Processing and Analysis of Belgian Car Data

Stef L.

Data Engineering Capstone Project

Table of Contents

Preface

This project aims to analyze the interaction between various municipal-level characteristics and the degree of car ownership in Belgium. To achieve this, we have developed a comprehensive workflow that facilitates the extraction, transformation, and analysis of relevant data.

The workflow begins with the extraction of data from online sources. This data is subsequently ingested into our Data Lake.
Next, we perform the transformation and loading of this data. Using batch-processing and transformations, after which we load the data into our Data Warehouse.
Finally, we connect an interactive dashboard to the Data Warehouse. This allows us to generate insightful reports and conduct detailed analyses of the data.

The primary research question guiding this project is: How do different municipal-level characteristics interact with the degree of car ownership? Key elements of our analysis include population density, province, city area size, the count of sub-municipalities, and household types.

Our data sources for this analysis include car ownership data from Statbel (statbel.fgov.be) and city demographic information from Wikipedia tables.

Tools Used

Overview

Languages:	Python, SQL
IDE:	Jupyter Notebook, Visual Studio Code
Workflow:	Mage-ai
Transformations:	PySpark, dbt Cloud
Infrastructure:	Docker, Terraform
Storage:	Google Cloud Storage (Data Lake), Google BigQuery (Data Warehouse)
Cloud environment:	Google Cloud
Other tools:	draw.io, Bash, MS Excel, MS PowerBi

Rationale for Tool Usage

Python and SQL are fundamental languages for working with data, offering straightforward methods to transform and interact with various data types, including unstructured data in the case of Python.

Jupyter Notebook serves as an excellent GUI for testing Python code with real-time feedback, while Visual Studio Code provides a streamlined coding environment with GIT integration and extensive support for add-ons.

Mage-ai is a relatively new open-source tool for workflow orchestration. I chose it over Apache Airflow due to its simplicity, making it ideal for smaller teams working on simpler data pipelines.

For transformations, I used PySpark and dbt Cloud. Dbt Cloud is a user-friendly platform for modeling data with SQL and scheduling SQL workflows, while PySpark is widely used in Data Engineering for batch processing at scale. Spark also supports user-defined functions, making it suitable for all our structured and unstructured data needs.

Terraform offers a simple way to deploy cloud infrastructure without manual configuration through a GUI, and Docker allows us to containerize the pipeline, isolating it from our local environment. This is also the recommended setup for Mage-ai pipelines.

Our cloud platform is Google Cloud, which meets all our infrastructure needs. It provides Google Cloud Storage as a staging area for our Data Lake and Google BigQuery to manage our Data Warehouse, containing structured, analysis-ready tables.

Additionally, I utilized a few more tools: draw.io for workflow design, Bash for terminal-based tasks, MS Excel for user-friendly data exploration, and MS Power BI for data visualization and reporting.

Technical Steps to Execute Project

1. Identifying the Sources

Tools: Bash, Excel

- First, we gather data from online sources:
  - We collect municipality-level data by scraping the relevant table from Wikipedia: https://nl.wikipedia.org/wiki/Tabel_van_Belgische_gemeenten
  - We download data on car ownership per municipality and household type from Statbel: https://statbel.fgov.be/sites/default/files/files/opendata/Aantal%20wagens%20volgens%20huishoudtype%20per%20gemeente/TF_CAR_HHTYPE_MUNTY.xlsx
- Next, we import and explore the data using Bash for data gathering and simple row counting and Excel for a closer inspection of the datasets.

2. Designing the Workflow

Tools: draw.io

- The workflow consists of the following steps:
  - **Extraction:** Data from online sources (Wikipedia and statbel.fgov.be) is extracted and loaded into a Data Lake, stored in Google Cloud Storage buckets.
  - **Transformation & Ingestion:** The data is retrieved from the Data Lake, batch processed using Spark, and then loaded into the Data Warehouse (Google BigQuery).
  - **Planning:** Both the Extraction and Transformation & Load steps are scheduled and managed by Mage, which acts as the workflow orchestrator.
  - **Additional Transformations:** Once in the Data Warehouse, the data is further transformed and prepared for analysis using dbt Cloud.
  - **Analysis:** Finally, an MS Power BI dashboard is connected to the Data Warehouse to generate insightful reports and perform data analysis.
- See *workflow_design.png* on the next page for a visual summary of the process.

