Department of Computer Science

Master in Data Science & Business Informatics

Part 1 & 2

Data Warehouse Design + MDX Data Analysis

(SSIS & SSMS) + (SSAS | MDX | BI)

LDS Decision Support System Module II

Submission Date: 05/01/2025

Group ID_31

Sana Afreen (681744)
Yordanos Girma Ayele (682486)
Shreyashree Das (701179)

Prof. **Anna Monreale**

Università di Pisa

Table of Contents

Part 1

Introduction

In Part 1 of this project, a Data Warehouse was developed using SQL Server Management Studio on **lds.di.unipi.it**. The database, **Group_ID_31_DB**, was designed to store and manage data on traffic incidents in Chicago from January 2017 to January 2019. The tasks involved creating and populating the database schema and solving assigned problems using SSIS, prioritizing client-side computations with minimal reliance on SQL commands.

Assignment 1

1.1 Data Understanding

We conducted initial data exploration and preprocessing activities to identify any instances of missing values, Unique values or redundant data as follows:

1.1.1 Crashes Dataset (257,925 instances, 36 attributes)

Each instance here represents a unique crash event. The attributes likely include details about each crash: date, time, location, weather conditions, road conditions, crash severity, and so on. Since it has the highest number of attributes, it's likely the most detailed dataset in terms of context around each event.

1.1.2 People Dataset (564,565 instances, 19 attributes)

This dataset provides information on individuals involved in the crashes. Each instance here corresponds to a unique individual. Attributes could include demographics (age, gender), roles in the crash (driver, passenger, pedestrian), injury severity, use of safety equipment (seat belts, helmets), etc. The fact that there are more people than crashes suggests that many crashes involve multiple people (drivers, passengers, or other non-vehicle occupants).

1.1.3 Vehicles Dataset (460,437 instances, 17 attributes)

Each record here corresponds to a vehicle involved in a crash. Attributes might include vehicle type, make/model, year, safety features, and possibly whether the vehicle was at fault. Since there are fewer vehicles than people, this could indicate that some vehicles carried multiple people, or that certain crashes involved only one vehicle.

1.2 Data Relationships and Potential Analysis

Each dataset includes a unique identifier and a shared key, **RD_NO**, enabling us to link the people and vehicles involved in the same crash for granular analysis. The **Crashes.csv** contains **RD_NO**, the **Vehicles.csv** includes **Vehicles_ID** and **RD_NO**, and the **People.csv** has **RD_NO**, **Vehicle_ID**, and **Person_ID**.

We first merged **People.csv** with **Vehicles.csv** on **Vehicle_ID** and **RD_NO**, followed by merging **Crashes.csv** with the pre-merged dataset on **RD_NO**. This revealed that while RD_NO in the crashes.csv is unique, it is repeated in the other datasets due to multiple vehicles and people being involved in the same crash. Consequently, the final merged dataset contains repeated RD_NO entries with distinct records for each related entity. And this also suggests that we can't use RD_NO as PK when we are going to create DB_Schema as it is not unique now for each record. So we need to create Key_ID for each Dimensional Table that we are going to use as FK in Fact_Table.

Assignment 2

2.1 Data Cleaning

Analyzing and addressing missing values was a critical step in preparing the merged dataset for further analysis. This process involved assessing the extent of missing data, reviewing unique value counts for each attribute, and implementing effective imputation strategies. By utilizing logical relationships between columns, we applied mapping-based solutions to infer missing values. When inference was not possible, we used "UNKNOWN" as a standardized placeholder to maintain consistency. However, as this was not the primary focus of the project or essential for performance evaluation, we prioritized preserving the originality of the provided information.

2.1.1 Crashes Dataset

The **Crashes Dataset** contained missing values across multiple columns, which were addressed using logical relationships and mappings within the data:

1. **REPORT_TYPE**: Missing values were inferred based on the **CRASH_TYPE** column. The CRASH_TYPE column indicates whether the crash involved injuries or was a drive-away scenario. It provides direct insight into whether the report was handled "At Scene" or "At Desk".

   "At Scene" was assigned for crashes labeled as "INJURY AND/OR TOW DUE TO CRASH."

   "At Desk" was assigned for crashes labeled as "NO INJURY / DRIVE AWAY."

2. **STREET_DIRECTION**: Mappings between **STREET_NAME** and **STREET_DIRECTION** were created from existing data. Missing values were filled using these mappings. Defaults were applied where no match was available.
3. **LATITUDE and LONGITUDE**: Geographic coordinates were inferred using a mapping of **STREET_NAME** and **STREET_DIRECTION**.
4. **LOCATION**: Missing values were constructed by concatenating available **LATITUDE** and **LONGITUDE** values as we already incorporated these two columns using the above method.
5. **MOST_SEVERE_INJURY**: Missing values were assumed to represent "No Injury" based on the dataset's context.
6. **STREET_NAME**: Missing values were inferred from **BEAT_OF_OCCURRENCE** mappings. If no mapping was available, "UNKNOWN STREET" was used as a placeholder.
7. **BEAT_OF_OCCURRENCE**: Inferred from STREET_NAME and LOCATION mappings, reports within the same area must be assigned to the same Beat. If no match is found, the Beat should be set to "UNKNOWN."

2.1.1.1 Python Implementation

The Python code used logical functions for each column to infer missing values. Predefined mappings and default values were applied within a structured data-processing pipeline. CSV reading and writing operations ensured efficient handling of the data.

2.1.2 Vehicles Dataset

The **Vehicles Dataset** also contained missing values across several columns. These were addressed using frequency-based mappings and default placeholders:

1. **MAKE**: Missing values were inferred from the **MODEL** column. A mapping of **MODEL** to **MAKE** was created using frequency counts. "UNKNOWN" was assigned where no association was found.
2. **MODEL**: Missing values were filled using mappings based on **MAKE** and **VEHICLE_YEAR**. If no match was found, "UNKNOWN" was used.
3. **VEHICLE_YEAR**: Missing values were inferred using the most frequent year associated with each **MODEL**. If no match existed, "UNKNOWN" was assigned.
4. **Other Columns**: All other columns with missing values (LIC_PLATE_STATE, UNIT_TYPE, VEHICLE_ID, VEHICLE_DEFECT, VEHICLE_TYPE, VEHICLE_USE, TRAVEL_DIRECTION, MANEUVER, OCCUPANT_CNT, FIRST_CONTACT_POINT were filled with "UNKNOWN" due to the absence of reference data.

2.1.1.2 Python Implementation

The Python code constructed dictionaries to map relationships between columns. Logical functions applied these mappings to infer missing values, with fallback defaults ensuring no column remained empty. A frequency-based approach ensured that the most common values were selected for imputation.

2.1.3 People Dataset

The people Dataset contains missing values across several columns.

1. **VEHICLE_ID:** Missing values might mean the person isn't associated with a specific vehicle. Replace missing values with a placeholder value, such as (-1, 0) for easier identification. Vehicle ID is common in both People.csv and Vehicles.csv, so the script fills missing VEHICLE_ID values in People.csv by matching the RD_NO column with Vehicles.csv. If no match is found, a default value (`-1`) is assigned.
2. **BAC_RESULT:** it is replaced with the default "test not offered" since we already have this category and total value is very big.
3. **Other columns:** All the other columns with missing values (CITY, STATE,SEX,AGE,SAFETY_EQUIPMENT, AIRBAG_DEPLOYED,EJECTION,INJURY_CLASSIFICATION,DRIVER_ACTION,DRIVER_VISION,PHYSICAL_CONDITION,DAMAGE) were filled with 'UNKNOWN' due to absence of reference data.

