UNIVERSITÀ DI PISA

Laboraroty of Data Science

Group 24 project

Professor:

Anna Monreale

Alessia Ferrero
683177

Erica Neri
682010

Giorgio Gobbin
683071

05 January 2025

Contents

Chapter 1

Part 1

1.1 Assignment 1 & 2: Data understanding and Cleaning

The first task of the project required us to analyze and complete the missing values within the three assigned datasets: Crashes, People and Vehicles. After carefully analyzing the data and identifying the missing values, we decided to take different approaches depending on the nature of the data and the context of the attributes. In particular, we chose to replace only those missing values that could be recovered with certainty. For most of the attributes, we exploited related information within the same dataset or between different datasets to infer the correct values.

For missing values of attributes such as *vehicle_id*, the absence of which indicated a real impossibility of recording an actual datum, as for pedestrian or bicycles, we preferred to replace them with identifying values as $'-1'$. This approach allows us to clearly distinguish cases where the data is justifiably missing from those where it may have been omitted or lost.

Missing values were replaced by the use of several specially developed functions:

- **replace_nulls**: This function replaces null or empty values in a specified column of a dataset with a predefined replacement value.
- **replace_nulls_with_conditions**: This function replaces null values in a column only if another column satisfies a specified condition. For example, it was used to replace some values of `vehicle_type` according to the values in the attribute `unit_type`.

For `latitude, longitude` and `location` special attention was given; we created a function that exploits Nominatim's geocoding API to obtain geographical coordinates from partial information, such as house number, direction and street name. These attributes were used to construct a complete address, which was then sent to the API to obtain latitude and longitude. If geocoding was successful, these values were returned and inserted into the respective columns of the record. If geocoding failed, the function continued without modifying that record.

Subsequently, the records were scanned again, trying to fill in the missing values using partial addresses derived from the previously specified columns and following the same procedure. This process allowed latitude, longitude and full address values to be filled in where they were initially missing, improving the completeness and reliability of the dataset.
For the remaining 300 or so records that were left with null values, we decided to replace the missing latitude and longitude values with the average of the geographic coordinates for the same street. Finally, for the last records that couldn't be filled in, we decided to assign -1 as the value.

1.2 Assignment 3: Data-warehouse schema

This task required designing the database schema using SQL Server Management Studio. Each table was carefully structured, and a specific data type was assigned to every attribute to ensure data accuracy and consistency. (See Figure 1.1).