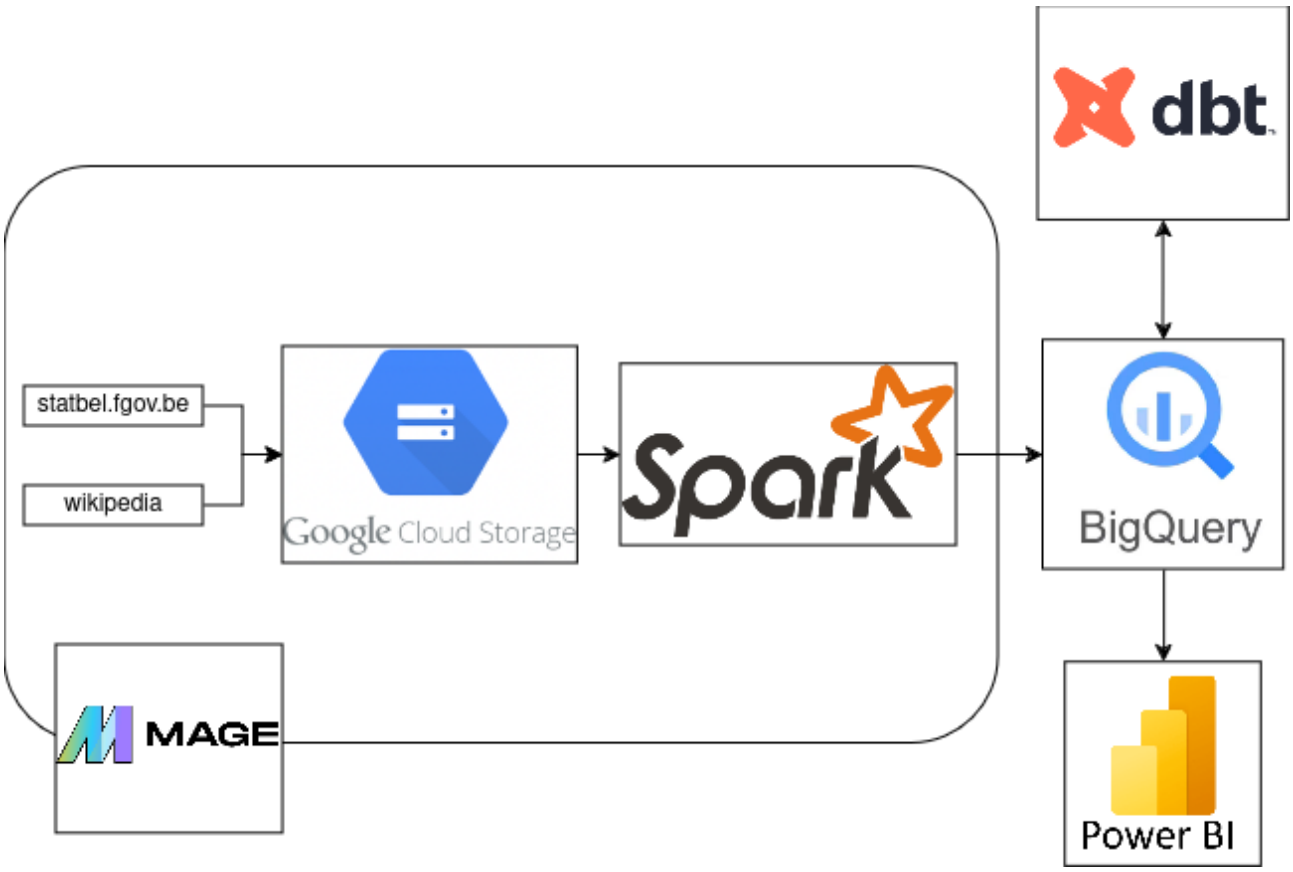


Figure 1: workflow_design.png

3. Managing the Data Workflow, Data Extraction, and Data Ingestion into Data Lake

Tools: Python, Docker, Mage-ai, Google Cloud Storage

- First, we configure the necessary buckets and service account for use with Mage scripts:
  - We begin by creating a bucket in Google Cloud Storage to store our data.
  - Next, we create a service account in the IAM Admin console and export the credentials as a JSON-file, for later use.
- Then, we set up the Mage AI instance locally:
  - We use a Mage-ai template provided by mage.
  - After that, we customize the *Dockerfile*, *.env* file, and *requirements.txt* based on our specific requirements.
  - The setup is completed by running the following Bash commands: *docker compose build && docker compose up.*
- Once the setup is done, we navigate to the user interface by going to http://localhost:6789.
- Next, we generate a pipeline.
- Finally, we configure the *IO_config.yaml* file to work with the service account and the storage buckets:
  - We define the location of the Google service account JSON file.
  - Additionally, we remove any unnecessary Google Cloud configuration lines to ensure the file is properly streamlined.

- Now, we configure the blocks for the data pipeline:
  - **Data Loader** (Python - API): This block is responsible for loading data from Wikipedia (see: *wikipedia_api.py*).
    - The data loader performs the following actions:
      - **Imports Libraries:** Imports the required libraries, including pandas, Wikipedia, requests, BeautifulSoup, and functions from Mage for data loading and testing.
      - Sets Language: Configures Wikipedia to fetch Dutch pages using wp.set_lang("nl").
      - **Fetches Wikipedia Page:** Retrieves the HTML content from the Dutch Wikipedia page titled "Tabel_van_Belgische_gemeenten" using the Wikipedia API.
      - **Parses HTML:** Uses BeautifulSoup to parse the HTML for easier manipulation and data extraction.
      - **Converts Belgian Number Formats:** Defines a function (convert_belgian_number_format) to handle Belgian-style numeric formats (e.g., 1.234,56 to 1234.56).
      - **Cleans Numeric Formats:** Applies the conversion function to all cells in the HTML tables, replacing string values with the cleaned format.
      - **Reads Cleaned Table into DataFrame:** Converts the cleaned HTML back into a string, then loads it into a pandas DataFrame using pd.read_html().
      - **Handles Edge Cases:** Ensures the correct table is selected if multiple tables are present, defaulting to the first one if needed.
      - **Converts to CSV:** Converts the table into CSV format and returns it as a pandas DataFrame.

  - **Data Transformer** (Python - Template): This block transforms the data (see: *wikipedia_transform.py*).
    - The transformer performs the following actions:
      - **Imports Libraries:** Imports necessary libraries such as pandas and io, along with conditional imports for transformer and test decorators from Mage if not already present.
      - **Drops Unnecessary Columns:** Removes irrelevant columns like Index, Index.1, Index.2, Index.3, and Index.4 to clean up the data.
      - **Fixes the Header:** Identifies rows where the "Rang" column contains 'Rang' to reset the header and update column names accordingly.
      - **Converts to CSV**: Converts the data into a CSV string, and reads it back into a DataFrame.

  - **Data Exporter** (Python - Google Cloud Storage): This block exports the transformed data to Google Cloud Storage (see: *wikipedia_to_gcs.py*).
    - The exporter performs the following actions:
      - **Imports Libraries:** Imports necessary modules for handling paths (get_repo_path, path), loading configuration (ConfigFileLoader), interacting with Google Cloud Storage (GoogleCloudStorage), and managing DataFrames (pandas.DataFrame).
      - **Configures Paths and Settings:** Defines the path to the configuration file (*io_config.yaml*) using get_repo_path() and specifies the configuration profile as 'default.'

        - **Specifies GCS Bucket and Object Key:** Identifies the Google Cloud Storage bucket (carproject_datalake) and file path (*raw/wp_municipality_data.csv*) where the data will be stored.
        - **Exports Data:** Uses GoogleCloudStorage.with_config() to load the configuration and export the DataFrame to the designated Google Cloud Storage bucket.

