green lines indicate a low density of vehicles over time.

```
SELECT stop_id, stop_loc, AVG(delay) AS avg_delay
FROM trip_stops
GROUP BY stop_id, stop_loc;
```

The result is shown in Fig. 11.18. We can see that there is a correlation between the locations of the stops with higher delay and the segments with high occupancy density in Fig. 11.17. In the figure, the blue and red dots indicate stops with an average delay higher than the other ones.

To analyze delays at segment granularity, we will compare the actual time that takes to traverse each segment against the scheduled one. We then take the averages and for each segment indicate if the difference between the averages is positive (indicating that on average the segment contributes to a trip delay) or negative (indicating that on average the segment is traversed on time or faster than scheduled). The query is as follows:

```
WITH agg_by_seg AS (
  SELECT start_stop_id, end_stop_id, seg_geo, COUNT(*) AS no_trips_seg,
    AVG(elapsed_time_actual) AS avg_span_actual,
    AVG(elapsed_time_schedule) AS avg_span_schedule
  FROM trips_join_segments
  GROUP BY start_stop_id, end_stop_id, seg_geo)
SELECT seg_geo, no_trips_seg,
  CASE WHEN avg_span_actual - avg_span_schedule < 0 THEN 1 ELSE 0 END
  AS has_delay
FROM agg_by_segm;
```

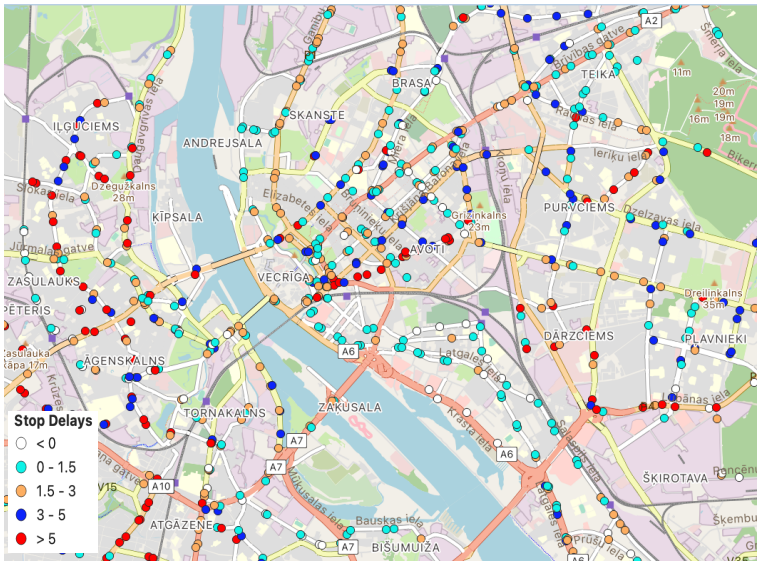


Fig. 11.18 Average delay at every stop for all public transport routes.

Table trips_join_segments contains all information needed by the query. The query above computes in table agg_by_seg the aggregate by segment, including the geometry of the segment to display the result. The main query computes if the difference between the actual and scheduled segment traversing times is positive or negative. Figure 11.19 shows the result. We can see that the downtown transport run with some delay on the average case.

The next step in our analysis would be to compute the delays between the scheduled and actual trips at every stop and analyze these delays as a time series. We define the delay as the difference between the time when a vehicle was scheduled to arrive at a stop and the time when it actually arrived. Since the schedules and actual times are stored as temporal sequences, we can compute the delay as the time difference between the corresponding points in the actual_trip and schedule_trip columns. The query below starts from the trip_stops table and after a number of steps transforms it into a time series to make it possible to display graphically the results in a dashboard. In this case, we use Grafana, like in Chap. 5. To make it easier to work with dashboards, we store the result of the query in a table called delay_time_series.

```
CREATE TABLE delay_time_series AS
WITH actual_schedule_trips AS (
  SELECT actual_trip_id, stop_id, schedule_trip_id, schedule_time, actual_time,
    tgeompointSeq(array_agg(tgeompoint(stop_loc, schedule_time)
      ORDER BY schedule_time)) AS schedule_trip,
    tgeompointSeq(array_agg(tgeompoint(trip_geom, actual_time)
      ORDER BY actual_time)) AS actual_trip
  FROM trip_stops
```