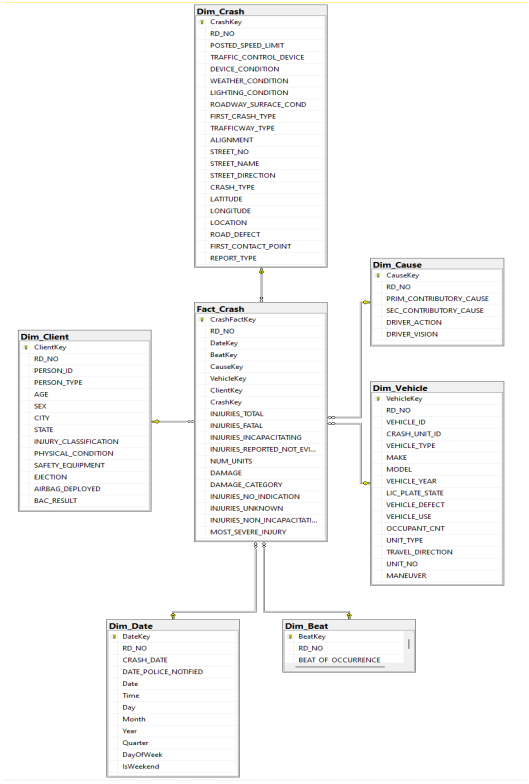
2.1.1.3 Python Implementation

The Python code utilized dictionaries to map relationships between RD_NO and VEHICLE_ID for quick lookups. Logical functions applied these mappings to fill missing values in People.csv, ensuring consistency across datasets. Fallback defaults (e.g, `-1` for VEHICLE_ID, "UNKNOWN" for other attributes) guaranteed no column remained empty. For other attributes, default values replaced missing entries, while a frequency-based approach could be applied for attributes requiring imputation based on common values.

Assignment 3

3.1 DW Schema

This task was one of the most creative aspects of the project. There were two potential approaches to creating the tables in the database: using Python or SQL commands. We chose to design the Data Warehouse (DW) schema using SSMS, executing SQL queries to create the tables (all the codes are attached to the zip file for reference). We decided to create six dimensional tables, each with a primary key (PK), linking to a central fact table via foreign keys (FK). Initially, we considered using **RD_NO** as the unique identifier to connect all the tables. However, as discussed earlier, after merging the datasets, **RD_NO** no longer served as a unique key for each record. Therefore, we generated a surrogate key by adding a new attribute, **KeyID**, to each table, which was an auto-generated, incremental value, as illustrated in Figure 3.1. This surrogate key was used for linking the tables, while **RD_NO** was retained in each table for reference to maintain the interconnections between them. By keeping the **RD_NO** column, we also ensured that each of our files contained the same number of records across all split datasets, which we will further explore in the next assignment. After successfully creating the tables, we designed our Data Warehouse (DW) Diagram using the prebuilt Diagram tool in SSMS. This was done by selecting all the previously created Dim_Tables and Fact_Table, ensuring a clear and structured visualization of the relationships within the schema.



3.1.1 Challenges & Insights

As per our own knowledge and understanding, An interesting challenge we encountered during the creation of the DW schema was that once a table was created,SSMS did not allow any modifications directly to the structure of the table. To make changes, we were required to drop and recreate the table. Additionally, we discovered that in order to drop a table, the **fact_table** had to be dropped first before any of the dimensional tables could be removed. This limitation required careful planning to ensure that the database schema was structured correctly from the outset.

Another challenge we faced was related to data types. While defining the data types and lengths for each column, we noticed discrepancies between the data types used in the CSV files and those required for the tables in the database. Some attributes had different data types in the CSV files than we initially defined when creating the tables. After realizing this issue, we had to drop and recreate the tables with the correct data types and lengths, which involved re-evaluating the data structure for each column to ensure proper compatibility and consistency. Once, we also tried to make a change directly to the pre-built table, it was giving us errors and asked us to drop and recreate the table.

Assignment 4

4.1 Data Preparation

As outlined in the data preparation and merging process, we carried out additional preprocessing steps to enrich our dataset. Using Python, we leveraged libraries such as **datetime** and **csv** to extract features like date, time, day, month, and quarter, adding new columns for each. Initially, the merged dataset contained 67 columns. After preprocessing and feature extraction, the dataset expanded to 75 columns.

To align with the schema designed in the previous assignment, we used the **csv** module to split the data into multiple CSV files. Instead of creating a new Key_ID column, we utilized the existing **RD_NO** field across all tables. This approach ensured consistent record counts across tables, simplifying the process of establishing relationships between tables during subsequent tasks.

Assignment 5

5.1 Data Uploading with Python

The process of populating the `Group_ID_31_DB` database was implemented through Python scripts designed to follow the specified schema. Each script read data from corresponding CSV files and leveraged efficient batch processing techniques to ensure data upload was both scalable and robust. **Batch Insertion** approach is consistently used in all scripts to enhance performance and minimize database communication overhead. Data from the CSV files is processed row by row, stored in a list, and inserted into the database in a single operation using **`executemany()`**. Comprehensive error-handling mechanisms were integrated to address potential issues and maintain data integrity during execution. **Database Connection Errors** errors were couched during database connection using `pyodbc.Error` and log them to avoid program crashes. **Batch Insertion Errors**: If a batch insertion fails, the entire batch is marked as failed, and the data is logged for further inspection. **File Handling Errors** Missing or unreadable CSV files were caught using `FileNotFoundError` or `IOError.` **Failed Records Logging** during insertion were written to a separate CSV file (failed_records.csv) for debugging purposes.

5.1.1 Challenges & Insights

1. **Inefficiency with Row-by-Row Insertion**
   Initially, the data was inserted into the database using a row-by-row approach (`cursor.execute()` for each record). This method caused significant delays, as each record required a separate database transaction. For large datasets, this approach was highly inefficient and time-consuming, making it impractical for completing the upload process in a reasonable timeframe.
2. **Primary Key Increment Issue After Execution Aborts**
   During the row-by-row insertion, the **primary key values** in the database continued to increment even when the execution was aborted due to errors or interruptions. When retrying the upload, the insertion resumed from the last incremented primary key value, leaving gaps in the primary

key sequence. This inconsistency required dropping and recreating the affected tables to reset the primary key counters, which added extra manual work and delays to the process.

