Distributed System Project Overview

MSc in Computer Science

Athens University of Economics and Business

Authors:

Fotis Bistas, Stavros Armeniakos, Anna Chatzipapadopoulou

Emails: fot.bistas@aueb.gr, sta.armeniakos@aueb.gr, ann.chatzipapadopoul@aueb.gr

Supervising Professor:

Vana Kalogeraki

Date: February 5, 2025

Contents

Chapter 1

Overview

1.1 Project Description

This project, titled **Distributed System Project**, focuses on designing and deploying a distributed system for real-time data processing and querying.

1.2 Document Structure

The report is structured as follows:

- Preprocessing
- Processing
- Queries
- Environment Deployment
- Conclusion

Chapter 2

Preprocessing

2.1 Introduction

The preprocessing module is responsible for preparing video data for further analysis. Its main functionality includes segmenting large video files into smaller chunks and uploading these chunks to the ‘input-segments-container‘ in Azure Blob Storage. This process ensures that video data is efficiently managed and can be processed in parallel by other parts of the system.

2.2 Files in Preprocessing

The module is composed of the following files:

- `Dockerfile`: Defines the containerized environment for running the preprocessing logic.
- `function_app.py`: Contains the core logic for segmenting video files and uploading them to Azure Blob Storage.
- `host.json`: Configures the logging and function host settings for the Azure Function.
- `requirements.txt`: Lists the Python dependencies required for the application.
- `upload_large_file.py`: A utility script for splitting and uploading large video files to the Azure Function.

2.3 Detailed Explanation of Files

2.3.1 `Dockerfile`

The `Dockerfile` defines the container environment that runs the preprocessing application. Here's what each part does:

- **Base Image**: It starts with an official Azure Functions Python image to provide the runtime for the application.

- **Environment Variables**: Configures paths and logging settings required by Azure Functions.
- **System Dependencies**: Installs `ffmpeg`, a tool for video processing, which is critical for splitting video files.
- **Python Dependencies**: Copies the `requirements.txt` file and installs the Python libraries required for the application.
- **Code Deployment**: Copies the application code into the container, ensuring it is ready to run.

This container allows the application to run consistently across different environments, such as local machines and Azure.

2.3.2 `function_app.py`

This is the heart of the preprocessing module. It defines an Azure Function that accepts video files, segments them, and uploads the chunks to Azure Blob Storage. Let's break it down:

Key Steps in the Function

1. **File Upload Handling**: - The function accepts HTTP POST requests containing a video file. If the file is missing or not in MP4 format, the function returns an error.

2. **Temporary File Management**: - The uploaded file is saved to a temporary directory on the server. This ensures the original file isn't modified and temporary data can be cleaned up after processing.

3. **Video Segmentation**: - The `segment_video` function splits the video into smaller chunks (default: 2-minute chunks). This is done using `ffmpeg`, a command-line tool for handling video files. - For each chunk, the function calculates the start time and duration and generates an output file.

4. **Uploading to Azure Blob Storage**: - The function connects to Azure Blob Storage using a connection string. It ensures the target container ('input-segments-container') exists, creating it if necessary. - Each video chunk is uploaded to the container with a unique name.

5. **Cleanup**: - After processing, the temporary video file and its chunks are deleted to free up disk space.

6. **Error Handling**: - If any step fails, the function logs the error and returns a detailed error message to the user.

`segment_video` Function

The `segment_video` function is a helper utility that uses `ffmpeg` to split videos. Here's what happens:

- It determines the total duration of the video by running an `ffprobe` command.
- Based on the chunk length, it calculates the number of chunks needed.
- For each chunk, it generates an `ffmpeg` command to extract a specific portion of the video and saves the output to a file.

2.3.3 `host.json`

The `host.json` file configures the Azure Function environment. In this project:

- **Logging**: Application Insights logging is enabled to monitor the function's performance and errors.
- **Extension Bundle**: Specifies the version of Azure Function extensions to use, ensuring compatibility with the platform.

2.3.4 `requirements.txt`

This file lists the Python dependencies required for the preprocessing module. Key libraries include:

- `azure-functions`: Provides the runtime support for Azure Functions.
- `azurefunctions-extensions-bindings-blob`: Enables blob storage integration for handling video chunks.