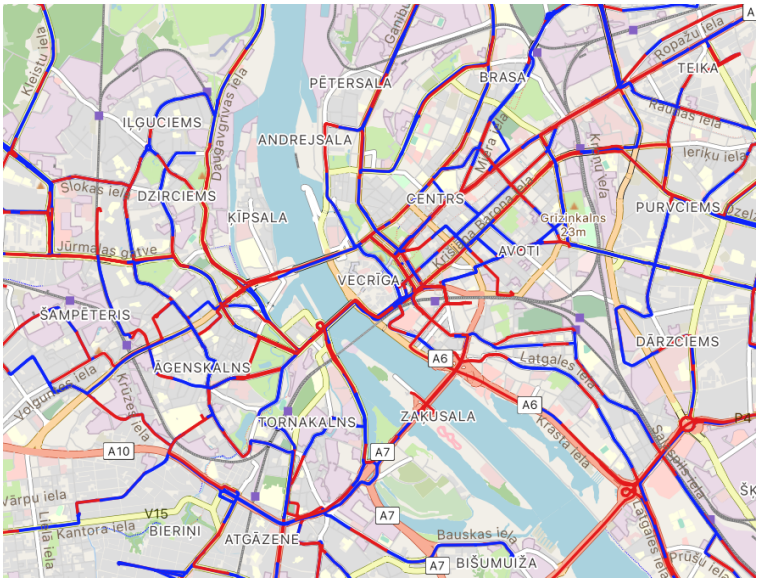


Fig. 11.19 Delays of public transports in Riga. Red lines indicate segments traversed in more time than it was scheduled.

```
  WHERE actual_trip_id LIKE 'TRAM1%' AND
    schedule_time BETWEEN '2024-09-24 16:00:00' AND '2024-09-24 18:00:00'
  GROUP BY actual_trip_id, schedule_trip_id, stop_id, schedule_time, actual_time )
SELECT schedule_trip_id, stop_id, schedule_time,
  EXTRACT(EPOCH FROM actual_time - schedule_time) / 60 AS delay
FROM (
  SELECT actual_trip_id, schedule_trip_id, stop_id, schedule_time, actual_time,
    unnest(timestamps(actual_trip)) AS actual_time,
    unnest(timestamps(schedule_trip)) AS schedule_time
  FROM actual_schedule_trips) t
ORDER BY schedule_trip_id, schedule_time;
```

The query above must be explained with certain detail because it involves many concepts studied in this book. The table actual_schedule_trips builds two sequences of scheduled and actual stops, denoted respectively schedule_trip and actual_trip, represented as spatiotemporal points built using the tgeompointSeq function. The two sequences are explained next.

- schedule_trip: The spatiotemporal sequence that contains the stops together with their scheduled arrival times. Notice that this trajectory is built using linear interpolation between the stops, therefore it does not follow the road network. This is mainly because the GTFS Schedule does not encode the speed variation between stops and only provides the arrival and departure times at such stops. Although we could linearly interpolate the vehicle positions between stops to match the network (i.e., assuming constant speed), it would not be relevant for the analysis presented here.

- actual_trip: The spatiotemporal sequence that contains the actual positions of the vehicle and their corresponding times matched with the stops. Similarly to the schedule_trip attribute, this trip trajectory skips all the vehicle positions between stops and assumes linear interpolation with constant speed. We have structured it in the same way as the schedule_trip to be able to compute delays.

In the query we focus on the trips of TRAM1 between 4 pm and 6 pm on September 24, 2024. Note that this query may fail when constructing trajectories for some trips due to errors in the trip_stops table, which can be caused by incomplete trips, trips that do not match their scheduled routes, or when two consecutive stops have the same timestamp, in which case the tgeompointSeq constructor would fail. Investigating these errors in more detail is beyond the scope of this section, since this would be specific to the Riga dataset and could not be generalized for other cases. The result of this part of the script is shown in Fig. 11.20.

