PROJECT 2: UNDERSTANDING QUERY PLANS DURING DATA EXPLORATION

SC3020 DATABASE SYSTEM PRINCIPLES

TOTAL MARKS: 100

Due Date: April 16, 2023; 11:59 PM

Real-world users may write a sequence of SQL queries to explore the underlying relational database for a specific task. For instance, a user may start with a SQL query Q, execute it, browse the results and then modify Q to Q' (e.g., by modifying certain predicates in the WHERE clause), reexecute it, and view the refined results. Such exploration can go on *iteratively* by executing a sequence of related SQL queries and browsing corresponding results repeatedly. The following example shows Q and Q' with changes to Q highlighted in red.

Query Q	Query Q'

`select *`
`from customer C, orders O`
`where C.c_custkey = O.o_custkey`
	`select *`
	`from customer C, orders O`
	`where C.c_custkey = O.o_custkey`
	`and customer.name like '%cheng'`

The RDBMS query optimizer will execute a **query execution plan** (QEP) to process each such SQL query during exploration. For instance, there will be two QEPs, P and P', associated with Q and Q', respectively, in the above example. These QEPs are typically displayed in the form of tree-structure by a DBMS software (e.g., PostgreSQL). Unfortunately, to an end user who is not proficient in database technology, this may not be the best way to understand how each of her queries has been executed during data exploration.

Your task is to write a program that **automatically generates user-friendly explanation (e.g., natural and visual language description) of the changes to the query execution plans that take place during data exploration**. Specifically, let $P_1, P_2, \ldots, P_n$ are the QEPs generated by the DBMS for executing a sequence of queries $Q_1, Q_2, \ldots, Q_n$, respectively, during data exploration. Note that the queries are related as they have evolved from the original query Q_1. Hence, the QEPs may also share common content among themselves. Your task is to generate user-friendly description of the way the **plans have evolved** during data exploration (e.g., a hash join in P_1 has now evolved to sort-merge join in P_2 due to changes in the WHERE clause in Q_2).

Hint: Design algorithm to efficiently identify the parts of a plan that have evolved in the query plan trees and explain those to the end user using a combination of visual and natural language form and connecting them with the changes to SQL.

To this end, your tasks are as follows:

- Design and implement an algorithm that takes as input the followings:
    a. Old query Q_1, its QEP P_1
    b. *New query Q_2, its QEP P_2*

    It generates a user-friendly description of **what has changed** from P_1 *to* P_2, and **why**. Your goal is to ensure generality of the solution (i.e., it can handle a wide variety of query plans on different database instances) and the user-friendly explanation should be **concise** without sacrificing important information related to the plan. The better is the algorithm design for the task, the more credit you will receive. Similarly, the more functionalities you support, the more credit you will receive.

- A user-friendly, graphical user interface (GUI) to enable the aforementioned goals.

You should use **Python** as the host language on **Windows** platform for your project. For students using **Mac** platform, you can **install Windows on your Mac** by following instructions in https://support.apple.com/en-sg/HT201468. The DBMS allowed in this project is **PostgreSQL.** The example dataset you should use for this project is **TPC-H** (see *Appendix*). You are free to use any off-the-shelf toolkits for your project.

Note that several parts of the project are left open-ended (e.g., how the GUI should look like? What are the functionalities we should support? How should you explain to an end user?) intentionally so that the project does not curb a group's creative endeavors. You are free to make realistic assumptions to achieve these tasks.

SUBMISSION REQUIREMENTS

You submission should include the followings:

- You should submit **three** program files: *interface.py, explain.py, and project.py*. The file *interface.py* contains the code for the GUI. The *explain.py* contains the code for generating the explanation. The *project.py* is the main file that invokes all the necessary procedures. **Note that we shall be running the project.py file** (either from command prompt or using the Pychamp IDE) to execute the software. Make sure your code follows good coding practice: sufficient comments, proper variable/function naming, etc. We will execute the software

to check its correctness using **different** query sets and dataset to check for the generality of the solution. We will also check quality of algorithm design w.r.t processing of the query plans.

- **Softcopy report** containing details of the software including formal descriptions of the key algorithms with examples. You should also discuss limitations of the software (if any).
- **Peer assessment report** from each member of the team. Each individual member of a team needs to assess contributions of the group members. Details of peer assessment form will be provided closer to the submission date.
- You must submit a document containing instructions to run your software successfully. You will not receive any credit if your software fails to execute based on your instructions.
- All submissions will be through NTU Learn.

Note: Late submission will be penalized.