    - **Data Loader** (Python - API): Another data loader fetches data from the Statbel (see: *statbelcar_api.py*).
        - This data loader performs the following actions:
            - **Imports Libraries:** Imports libraries like pandas for data handling, requests for API calls, and conditionally imports Mage's data_loader and test decorators if not already present.
            - **Fetches Data:** Downloads an Excel file from Statbel (URL: https://statbel.fgov.be/...), containing data on car ownership by household type for each municipality.
            - **Reads Excel Data:** Loads the data into a pandas DataFrame from the "données" sheet of the Excel file using pandas.read_excel().

    - **Data Exporter** (Python - Google Cloud Storage): This block exports Statbel data to Google Cloud Storage (see: *statbelcar_to_gcs.py*).
        - The exporter performs the following actions:
            - **Imports Libraries:** Imports modules for file paths (get_repo_path, path), configuration loading (ConfigFileLoader), interaction with Google Cloud Storage (GoogleCloudStorage), and DataFrame management (pandas.DataFrame).
            - **Configures Paths and Settings:** Defines the path to the configuration file (*io_config.yaml*) and sets the configuration profile to 'default.'
            - **Specifies GCS Bucket and Object Key:** Identifies the Google Cloud Storage bucket (carproject_datalake) and file path (*raw/statbelcar_data.csv*) for storage.
            - **Exports Data:** Uses GoogleCloudStorage.with_config() to export the DataFrame to the specified Google Cloud Storage location.
- Finally, we run all blocks. After running, the data is extracted from the sources, cleaned, and the resulting CSV files are stored in Google Cloud Storage buckets (our Data Lake).

4. Transforming Data Using Batch Processing and Loading into Data Warehouse

Tools: Python, Spark, Google BigQuery, Google Cloud Storage, Mage-ai

- First, we design the batch process in a Jupyter Notebook:
  - This is documented in the file *batch_processing.ipynb*, where we outline and test the batch processing workflow.
- Next, we convert the notebook to a Python script:
  - Using a Bash command: *jupyter nbconvert --to=script batch_processing.ipynb*, we convert the Jupyter Notebook into a Python script (refer to *batch_processing.py*).
- Then, we adapt the Python script to connect with Google Cloud Storage and write to BigQuery:
  - The necessary modifications are made in the *batch_processing.py* script
- After that, we set up a Mage instance with PySpark installed:
  - We start by downloading the required PySpark *Dockerfile* from Mage's GitHub repository at https://github.com/Mage-ai/Mage-ai/blob/master/integrations/spark/Dockerfile.
  - We build the Docker image using Bash: docker build -t mage_spark .
  - Once the image is built, we run the container using Bash:
    docker run -it --name mage_spark -e SPARK_MASTER_HOST='local' -p 6789:6789 -v $(pwd):/home/src mage_spark /app/run_app.sh mage start spark_project
- We specify the required metadata for the Spark session in the project's *metadata.yaml* file.
- We then integrate the batch processing script into Mage as a custom block:
  - The final version of this block performs the following actions:
    - **Imports Libraries:** Imports the required modules such as PySpark for distributed data processing, pandas for data handling, and PySpark functions and types for transformations.
    - **Conditionally Imports Decorators:** Checks for the custom and test decorators from Mage, importing them if they are not already present.
    - **Initialize Spark Session:** A new Spark session is created with the necessary configurations to use Google Cloud Storage (GCS) connectors and credentials for accessing GCS and BigQuery.
    - **Read Input Data:** The script reads two CSV files (one for car data and one for municipality data) from GCS into Spark DataFrames.
    - **Define Schemas:** Specific schemas for the car and municipality data are defined and applied to ensure the correct structure.
    - **Clean and Transform Data:**
      - Columns with unwanted characters (e.g., "*") are cleaned using regexp_replace.
      - New calculated columns are added, such as pop_perc_increase_2024v2000 to calculate the population percentage increase.
      - Translations are applied to household types (e.g., French to English), and new columns indicating car ownership status (has_car or no_car) are added using user-defined functions (UDFs).
    - **Write Data:** The transformed data is written back to GCS in Parquet format and also exported to BigQuery tables.
    - **Configure GCS and BigQuery:** The script ensures proper authentication and project configuration for seamless data exchange with GCS and BigQuery.

- Finally, we combine the data ingestion pipeline with the Spark process:
  - The earlier data ingestion pipeline in Mage is integrated with the Mage setup that holds the Spark process, forming a unified pipeline. The code is stored in the *mage_load_and_spark* folder.
  - For a visual summary of this combined pipeline, refer to the diagram in *mage_pipeline.png*.