2.3.5 `upload_large_file.py`

This script is a utility for testing and uploading large video files to the preprocessing function. Here's how it works:

- **Video Segmentation**: - It uses the same `segment_video` function as the Azure Function to split large video files into smaller chunks locally.
- **HTTP Upload**: - For each chunk, the script sends an HTTP POST request to the Azure Function, uploading the chunk along with metadata (e.g., chunk number and total chunks).
- **Logging**: - Each upload is logged, including success and failure messages, to help monitor the process.

This script is particularly useful for debugging and batch uploading videos during development.

2.4 Use Cases

The preprocessing module addresses the following challenges:

- **Handling Large Video Files**: By splitting videos into smaller chunks, it avoids memory and processing limitations and enables parallel processing in later stages.
- **Seamless Integration with Azure**: The module leverages Azure Blob Storage to store and manage video chunks efficiently.
- **Automated Cleanup**: Temporary files are automatically removed, ensuring the system remains clean and efficient.

Chapter 3

Processing

3.1 Introduction

The processing module manages object detection and tracking in video streams.

3.2 Files in Processing

- `Dockerfile`: Containerizes the processing module.
- `function_app.py`: Core logic for processing video data.
- `host.json`: Configuration for the Azure Function.
- `requirements.txt`: Lists the dependencies, including `numpy` and `opencv-python-headless`.
- `db.py`: Handles database interactions for storing results.
- `object_detection.py`: Implements object detection logic.
- `object_tracker.py`: Tracks objects across frames.
- `vehicle_utils.py`: Utility functions for vehicle-related computations.

3.3 Object Tracker

The `ObjectTracker` class is a core component of the processing module that handles real-time tracking of objects across video frames. It integrates object detection results, assigns unique IDs to detected objects, and tracks their movements over time. This section explains its functionality and features.

3.3.1 Key Features

- **Object Detection Integration:** The `ObjectTracker` integrates with the object detection pipeline, using bounding box information to identify objects of interest, such as cars and trucks.

- **Tracking Across Frames:** Objects are tracked across consecutive frames by comparing their positions. The tracker matches objects based on proximity and assigns new IDs to untracked objects.
- **Visualization:** The tracker visualizes bounding boxes, object IDs, and additional metadata (e.g., direction and type) on the video frames. This feature is particularly useful for debugging and monitoring.
- **Database Logging:** Inactive objects (those no longer detected) are logged to a PostgreSQL database for further analysis and real-time querying.

3.3.2 Core Functionalities

- **Initialization:** The tracker initializes with configurable parameters such as the minimum number of frames required to confirm an object and whether to visualize tracking results. It also sets up database logging, if enabled.
- **Object Matching:** Objects are matched between frames by comparing the Euclidean distances of their centers. Objects with distances below a predefined threshold are considered the same object.
- **Visualization:** Visual feedback is provided by drawing bounding boxes and adding labels to objects in the video. This feedback includes the object's ID, type (e.g., car or truck), and its direction of movement.
- **Position Detection:** The tracker identifies the center positions of objects in each frame using bounding box coordinates.
- **Database Logging:** When objects are no longer detected in the current frame, their tracking history is logged to the database. This enables the system to analyze object movement and behavior over time.

3.4 YOLO-Based Video Processing

This part of the processing module uses the YOLO (You Only Look Once) object detection model to identify and localize objects within a specific region of interest in video frames. The YOLO model processes the frames and detects objects such as cars and trucks, which are then used as input for tracking.

3.4.1 Key Features

- **Region of Interest (ROI):** The video frame is divided into a region of interest, which is defined as a rectangle extending from the left lane of the road to the right lane. This approach focuses on specific areas (e.g., roads) for object detection, reducing computational overhead and eliminating unnecessary detections.
- **YOLO Integration:** The YOLO model is configured with pre-trained weights and a configuration file for efficient object detection. The model detects objects like cars and trucks with high confidence.