Appendix

I. Creating TPC-H database in PostgreSQL

Follow the following steps to generate the TPC-H data:

1) Go to http://www.tpc.org/tpc_documents_current_versions/current_specifications5.asp and download TPC-H Tools v2.18.0.zip. Note that the version may defer as the tool may have been updated by the developer.
2) Unzip the package. You will find a folder "dbgen" in it.
3) To generate an instance of the TPC-H database:
    - Open up tpch.vcproj using visual studio software.
    - Build the tpch project. When the build is successful, a command prompt will appear with "TPC-H Population Generator <Version 2.17.3>" and several *.tbl files will be generated. You should expect the following .tbl files: customer.tbl, lineitem.tbl, nation.tbl, orders.tbl, part.tbl, partsupp.tbl, region.tbl, supplier.tbl
    - Save these .tbl files as .csv files
    - These .csv files contain an extra " | " character at the end of each line. These " | " characters are incompatible with the format that PostgreSQL is expecting. Write a small piece of code to remove the last " | " character in each line. Now you are ready to load the .csv files into PostgreSQL
    - Open up PostgreSQL. Add a new database "TPC-H".
    - Create new tables for "customer", "lineitem", "nation", "orders", "part", "partsupp", "region" and "supplier"
    - Import the relevant .csv into each table. Note that pgAdmin4 for PostgreSQL (windows version) allows you to perform import easily. You can select to view the first 100 rows to check if the import has been done correctly.
      If encountered error (e.g., ERROR: extra data after last expected column) while importing, create columns of each table first before importing. Note that the types of each column has to be set appropriately. You may use the SQL commands in Appendix II to create the tables.

Alternatively, you can also refer to https://docs.verdictdb.org/tutorial/tpch/ for additional help on creating the TPC-H database

II. SQL commands for creating TPC-H data tables

Region table

```
CREATE TABLE public.region
(
    r_regionkey integer NOT NULL,
    r_name character(25) COLLATE pg_catalog."default" NOT NULL,
    r_comment character varying(152) COLLATE pg_catalog."default",
    CONSTRAINT region_pkey PRIMARY KEY (r_regionkey)
)
WITH (
    OIDS = FALSE
)
TABLESPACE pg_default;

ALTER TABLE public.region
    OWNER to postgres;
```

1) Nation table

```
CREATE TABLE public.nation
(
    n_nationkey integer NOT NULL,
    n_name character(25) COLLATE pg_catalog."default" NOT NULL,
    n_regionkey integer NOT NULL,
    n_comment character varying(152) COLLATE pg_catalog."default",
    CONSTRAINT nation_pkey PRIMARY KEY (n_nationkey),
    CONSTRAINT fk_nation FOREIGN KEY (n_regionkey)
        REFERENCES public.region (r_regionkey) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
WITH (
    OIDS = FALSE
)
TABLESPACE pg_default;

ALTER TABLE public.nation
    OWNER to postgres;
```

2) Part table

```
CREATE TABLE public.part
(
    p_partkey integer NOT NULL,
    p_name character varying(55) COLLATE pg_catalog."default" NOT NULL,
    p_mfgr character(25) COLLATE pg_catalog."default" NOT NULL,
    p_brand character(10) COLLATE pg_catalog."default" NOT NULL,
    p_type character varying(25) COLLATE pg_catalog."default" NOT NULL,
    p_size integer NOT NULL,
    p_container character(10) COLLATE pg_catalog."default" NOT NULL,
    p_retailprice numeric(15,2) NOT NULL,
    p_comment character varying(23) COLLATE pg_catalog."default" NOT NULL,
    CONSTRAINT part_pkey PRIMARY KEY (p_partkey)
)
WITH (
    OIDS = FALSE
)
TABLESPACE pg_default;

ALTER TABLE public.part
    OWNER to postgres;
```

3) Supplier table

```
CREATE TABLE public.supplier
(
    s_suppkey integer NOT NULL,
    s_name character(25) COLLATE pg_catalog."default" NOT NULL,
    s_address character varying(40) COLLATE pg_catalog."default" NOT NULL,
    s_nationkey integer NOT NULL,
    s_phone character(15) COLLATE pg_catalog."default" NOT NULL,
    s_acctbal numeric(15,2) NOT NULL,
    s_comment character varying(101) COLLATE pg_catalog."default" NOT NULL,
    CONSTRAINT supplier_pkey PRIMARY KEY (s_suppkey),
    CONSTRAINT fk_supplier FOREIGN KEY (s_nationkey)
        REFERENCES public.nation (n_nationkey) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
WITH (
    OIDS = FALSE
)
TABLESPACE pg_default;

ALTER TABLE public.supplier
    OWNER to postgres;
```