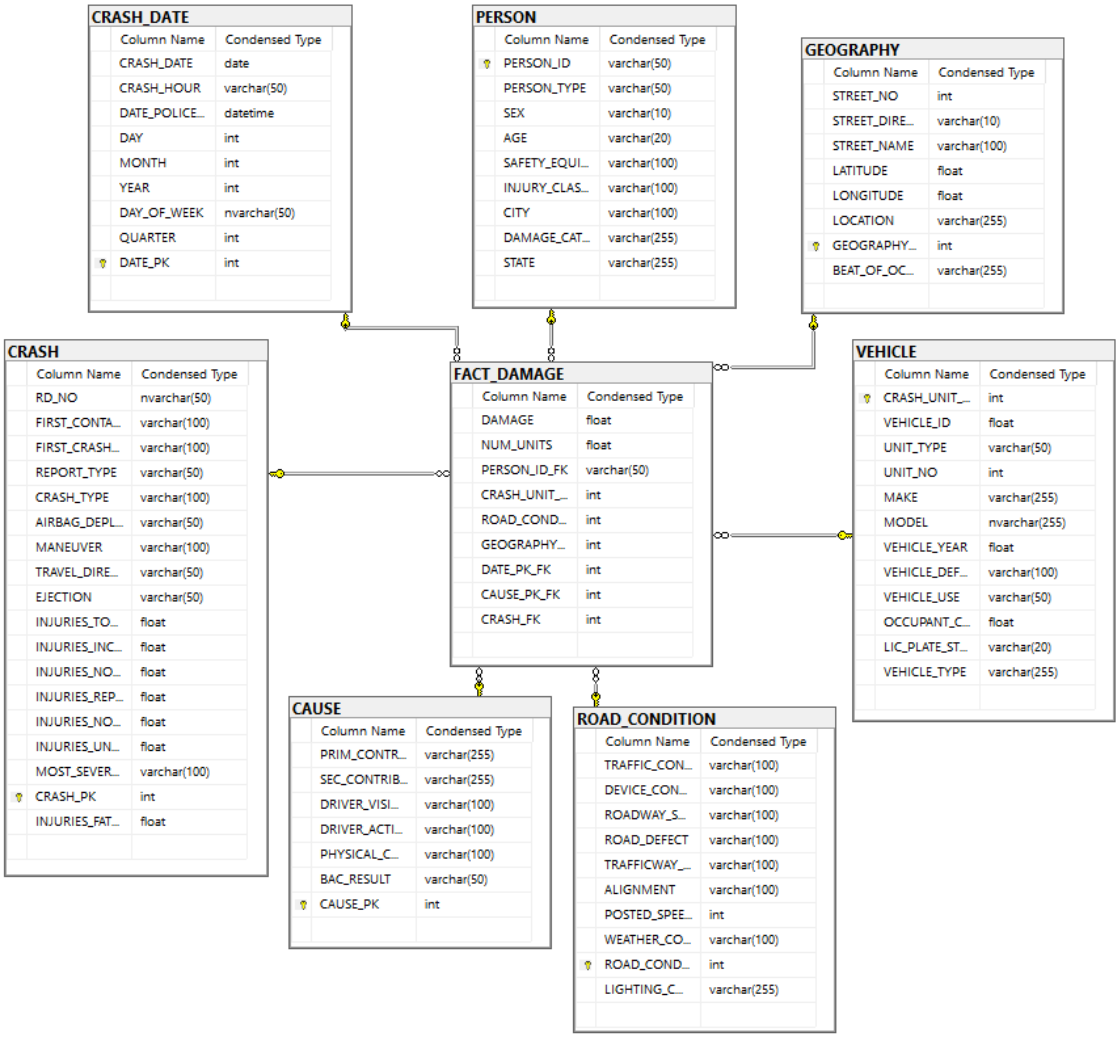

Figure 1.1: Data warehouse diagram.

In the fact table, all the primary keys of the dimension tables are present, serving as foreign keys within it. The combination of these attributes constitutes the primary key of the *Damage* table, linking it to the corresponding entries in the dimension tables.

1.3 Assignment 4: Data Preparation

In Assignment 4, the task involved developing a Python program to process the contents of three datasets (Crashes.csv, People.csv, and Vehicles.csv) and generate eight separate tables: *Crashes*, *Vehicles*, *Person*, *Crash_date*, *Geography*, *Road_condition*, *Cause*, and *Damage*.

The first step in our approach was to develop a function to unify the datasets into a consistent structure, enabling an easier data manipulation. The merge was performed using the *people* dataset as the base, with the `RD_NO` column serving as the unique key to integrate information from the vehicles and crashes datasets.

Next, we developed several functions to process and structure the data for different tables. During this step, we focused on identifying and removing any potential duplicate records to ensure data integrity and avoid redundancies in subsequent analyses. These functions can be categorized into two types: generic functions for most tables and specialized functions for tables requiring custom logic.

On the generic functions we have also *create_table_pk*, that is constructed to add a primary key column to each row, assigning a unique identifier to ensure that all records are distinct and can be linked effectively to other tables in the database.

A more tailored function, *create_csv_for_data*, was implemented to handle the Crash Date table. Here, we incorporated utilities from Python's datetime module to extract, format, and standardize temporal information. We normalized dates to the format YYYY-MM-DD, ensuring consistency across the database, and we derived temporal attributes, creating additional columns, such as day, month, year, day of the week, and quarter of the year, computed in order to facilitate temporal segmentation and further analysis.
Similarly, the `date_police_notified` attribute was transformed using the same approach.