- **Bounding Box Extraction:** For each detected object, YOLO provides bounding boxes that are adjusted to align with the ROI.
- **Video Output:** The processed video is saved with bounding boxes and labels for detected objects, providing a visual representation of the detections.

3.4.2 Core Functionalities

- **Input and Output Video:** The system reads an input video and outputs a processed video with detected objects annotated.
- **Object Detection Pipeline:** The YOLO model processes each frame to identify objects within the defined ROI. Objects like cars and trucks are filtered based on their detection confidence scores.
- **Visualization and Saving:** Detected objects are drawn on the frames, and the processed frames are saved as a new video for further analysis.

3.4.3 Visualizing the Transformation

The following images illustrate the processing workflow:

- The first image in Figure 3.1 shows the raw video frame received by the system.
- The second image in Figure 3.2 demonstrates the frame after applying the region of interest (ROI) and performing YOLO object detection.

Figure 3.1: Raw video frame as received by the system.

As illustrated in Figure 3.1, the raw video frame contains the complete scene captured by the camera. After applying the region of interest, as shown in Figure 3.2, the focus is narrowed to the relevant area (e.g., the road). YOLO object detection adds bounding boxes and labels to highlight detected vehicles, such as cars and trucks.

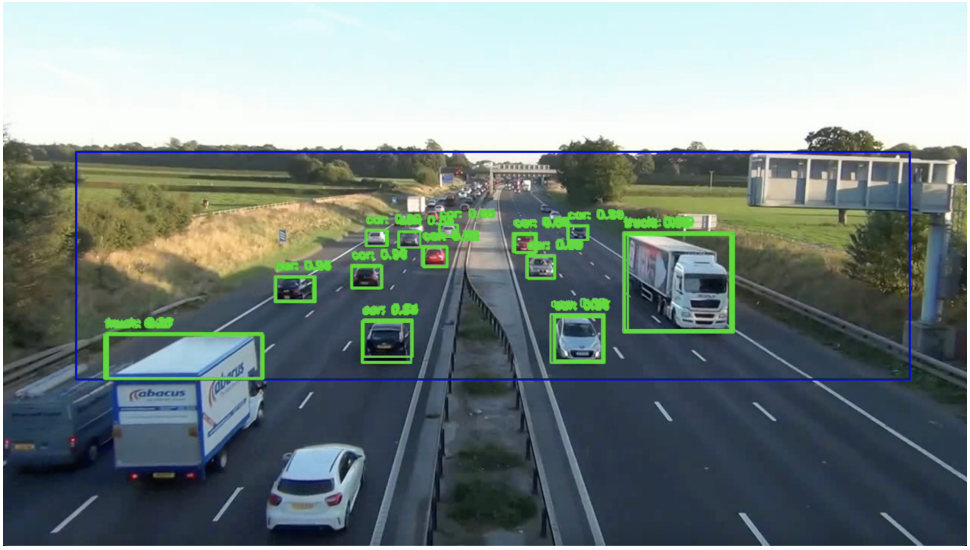


Figure 3.2: Video frame after applying ROI and YOLO object detection.

Chapter 4

Queries

4.1 Introduction

The querying module provides the tools to analyze the processed data and extract meaningful insights, such as vehicle counts, speed trends, and compliance with speed limits. By leveraging PySpark, a distributed computing framework, the module processes large volumes of data efficiently, taking advantage of PostgreSQL integration for database access.

4.2 Files in Queries

The querying module comprises the following scripts:

- `create_log_file.py`: Aggregates vehicle speed data over 5-minute intervals and writes a log file summarizing average speeds per stream.
- `vehicles_above_limit.py`: Identifies vehicles that exceed predefined speed limits and computes the total count of violations.
- `vehicles_per_stream_per_minutes.py`: Counts vehicles per stream over 5-minute windows to analyze traffic trends.
- `vehicles_per_stream.py`: Provides the total count of vehicles per stream direction for an overall view of traffic volume.
- `requirements.txt`: Lists the dependencies needed for the querying module, including PySpark and PostgreSQL drivers.

