The aim of the third project is to obtain a better understanding in database optimization. It is highly recommended that you read the instructions **fully** before starting with the project.
The deadline of the project is **Friday April 26, 2024**. You will need to submit your work on INGInious. Once the submission interface is available, this will be announced on Moodle. You are allowed to work in groups of up to 2 students.

Instructions

In this project, you are taking the role of a database administrator at a company that sells a small set of 6 products world wide. You are responsible for maintaining the central database of the company, which is used to provide information to the management of the company. The central database of the company consists of two important components:

- geographical data, which was obtained from the `mondial` database that was also used in the first project;
- purchase data for each product world wide.

The purchase data is forwarded to you every 10 days, by another database administrator. This administrator provides you a single SQLite database file with one table named `Purchases`. Every 10 days, you are required to:

- add this data to the central database;
- run a number of queries on the updated database.

Your task is to organize the database such that every ten days:

- the data can be added to the database as quickly as possible;
- the answers to the queries are calculated as quickly as possible.

You are allowed to *add* tables to the database that store the output of queries. This can be helpful to make the subsequent execution of other queries faster.
The sections below will describe in more detail:

- the data that will be provided to you;
- the queries you will need to write for this data;
- the methods used for evaluating your queries;
- the recommended steps for working on this project.

Data

The first part of data consists of the Mondial database that was also the basis for the first project. You may consult the documentation of the first project for more information. Note that in this project we will not be using the Covid-related data.
The second part of data consists of purchase data. The purchase data is included in a table with the following schema:

```
CREATE TABLE Purchases (
  id INTEGER PRIMARY KEY,
  time INTEGER NOT NULL,
  province INTEGER NOT NULL,
  product INTEGER NOT NULL,
  qty INTEGER NOT NULL
);
```

The interpretation of the attributes is as follows:

- `id` is an identifier for the purchase;
- `time` indicates the day of the purchase;
- `province` indicates the province in which the purchase was made; it can be considered a foreign key that points towards the `rowid` attribute of the `Province` relation;
- `product` indicates the identifier of a product in the range `0...5`;
- `qty` indicates the quantity of products included in the purchase: it is possible that a customer purchases multiple copies of the same product on the same day.

Note that we can hence distinguish:

The total number of items sold, which takes into account the quantity of products sold per purchase;

The total number of purchases, which does not take into account the quantity of items sold per purchase.

An example of data is provided below.

id	time	province	product	qty
1	2019-03-01	311	4	1
2	2019-03-01	311	0	2
3	2019-03-01	314	1	1
4	2019-03-01	314	0	1
5	2019-03-01	900	2	2
6	2019-03-11	311	2	2
7	2019-03-11	311	0	1
8	2019-03-11	314	1	2
9	2019-03-11	900	0	3
10	2019-03-11	900	2	4

The first row in this table represents one purchase; this purchase was made on the first of March, 2019, in Province 311 (`Brabant` in the Mondial database); it involved a product of type 4, and only one item of this type was bought. The second purchase is for the same day and the same province, but involves the product of type 0, of which 2 items were bought.

Queries

The queries that you need to write can be put in three categories:

- queries that don't consider the time attribute;
- queries that involve only the last ten days of purchases;
- a query that involves the time attribute and is not limited to the last ten days.

Many of the queries involve the calculation of counts. To simplify the queries, it is not required that you include rows for which the count is zero.

Queries that don't consider time

1. Determine the 10 countries with the most purchases; list the names of these countries, as well as the number of purchases; sort the results in decreasing order of purchases.

   On the earlier tiny example, the result should be:

name	cnt
Belgium	7
United States	3

   Note that province 314 (Liege) is located in Belgium as well, and that province 900 (New York) is located in the United States.

2. Determine the total number of purchases for each product. List the number of the product, as well as the total number of purchases.

   On the earlier tiny example, the result should be:

product	cnt
0	4
1	2
2	3
4	1

3. Determine for each country the province in which the total number of purchases of product 0 is the highest; list the name of the country, the name of the province, and the number of purchases, both absolute and as a proportion of the total number of purchases in that country.