For the Damage table, the *create_csv_for_damage* function was implemented. This function consolidates data from multiple indexed tables (e.g., geography, cause, and crash data) into a single output table. The function also integrates primary keys from related tables, ensuring that relationships are preserved and enabling seamless database integration.

1.4 Assignment 5-6: Data uploading with Python and SSIS tables

We developed a Python script to facilitate the population of SQL Server tables using data from CSV files. This script employs the *pyodbc* library to establish a connection to the SQL Server database.

The core of the script is the *populate_database* function, which handles the straightforward insertion of CSV data into SQL tables. This function begins by reading the CSV file, extracting the column names from the header row to align the data structure with the target database table. A parameterized SQL INSERT statement is then dynamically constructed to match the columns and values, ensuring flexibility and security. The function processes the data in batches of 1000 rows, significantly improving performance and reducing the risk of overloading the database with a single large transaction. It does not explicitly address duplicate entries since they were handled in the previous step. Any errors during this process, such as missing columns or database issues, are logged for further inspection.

For scenarios where the column names in the CSV file differ from those in the database table, the *populate_database_with_mapping* function provides a more versatile approach. This function uses a mapping dictionary to define the correspondence between the CSV file and the database columns. It transforms the header row of the CSV file according to this mapping, aligning the data structure with the target table. Each row of the CSV is processed and adjusted to reflect the column mapping before being inserted into the database. Like the previous function, it employs batch processing for efficiency and generates a dynamic SQL statement to handle the insertion. Errors, such as missing mappings or data alignment issues, are captured and reported to maintain process reliability.

At this point, we duplicated each table created in the earlier stages, excluding the records, and renaming the new tables by appending the suffix *"_SSIS"* to their names.

1.4.1 Assignment 6: Uploading 10% on the SSIS tables

After duplicating the tables, we proceeded to create a SSIS project designed to populate the new *"_SSIS"* tables. The project was configured to load a subset of the data prepared before, specifically 10% of the total records. This selective data transfer was achieved by applying a sampling percentage on the fact table, saving the result on the OLE DB destination *Fact_Damage_SSIS*. From this table we opened a new Data flow to populate the other tables: the procedure initiated with a multicast of the SSIS *Fact* table to get a duplicate of the starting data and then, for each one of the dimensions, we computed a *lookup* and saved the saplings into the new _SSIS tables.

Figure 1.2: External view: Upload 10%

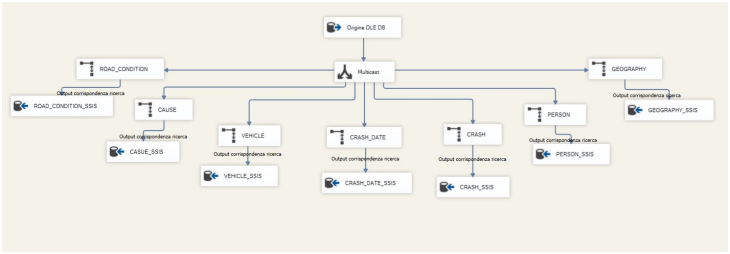

Figure 1.3: Internal view: Upload 10%

1.5 Assignment 7: Query

The last step we did was the implementation of four queries in SQL Server Integration Services (SSIS) using Visual Studio. Each query is described in terms of the steps taken in SSIS and the final result achieved. The last query has been defined by our group.

6b: Show all participants ordered by the total damage costs for every vehicle type

We configured the OLE DB source with our *fact_damage*. We used a *lookup* to join the *Fact* table with the *Vehicle* table, in order to enrich the dataset with the *vehicle_type*. At this point, thanks to a *multicast*, we did two different operations: from one side, we just sorted the dataset on the *vehicle_type* attribute; on the other side, we grouped the data by *vehicle_type* and compute the total of the damage for each one. Subsequently, we merged the two results on *vehicle_type*, obtaining a table having, for each *person_id*, its *vehicle_type* and the total damage associated to that category.