4.3 Use of PySpark

The querying module relies heavily on PySpark for its ability to process distributed data across large datasets efficiently. PySpark enables:

- Reading and writing data from and to PostgreSQL using the JDBC driver.
- Applying transformations and performing aggregations on time-based windows.
- Generating actionable insights by filtering, grouping, and analyzing the data.

4.4 Detailed Explanation of Scripts

4.4.1 `create_log_file.py`

This script computes the average speed of vehicles for each stream direction over 5-minute intervals and logs the results.

Step-by-Step Explanation

1. **Spark Initialization:** The script begins by initializing a SparkSession, configuring it to stop gracefully on shutdown, and setting up the necessary dependencies for PostgreSQL and Kafka.

2. **Database Connection:** Connection details for the PostgreSQL database are configured dynamically using environment variables for host, port, database name, user, and password.

3. **Loading Data:** Data is read from the PostgreSQL table `tracking_data` into a Spark DataFrame. This table contains vehicle data, such as speed, direction, and timestamps.

4. **Time-Based Aggregation:** The DataFrame is grouped into 5-minute time windows for each stream direction. The average speed within each window is calculated using PySpark's `groupBy` and `agg` functions.

5. **Ordinal Interval Indexing:** Each 5-minute window is assigned a continuous index for better readability in the output.

6. **Result Logging:** The aggregated results are written to a log file in a human-readable format. For example:

   (North, 1st 5min, 50.34 km/h)
   (South, 1st 5min, 60.12 km/h)

7. **Spark Session Termination:** The Spark session is stopped to release resources.

4.4.2 `vehicles_above_limit.py`

This script identifies vehicles exceeding speed limits and computes the total number of violations.

Step-by-Step Explanation

1. **Spark Initialization:** A SparkSession is initialized with the same configuration as in `create_log_file.py`.

2. **Speed Limit Definition:** Speed limits are defined as 90 km/h for cars and 80 km/h for trucks.

3. **Data Transformation:** A new column is added to the DataFrame, `exceeded_speed_limit`, which flags vehicles that exceed the speed limit based on their type (car or truck).

4. **Violation Counting:** The script filters rows where `exceeded_speed_limit` is 1 (violations) and counts these rows using PySpark's `count` function.

5. **Result Display:** The total number of speed violations is displayed, providing actionable data for traffic enforcement.

4.4.3 `vehicles_per_stream_per_minutes.py`

This script computes the number of vehicles passing through each stream direction in 5-minute intervals.

Step-by-Step Explanation

1. **Time Windowing:** The script groups vehicle data into 5-minute intervals using PySpark's `window` function. This allows for time-based aggregation.

2. **Vehicle Counting:** For each time window and stream direction, the total count of vehicles is calculated using `count`.

3. **Formatting Results:** The output includes the start and end times of each 5-minute window, the direction, and the vehicle count. The results are ordered for better readability.

4. **Use Case:** This script is valuable for traffic flow analysis, helping identify peak traffic times and directions.

4.4.4 `vehicles_per_stream.py`

This script calculates the total number of vehicles passing through each stream direction.

Step-by-Step Explanation

1. **Direction-Based Grouping:** Data is grouped by `direction`.

2. **Total Count:** For each direction, the total number of vehicles is counted using `count`.

3. **Result Display:** The total vehicle count per direction is displayed. This provides a high-level overview of traffic volume in each direction.

4. **Efficiency:** By avoiding time-based windows, this script is more efficient for cumulative traffic analysis.

4.4.5 `requirements.txt`

The `requirements.txt` file includes all dependencies required for running the querying module:

- `pyspark`: Provides distributed data processing capabilities.
- `org.apache.spark:spark-sql-kafka-0-10_2.12`: Enables integration with Kafka for streaming data.

- `org.postgresql:postgresql`: Provides the JDBC driver for PostgreSQL connectivity.

Chapter 5

Questions and Answers

5.1 Introduction

This chapter provides answers to the questions related to the processed vehicle data. Each question is addressed with a corresponding explanation and an image showing the results obtained from the system.

5.2 Questions

Q1: The speed at which each vehicle is moving.

The system computes and logs the speed of each vehicle using the processed data. The speed is calculated based on the timestamps and positions recorded for each vehicle in the video.