   On the earlier tiny example, the result should be:

country	province	abs	prop
United States	New York	1	1
Belgium	Brabant	2	0.6667

4. List for each continent and each possible number of items per purchase (in the range 1...10), the corresponding number of purchases. Only consider countries that are 100% located in their respective continents.

   On the earlier tiny example, the result should be:

continent	qty	cnt
Europe	1	4
Europe	2	3
America	2	1
America	3	1
America	4	1

Queries that consider the last ten days

5. Determine the total number of purchases for each product. List the number of the product, as well as the total number of purchases.

   On our tiny example, the expected output would be:

product	cnt
0	2
1	1
2	2

6. Determine the total number of purchases for each of the last ten days. List the date, as well as the total number of purchases.

   On the tiny example, the output would be:

time	cnt
2019-03-11	5

7. Determine the identifiers of purchases performed in Brabant on the last day present in the database.

   On the tiny example, this would create one table with `ids` 6 and 7.

8. Determine the identifiers of purchases performed in Vienna on the last day present in the database.

   As the tiny example does not contain information regarding the province Vienna, the resulting table would be empty.

9. List for each continent and each possible number of items per purchase (in the range 1...10), the corresponding number of purchases. Only consider countries that are 100% located in their respective continents. Note that the difference with the earlier question is that we only consider the last 10 days.

   On the earlier tiny example, the result should be:

continent	qty	cnt
Europe	1	1
Europe	2	2
America	3	1
America	4	1

Query that involves time

10. List the top-10 of provinces for which the difference between the total number of items in purchases of the last 10 days and the preceding period of 10 days is the largest (i.e., the increase is large; we do not look for the largest decreases); list the full name of the province, the full name of the country, and the number of purchases in the last 10 days, as well as in the preceding period of 10 days; order the results by the size of the difference. Note that if there is no preceding period, the outcome is supposed to be empty.

    On the earlier tiny example, the result should be:

province	country	qty_last10	qty_1020
New York	United States	7	2
Brabant	Belgium	3	3
Liege	Belgium	2	2

Optimization

The main challenge that you will face in this project is to write queries that calculate the desired results as quickly as possible, every ten days.
To optimize the queries, you are allowed to take the following steps:

- you may extend the database by adding indexes and tables that store the results of queries;
- you may optimize the SQL expressions as much as you like.

Adding tables to a database may be useful if multiple queries can be answered more efficiently from the added tables.
To provide one concrete example: if we calculate the total number of purchases for every combination of province and product in advance, we can use the resulting table to calculate additional aggregated results, such as the total number of purchases per product and the total number of purchases per province.
Note however that adding a table to the database will also take time.
To evaluate the total performance of your queries, we will sum the run times of the following steps:

- the run time for extending the database (with indexes and additional tables) before purchase data arrives (E0);
- the run time for adding a first batch of purchases (`period1.db`) to the database, corresponding to the first ten days (P1);
- the run time for extending the database (with additional tables) after the first batch of purchases (E1); for example, the run time for adding tables that aggregate the purchases in the database;
- the run time for executing queries Q1-Q10 on the resulting database, using the extended database;
- the run time for adding a second batch of purchases (`period2.db`) to the database, corresponding to the next ten days;
- the run time for extending the database (with additional information in the added tables) based on the second batch (E2);
- the run time for executing queries Q1-Q10 again on the resulting database;
- the run time for adding a third batch of purchases (`period3.db`) to the database, corresponding to the final ten days;
- the run time for extending the database based on the third batch (E3);
- the run time for executing queries Q1-Q10 again.

Hence, you will not only need to write SQL statements for queries Q1-Q10, but also for:

- extending the initial database (E0). This may be useful to add tables to the initial Mondial database, or to define indexes.
- extending a database in which a batch of purchases have been added (E1-E3). The same set of statements will be used every 10 days.

To import a batch of data (`period1.db` in this example) into the central database, you are required to use the following statements:

```
ATTACH DATABASE 'period1.db' as period1;
INSERT INTO Purchases SELECT * FROM period1.Purchases;
```

For this purpose, the initial Mondial database has an empty Purchases table; this table will be filled using the above statement.
Note that importing the data could take a number of seconds, as each batch involves around 200MB of data.
Further hints on how to add tables and indexes in SQLite are provided in the Appendix.