7b: For each month, calculate the percentage of the total damage costs caused by incidents occurring between 9 pm and 8 am and incidents occurring between 8 am and 9 pm, with respect to the average total damage costs for all months within the same year

Once the fact table had been configured as the starting point, a lookup was performed with the *crash_date* table. Subsequently, the hour attribute, which was in hh:mm:ss format, was transformed to extract just the hour, and a duplicate of the resulting table was created with the multicast operator. This procedure enabled us to perform a grouping on the year to extract the average damage and, conversely, to calculate the total damage grouped by month. Furthermore, we employed a conditional split to obtain the nighttime (between 21 and 8) and daytime (between 8 and 21) data. Subsequently, we were able to perform a merge join based on the year

and then a union to finally calculate the requested percentages.

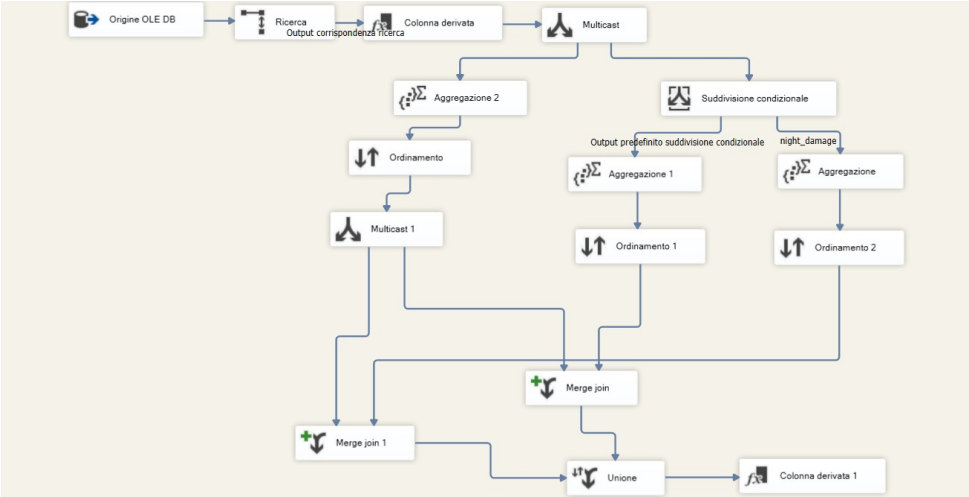

Figure 1.4: Query 7b

8b: Show the total crash damage costs for each vehicle type and weather condition

As in the other queries, we first configured the origin with the *Fact* table, implement two subsequent *join* with the *vehicle* table to get the *vehicle_type* and with *road_condition*, to take the *weather_condition* attribute. At this point, using *aggregation*, we found the total damage for each combination of weather and vehicle type. In this way, we obtained a table containing *vehicle_type*, *weather_condition* and the total_damage.

9b: Show the total damage for personal vehicle use, for each primary cause and secondary cause. We thought this could be interesting to identify which combinations of causes contribute the most to overall damage

In this case as well we configured the fact table as the origin. Then, we used a *look up* to join the *vehicle* table in order to do a conditional division on the *vehicle_use*, keeping only those who were personal. Then, we did a join to connect the dataset with the *Cause* table and get the *prim_contributory_cause* and *sec_contributory_cause*. We used *aggregation* to *group by* primary and secondary cause and sum the total damages. The result is a table with *prim_contributory_cause*, *sec_contributory_cause* and *damage_tot*.

Chapter 2

Part 2

2.1 Assignment 1: Build a DataCube

In creating the data cube, the decision was made to retain all attributes associated with each dimension, rather than focusing on a subset of attributes that would directly address the specific business questions at hand. This approach enables the system to answer potential future business inquiries. The **dimensions** were set up as follows:

Date: The date_id was used as the key, and a single hierarchy was established, comprising the *year_month_day* hierarchy. It's granularity spans from the year to the crash date, passing through the quarter, month, and day. We decided to add the crash_day as the last element, because we also have the time when the incident happened, which deepens the granularity.