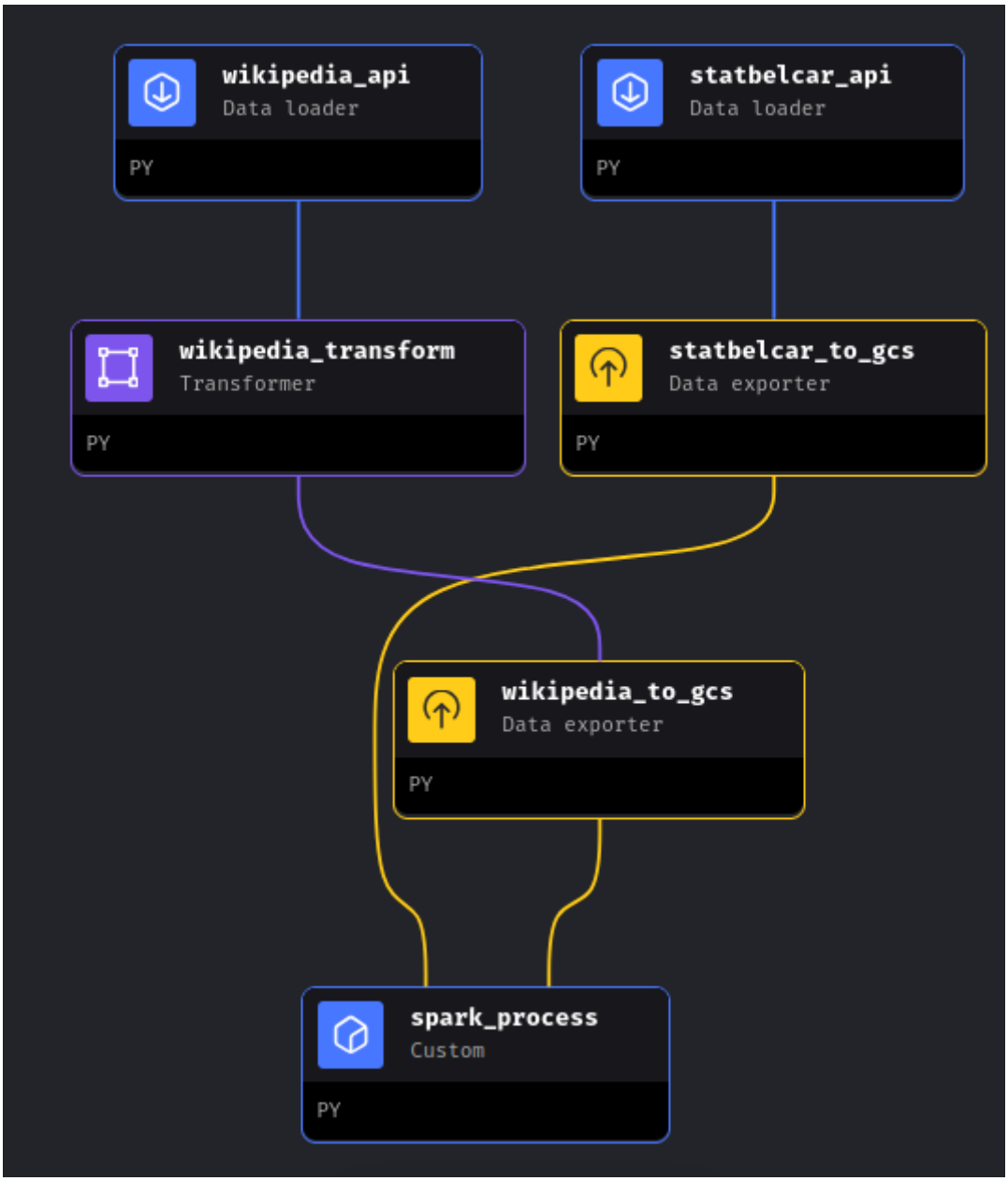


Figure 2: mage_pipeline.png

5. Deploying the Unified Mage Pipeline to the Cloud

Tools: Terraform, Google Cloud, GIT, Mage-ai, Docker

- We push our custom-made *Dockerfile* for Mage to Google Artifact Registry using the GCloud CLI:
  - First, we navigate to the directory that contains the Dockerfile for the Mage pipeline.
  - We run the following Bash command to authenticate your Google Cloud account: gcloud auth login
  - We set the project to the appropriate Google Cloud project using the Bash command: *gcloud config set project carprojectbelgium*
  - We create a new Docker repository in Google Artifact Registry using the Bash:
    *gcloud artifacts repositories create my-new-repo *
    *--repository-format=docker *
    *--location=us-central1 *
    --description="My new Docker repository"
  - Authenticate Docker with the Google Artifact Registry using the access token:
    cloud auth print-access-token | docker login -u oauth2accesstoken --password-stdin https://us-central1-docker.pkg.dev
  - Build the Docker image with the following command:
    docker build -t mageai-customsparkfile:latest .
  - Tag the Docker image for the new repository:
    docker tag mageai-customsparkfile:latest us-central1-docker.pkg.dev/carprojectbelgium/my-new-repo/mageai-customsparkfile:tag1
  - Finally, we push the tagged image to the Google Artifact Registry with bash:
    docker push us-central1-docker.pkg.dev/carprojectbelgium/my-new-repo/mageai-customsparkfile:tag1
- We create Terraform files to deploy our virtual machine hosting the Mage pipeline:
  - We begin by enabling the Service Usage API on Google Cloud Platform (GCP).
  - Authenticate your application defaults by running:
    gcloud auth application-default login
  - Navigate to the directory that contains the Terraform files, specifically the *terraform_deploy* folder.
  - The Terraform main file will include the following setup:
    - **Configure Google Cloud Provider:** This section configures the Google Cloud provider using credentials from a local JSON file named mage_planner_key.json. It specifies the GCP project (carprojectbelgium) and the region (us-central1) where the resources will be created.
    - **Instance Configuration:** This part provisions a Compute Engine instance named mageai-instance in the us-central1-a zone, utilizing an e2-medium machine type. The instance is initialized with a Debian 11 boot disk.
    - **Service Account:** A Google service account (mage-planner@carprojectbelgium.iam.gserviceaccount.com) with Cloud Platform access is attached to the VM to facilitate resource management.
    - **Network and Tags:** The instance utilizes the default network with an external access configuration and is tagged as http-server for firewall rules.
    - **Docker Installation:** A startup script is included to install Docker and configure it to start on boot.

    - **GCloud Authentication:** The script configures Docker to pull images from Google Artifact Registry using GCloud.
    - **Run Docker Container:** A startup script is configured where the Docker container (mageai-customsparkfile:tag1) is executed from Google Artifact Registry (us-central1-docker.pkg.dev/carprojectbelgium/my-new-repo). The container is started with specific environment variables and port mappings (port 6789) to initiate the Mage-ai Spark_project using the command /app/run_app.sh mage start spark_project.
    - **Allow HTTP Traffic:** A firewall rule named allow-http is created to permit TCP traffic on port 6789.
    - **Security Settings:** This firewall rule is configured to only allow traffic from a specific source IP (MY_PUBLIC_IPV4/32), ensuring controlled access. It specifically targets instances with the http-server tag applied to the mageai-instance.
- We initialize Terraform with the following command:
  terraform init
- We apply the Terraform configuration using:
  terraform apply
- We add the service account JSON using Bash, by establishing an SSH connection to the new Virtual Machine.
- We set up Git synchronization with the mage_load_and_spark folder in my GitHub repository, following the Mage-ai documentation.
- We utilize Mage-ai to plan jobs as needed and run them. After running the job, the data is processed and stored in our Data Warehouse (Google BigQuery)
- After completing the jobs, we destroy the Virtual Machine to clean up resources by executing: *terraform destroy*

6. Designing the Data Warehouse and Executing Additional Transformation Jobs

Tools: SQL, DBT, Google BigQuery, GIT

- First, we define our grain and design fact and dimension tables using the Kimball model. Refer to *table_design.svg* for a visual representation of our table design
- Our grain is defined at the level of a unique combination of household_type_en and city_id. Each record in the table corresponds to a distinct pairing of these two attributes, meaning that for every household_type in a given municipality, there is a single record representing that combination. Each instance will have a surrogate key generated through our dbt model.