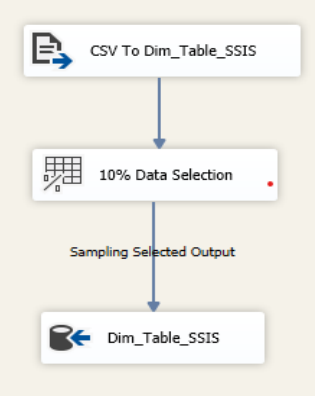
5.1.2 Foreign Key Initialization and Planned Improvements for Fact_Crash Table

The python script for populating the Fact_Crash table initializes foreign keys (DateKey, BeatKey, etc.) at 1 and increments them sequentially for each record, bypassing the dimension tables (Dim_Date, Dim_Beat, etc.). This approach was chosen due to the lack of descriptive data in the CSV file to match against the dimension tables, which made dynamic foreign key resolution impossible. Additionally, time constraints and the focus on building an initial functional pipeline influenced the decision to adopt this simplified method. While this ensures that data can be inserted into the Fact_Crash table for testing purposes, it is not an optimal solution, as it does not maintain data integrity or meaningful relationships between tables. Moving forward, the plan is to enrich the CSV file with descriptive fields (e.g., DateValue, BeatName) that correspond to dimension table columns, enabling dynamic foreign key lookups. The script will be updated to fetch foreign keys from the dimension tables and validate their correctness before insertion. This interim solution is acknowledged as a compromise, with the goal of implementing a more robust, relational design in subsequent phases of the project.

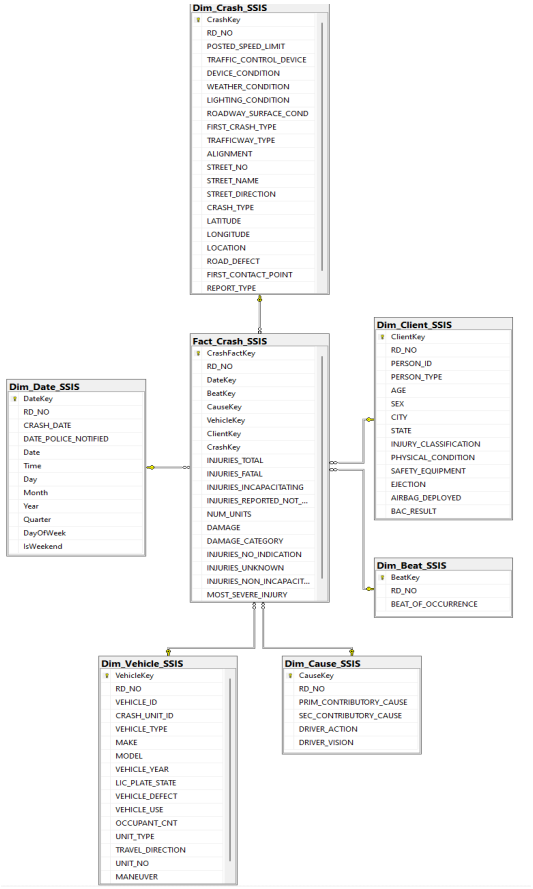
Assignment 6

6.1 Data Uploading SSIS

To create duplicate tables containing 10% of the original data, there were two ways, either to use SQL command "Select top 0 * into Dim_Table_SSIS, From Dim_Table" directly on SSMS by opening a New Query. Or The second approach was to create duplicate Tables using SSIS tools. We utilized Visual Studio SSIS **(SQL Server Integration Services**). The process began by creating multiple packages in SSIS for each Dim_Table. Then using the **Execute SQL Task** within the Control Flow to connect to the server and create a duplicate table based on the pre-existing schema, for reference we added all the codes we used for it in our SQL Query file and as mentioned above.



After establishing the structure of the duplicate table, we moved to the **Data Flow Task** to handle the data transformation and transfer.



Within the Data Flow, we used the **Flat File Source** to import the CSV files from the local host that we created during the pre-processing step (splitted files). For the 10% random sampling, we employed the **Percentage Sampling Transformation** from the Transformation section of the SSIS Toolbox. This component enabled us to randomly select 10% of the data records from the source. Subsequently, we added a **Destination** node, connected it to the server database and mentioned the specific table where we wanted to populate the data.

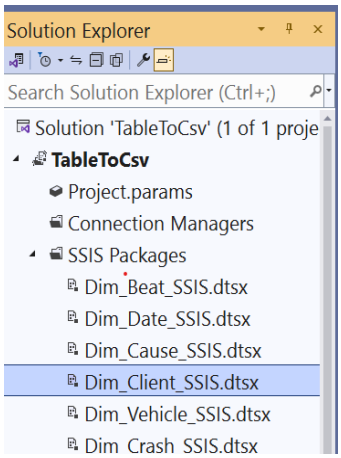
6.1.1 Challenges & Insights

During the process, we encountered several challenges. Initially, we were unable to create a duplicate table directly through SSIS. However, we successfully achieved this using SSMS by executing SQL commands. But, after multiple trials and adjustments we were able to create tables using SSIS. Once the table was created via SSIS, we had to redefine the primary key (PK) for each table, as PKs are not retained during table duplication.

The primary challenge arose when we added a new KeyID column as the PK for our `Dim_Table_SSIS`. This created issues during data upload from the CSV files, as the split files were not designed to include the `KeyID` column. Additionally, the KeyID column in the 'Dim_Table' and 'Dim_Table_SSIS' was configured to increment automatically starting from `(1,1)`, which caused mapping inconsistencies with the incoming data. These issues required careful adjustments to ensure proper alignment between the schema and the dataset.

6.1.2 Approaches

As explained in our **Data Uploading with Python** section, creating the original Dim_Table was achievable even though the split datasets lacked the KeyID column, as the column was already present in our server table but we managed it by initializing the key column and late it started generating in incremental order. Unfortunately, this approach did not work using SSIS. To resolve this, we decided to extract the required data, including all necessary attributes, directly from the server table into a CSV file for mapping purposes.



To accomplish this, we opened Visual Studio (VS) and created a new package in SSIS. We selected the **Data Flow Task** From the **Control Flow** and proceeded with the following steps:

1. In the **Data Flow** section, we added an **OLEDB Source** node from the SSIS Toolbox since we were connecting to a server.
2. We created a connection to the server and specified the table we wanted to fetch data from.
3. To transform the data and limit the dataset to 10% of the total records, we added a transformation node (such as a **Percentage Sampling, Row Sampling** or **Conditional Split**, depending on the specific requirements).
4. Finally, we configured the destination to output the selected and transformed data into a CSV file.

This approach allowed us to extract only the necessary data subset while ensuring the format and attributes aligned with our requirements for further processing and mapping.

After completing the extraction process, we proceeded with the same steps as before to populate the data into the duplicate Dim_Table_SSIS. Using the newly created CSV file, which contained the required attributes, we were able to efficiently map and upload the data into the table, successfully addressing the initial challenges.

This time, we did not need to use the "Percentage Sampling" node, as the CSV file created during the extraction process was already sampled at 10%. This streamlined the workflow and ensured that the **Dim_Table_SSIS** was populated accurately, aligning with the schema and data requirements.

However, after random sampling, we encountered an issue where the data no longer aligned perfectly across columns with matching IDs. This misalignment created difficulties in accurately connecting the datasets. Recognizing the importance of maintaining data integrity, we actively explored solutions to address this challenge.