actual_trip_id text	stop_id text	schedule_trip_id text	schedule_time timestamp with time zone	actual_time timestamp with time zone	schedule_trip tgeompoint	actual_trip tgeompoint
TRAM1-01-1-2…	3012	TRAM1-01-1-240…	2024-09-24 17:27:00+02	2024-09-24 17:28:53.0635…	[0101000020E…	[010100002…
TRAM1-01-1-2…	3013	TRAM1-01-1-240…	2024-09-24 17:29:00+02	2024-09-24 17:31:12.5319…	[0101000020E…	[010100002…
TRAM1-01-1-2…	3014	TRAM1-01-1-240…	2024-09-24 17:31:00+02	2024-09-24 17:34:31.3534…	[0101000020E…	[010100002…
TRAM1-01-1-2…	3043	TRAM1-01-1-240…	2024-09-24 17:33:00+02	2024-09-24 17:35:41.7491…	[0101000020E…	[010100002…
TRAM1-01-1-2…	3044	TRAM1-01-1-240…	2024-09-24 17:34:00+02	2024-09-24 17:36:41.5197…	[0101000020E…	[010100002…
TRAM1-01-1-2…	3045	TRAM1-01-1-240…	2024-09-24 17:35:00+02	2024-09-24 17:37:33.5073…	[0101000020E…	[010100002…
TRAM1-01-1-2…	3046	TRAM1-01-1-240…	2024-09-24 17:37:00+02	2024-09-24 17:38:53.9985…	[0101000020E…	[010100002…
TRAM1-01-1-2…	3047	TRAM1-01-1-240…	2024-09-24 17:38:00+02	2024-09-24 17:41:12.0965…	[0101000020E…	[010100002…
TRAM1-01-1-2…	3048	TRAM1-01-1-240…	2024-09-24 17:40:00+02	2024-09-24 17:43:18.1863…	[0101000020E…	[010100002…

Fig. 11.20 Table containing, for each trip, the actual and scheduled arrival times at each stop, together with the corresponding spatiotemporal points.

The main query extracts the timestamps from the actual_trip and schedule_trip sequences using the unnest function. The difference between each actual and scheduled timestamp is computed using the EXTRACT function, which converts the time difference into seconds. Dividing by 60 transforms the result into minutes, providing a more human-readable delay value. We have now transformed the temporal sequence into a time series ready for visualization in a specialized dashboard.

We now show how the results can be displayed in Grafana. In our queries we use the table delay_time_series as follows:

```
SELECT schedule_trip_id, schedule_time, delay
FROM delay_time_series
WHERE schedule_trip_id in ($trip);
```

The query is parameterized with a variable $trip that is instantiated at runtime with the identifier of the trip whose delay we want to show. Thus, we can design a dynamic dashboard, where the user can choose the trip to display.

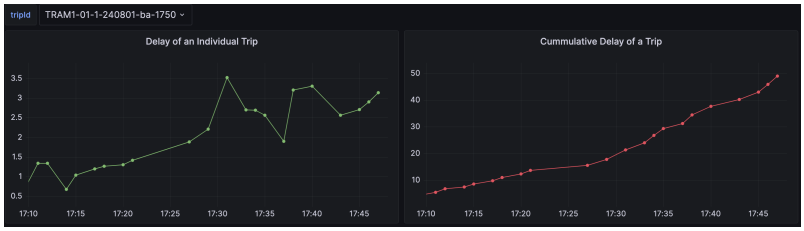


Fig. 11.21 Delay of a trip across time.

We depict a panel of the dashboard in Fig. 11.21, which, on the left-hand side, shows the result of the query for the trip number 1750. On the left-hand side of the figure we can see the delay at each sample time instant (indicated by the dots). On the right-hand side we display the cumulative sum of the delay across time. The query that computes this graph is:

```
WITH Temp AS (
  SELECT schedule_time, stop_id, delay, SUM(delay) OVER (PARTITION BY
    schedule_trip_id ROWS UNBOUNDED PRECEDING) AS total_delay
FROM delay_time_series)
SELECT schedule_time, total_delay
FROM Temp
WHERE schedule_trip_id IN ($trip);
```

We compute the cumulative sum using a window function. Note that the delay at the end of the curve does not represent the total delay of the trip, because it adds the delays at each sampled time instant.