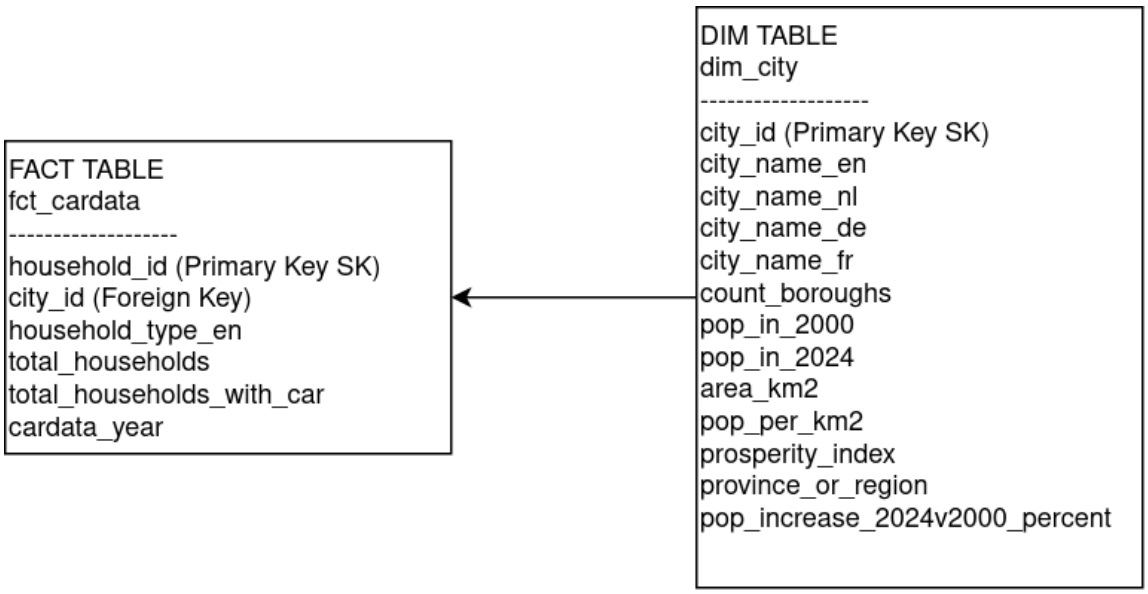


Figure 3: table_design.svg

- Next, we set up the dbt Cloud project to manage our data transformation workflows.
- Then, we create a subdirectory and initialize the dbt project within that subdirectory.
- After that, we configure the *dbt_project.yml* file to define project settings and specifications.
- Subsequently, we develop models and source files, refer to the dbt folder for code
  - The file dbt/models/staging/**src_carprojectbelgium.yml** includes the following:
    - **Source Definition:** The source name is carprojectbelgium, which refers to a GCP BigQuery dataset named belgium_cardata.
    - **Tables:** Two tables from the belgium_cardata dataset are specified:
      - **spark_cardata2017-2023:** This table contains car ownership data in Belgium from 2017 to 2023, processed using Spark.
      - **spark_municipalitydata:** This table includes data related to towns and cities in Belgium, also processed using Spark.
  - The file dbt/models/staging/**stg_carprojectbelgium__spark_cardata2017-2023.sql** consists of:
    - **Data Selection and Transformation:** The main SELECT statement retrieves specific columns from the carsource CTE:
      - **Trimmed City Names:** For each language (French, Dutch, English, and German), the process removes any text within parentheses and trims leading and trailing whitespace
      - Other attributes of the car ownership data
  - The file dbt/models/staging/**stg_carprojectbelgium__spark_municipalitydata.sql** consists of:
    - **Data Selection and Transformation:** The main SELECT statement retrieves and transforms several fields from the citysource CTE:
      - **Standardized Municipality Names:** A CASE statement is used to standardize the names of certain municipalities for consistency.
      - **area_km2:** Area of the muncipality, rounded to 2
      - **pop_per_km2:** Population density casted as an integer
      - Other demographic and geographic attributes

- The file dbt/models/intermediate/**dim_city.sql** consists of:
  - **Materialization:** The model is configured to be materialized as a table, meaning the results will be stored physically in the database.
  - **Data Selection and Transformation:** The main SELECT statement retrieves various fields, including:
    - **city_id:** A unique identifier for each city, generated using ROW_NUMBER() based on province and city name.
    - **city_name_nl, city_name_en, city_name_fr, city_name_de:** The municipality names in Dutch, English, French, and German, respectively, gathered through a LEFT JOIN with car data.
    - Other demographic and geographic attributes
    - Columns are renamed for clarity

- The file dbt/models/intermediate/**fct_cardata.sql** includes the following components:
  - **Materialization:** This model is configured to be materialized as a table, meaning the results will be physically stored in the database.
  - **Pivoting CTE:** This CTE aggregates data from car data by calculating:
    - **cardata_year, TX_MUNTY_DESCR_FR (city name), household_type_en**
    - **total_households_with_car:** The sum of households that own at least one car, grouped by year, municipality name in French, and household type.
    - **total_households:** The total sum of households for each group.
    - The results are grouped by year, city name and household type, providing a yearly breakdown of car ownership statistics.
  - **Data Selection and Transformation:** The main SELECT statement retrieves the following fields:
    - **household_id:** A unique identifier for each household, generated using ROW_NUMBER() based on year, city ID, and household type.
    - **city_id:** The identifier for the city, matched using the city name in the LEFT JOIN
    - **household_type_en:** The type of household in English.
    - **total_households:** The total number of households for each category.
    - **total_households_with_car:** The total number of households with cars.
    - **household_hascar_perc_of_total:** The percentage of households with cars, calculated by dividing total_households_with_car by total_households and rounding to two decimal places.
    - **cardata_year:** The year of the car data.
    - Columns are renamed for clarity