6.1.3 Alternate Approach for Data Alignment and Consistency

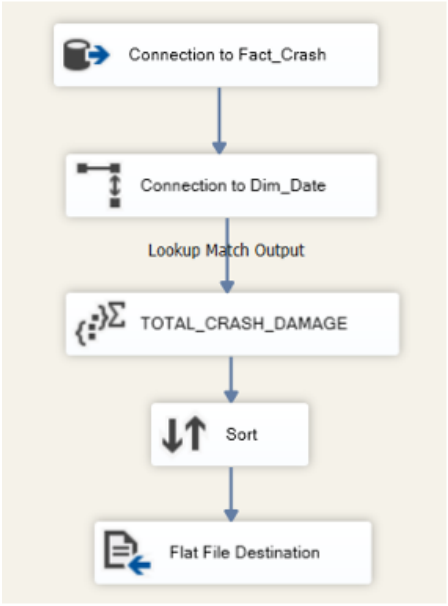
The random sampling approach initially attempted did not meet our goal of creating identical records and matching keys for `Dim_Table_SSIS` and `Fact_Crash_SSIS`. As a result, we reverted to our original method, using the SQL command `SELECT TOP 10 PERCENT * INTO Dim_Table_SSIS FROM Dim_Table ORDER BY Table_Keys` to populate `Dim_Table_SSIS`, and created `Fact_Crash_SSIS` by joining the top 10% of the related dimension tables. To ensure consistency, we checked for key mismatches between the two tables, and the query returned no discrepancies, confirming the alignment of records. Although we tested this method, we realized it wasn't the intended approach and decided to explore a more efficient solution using SSIS tool nodes.

6.1.3 Final Approach using SSIS

It's surprising how straightforward the solution was despite the time it took us to realize it. We configured an OLEDB Source, set the OLEDB Editor to **SQL Command** to query the top 10% from `Dim_Table`, and connected it to the `Dim_table_SSIS` destination. This is how we populated our SSIS tables. And Confirm the consistency of records by cross checking with fact_crash and Dim_Tables.

Assignment 6c

In this assignment, the task was to calculate the total crash damage costs for each incident per year. Below is a detailed breakdown of how the solution was implemented.



Step 1: OLE DB Source (Connection to Fact_Crash Table)

The first step in the ETL process was to connect to the Fact_Crash table in the source database using an OLE DB Source. This was essential for extracting the data relevant to crash incidents. The necessary columns such as RD_NO, DateKey, and DAMAGES were selected for further processing.

Step 2: Lookup Transformation (Connection to Dim_Date Table)

Next, a Lookup Transformation was applied, connecting the Fact_Crash table with the Dim_Date table. The goal was to bring in the Year column from the Dim_Date table.

Step 3: Aggregate Transformation

In the Aggregate Transformation, the data is grouped by RD_NO (Incident ID) and Year, and the DAMAGES column is summed up to calculate the total damage costs for each incident, categorized by year. This step allows for the aggregation of damage values for each incident across the different years. The output is stored as TOTAL_CRASH_DAMAGE.

Step 4: **Sort Transformation**

After the aggregation, a Sort Transformation is applied to the data. The dataset is sorted by the Year column in ascending order. Sorting helps in maintaining the correct chronological order of the crash incidents across different years.

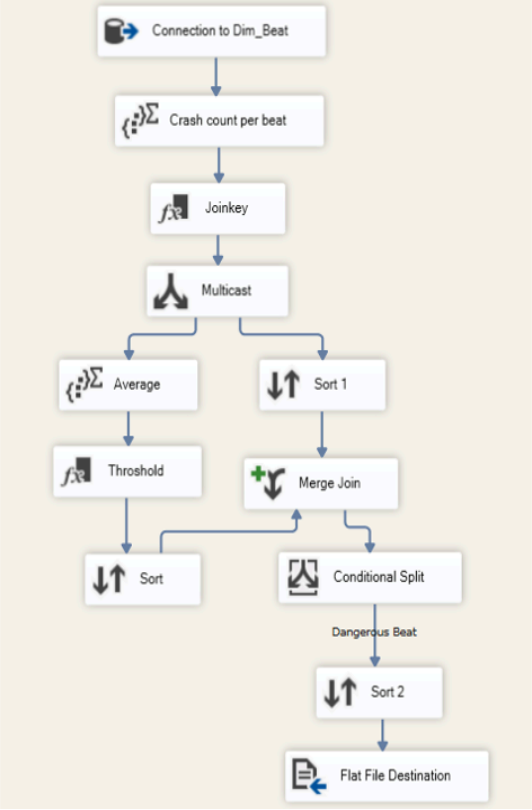
Step 5: Flat File Destination

The final output of the data flow is written to a Flat File Destination, where the results are stored. The output consists of the RD_NO (Incident Identifier), the Year, and the corresponding TOTAL_CRASH_DAMAGE for each incident. This file provides a summary of the total damage costs for each incident across the years.

Assignment 7c

A beat is classified as dangerous if the number of crashes within that beat exceeds the average number of crashes across all other beats by more than 10%. The task is to list all the beats that meet this criterion

---



Step 1: OLE DB Source (Connection to Fact_Crash Table)

The data flow starts by connecting to the Dim_Beat table, which holds information about different beats. The necessary columns RD_NO and BEAT_OF_OCCURANCE were selected for further processing.

Step 2: Aggregate Transformation (Crash count per beat)

The aggregation node counts the number of crashes occurring in each beat. This is a key step in determining which beats meet the dangerous classification.

Step 3: Derived Column Transformation (Join Key)

The Derived Column transformation is used after the aggregation step to create additional columns in the data. This column ensures that the dataset has identical structure for later merging in the Merge Join node.

Step 4: Multicast Transformation

The Multicast transformation is used to duplicate the data flow, allowing the same dataset to be used in the next aggregation branch node (to calculate average crashes) and the merge join node.

Step 5: Aggregation Transformation (Average)

The Aggregation transformation is used to compute the average number of crashes across all beats.

Step 6: Derived Column Transformation (Threshold Calculation)

The Derived Column Transformation is used to calculate the threshold for the number of crashes in a beat. This threshold is determined by multiplying the average number of crashes across all beats (calculated earlier) by 1.10. If a beat has a crash count greater than this threshold value, it is classified as dangerous.

THRESHOLD = AVERAGE_CRASH * 1.10

This calculation ensures that beats with crash counts exceeding 10% of the average are flagged as "dangerous."

Step 7: Merge Join Transformation

The Merge Join transformation is used to combine the data from two sources based on the join key. This transformation brings together the necessary columns for the final analysis. Specifically, it merges the crash count data with the calculated threshold, and the average crash data, ensuring that all the required columns are available in the final dataset.

Step 8: Conditional Split Transformation

The Conditional Split Transformation is used to route the data based on a condition. It checks if the CRASH_COUNT for each beat exceeds the Threshold (calculated earlier). If the condition CRASH_COUNT > Threshold is met, the data is sent to the "Dangerous Beats" output, identifying the beats with a crash count more than 10% higher than the average.

Step 9: Sort Transformation

A Sort transformation is used to organize the data. This sorting helps present the dangerous beats in an ordered manner, ready for final output.

Step 10: Flat File Destination (Final Output)

The processed data, containing the list of dangerous beats, is written to a flat file destination for final output. This is the report listing all the beats that meet the dangerous criterion.

