UNIVERSITÀ DI PISA

**Laboratory of Data Science**

Group_10 Project - Part 1

---

Assignments:

# *Data preparation, warehousing and population*

---

**Authors**

Bernardo D'Agostino
Luca Coda-Giorgio

*Data Science & Business Informatics*

Academic Year 2023/2024

## 1 Introduction

For this part of the project, we had to create a Data Warehouse from a collection of files given in several formats, specifically CSV, XML and JSON. The whole collection regarded criminal activities in the United States stored in a single CSV file which we had to separate into different dimensional tables following a given Data Warehouse star schema.

We then had to create the aforementioned schema, once implemented the appropriate design solution, in SQL Server, making use of SQL Server Management Studio. In the end, we had to upload the tables to the DBMS. For every partial step of the pipeline, we created a set of functions for the specific tasks at hand and python programs implementing these last.

## 2 Tables creation pipeline

### 2.1 Data preparation and splitting

**Crime Gravity** The first thing we did was to read the Police.csv file and then calculate the Crime Gravity measure for each row. To do so we created the `add_crime_gravity()` function which takes as input a list of lists containing each row of the CSV and then uses the information contained in the JSON files to do a simple multiplication and append the computed measure to each row.

**Creation of the tables from Police.csv** We define the function `add_unique_ids()` to partition the *Police* table into *Gun*, *Geography*, and *Participant* tables. This function accepts several parameters: A list of lists, where each sublist represents a row from a data frame. A list of column names for the new table. The name of the key, created for maintaining a lossless join between the original and new tables. The file path for storing the newly created table.

The function executes the following steps: Initialize the new data frame (*df*) with the first row containing the column names and a new identifier (*id*). Create an empty dictionary, mapping column values to their respective *id* values. Iterate over each row of the original *df*, performing the following actions: Create a row for the new *df* using the values from the specified columns. Check if these values exist in the dictionary: If not, append the row to the new table and the dictionary, assigning a new *id*, and add this *id* to the original *df*. If yes, retrieve the *id* from the dictionary and add it to the original *df*. Saved the new *df* as a CSV and kept the old *df* as a list of list with the modified data

(The first version of our function used a list instead of a dictionary but we found it to be excessively slow computationally)

After applying the function to create the Gun, Geography, and Participant the remaining list of lists was saved as the Custody table.

**Date.csv** In order to create the Date dimension's table, since the original data was formatted in XML (element format), we first needed to develop functions (`functions_xmlToCsv.py`) to convert such representation to the more manageable CSV format, therefore also making it coherent with the ones used for the other tables.

Furthermore, the 1633 dates.xml *Element*s had sub-elements, interpreted as attributes, *date* and *date_pk*, of which the first' values included not only the year, month and day, but also the timestamps. Since they all had the same times, we decided to exclude them from the final values. As for the primary key (*date_pk*), we decided to rename it as *date_id* to make it consistent with the other PKs|FKs.

Consequently, we used the `xml.etree.ElementTree` API module for parsing the data, exploiting its *Element* object attributes, *tag* and *text*, to extract respectively each attribute's name and values.

The main function reads the XML file in input and opens a new output CSV file in writing mode, instantiates an `ElementTree` object and gets its root, which is necessary. It then checks if the file has any rows and if this is true, gets the attributes' name and values and writes them in the output file using the `csv.writer` writerow method. The header (list of attribute names) gets written first, then all the subsequent lines (lists of ordered values based on the header).

**Incident** In this case, even if it was present in the star schema followed to construct the tables, a separate dimensional table was useless since the only attribute was the *incident_id* PK, therefore we treated it as a degenerate dimension and included it in the fact table *Custody*. Such attribute serves the purpose of selecting *custodies* that happened in the same incident.

## 2.2 Dimensional attributes and hierarchies

**Date attributes.** Once obtained the *dates.csv* temporary file, we proceeded to add the following derived attributes: *date* (converted to `datetime.date` format to be able to apply specific SQL functions in the future analyses), *day*, *month* and *year* as integers in the form YYYYMMDD, YYYYMM and YYYY in order to represent the hierarchical relationship between them, *quarter* as a string YYYY"Q"Q and *weekday* as capitalized strings.

As a means to do so, we defined functions taking as parameters the temporary *dates.csv*'s *date* as string, and converting it to the aforementioned formats, as explained in `functions_create_table.py` documentation, making use of the `csv.DictReader|DictWriter` objects. To retrieve the values of the *weekday* attribute, we made use of the `datetime.date.strftime` method applied to a list of parameters obtained directly from *date*.