Figure 11.22 shows a dashboard where we define a dynamic variable that allows selecting multiple trips and compare the delays. On the left-hand side we can see the development of the delays of four trips. We can also see that the lengths of the trips differ from each other because of the sampling time interval previously explained. Two of them (the green and the orange ones) have increasing delays, meaning that at most stops new delays occurred. This results in steeper cumulative curves on the right-hand side since the delays are never reduced along the way. A smoother slope in the cumulative curve means that some delays are "recovered" along the way, as can be seen on the left-hand side, where we observe negative values in the delays.

To conclude this section, we consider a group of tram lines and compare the total delay throughout the trip. We start with the next query:

```
CREATE TABLE trip_delay_total AS
SELECT actual_trip_id, schedule_trip_id,
  tgeompointSeq(array_agg(tgeompoint(stop_loc, schedule_time)
    ORDER BY schedule_time)) AS schedule_trip,
  tgeompointSeq(array_agg(tgeompoint(trip_geom, actual_time)
    ORDER BY actual_time)) AS actual_trip
FROM trip_stops
WHERE actual_trip_id LIKE 'TRAM%-%-ab-%'
GROUP BY actual_trip_id, schedule_trip_id;
```

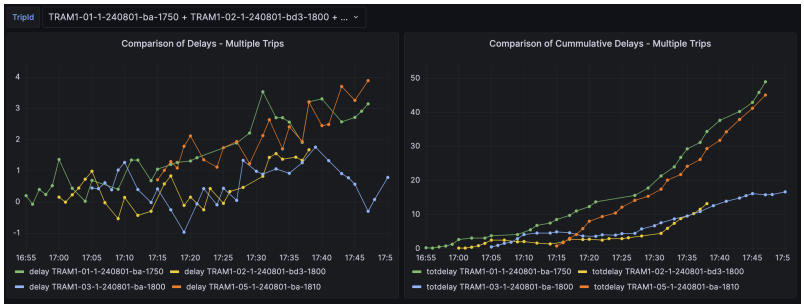


Fig. 11.22 Comparison of the delays of multiple trips.

Note that we filter the tram lines with the expression in the WHERE clause.

```
SELECT actual_trip_id, duration(timespan(actual_trip)) AS duration_actual,
  duration(timespan(schedule_trip)) AS duration_schedule,
  EXTRACT(EPOCH FROM (duration(timespan(actual_trip)) -
    duration(timespan(schedule_trip)))) / 60 AS delay,
  trajectory(actual_trip) AS trajectory
FROM trip_delay_total;
```

We do not get into the details of this query since it is similar to the one previously explained. The expressions duration(timespan(·)) compute the total duration of the actual and scheduled spatiotemporal trajectories. Figure 11.23 shows that most of the trips considered in the figure have a total delay of less than three minutes.

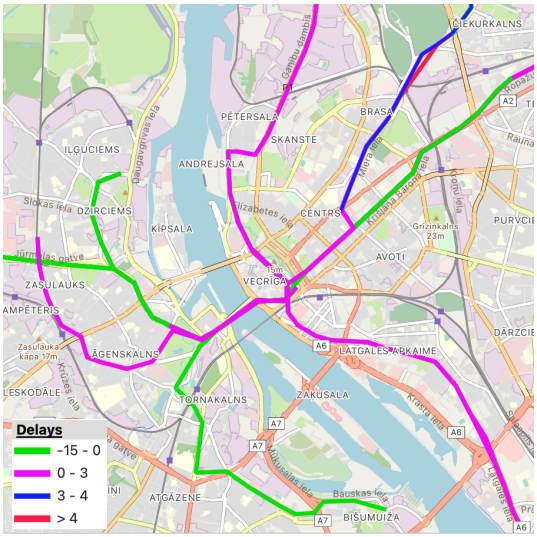


Fig. 11.23 Trip delay sampled at every stop.

11.6 Final Considerations

It is worth mentioning that the delay, defined as the difference between schedule and actual vehicle arrival at stops, is not always interesting for public transport companies. The major concern for public transport is passenger satisfaction. This is translated into *punctuality* and *regularity* of vehicle arrival times at stops. Punctuality, similarly to what we analyzed in this section, refers to whether the vehicles arrive at the stops on schedule as indicated in the GTFS file. Regularity is related to the notion of *headway*, which is the time interval between two consecutive vehicles. For frequent lines, i.e., where the headway is lower than 10 minutes, regularity is more relevant than punctuality as a measure of quality of service (QoS). For these lines, passengers tend to go to the stop without looking at the exact arrival times, as they know that they will have a maximum wait of a few minutes. For these lines, the goal would be to analyze whether the vehicles indeed arrive in the planned headway, rather than analyzing their exact arrival times. Notice that the planned headway per line changes depends on the hour of day. For instance, in peak hours the headway is shorter than in early and late night hours.