Assignment 8c

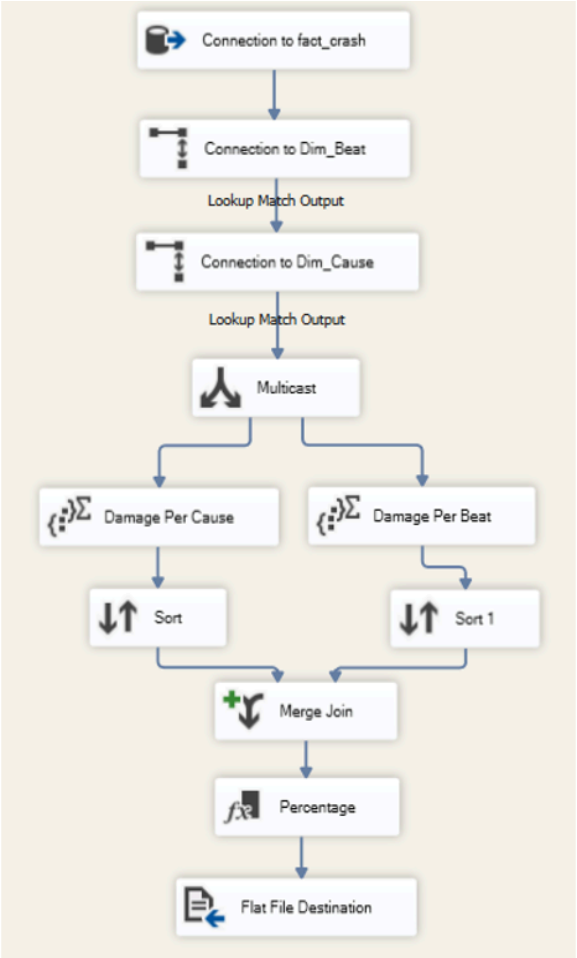
The task is to identify the primary cause behind each crash registered in a specific beat, calculate the damage caused by each of those causes, and determine the contribution of each cause to the total damage of the beat in percentage.

Step 1: OLE DB Source (Connection to Fact_Crash Table)

The flow begins by connecting to the Fact_Crash table, which contains the relevant crash data. The necessary columns such as BeatKey, CauseKey, and DAMAGES were selected for further processing.

Step 2: Lookup Transformations (Connection to Dim_Beat)

Lookup transformation is used to join the previously extracted data with the Dim_Beat table by using the BeatKey column. This enriches the data by adding the BEAT_OF_OCCURANCE for each crash.



Step 3: Lookup Transformations (Connection to Dim_Cause)

Another Lookup transformation is used to join the previously extracted data with the Dim_Cause table by linking the two tables using the Causekey column. This enriches the data by adding the PRIMARY_CONTRIBUTORY_CAUSE for each crash.

Step 4: Multicast Transformation

The Multicast transformation is used to duplicate the data, passing the same data from the source to two different branches for further processing. This allows the same dataset to be used in the next aggregation node without altering the original data.

Step 5: Aggregation Transformations: (Damage per Cause and Damage per Beat)

The first aggregation operation groups the data by PRIM_CONTRIBUTORY_CAUSE and BEAT_OF_OCCURRENCE, summing the DAMAGE for each cause within each beat. This gives the total damage caused by each primary contributory cause for every beat.

The second aggregation operation groups the data by BEAT_OF_OCCURRENCE and sums the DAMAGE for each beat. This gives the total damage for each beat, regardless of the cause.

Step 6: Sort transformation

A Sort transformation is applied to organize both the output of Damage Per Cause and Damage Per Beat data based on the BEAT_OF_OCCURRENCE column.

Step 7: Merge Join

A Merge Join transformation is used to join the two sorted datasets (Damage per Cause and Damage per Beat) based on the matching columns BEAT_OF_OCCURRENCE. This step combines the data from both sources for further processing.

Step 8: Derived Column Transformation (Percentage Calculation)

A Derived Column transformation calculates the percentage of total damage caused by each contributory cause. The expression used for this calculation is:

(DAMAGE_PER_BEAT != 0) ? (DAMAGE_PER_CAUSE / DAMAGE_PER_BEAT) * 100 : 0

This formula calculates the percentage contribution of the primary contributory cause to the total damage for each beat.

Step 9: Final Output - Flat File Destination:

The processed data is then written to a flat file destination for the final output.

Assignment 9c

Show the total damage cost of all dangerous beats and what percentage of the total crash damage costs they account for.

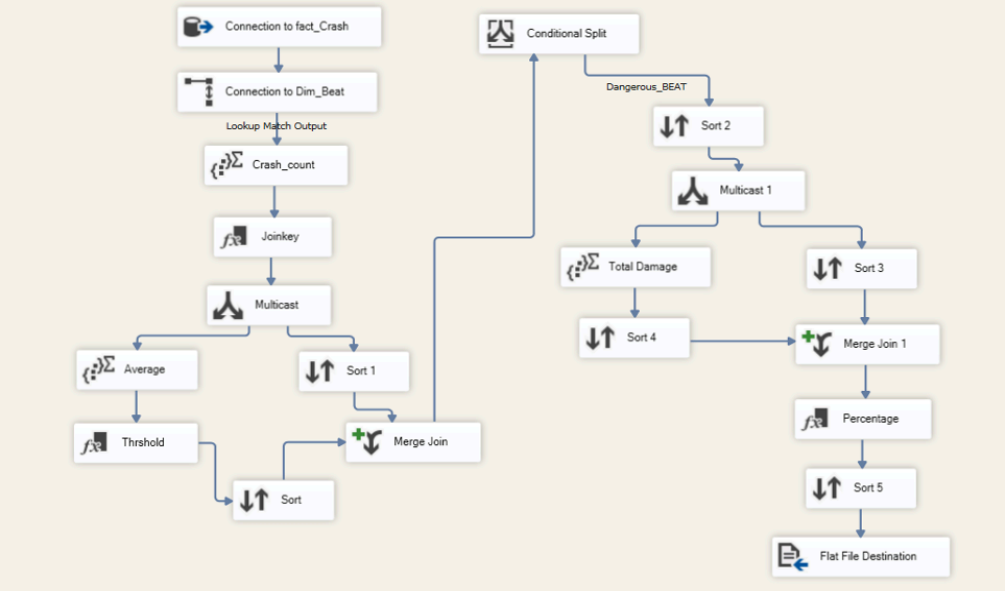
The question for Assignment 9c was developed to explore the total financial burden of dangerous beats and understand how significantly they contribute to overall crash damage costs. This question aims to consolidate the findings from previous tasks into a broader financial impact analysis.

Step 1: Data Extraction

The OLE DB Source was used to extract relevant columns from the fact_Crash table. A Lookup Transformation was used to extract the required columns from the Dim_Beat table, matching the crash data with the beat details.

Step 2: Crash Count and Total Damage Calculation:

An Aggregate Transformation was used to calculate both the total number of crashes per beat and the total damage cost per beat in parallel. Following this, a derived Column Transformation named **Joinkey** was used to create a common column to facilitate later joins. Multicast Transformation was applied to split the data into multiple streams.