- The file dbt/models/mart/**mart_latestyear_by_city.sql** consists of the following elements:
  - **Materialization:** This model is configured to be materialized as a table, meaning the results will be stored in the database.
  - **Aggregation CTE:** This CTE aggregates car data grouped by city for the most recent year, calculating:
    - **year_of_cardata**
    - **total_households:** The total sum of households for each city.
    - **total_households_with_car:** The total sum of households that own at least one car.
    - **percentage_car_having_households_of_total_households:** The percentage of households with cars, rounded to two decimal places calculated and rounded up to two decimal places.
  - **Data Selection and Transformation:** The main SELECT statement retrieves the following fields:
    - **city_name_in_dutch:** The name of the city in Dutch.
    - **city_id:** The unique identifier for the city.
    - **year_of_cardata:** The most recent year of car data.
    - **total_households and total_households_with_car:** The counts of overall households and those owning cars.
    - **percentage_car_having_households_of_total_households:** The calculated percentage of households with cars.
    - Additional demographic information from the city table,
    - **Join:** A LEFT JOIN is performed between the city and car_grouped CTEs using city_id to combine city demographics with car ownership statistics
    - Columns are renamed for clarity

- The file dbt/models/mart/**mart_latestyear_by_householdtype.sql** includes the following components:
  - **Materialization:** This model is configured to be materialized as a table, meaning the results will be saved in the database.
  - **Data Selection and Transformation:** The query retrieves the following information:
    - **year_of_data, household_type_in_english,**
    - **Sum of total_households:** The total number of households for each household type.
    - **Sum of total_households_with_car:** The number of households that own at least one car for each household type.
    - **percentage_car_having_households_of_total_households:** Column calculating percentage of households with cars, rounded to two decimal places.
    - **Filtering and Grouping:** The data is filtered to include only records from the most recent year of data, and the results are grouped by household_type_en to provide insights specific to each household type.
    - Columns are renamed for clarity

- Next, we set up the production environment in dbt to ensure a consistent data workflow.
- After that, we configure jobs and triggers using the production environment and dataset to automate data processing tasks, after which we run the job.
- Refer to *dbt_pipeline.png* on the next page, for a visualization of the dbt job execution.

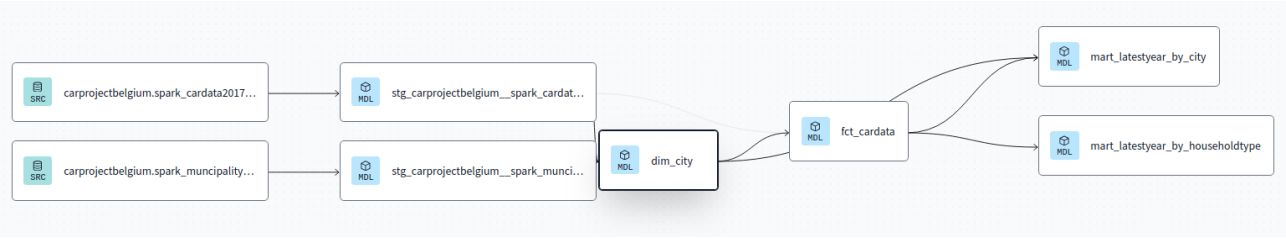


Figure 4: dbt_pipeline.png

- Finally, we optimize the Data Warehouse tables for performance. To enhance storage efficiency within our Data Warehouse, we will implement partitioning and clustering strategies.
  - **Partitioning** divides a table into smaller segments based on a specified column, such as a date or integer range. This reduces the amount of data scanned by queries, enhancing performance and lowering costs.
  - **Clustering** sorts data within partitions or the entire table based on specific column values. It improves query performance for filters or joins, minimizing scanned data by grouping related rows together.
  - The **ideal use case for Partitioning** is time-based data; in our case, we would partition the fct_cardata table by the cardata_year column.
  - **Clustering works well** on columns frequently used for aggregation, like city_id and household_type_en.
  - In practice we will optimise our fact table using the following query
    CREATE TABLE dwh_belgium_cardata.fct_cardata_partitioned
    PARTITION BY DATE(cardata_year)
    CLUSTER BY city_id, household_type_en AS
    *SELECT * FROM dwh_belgium_cardata.fct_cardata;*

7. Data Visualization and Reporting

Tools: MS PowerBi

- First, we will connect the BigQuery data to Power BI.
- Next, we design the dashboard. (Refer to *dashboard_cardata.pbix)*
  - The dashboard features three panels:
    - The first panel will include bar charts that display car ownership levels by province and by household type.
    - The second panel will consist of scatter plots that illustrate the relationship between car ownership and municipality size, as well as between car ownership and municipality population density.
    - The third panel will present a line graph showing the correlation between car ownership and the prosperity index of the municipality.
- After designing the dashboard, we export it as a PDF file for easier distribution. (Refer to *dashboard_cardata_print.pdf*, attached in the last three pages of this paper.)

Analysis

Following the deployment of our data pipeline and the creation of our interactive reports, we can draw the following conclusions regarding car ownership patterns in relation to demographic and geographic data at the city level.

The analysis of car ownership across various municipalities reveals several significant patterns. As shown in the bar graphs, most provinces exhibit a relatively similar level of car ownership, with the notable exception of the Brussels region, which has a markedly lower degree of car ownership. This discrepancy can be attributed to the region's densely populated urban landscape, where the availability of public transport and limited space likely reduce the need for private vehicles.

Furthermore, the second bar graph highlights that car ownership is much higher among couples, both with and without children, compared to non-couples. In contrast, one-person households are the least likely to own a car, and even single-parent households exhibit significantly lower car ownership rates compared to couples.

The scatter plot analysis adds another dimension, demonstrating that larger towns and cities, in terms of area, tend to have higher rates of car ownership, whereas higher population density correlates with lower car ownership levels.