Q2: The number of vehicles per stream.

The total number of vehicles in each stream (inbound and outbound) is calculated by grouping data by stream direction.

Q3: The number of vehicles that have exceeded the speed limit for the entire video.

The system calculates the total number of vehicles that exceeded the speed limits (90 km/h for cars and 80 km/h for trucks) during the 32 minutes of video.

Q4: Real-time alert for vehicles moving at more than 130 km/h.

For vehicles moving faster than 130 km/h, real-time alerts are generated and printed in the log of the real-time alerts function.

Q5: The number of vehicles per stream and every 5 minutes.

The number of vehicles is calculated for each stream and aggregated over 5-minute intervals. This provides insights into traffic flow over time.

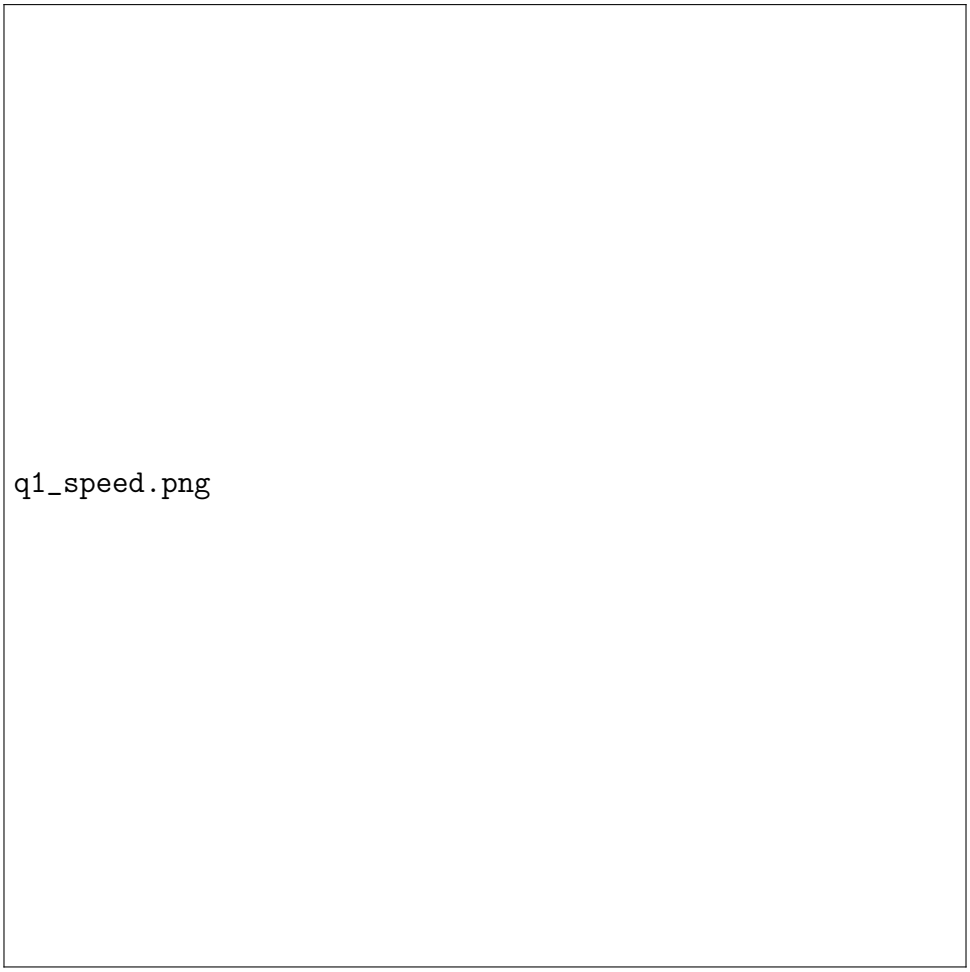


Figure 5.1: Speed of each vehicle recorded during the video processing.