Recommended Steps

You are recommended to work as follows:

1. First implement the SQL queries in a naive manner; a small database file `toy.db` is provided, on which you can already evaluate your queries without caring too much about performance. Check the Appendix for a number of hints on how to implement aggregation and ordering in SQL! You are free to create any additional table that you find helpful to answer the queries.
2. Load the initial batch of complete data (`period1.db`) in the original (non-toy) Mondial database (`mondial.db`) and test how well your queries perform on this data.
3. Improve the performance of your initial queries based on the first period of data, by either rewriting them, adding tables, or adding indexes;
4. Improve the performance of your queries by adding batches of purchases.
5. Test the performance of your queries, by submitting them to INGInious for a full evaluation;
6. Improve the performance of your queries further.

Appendix: SQLite in Query Optimization

When optimizing SQLite databases, you may find the following commands practical:

- `.tables`
  Provides an overview of all tables in the database.
- `.schema Table`
  Show the schema of `Table`, including the primary keys and foreign keys.
- `.indexes`
  Show the indexes present in the database.
- `explain query plan select * from table;`
  Show the query plan for the `select * from table` query, including which indexes are used. Consider the following query:

  ```
  EXPLAIN QUERY PLAN
  SELECT c.name
  FROM Country c, Borders b
  WHERE c.code = b.country2 and b.country1 = 'B';
  ```

  For this query, we get the following output:

  ```
  0|0|1|SEARCH TABLE Borders AS b USING COVERING INDEX sqlite_autoindex_Borders_1 ...
  0|1|0|SEARCH TABLE Country AS c USING INDEX sqlite_autoindex_Country_2 (code=?)
  ```

  This output should be interpreted as follows:

  – The numbers in the second column indicate a nesting level;
  – The numbers in the third column indicate the number of a condition in a WHERE statement;
  – The texts in the fourth column indicate the indexes that are used.

  In SQLite, a join is calculated using nested joins: for every element in one table, a corresponding element in the second table is determined. In our example, SQLite will iterate over the Borders table first, recovering bordering countries for the given country 'B' from the index that SQLite created automatically for the primary key; the nesting level of this loop is 0. The condition that is evaluated in this loop is the condition `b.country1='B'`, which is the second condition in the `WHERE`. For each bordering country, it will search the corresponding country name. This search is performed at nesting level 1, where the condition `c.code=b.country2` is evaluated; this is the first condition in the `WHERE`.

- `.timer on`
  Switch on a timer that measures the execution time of every query.
- `CREATE TABLE new_table AS (SELECT * FROM old_table);`
  Creates a new table that stores the result of evaluating another query.
- `SELECT * FROM COMPANY INDEXED BY salary_index WHERE salary > 5000 and id=1;`
  Forces SQLite to prefer the `salary_index` during the evaluation of the query, if there is a query plan in which `salary_index` can be used.
- `CREATE TABLE Company ( id INT PRIMARY KEY );`
  `INSERT INTO Company SELECT * FROM CompanyB;`
  Insert tuples from relation `CompanyB` into newly created relation `Company`.
- `CREATE INDEX Company_index ON Company(id);`
  Create an index on the `id` attribute of the `Company` relation.
- `DROP INDEX Company_index;`
  Remove the indicated index.
- SQLite may decide to create temporary indexes itself. This is indicated using
  `USING INDEX sqlite_autoindex` (for an index that SQLite creates to retrieve records more efficiently). A special type of index is the `COVERING INDEX`, which is an index from which SQLite can answer the query without going back to the data.
- Don't forget the use of `WITH` to split a query into multiple subqueries.
- To create a table every 10 days, you should make sure to remove any old table, if present. This can be done using the following statement:

  ```
  DROP TABLE IF EXISTS NameOfTable;
  ```
- To determine the top-k of results according to a certain column, use the following construction:

```
SELECT id, cnt
FROM TableWithIDandCnt
ORDER BY cnt DESC
LIMIT 10;
```

- Note that the division of two integers gives an integer in SQLite. If you wish the outcome to be a float, use for instance `CAST(a AS FLOAT)/b`.