11.7 Summary

This chapter showed how mobility analysis can be applied to an urban mobility use case, also integrating many of the concepts studied in this book. We focused on the study of delay analysis in public transportation. We explained the Google Transit Feed Specification (GTFS) Schedule and Realtime standard data formats. Using GTFS data from Riga and Genova, the chapter explained the processes of loading, segmenting, and analyzing GTFS schedule data. We described a method to obtain estimated traffic speeds over the transport routes, which can benefit applications other than public transport. Further, a detailed methodology for joining schedule and real-time data was discussed, showing the process of calculating vehicle speed over network segments and analyzing delays at specific stops and along entire routes. We also explained that, to align the two data sources, a series of data transformations are necessary to match scheduled and actual vehicle positions, highlighting the challenges of working with heterogeneous data formats.

11.8 Bibliographic Notes

Analysis of public transport data is possible due to the existence of mobility data specifications such as GTFS and GBFS, which provide essential data formats for interoperability in transit systems. Various tools transforming

such representations for data analyses have been proposed. In this chapter, we used gtfs-via-postgres and gtfs_functions. In [167], Pereira et al. introduce gtfs2gps, an open-source R package that converts GTFS feeds from relational text files into a trajectory data table, similar to GPS records, with the timestamps of vehicles in every trip. In this chapter, we also used the Apache Parquet file format, which was primarily developed as an open-source project by Twitter and Cloudera. Due to its efficient, columnar storage and compression features, Parquet is widely adopted in big data and data lake systems. This has inspired the development of extensions for geospatial data [181] and for spatiotemporal data [119] to support mobility data in applications like transportation and logistics. Tools like DuckDB [171] and Apache Arrow DataFusion [122] can also be used for data processing and analysis of such kind of data.

There is an abundant literature analyzing key areas in urban mobility, in particular public and shared transport. Aemmer et al. present in [4] a method for extracting transit performance metrics from GTFS Realtime and aggregating them to roadway segments. This data is then used in terms of consistent, predictable delays, and random variation on a segment-by-segment basis. In public transport systems, schedule padding is the extra time added to transit schedules to account for random delays that may occur. Wessel and Widener [228] present a method to detect schedule padding in transport networks by analyzing transit schedules and real-time vehicle location data at the level of segments. Similar to the work presented in this chapter, Godfrid et al. [82] present a study of public transport in the city of Buenos Aires, Argentina, also including an analysis of delays of public buses. The work in [186] evaluates user-centered approaches for traffic analysis. Identifying bottlenecks in public transport has been addressed in [127, 247], which allows transit authorities to target infrastructure improvements and adjust schedules for better efficiency. The integration of different modes of transport, e.g., public transport and micromobility presents many challenges. Rider safety at intersections has been addressed in [202]. The effectiveness of bike-sharing services in urban transport has been assessed in [134], which suggested that bike-sharing may not boost public transit efficiency and can create inequalities.

The advances in machine learning and artificial intelligence have lend solutions for urban mobility. Work in graph convolutional neural networks for traffic analysis [187] point to advanced methodologies for data augmentation, enabling more accurate demand forecasting. The availability of large trajectory datasets has encouraged using them in map making. Works such as [45, 71, 148], focus on extracting road elements from GPS data. Huang et al. [104] review work in context-aware machine learning for intelligent transportation systems, examining key techniques across contextual data (e.g., location, time, weather, road conditions, and events), applications, modes, and learning methods. Another important topic in urban transport and vehicle performance is the fleet electrification and energy optimization. For

instance, studies on electrical vehicles and their energy consumption in urban settings, including roundabouts, provide insights into energy efficiency strategies [55, 56]. A survey on the major data management challenges, open issues, and applications of urban green mobility can be found in [47].