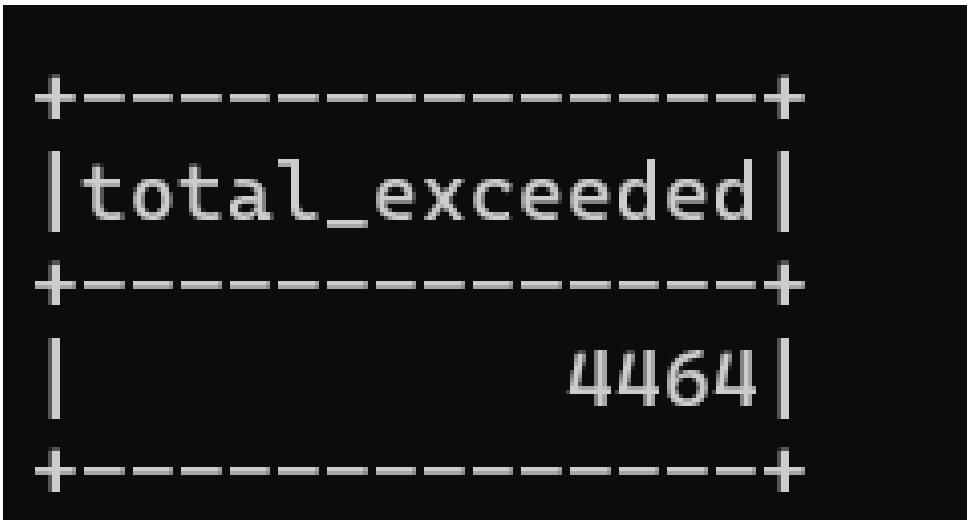
Q6: The average speed of the traffic per stream and every 5 minutes.

The average speed of traffic is computed for each stream in 5-minute intervals. The results are logged in a user-friendly format, such as:

(inbound, 1st 5min, 60 km/h), (outbound, 2nd 5min, 50 km/h), ...

direction	vehicle_count
outbound	3758
inbound	4525

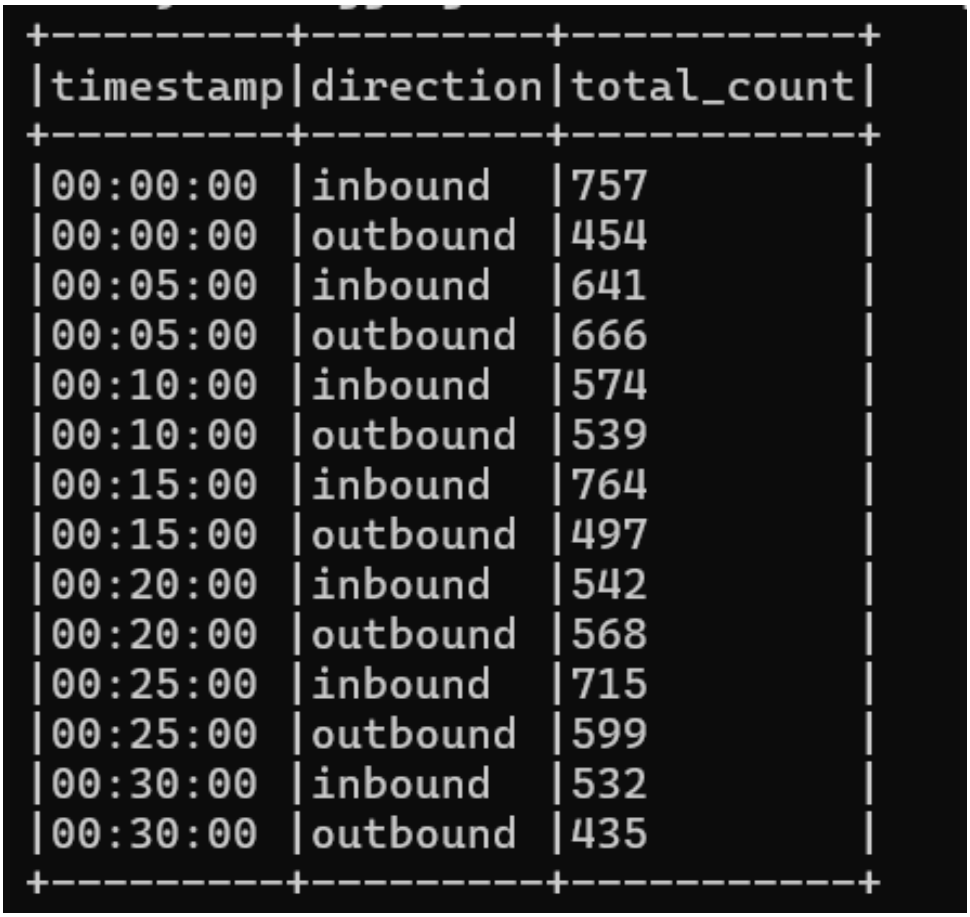
Figure 5.2: Number of vehicles per stream (inbound and outbound).