The attribute *month* has values from 1 to 12, and a new attribute was generated to facilitate the sorting process; the variable designated is *order_month* (utilizing 'new calculation' methodology). While this attribute is typically sorted alphabetically by default, the new attribute was created to adapt to potential future MDX queries searching for, for example, the "value of the preceding one." This attribute was generated using the case method, assigning a number from one to nine for the months from January to September and then ninety, ninety-one and ninety-two for the remaining ones (e.g., 1 = 1, 12 = 92). The ordering of the month was then set using the numeric attribute in the advanced properties.

Regarding the other dimensions, we did not define any other hierarchies outside of the one mentioned above, as we did not find any attributes that, considered hierarchically, returned more specific information as the granularity increased.

In the process of creating dimensions, it was determined that a specific attribute would not be required as the minimum element of the tables. This decision was influenced by the substantial number of missing values identified at the beginning of the analysis of the datasets. Consequently, neither attribute appeared to be the optimal selection.
A secondary rationale for this decision pertained to the absence of attributes designated as essential in the development of dimensions within the SQL management studio. To avoid potential complications, the dimensions were created, considering solely the primary key.

The sole **measures** employed were those automatically generated by the system at the time of data cube creation. The measures in question are as follows:

- *Damage* and *num_unit* item are already present in the fact table, so they are translated into a measure.
- The *Conteggio di Fact Damage* item contains a row count based on the previous measures.

2.2 MDX Query: 2-3-4-6-8b

In the majority of the executed queries, the decision was made to implement the solution using a smaller set of values (e.g., Weather Condition instead of Location) due to the "out of memory" error returned by SQL Server Management Studio. This approach enabled the testing of the query results and the subsequent substitution of the required attributes.
It was also determined that the output of the queries would include measures calculated during the resolution. The rationale for this decision was to enhance the readability of the results.

2. For each month, show the total damage cost for each location and the grand total with respect to the location.

To compute the total damage cost for each location, we utilized the `SUM` function over the location attributes. This approach allowed us to aggregate the damage values for all records associated with each location. Additionally, the query incorporates filtering using the `nonempty` function to ensure that only records with non-zero damage values are included in the results. This method provides both the total damage for each location and the grand total across all locations, organized by month.

3. Compute the average yearly damage costs as follows: for each crash, calculate the total damage to the user divided by the number of distinct people involved in the crash. Then, compute the average of these values across all crashes in a year.

To compute the average yearly damage costs, we employed a systematic approach. First, we calculated the number of distinct individuals involved in each crash by using the `COUNT` function on the set of `PERSON ID` members filtered to include only crashes with nonzero damage values. This allowed us to determine the number of people impacted by each incident.

Next, we determined the total number of crashes per year by summing over all crash records (`RD NO`) and counting the non-empty damage values associated with each crash. This step ensured an accurate count of crashes contributing to the yearly damage calculation.

To calculate the average damage for each crash, the total damage value was divided by the number of individuals involved in the crash. If no individuals were associated with a crash, the resulting value was set to zero to ensure robust handling of division by zero scenarios.

Finally, the average yearly damage cost was computed by summing the average damage per crash across all crashes in a year and dividing this total by the number of crashes for that year. If no crashes were recorded in a given year, the value was set to zero to account for missing data.