4) Partsupp table

```
CREATE TABLE public.partsupp
(
    ps_partkey integer NOT NULL,
    ps_suppkey integer NOT NULL,
    ps_availqty integer NOT NULL,
    ps_supplycost numeric(15,2) NOT NULL,
    ps_comment character varying(199) COLLATE pg_catalog."default" NOT NULL,
    CONSTRAINT partsupp_pkey PRIMARY KEY (ps_partkey, ps_suppkey),
    CONSTRAINT fk_ps_suppkey_partkey FOREIGN KEY (ps_partkey)
        REFERENCES public.part (p_partkey) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION,
    CONSTRAINT fk_ps_suppkey_suppkey FOREIGN KEY (ps_suppkey)
        REFERENCES public.supplier (s_suppkey) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
WITH (
    OIDS = FALSE
)
TABLESPACE pg_default;

ALTER TABLE public.partsupp
    OWNER to postgres;
```

5) Customer table

```
CREATE TABLE public.customer
(
    c_custkey integer NOT NULL,
    c_name character varying(25) COLLATE pg_catalog."default" NOT NULL,
    c_address character varying(40) COLLATE pg_catalog."default" NOT NULL,
    c_nationkey integer NOT NULL,
    c_phone character(15) COLLATE pg_catalog."default" NOT NULL,
    c_acctbal numeric(15,2) NOT NULL,
    c_mktsegment character(10) COLLATE pg_catalog."default" NOT NULL,
    c_comment character varying(117) COLLATE pg_catalog."default" NOT NULL,
    CONSTRAINT customer_pkey PRIMARY KEY (c_custkey),
    CONSTRAINT fk_customer FOREIGN KEY (c_nationkey)
        REFERENCES public.nation (n_nationkey) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
WITH (
    OIDS = FALSE
)
TABLESPACE pg_default;

ALTER TABLE public.customer
    OWNER to postgres;
```

6) Orders table

```
CREATE TABLE public.orders
(
    o_orderkey integer NOT NULL,
    o_custkey integer NOT NULL,
    o_orderstatus character(1) COLLATE pg_catalog."default" NOT NULL,
    o_totalprice numeric(15,2) NOT NULL,
    o_orderdate date NOT NULL,
    o_orderpriority character(15) COLLATE pg_catalog."default" NOT NULL,
    o_clerk character(15) COLLATE pg_catalog."default" NOT NULL,
    o_shippriority integer NOT NULL,
    o_comment character varying(79) COLLATE pg_catalog."default" NOT NULL,
    CONSTRAINT orders_pkey PRIMARY KEY (o_orderkey),
    CONSTRAINT fk_orders FOREIGN KEY (o_custkey)
        REFERENCES public.customer (c_custkey) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
WITH (
    OIDS = FALSE
)
TABLESPACE pg_default;

ALTER TABLE public.orders
    OWNER to postgres;
```

7) Lineitem table

```
CREATE TABLE public.lineitem
(
    l_orderkey integer NOT NULL,
    l_partkey integer NOT NULL,
    l_suppkey integer NOT NULL,
    l_linenumber integer NOT NULL,
    l_quantity numeric(15,2) NOT NULL,
    l_extendedprice numeric(15,2) NOT NULL,
    l_discount numeric(15,2) NOT NULL,
    l_tax numeric(15,2) NOT NULL,
    l_returnflag character(1) COLLATE pg_catalog."default" NOT NULL,
    l_linestatus character(1) COLLATE pg_catalog."default" NOT NULL,
    l_shipdate date NOT NULL,
    l_commitdate date NOT NULL,
    l_receiptdate date NOT NULL,
    l_shipinstruct character(25) COLLATE pg_catalog."default" NOT NULL,
    l_shipmode character(10) COLLATE pg_catalog."default" NOT NULL,
    l_comment character varying(44) COLLATE pg_catalog."default" NOT NULL,
    CONSTRAINT lineitem_pkey PRIMARY KEY (l_orderkey, l_partkey, l_suppkey, l_linenumber),
    CONSTRAINT fk_lineitem_orderkey FOREIGN KEY (l_orderkey)
        REFERENCES public.orders (o_orderkey) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION,
    CONSTRAINT fk_lineitem_partkey FOREIGN KEY (l_partkey)
        REFERENCES public.part (p_partkey) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION,
    CONSTRAINT fk_lineitem_suppkey FOREIGN KEY (l_suppkey)
        REFERENCES public.supplier (s_suppkey) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
WITH (
    OIDS = FALSE
)
TABLESPACE pg_default;

ALTER TABLE public.lineitem
    OWNER to postgres;
```