COMP63301-Data Engineering Concepts Coursework 2

Instructions

This coursework is **100% summative**, meaning it is designed to support your learning and skill development.

- **Marks awarded** will **contribute** to your final course grade.
- You are strongly encouraged to engage fully with the tasks, as they align closely with the learning outcomes of the module and will prepare you for other summative assessments.

This coursework addresses key activities within the Data Engineering Lifecycle, specifically:

- Data Cleaning through Value Imputation: assessing the suitability of multiple techniques for filling missing values, based on the properties of the data, to enhance data completeness, standardisation and interpretability.
- Data Summarisation: Using SQLite SQL to improve your understanding of the data and the limitations of specific techniques for a given data context.

You will work with refined, modelled data stored in an SQLite database that is, nonetheless historical and time series, applying SQL queries embedded in Python to explore, clean, and summarise the data.

All tasks must be completed using **SQLite SQL embedded within Python**. You are expected to write SQL queries and execute them through Python, rather than using Python code to replicate functionality that is already available in SQL. In other words, use SQL for operations such as data selection, filtering, aggregation, manipulation, etc.. Python should primarily serve as the interface for executing SQL queries and handling the results—not as a substitute for SQL logic. Failure to adhere to this requirement may result in reduced marks, as the purpose of this coursework is to assess your ability to work effectively with SQL within a Python environment. Minimum Python version: 3.10 or higher.

You can use the ONLY the Python libraries specified in the code stub for this exercise.

You are required to complete **eight data-related requests**, each designed to simulate a real-world data engineering scenario. The requests involve reading and understanding the dataset, while applying appropriate Python techniques.

Submission Requirements

You are required to submit the single Python code stub provided for this exercise. This file will serve as the container for your solutions to each of the data requests outlined in the coursework.

- Please ensure that your responses are written directly within the designated sections of the code stub, following the instructions and comments embedded in the file.
- Do not rename or restructure the file, as it is configured to work with our Gradescope automated marking system.
- The submission link is provided on our Canvas space (where this coursework information is provided).

You may submit this coursework multiple times. However, we **strongly recommend** that you achieve a **successful submission**—one that is fully processed and marked by our **Gradescope automarker**, with feedback provided—**well in advance of the deadline for the coursework!**
This will allow you to **avoid a ZERO mark for the coursework!** And, also, to:

- **Practise submitting your coursework** in advance of the deadline on the Gradescope platform.
- Ensure your submission is in the **correct format** (i.e., the original stub with your completed code).
- **Submit as many times as needed**—only your most recent submission will be considered.
- Aim to achieve a **successful submission** (i.e., one that is fully marked by the Gradescope automarker and returns feedback) **well before** the deadline for the coursework.
- You avoid technical issues or delays close to the deadline.

Need Help?

If you experience any issues with the code stub, the automarker, or the submission process

- Post your question in the course discussion forum.
- Attend the laboratory drop-in session of this week.

Exercises

Important Coursework Notes

- **Interdependency of Questions**
  Please be aware that any changes made to the database in response to one question may affect the queries and results in subsequent questions. You are advised to plan and execute your changes carefully, keeping in mind the cumulative impact across the entire coursework.
- **Output Format Requirement**
  The output for each question **must strictly follow** the format specified in the code stub provided with this coursework. Failure to comply with the required structure will result in a mark of **zero** for that question.

Relational Data Cleaning and Value Imputation using SQL

1. The `trip` table contains a column named `end_location_name`, which includes missing (null/NaN) entries.
   Now, consider the following *'filling-missing-value'* strategies*:*
   i. Replace each missing `end_location_name` value in the table with the placeholder string `'Stop St.'`, which serves as a meaningful default value.
   ii. Replace each missing `end_location_name` value in the table with the most frequent `end_location_name` value in the table.
   iii. Replace each missing `end_location_name` value in the table with the corresponding `start_location_name` value, if available. If not available, then use replacement strategy *ii*.