4. For each location, show the damage costs increase or decrease, in percentage, with respect to the previous year.

An experiment was conducted in which different versions of the query were tested. The versions tested included `PARALLELPERIOD`, `LAG`, and `PREVMEMBER`. After a thorough evaluation of the results obtained from each approach, it was determined that the version employing the PREVMEMBER function offered the most optimal outcome, given its simplicity, readability, and the absence of necessity to traverse distinct levels within the date hierarchy.
After careful consideration, we opted to leave the `NULL` values in the text because we thought it would be interesting to analyze the differences in the location of the incidents over time.

```
With member [Measures].[Change] as
    ([Measures].[DAMAGE] -
    ([Date_info].[year_month_day].prevmember, [Measures].[DAMAGE]))/
    ([Date_info].[year_month_day].prevmember, [Measures].[DAMAGE]),
```

```
format_string = "Percent"

SELECT {[Measures].[DAMAGE], [Measures].[Change] } ON COLUMNS,
([Date_info].[year_month_day].[YEAR], [GEOGRAPHY].[LOCATION].[LOCATION]) ON ROWS
FROM [Group_ID_24_cubo]
```

6. For each vehicle type and each year, show the information and the (total) damage costs of the person with the highest reported damage.

We decided to construct custom members to compute `PersonAge` and `PersonSex`, representing the information of individuals involved in incidents. To effectively analyze the data, we employed two nested `GENERATE` functions on the *vehicle_type* and *year* dimensions. To prevent summing the *Damage* measure across the entirety of the attribute space, we utilized the `EXCEPT` function to exclude specific members from the aggregations.

Within the inner loop, a `CROSSJOIN` was performed between the current `vehicle_type`, `year`, and `person_ID`. From this result set, we selected the topmost element, corresponding to the person that produced the highest `DAMAGE`.

8b. For each year, show the most risky crash type and its total damage costs. To measure how risky a crash type is, you should assign a weight to each type of injury you encounter in the data (for example, a fatal injury weighs 5 times an incapacitating one, which weighs twice a non-incapacitating injury).

To begin, we calculated the number of injuries for each category annually defining separate members for each injury classification to facilitate further calculations. This was achieved using the `COUNT` function in conjunction with `nonempty` to exclude null results.
Following this, To assess the riskiness of each crash type, we assigned the following weights to the respective injury categories:

- 60 for Fatal Injuries
- 30 for Incapacitating Injuries
- 15 for Non-Incapacitating Injuries
- 5 for No Indication of Injuries
- 2 for Injuries Reported but Not Evident
- 0 for Unknown Injuries

Although the severity of Unknown Injuries could not be determined, we included them in our analysis by assigning a weight of 0, thereby ensuring comprehensiveness without introducing undue bias.

The riskiness of a crash type was determined by applying the `MAX` function over the set of crash types, using the sum of weighted injuries as the measure. This enabled us to identify the riskiest crash type through the `TOPCOUNT` operator. Finally, we computed the total damage cost for the identified riskiest crash type by summing the damage values of all crashes categorized under it.

2.3 PowerBI

These assignments focus on creating interactive dashboards to analyze key aspects of traffic incidents, including geographical damage distribution, road conditions, and the characteristics

of people involved in crashes. Each dashboard integrates dynamic visualizations and filters to enhance data exploration and insight generation.

It is worth mentioning that, due to some dashboard issues with the cube connection, we opted to use the database connection for these assignments. This approach was further supported by the observation that, during the cube's creation, the database was fully utilized. While the cube includes some hierarchies, they were not employed for these assignments, so the overall structure remains unchanged.

2.3.1 Assignment 9

For the first assignment, we created a dashboard to show the geographical distribution of total damage costs by vehicle category. We calculated the total damage costs using the formula

$$\text{SUM}(\text{`Group\_ID\_24 FACT\_DAMAGE'}[\text{DAMAGE}]),$$

which determined the size of the bubbles on the map. Then, we used latitude and longitude data from the dimension table `Geography` to create the map, setting the vehicle type as the legend and adding a slicer to allow the selection of one or more specific vehicle types for a more focused analysis of damage costs.

We also added zoom-in and zoom-out controls to the map, enabling users to focus on specific areas of interest. It is important to note the presence of some damages located in the Atlantic Ocean. This anomaly is due to the fact that some missing latitude and longitude values were filled with -1, and these points (with latitude -1 and longitude -1) should be disregarded as they do not represent real locations.

