DATA ENGINEER CHALLENGE

Task 1 - Joining Data Sets

For the task, assume that we have a database with the following schema:

```
CREATE TABLE transactions (
    transaction_id          UUID,
    date                    DATE,
    user_id                 UUID,
    is_blocked              BOOL,
    transaction_amount      INTEGER,
    transaction_category_id INTEGER
);

CREATE TABLE users (
    user_id   UUID,
    is_active BOOLEAN
);
```

Example data for these tables is stored in the corresponding CSV files `transactions.csv` and `users.csv`, which can be generated from the `generate_data.py` script.

We want to compute the result of the following query:

```
SELECT
  t.transaction_category_id,
  SUM(t.transaction_amount) AS sum_amount,
  COUNT(DISTINCT t.user_id) AS num_users
FROM transactions t
JOIN users u USING (user_id)
WHERE t.is_blocked = False
  AND u.is_active = 1
GROUP BY t.transaction_category_id
ORDER BY sum_amount DESC;
```

But unfortunately the query planner of our database can not optimize the query well enough in order to get the results.

Your task is now to write a Python program using only the Python Standard Library[1,2] (preferably not using external libraries at all), which reads data from CSV files `transactions.csv` and `users.csv`, and computes the equivalent result of the SQL query in an *efficient way* that would be scalable for large data sets as well. The result should be printed to stdout.

Please note that the scope of the task is **not** to parse the SQL query or to generalize the computation in any way, but only to write a program which computes the result of this one specific query in an *efficient way*.

Reviewing this task, we pay special attention not only to the *correctness* of results, but especially to the *code quality* and *efficiency* of the data structures and algorithms used.

Remember that it is easier for us to review your task if we can test & run it. Providing Dockerfile and/or Makefile would be helpful (but not strictly required).

[1] Please note that *pandas* is not part of the Python Standard Library
[2] Let's not consider *sqlite* as Standard Library for this exercise

Task 2 - Feature Table Computation

We want to compute an ML feature table for the transactions. For every transaction of a user it should compute the number of transactions the user had within the ***previous*** seven days. So the resulting table should look somehow like the following example:

transaction_id	user_id	date	# Transaction within previous 7 days
ef05-4247	becf-457e	2020-01-01	0
c8d1-40ca	becf-457e	2020-01-05	1
fc2b-4b36	becf-457e	2020-01-07	2
3725-48c4	becf-457e	2020-01-15	0
5f2a-47c2	becf-457e	2020-01-16	1
7541-412c	5728-4f1c	2020-01-01	0
3deb-47d7	5728-4f1c	2020-01-12	0

Your task is now to write an SQL query, which computes this table based on the transactions table from the described schema.

Also elaborate (in a free text form) what happens in the database server under the hood when one executes a query. What would a database engine consider to execute your query effectively?

Please note that the day of the current transaction shouldn't be included in the calculation. Postgresql or ANSI Compliant SQL implementation is preferred but not strictly required.

Task 3 - Dimension Deduplication

Given the table **`dim_dep_agreement`**, containing information about the attributes of a deposit agreement and the history of their changes. A new row in the table is created if at least one of the 3 business attributes (`client_id, product_id, interest_rate`) has changed for a given agreement (`agrmnt_id`).

sk	agrmnt_id	actual_from_dt	actual_to_dt	client_id	product_id	interest_rate
Surrogate Key	Agreement ID	Row effective date	Row end date	Client ID	Product ID	Interest Rate
1	101	2015-01-01	2015-02-20	20	305	3.5%
2	101	2015-02-21	2015-05-17	20	345	4%
3	101	2015-05-18	2015-07-05	20	345	4%
4	101	2015-07-06	2015-08-22	20	539	6%
5	101	2015-08-23	9999-12-31	20	345	4%
6	102	2016-01-01	2016-06-30	25	333	3.7%
7	102	2016-07-01	2016-07-25	25	333	3.7%
8	102	2016-07-26	2016-09-15	25	333	3.7%
9	102	2016-09-16	9999-12-31	25	560	5.9%
10	103	2011-05-22	9999-12-31	30	560	2%

However, due to an error in the ETL process, redundant change records were inserted into the table (when in fact, none of the attributes changed). In the above table's sample of 10 records, such a redundancy can be observed within records 2, 3 and 6, 7, 8.

Your task:

1. Write an SQL script to create a new table **`dim_dep_agreement_compacted`**, where all such redundant records will be "collapsed". For example, instead of records 2-3 there should be one record with period `2015-02-21 - 2015-07-05`. Note: it should be a single statement, i.e. without using intermediate/temporary tables, updates or deletes.

Resulting table should have “smooth history” for every agreement (`agrmnt_id`) - without any gaps or intersections in row validity intervals (`actual_from_dt - actual_to_dt`).

2. Describe script logic in detail.

Please provide a Python script to solve Task 1 and SQL queries for Task 2 and Task 3. Put any free text answers or additional explanations into a README.md file.

Please don't hesitate to contact us in case something needs to be clarified.