Step 3. Threshold Calculation

The average number of crashes across all beats was computed using another Aggregate Transformation. A Derived Column Transformation calculated the threshold as:

$$\text{Threshold} = \text{AVG}(\text{CRASH\_COUNT}) * 1.10$$

Step 4. Identification of Dangerous Beats

A Conditional Split Transformation filtered rows where CRASH_COUNT > Threshold. Rows meeting this condition were categorized as Dangerous Beat.

Step 5. Total Damage Cost for All Dangerous Beats

The total damage cost across all dangerous beats was calculated using the results from the initial aggregation in step 2. After this, a **Merge Join Transformation** was applied to combine the total damage results with other relevant columns for further calculation.

Step 6. Percentage Contribution Calculation

A Derived Column Transformation calculated the percentage contribution of each dangerous beat to the total crash damage costs using the formula:

$$\text{Percentage} = (\text{Total Damage for Beat} / \text{Total Crash Damage for All Beats}) * 100$$

Step 7. Data Output

The results were sorted using a Sort Transformation and exported to a Flat File Destination for final output.

Conclusion

In conclusion, Part 1 of the project successfully established a solid data warehouse framework for traffic incident analysis. By carefully cleaning, organizing, and structuring the data, we ensured it was ready for deeper analysis in future stages. The use of Python for data preparation, batch insertion, and SQL Server Integration Services (SSIS) for efficient data processing allowed us to create manageable datasets, calculate key metrics, and address important assignments. This foundational work sets the stage for advanced decision-support tasks, such as identifying patterns and making data-driven insights into traffic incidents, ensuring the project progresses seamlessly into its next phases.

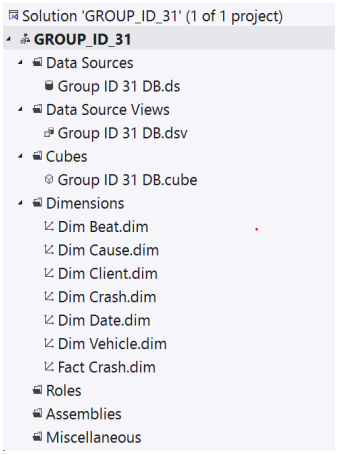
Part 2

Introduction

The second part of this project focuses on using SQL Server Analysis Services (SSAS) for multidimensional query analysis, enabling advanced insights and efficient data exploration through the design and analysis of data cubes. For this approach we use Visual Studio with Analysis service and created a new project Named Group_ID_31

Assignment 1

1.1 Connection To Data Source



In the Solution Explorer on the left side of our environment, after creating the Group_ID_31 project, the first step is to right-click on the project and select "Properties." From there, we change the deployment location from local to our server at: http://lds.di.unipi.it/olap/msmdpump.dll. This allows us to deploy all changes directly to the server. Additionally, we input our credentials for accessing the server. We use Analysis Services to establish the connection for this deployment.

1.1.1 Challenges & Insights

During the connection to the server, we encountered a 503 error. After testing various solutions, we discovered that connecting to the VPN prevented the connection. However, when we disconnected from the VPN, the connection was successful. This issue was both time-consuming and frustrating. Additionally, attempting to deploy without the VPN connection led to failure. Therefore, we realized that the VPN connection was required for deployment, but for running MDX queries on SQL Server, the VPN had to be disconnected.

1.2 Data Source View

After successfully connecting to the server and deploying our database, we navigated to the "View" section where our database was now ready to explore. Upon examining the dataset, we realized that additional columns were needed in our Dim_Date table. The months were in numeric format, but the days of the week were represented by names only. To improve the hierarchy and sequence, we decided to add two new columns: "MonthOfYear" and "DaysOfWeekInNum." By right-clicking and selecting "New Named Calculations," we were able to create these columns using a case expression to meet the required format.

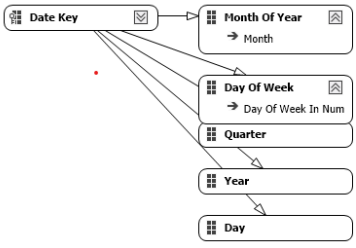
1.3 Creation of Dimensions

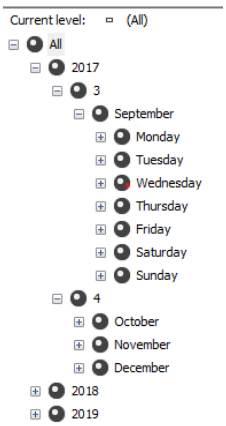
For cube creation, the first step is to define the dimensions with the proper hierarchy, as this forms the foundation for our data structure. These hierarchies can be adjusted at any time to

optimize performance and meet the query requirements. For our project, we selected the necessary dimensions, such as **[Dim_Date]**, **[Dim_Client]**, **[Dim_Causes]**, **[Dim_Vehicle]**, **[Dim_Beat]** and **[Dim_Crash]** , as these were essential for enhancing query performance and aligning with the analysis needs of the cube. By carefully selecting and organizing these dimensions, we ensured that the cube would provide accurate and efficient results for our analysis.

1.4 Defining Hierarchies

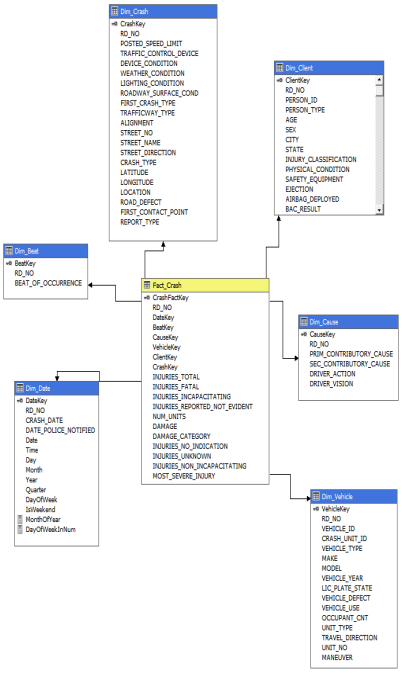
We created the data hierarchy as follows: **Year** → **Quarter** → **Month** → **DaysOfWeek** → **DateKey**, with additional columns **MonthOfYear** and **DaysOfWeekInNum** to ensure the sequence, as the original data did not follow a proper sequence. To achieve this, we adjusted the settings by changing the "Order By" option from **Key** to **AttributeKey** for better control over the hierarchy. Similarly, for the **[Dim_Client]** dimension, we established the hierarchy **State** → **City**, ensuring that the data was organized correctly for efficient analysis. These adjustments allowed us to maintain a logical sequence and optimize the performance of the cube.





1.5 Creation of DataCube

After creating the dimensions and setting the hierarchy, the next step was to create the cube. For this, we selected **Damage** as our measure from the **Fact_Crash** table, and added another new measure, **Fact_Crash_Count**, as per the assignment requirement. The dimensional tables we created before used to create the cube. This ensured that the cube would have the necessary measures to analyze both the total damage and the crash count data, fulfilling the specified criteria for the project.