Additionally, we included:

- A Q&A section where users can input simple questions to better understand the characteristics of the database.
- A bar chart to analyze damage costs by vehicle type. The bar chart includes a red line representing the average damage cost, helping users compare categories more effectively.

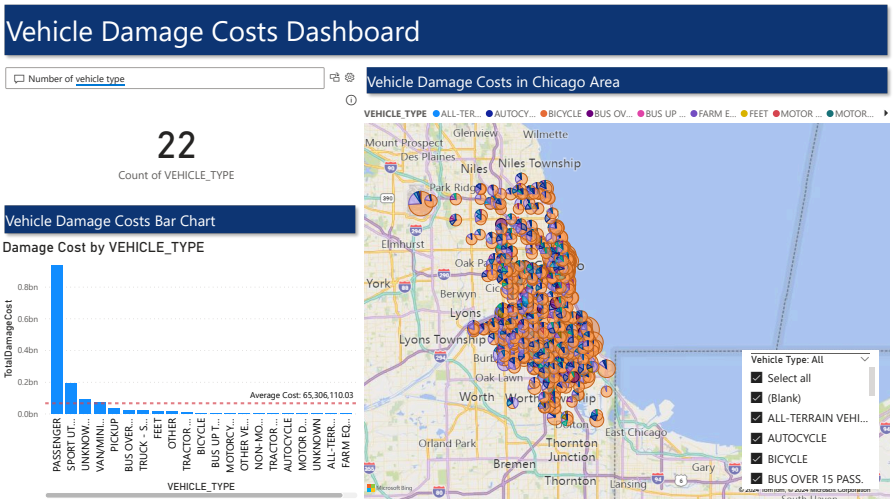

Figure 2.1: Geography Dashboard

2.3.2 Assignment 10

For the second assignment, we created a dashboard focusing on the table Road Condition. We calculated the total number of incidents creating the measure

COUNT('Group_ID_24 FACT_DAMAGE'[NUM_UNITS]).

The dashboard features are:

- A pie chart showing the total number of incidents by `Trafficway Type`.
- A slicer to filter by `Traffic Control Device`.
- A line chart illustrating the trend of incidents over time based on `Traffic Control Devices`.

Cross-filtering is enabled between `Traffic Control Device` and `Trafficway Type` by selecting an option from the pie chart legend, and the line chart includes controls to allow users to select a specific time period.

2.3.3 Assignment 11

For the third assignment, we built a dashboard focused on the characteristics of people involved in crashes. We included two bar charts:

- One displaying the classification of injuries.
- Another highlighting the primary contribution causes of incidents.

Additionally, we created an interactive decomposition tree based on the total number of incidents, enabling the analysis of data related to people involved in crashes. The variables included in the decomposition tree were:

- `Gender`
- `Age`
- `Driver Vision`
- `Physical Condition`
- `Primary Contribution Cause`

These variables were selected from the dimension tables `Person` and `Cause` to ensure relevance and clarity in understanding the characteristics of the individuals involved in accidents. The data was not only taken from the `People` dimension table but also from the `Crash` dimension table. In particular, we included `Primary Contributory Cause` because it directly relates to the actions of individuals. Understanding the main cause of an incident provides critical insights for identifying and addressing risk factors.

This dashboard supports multiple cross-filters. For example, selecting "Fatal Injury" from the first bar chart highlights all corresponding `Primary Contribution Causes` leading to fatal incidents. The decomposition tree updates dynamically to reflect fatal crashes based on the selected characteristics. This ensures a dynamic and detailed exploration of the data for the user.

Chapter 3

Changes from the first delivery

Due to missing attributes in the tables because of a transcription error, we updated the split files (Utility_split and splitting_data) so that they considered all the necessary attributes. In particular, the following were modified:

- The function create_table_for_data.
- The dictionary files_to_process

In particular, the function did not consider all the attributes necessary for the proper definition of foreign keys in the fact table and thus connections between tables.