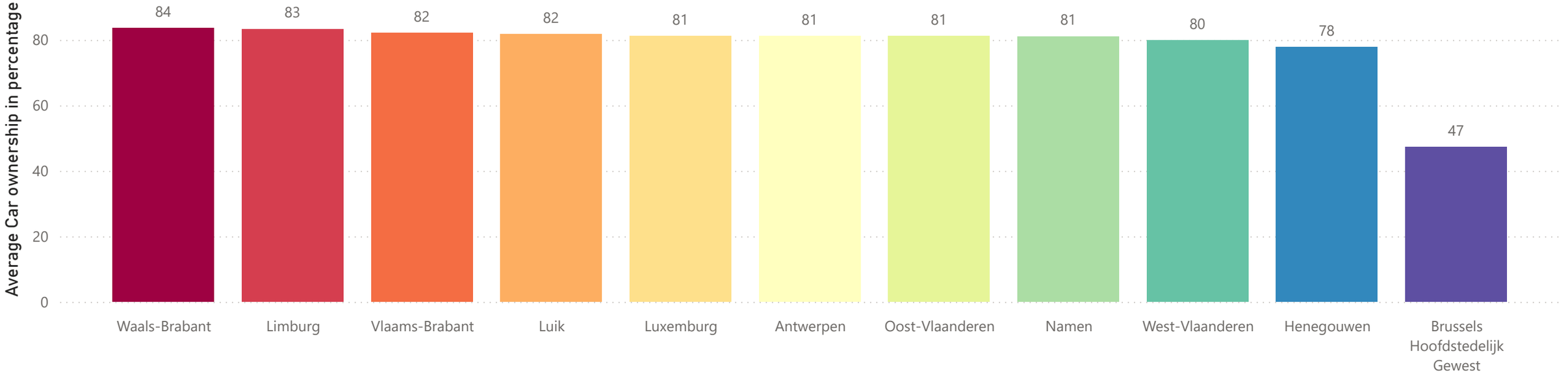
Lastly, the final panel suggests a socioeconomic factor in car ownership trends, showing that regions with lower prosperity indices also tend to have reduced car ownership rates. The combination of factors—urban density, household composition, town size, and economic prosperity—paints a comprehensive picture of the variables influencing car ownership across different regions.

Conclusion

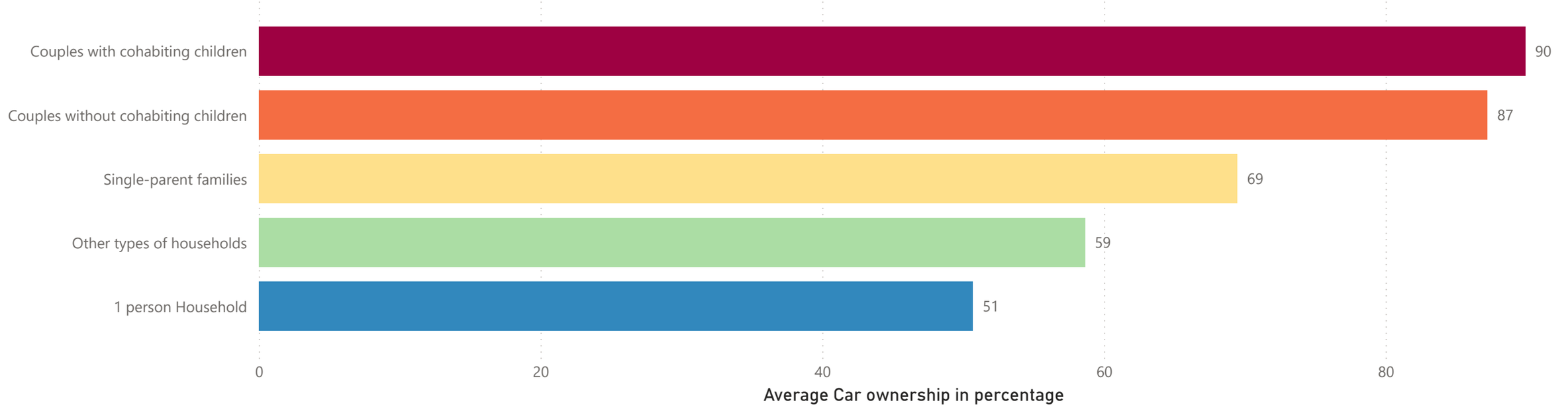
This project explored how various municipal characteristics relate to car ownership levels in Belgium, offering useful insights into regional transportation patterns. Using a data pipeline and a range of widely-used data processing tools, we analyzed the impact of factors like population density, household composition, and city size on car ownership. The findings show that car ownership tends to be lower in densely populated areas like Brussels and higher among couples, particularly in larger towns.

These tools are commonly applied in data engineering projects due to their reliability and scalability, making them essential for handling large datasets like the ones used in this analysis. Overall, the results provide a helpful overview of how different factors influence car ownership, which could inform future studies.