**Geography attributes.** In this case, to add the attributes we needed to design the DW schema (*city*, *state* and *country*, all of type string) we defined and exploited a set of functions.

One to retrieve the values for such attributes given the *latitude, longitude* through the use of external sources accessed with the `"3geonames.org"` geolocation API and the `request` native python module. The results of the requests were given in JSON format, therefore we looked up the appropriate keys to retrieve the needed data.

Once the new dimensional table was obtained making use of the `csv.DictReader|DictWriter` objects, even if no missing values were found, we proceeded with a data cleaning process involving the removal of some useless information in the values of the *city* and *state*, which were in the form of text between parentheses. We also removed commas inside the attributes' values if found, to remove complexity from the following tasks. We also found that the only distinct *country* value was "US".

# 3 Data Warehouse schema

In this section we will explain how we exploited the functionalities of *SQL Server Management Studio (SSMS)* to create the Data Warehouse schema in order to then proceed to populate it with the data extracted and integrated in Section 2.

The first step of this process consisted of accessing the *SSMS* server's database engine through the use of our SQL server authentication credentials, specifying the server's name (`lds.di.unipi.it`) and therefore accessing the correct database (`Group_ID_10_DB`). Before doing so, we needed to connect to the University of Pisa's *VPN* making use of *Sonic Wall Connect Tunnel*.

## 3.1 Dimensions' schemas

We began creating the schemas for the *Date*, *Geography*, *Gun* and *Participant* dimensional tables. In the following, we will list for each dimension the attribute names and their respective *T-SQL* types. `NOT NULL` was set as constraint for each dimensional attribute.

**Date** (*date_id*, int), (*date*, date), (*day*, int), (*month*, int), (*year*, int), (*quarter*, char(6)), (*weekday*, varchar(9)).
**Geography** (*geo_id*, int), (*latitude*, float), (*longitude*, float), (*city*, nvarchar(50)), (*state*, nvarchar(50)), (*country*, nvarchar(50).
**Gun** (*gun_id*, int), (*gun_stolen*, varchar(50)), (*gun_type*, varchar(50)).
**Participant** (*participant_id*, int), (*participant_age_group*, varchar(50)), (*participant_gender*, varchar(50)), (*participant_status*, varchar(50)), (*participant_type*, varchar(50)).

We then proceeded to set **_id* as PK for each table.

## 3.2 Fact schema

We then constructed the fact table's schema, *Custody*, setting as PK *custody_id* (type int), *crime_gravity* (type int) as measure and a FK attribute for each aforementioned dimension. As said before, *incident_id* was set as an attribute in this table and `NOT NULL` was also set as constraint for each attribute.

As a means to establish relationships between the tables, we used `relation` SSMS tool connecting each PK of the dimensions and respective FK in the fact table, named with the form `FK_"fact"_"dimension"`.

## 4 Data Warehouse population

**Connecting to the DBMS** To upload our data in the final CSV to the Data Warehouse we first created a connection to the DBMS using the `pyodbc` module.

**Table information** We then created the `get_tables_col_types()` function that has the connection as input and which runs the query `"SELECT table_name FROM information_schema.tables WHERE table_type='BASE TABLE'"` to get the table names in the dataset. It skips the first element and then runs for each table the additional query `f"SELECT column_name, data_type FROM information_schema.columns WHERE table_name = 'table'"` to get the column names and types for each table and storing them into a list returned by the function.

**Upload the data** We introduced the `upload_table()` function, which requires the following inputs: The name of a table, utilized for constructing the path to the corresponding CSV file. The connection to the database. The column names and types for the specified table in the Database Management System (DBMS).

The function operates as follows: Iterates over all rows in the CSV file. Compares the order of columns in the CSV with that in the DBMS table. Reorders the data using the `reorder_values()` function. Checks the datatype of each column. For types in {'nvarchar', 'date', 'char', 'varchar'}, it performs the following transformations: Converts each apostrophe (') into double apostrophes ("). Encapsulates the string as `f"'values[i]'"`. Constructs a single string by joining all values, separated by a comma, and adds this to a list named *values_list*.

Once the length of *values_list* reaches 1000: Joins all strings in the list, separated by commas. Forms a query in the format `"INSERT INTO {table_name}({columns_s}) VALUES {values_str}"` and executes it. After processing the last row of the CSV, uploads any remaining values. Finally, the function is used to create and execute queries for each table, followed by committing these changes and closing the database connection.