Assignment 2

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

```
-- For each month, the total damage costs for each location
SELECT {[Measures].[DAMAGE]} ON COLUMNS,
  FILTER(NONEMPTY( CROSSJOIN([Dim Date].[Year].[Year],[Dim Date].[DayMonthYear].[Month Of Year],[Dim Client].[STATE].[STATE],
  [Dim Client].[Geography].[CITY])), [Measures].[DAMAGE] > 0 ) ON ROWS
FROM [Group ID 31 DB]
```

```
-- The grand total with respect to the location
WITH MEMBER GrandTotal AS ([Measures].[DAMAGE])
SELECT {GrandTotal} ON columns,
    nonempty([Dim Client].[STATE].[STATE]) on rows
from [Group ID 31 DB]
```

Assignment 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.

```
-- the average of these values across all crashes in a year
WITH MEMBER PersonCount AS COUNT(EXISTING [Dim Client].[PERSON ID].[PERSON ID])
MEMBER DamagePerPerson AS IIF(PersonCount > 0, [Measures].[DAMAGE] / PersonCount, NULL)
MEMBER AverageYearlyDamage AS AVG(EXISTING [Dim Crash].[RD NO].[RD NO], DamagePerPerson)
SELECT {[Measures].[DAMAGE], PersonCount, DamagePerPerson,AverageYearlyDamage} ON COLUMNS,
    ([Dim Date].[Year].[Year].MEMBERS) ON ROWS
FROM [Group ID 31 DB]
```

Assignment 4

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

```
WITH MEMBER DamageChange AS
[Measures].[DAMAGE] - (PARALLELPERIOD([Dim Date].[DayMonthYear].[Year],1,[Dim Date].[DayMonthYear].CURRENTMEMBER),
SELECT  {[Measures].[DAMAGE],DamageChange} ON COLUMNS,
FILTER(NONEMPTY(CROSSJOIN([Dim Date].[DayMonthYear].[Year],[Dim Client].[STATE].[STATE])),
[Measures].[DAMAGE] <> 0 AND NOT UNKNOWN) ON ROWS
FROM [Group ID 31 DB]
```

While solving this query, we faced an issue with the `Format_String = "Percentage"` command, which was intended to convert the output into a percentage format. However, executing this command resulted in an error, preventing us from achieving the desired output format.

Assignment 6

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

```
WITH MEMBER MaxDamage AS MAX([Dim Client].[Person ID].[Person ID].MEMBERS, [Measures].[DAMAGE])
MEMBER TopPerson AS TOPCOUNT([Dim Client].[Person ID].[Person ID].MEMBERS, 1,[Measures].[DAMAGE]).ITEM(0).NAME
SELECT {[Measures].[DAMAGE],TopPerson, MaxDamage} ON COLUMNS,
    NONEMPTY(CROSSJOIN( [Dim Date].[Year].[Year].MEMBERS,[Dim Vehicle].[VEHICLE TYPE].[VEHICLE TYPE].MEMBERS)) ON ROWS
FROM [Group ID 31 DB]
```

Assignment 8

For each year, show the most frequent cause of crashes and the corresponding total damage costs. The primary crash contributing factor is given twice the weight of the secondary factor in the analysis. Additionally, show the overall most frequent crash cause across all years.

```
WITH
-- Weighted Frequency Calculation
MEMBER WeightedFrequency AS ([Dim Cause].[PRIM CONTRIBUTORY CAUSE].CurrentMember, [Measures].[Fact Crash Count]) * 2
+([Dim Cause].[SEC CONTRIBUTORY CAUSE].CurrentMember, [Measures].[Fact Crash Count])
-- Total Damage Cost Calculation
MEMBER [Measures].[TotalDamageCost] AS SUM([Dim Cause].[PRIM CONTRIBUTORY CAUSE].CurrentMember,[Measures].[DAMAGE])
SELECT {[Measures].[WeightedFrequency], [Measures].[TotalDamageCost]} ON COLUMNS,
NON EMPTY ([Dim Date].[Year].[Year].MEMBERS *[Dim Cause].[PRIM CONTRIBUTORY CAUSE].MEMBERS) ON ROWS
FROM [Group ID 31 DB]
```

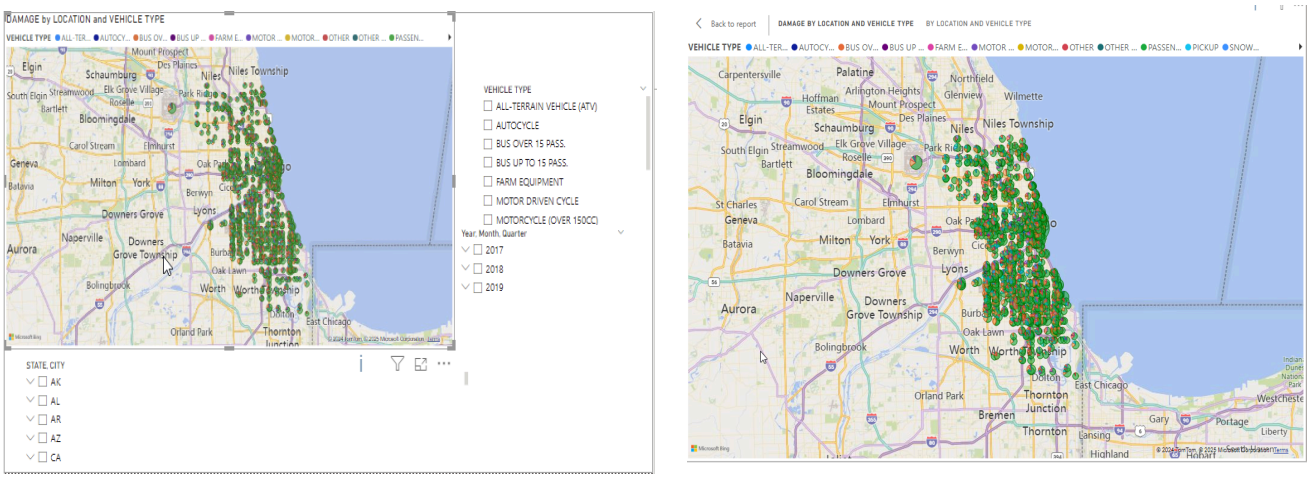
BI Dashboard and Data Visualization

The BI part of the project focuses on converting raw data into meaningful visual insights using **Power BI**. The **Fact_Crash** table serves as the central hub, connected to dimension tables like **Dim_Vehicle**, **Dim_Client**, **Dim_Date**, **Dim_Cause**, **Dim_Crash**, and **Dim_Beat** based on the schema. Relevant columns were selected to meet visualization requirements, enabling the creation of interactive dashboards that reveal trends and support decision-making.

Assignment 9

Create a dashboard that shows the geographical distribution of the total damage costs for each vehicle category

---



MAP VISUALIZATION

9.1 DAMAGE by LOCATION and VEHICLE TYPE (Map Visualization)

9.1.1 Visualization Process

The process integrates **Fact_Crash** with **Dim_Crash** and **Dim_Vehicle** to map crash records by location and vehicle type. Data is aggregated by **LATITUDE**, **LONGITUDE**, and **VEHICLE_TYPE**. Using Power BI, crashes are plotted on interactive maps with filters for vehicle type and zoom functionality, enabling detailed analysis of crash densities and patterns.