total_exceeded
4464

Figure 5.3: Number of vehicles that exceeded the speed limit during the entire video.

Figure 5.4: Real-time alerts generated for vehicles exceeding 130 km/h.



timestamp	direction	total_count
00:00:00	inbound	757
00:00:00	outbound	454
00:05:00	inbound	641
00:05:00	outbound	666
00:10:00	inbound	574
00:10:00	outbound	539
00:15:00	inbound	764
00:15:00	outbound	497
00:20:00	inbound	542
00:20:00	outbound	568
00:25:00	inbound	715
00:25:00	outbound	599
00:30:00	inbound	532
00:30:00	outbound	435

Figure 5.5: Number of vehicles per stream in 5-minute intervals.

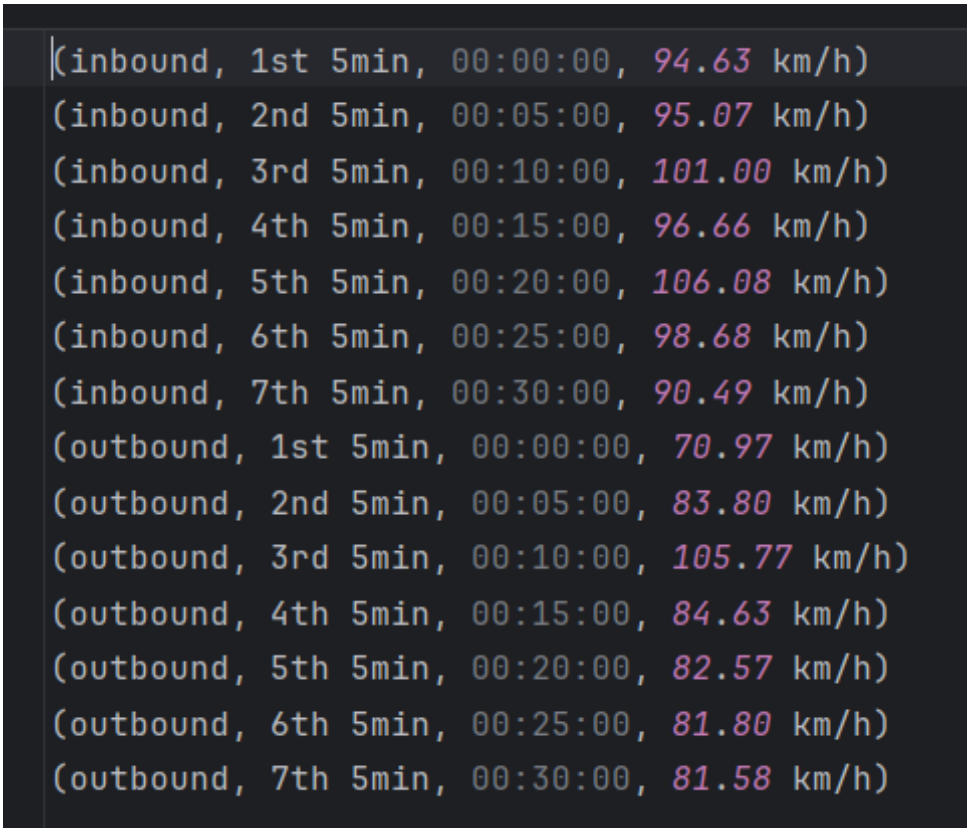


Figure 5.6: Average speed of traffic per stream in 5-minute intervals.