   Your task is to:
   a. Identify all missing values in the whole `end_location_name` column, saving the `trip_id` associated with the first 10 rows with a missing value for `end_location_name`.
   b. Replace each missing entry in the entire `end_location_name` column, using each of the three strategies described above, one-at-a-time. You will need to **restore** each of the rows that had the value of `end_location_name` filled to its state **before its `end_location_name` value was filled** (i.e., leaving it again with a missing `end_location_name`), before applying the next strategy. Note that, after applying the last strategy (strategy iii), you should NOT restore the missing values in `end_location_name`, leaving them filled with values according to strategy iii.
   c. For each strategy and for each of the first 10 replacements, compute and return the total number of occurrences of the associated, new, pair `<start_location_name, end_location_name>`.

   Important:

   - Your final output should be 10 integers representing the requested count for each of the relevant 10 pairs, per strategy. So, in total, 30 integers.
   - Use the following column names to structure your output (**note that, for this and all the following questions, the formatting of your output is defined in the code stub provided with this coursework**), ordering the output by showing, firstly, the 10 integer-results obtained by applying strategy *i*, then strategy *ii* and, finally, strategy *iii*.

     Column names to be used:
     `trip_id, start_location_name, end_location_name, pair_count.`

   (20/100)

2. The `trip` table contains a column named `start_location_name`, which includes missing (null/NaN) entries.
   Your task is to:

a. Identify all missing values in the whole `start_location_name` column, saving the `trip_id` associated with the first 6 rows with a missing value for `start_location_name`.
b. Replace each missing entry in the whole `start_location_name` column with the most frequent `start_location_name` value in the table.
c. For each of the first 6 replacements, compute and return the total number of occurrences of the associated, new, pair <`start_location_name`, `end_location_name`>.

Important:

- Your final output should be 6 integers representing the requested count for each of the relevant 6 pairs, ordered as they appear in the table.

  Use the following columns names to structure your output, as specified in the code stub provided for this coursework:
  `trip_id, start_location_name, end_location_name, pair_count.`

(10/100)

3. Standardise the naming convention of the `DURATION` column to align with typical lowercase formatting used in data schemas, by having this column name replaced with `duration`. Once the standardisation is performed, the result to be returned should include the (new) column name and the first 5 values in this column, structured as specified in the code stub for this coursework.

   (5/100)

4. Standardise the values in the `month` column so that all entries are in lowercase (i.e., *may*), regardless of their original format. Once the standardisation is performed, return, for only the following columns, the first 5 data values. Note that the column names should appear in the results (which should be formatted as specified in the code stub for this coursework): `month`, `trip_id`, `duration`.

   (5/100)

5. For rows in the `trip` table where the `duration` column contains missing values (i.e., `NaN`), excluding entries where the value is `'00:00:00'`, compute the correct duration using the `started_at` and `ended_at` columns.

   After filling in the missing durations, return:

   - The total number of remaining `NaN` values in the `duration` column (this count should be returned as a single number, without column headers/labels).

- The `trip_id` and `duration` values for the following specific `trip_id`s: `1821126`, `1821158`, `1821204`, `1821289`, and `2047623`. The column names should appear in the output (`trip_id, duration`).
- Don't forget that the output should be formatted as specified in the code stub for this coursework.

(10/100)

Understanding the Data

6. Identify the shortest trip duration recorded in the database. Then, determine how many unique `<start_location_name, end_location_name>` pairs are associated with this specific duration value.
   Important: The results must be returned without any column names or labels—only the raw values should be included.

   (10/100)

7. Calculate the median trip duration from the database. Then, determine how many trips have this exact duration value.

   Important: The results must be returned without any column names or labels—only the raw values should be included.

   (10/100)

8. Using the Haversine formula and the coordinate data available in the `geocode` table, calculate the distance in miles between the `start_location_name` and `end_location_name` for each of the following `trip_id`s: `1637299`, `1640777`, and `1652952`.
   Important: The results must be returned without any column names or labels—only the raw distance values should be included. SQL should be used in the calculation. It is worth pointing out that the output should retain the full floating-point precision as computed by SQLite. So, do not use apply rounding strategies or formatting over the floating numbers.

   (30/100)