Policy frameworks such as the New EU Urban Mobility Framework [73] and emerging urban trends, like those discussed by Adler and Florida [3], support the integration of innovative data-driven solutions in urban planning. Digital mobility twins enable a better understanding of different kinds of mobility, for example urban [182, 184] and maritime [91]. These studies show the role of digital representations of these environments for optimizing transportation systems and managing mobility data. An architecture for Mobility Data Spaces is proposed in [64], with a focus on energy-efficient data.

11.9 Review Questions

11.1 What are the key data sources and formats used in urban mobility, specifically in the context of public transportation?
11.2 What are GTFS Schedule and GTFS Realtime, and how do they differ in the purpose, encoding, and data structure?
11.3 What is the significance of data standards such as GTFS in improving real-time route recommendations and navigation services?
11.4 How do open data initiatives contribute to the optimization of urban mobility, particularly in public transport, traffic flows, and parking availability?
11.5 How can analyzing delays in public transportation impact urban mobility and passenger satisfaction?
11.6 What tools or libraries can be used to load and analyze GTFS Schedule data. How do they simplify the integration and analysis process?
11.7 What are the main tables created by `gtfs-via-postgres` in a PostGIS database and what do they represent?
11.8 Explain the purpose of the `arrivals_departures` and `connections` views created by `gtfs-via-postgres`.
11.9 What Python libraries are used to visualize individual transit lines, and how do they interact with a PostGIS database?
11.10 What is the role of the planning department in estimating vehicle speeds in public transportation?
11.11 What information can be derived from speed distributions across network segments?
11.12 What type of messages can be found in GTFS Realtime feeds?
11.13 What are protocol buffers (protobufs), and why are they used in GTFS Realtime?

11.14 How does the extract_vehicle_positions function filter out vehicle position messages from the feed?
11.15 What kind of information is stored in each vehicle_position entity in the GTFS Realtime feed?
11.16 What are some advantages of the Parquet file format?
11.17 How does DuckDB enable querying Parquet files?
11.18 What is the significance of analyzing delays as a continuous temporal function?
11.19 How can the delay function help in identifying operational issues in a transit system?

11.10 Exercises

Exercise 11.1. Find and download the GTFS Schedule data for your city or a city you are familiar with. Complete the following tasks:

a. Load the GTFS Schedule data into a PostGIS database.
b. Create a visualization of the city's transport network.
c. Calculate the total distance (in kilometers) that is scheduled to be traveled by all public transport routes.
d. Count the total number of trips scheduled for each route over the course of one month.
e. Find a map of the administrative regions in the city and calculate the total scheduled kilometers traveled within every region.

Exercise 11.2. Modify the script GTFS-Real-Time-Riga provided in the companion GitHub repository to collect real-time vehicle position data for two hours. Experiment with the following:

a. Save the response from each call to the Rīgas Satiksme GTFS Realtime endpoint into a separate GTFS Realtime file (in protobuf format).
b. Save the response from each call into a separate Parquet file.
c. Use the collect_vehicle_positions function to collect the entire two-hour dataset and store it in a single Parquet file.
d. Compare the total file sizes across all the tasks and analyze the results.
e. Experiment with various Parquet compression formats (such as Snappy and GZIP) and establish a comparison of them.

Exercise 11.3. A quantitative measure that captures regularity is the Excess Waiting Time (EWT), which is computed as the difference between the Actual Waiting Time (AWT) and the Scheduled Waiting Time (SWT). Compute the EWT for the public transport lines of Riga, focusing on the most frequent lines where headways are less than 10 minutes.

a. Identify relevant lines with frequent service. Use the GTFS Schedule data of Riga to identify the lines that operate with a planned headway of less than 10 minutes during peak hours. These are the lines where regularity (and not punctuality) is the primary concern.
b. Extract actual vehicle arrival times. Using the GTFS Realtime data collected from Riga, extract the actual arrival times of vehicles at each stop for the identified lines during a specific time period (e.g., a few hours of peak service).
c. Compute the observed headways. For each relevant line, compute the observed headways based on the actual vehicle arrivals at each stop.
d. Use the planned and observed headways to calculate the EWT for each line.
e. Visualize the EWT across the network for the most frequent lines on a map of the city. Use a tool like Folium or Plotly to create an interactive map, where the stops or lines are color-coded based on their EWT values.