9.1.2 Purpose

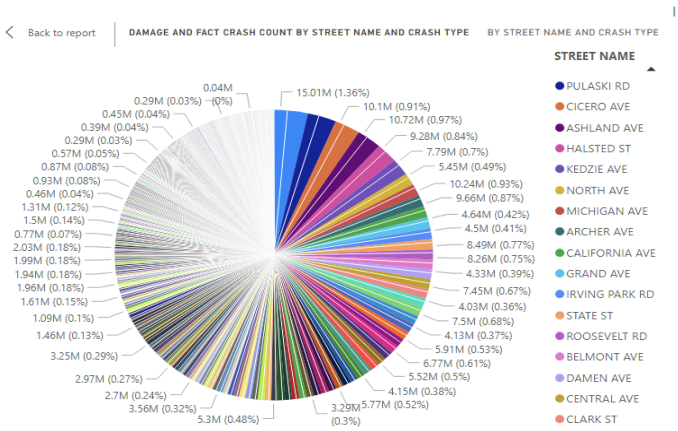
This map identifies high-density crash areas and frequently involved vehicle types, offering insights into urban risk zones and vehicle-specific trends.

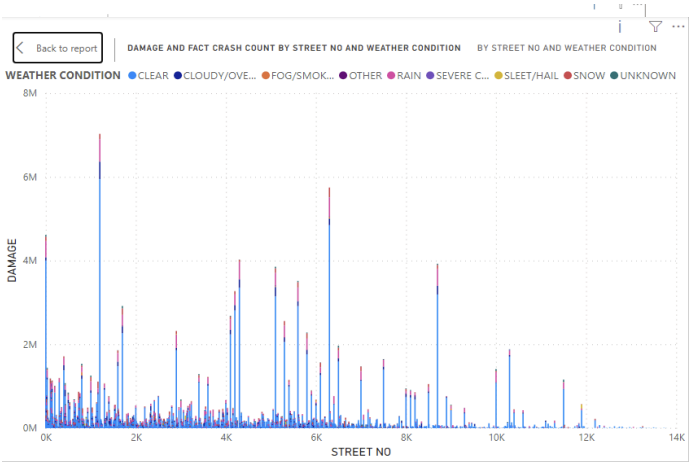
9.1.3 Output

The map highlights the very interactive graph of Chicago as a crash hotspot. Passenger vehicles are most commonly involved, while motorcycles and buses show clustered incidents in certain high-risk zones.

Assignment 10

Create a plot/dashboard that you deem interesting w.r.t. the data available in your cube, focussing on data about the street.





PIE CHART **BAR CHART**

10.1 DAMAGE and Fact Crash Count by STREET NAME and CRASH TYPE (Pie Chart)

10.1.1 Purpose

This visualization displays crash counts and associated damage by street name and crash type, providing insights into which streets experience the highest number of incidents and the types of crashes that occur most frequently. By analyzing this data, the chart highlights high-risk streets and common crash patterns, aiding in targeted interventions.

10.1.2 Output

The pie chart reveals that major streets like **Western Ave**, **Pulaski Rd**, and **Cicero Ave** report the highest number of crashes, accounting for a significant portion of total incidents. The bar chart further highlights that certain crash types, such as rear-end collisions and turning-related crashes, dominate specific streets. High-damage incidents are clustered around high-traffic areas, indicating zones that require enhanced traffic control and safety measures.

10.2 DAMAGE and Fact Crash Count by STREET NO and WEATHER CONDITION (Bar Chart)

10.2.1 Purpose

This bar chart illustrates the relationship between crash count and damage in correlation with street numbers and varying weather conditions. By visualizing this data, the chart highlights the influence of weather on crash severity and frequency across different streets.

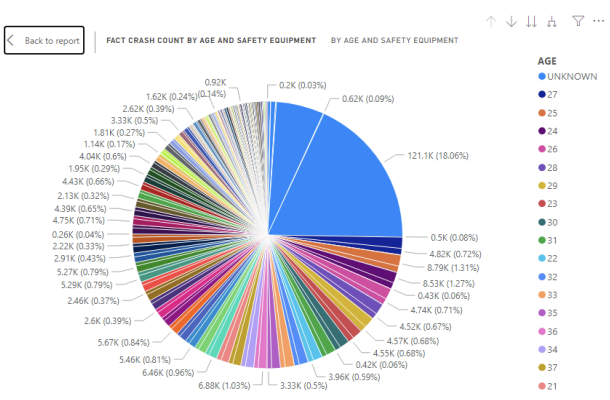
10.2.2 Output

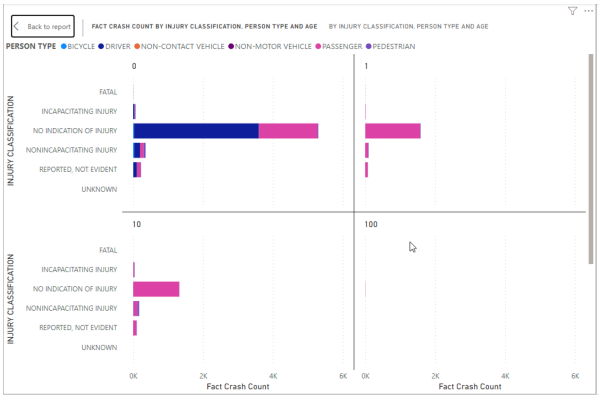
The visualization reveals that most crashes occur during clear weather, resulting in the highest crash counts and damage across many streets. Significant peaks in damage are observed around specific street numbers, such as 5000 and 10000, indicating areas with frequent or severe accidents. Although crashes under rainy and foggy conditions occur less frequently, they still contribute notably to overall crash data.

Assignment 11

Create a plot/dashboard that you deem interesting w.r.t. the data available in your cube, focussing on data about the people involved in a crash

---





PIE CHART **BAR CHART**

11.1 Fact Crash Count by AGE and SAFETY EQUIPMENT (Pie Chart)

11.1.1 Purpose

The pie chart visualizes the distribution of crash counts across different age groups and highlights the usage of safety equipment during these incidents. This visualization provides insight into how safety measures correlate with crash frequency across various demographics, helping to identify trends and areas for potential intervention.

11.1.2 Output

The pie chart reveals that younger age groups, especially those between 20 and 35, account for a significant portion of crash incidents. A considerable percentage of individuals within these age groups are recorded as not using safety equipment, contributing to higher injury rates. The “UNKNOWN” age category also shows a notable crash count, emphasizing the need for more comprehensive data collection. Conversely, older demographics show lower crash counts, but with relatively higher safety equipment usage, indicating greater adherence to safety measures.

11.2 Fact Crash Count by INJURY CLASSIFICATION, PERSON TYPE, and AGE (Bar Chart)

11.2.1 Purpose

This bar chart analyzes crash counts by injury classification, person type (such as drivers, passengers, or pedestrians), and age. The goal is to provide an overview of how different types of injuries are distributed

across various age groups and roles in crashes. By identifying trends, it becomes easier to focus on high-risk populations and implement targeted interventions.

11.2.2 Output

The bar chart reveals that drivers constitute the majority of crash victims across all age groups. Pedestrians and passengers represent a smaller but significant portion, particularly in cases of incapacitating or fatal injuries. Young adults and elderly individuals are more susceptible to severe injuries, while middle-aged drivers experience a higher number of non-incapacitating crashes.