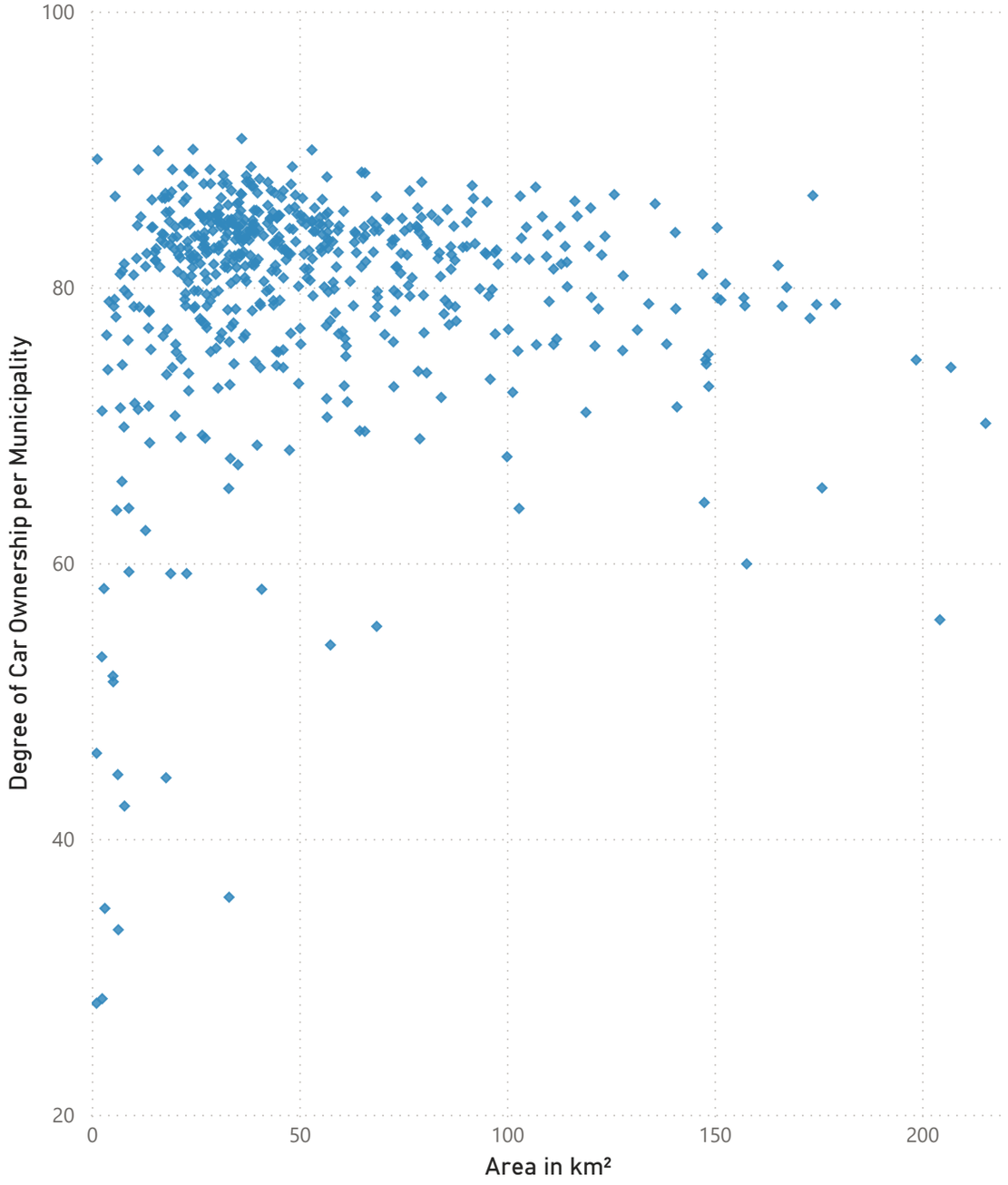
Degree of Car Ownership by Province



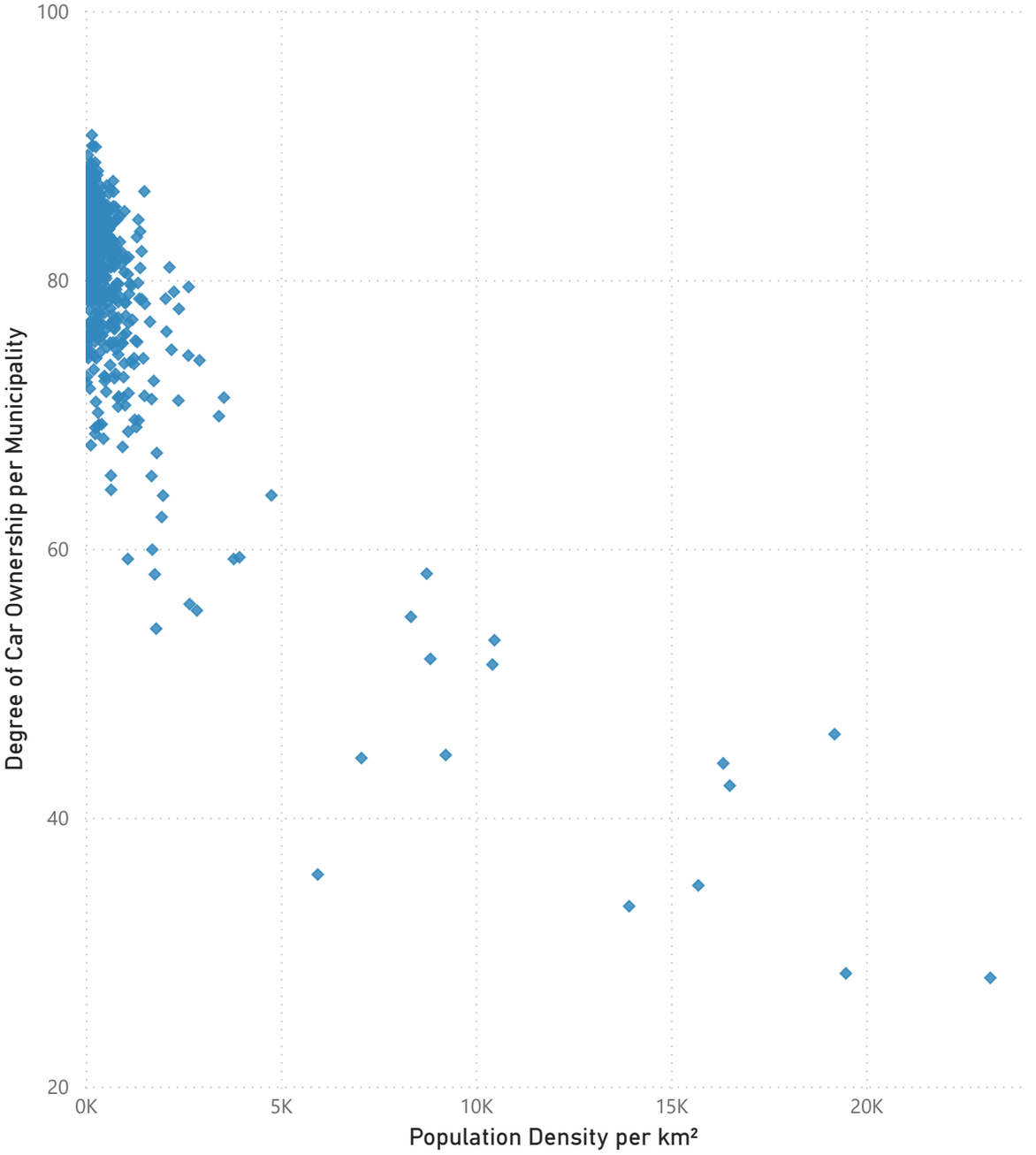
Degree of Car Ownership by Household Type



Degree of Car Ownership vs Municipality Size



Degree of Car ownership vs Population Density of Municipality



Degree of Car Ownership vs Prosperity Index of Municipality