PROJECT 2: CONNECTING YOUR QUERY WITH QUERY PLANS

CX4031 DATABASE SYSTEM PRINCIPLES

TOTAL MARKS: 100

Due Date: Nov 13, 2022; 11:59 PM

Real-world users may write SQL queries to search relational databases for different tasks. The RDBMS query optimizer will execute a query execution plan (QEP) to process each query, which is chosen from a large number of *alternative query plans* (AQP). Typically, the plan selected as the QEP is estimated to have least/lower cost than other AQP. One can retrieve and view these plans in visual (tree-structured view) or text format (e.g., json, XML) in an off-the-shelf DBMS software (e.g., PostgreSQL). In particular, PostgreSQL allows one to retrieve AQPs containing specific operators using the *planner method configuration* (https://www.postgresql.org/docs/9.2/runtime-config-query.html#RUNTIME-CONFIG-QUERY-CONSTANTS). Unfortunately, an SQL query and its query plan-related information are disconnected. In this project, the broad goal is to integrate them by retrieving relevant information from a QEP and AQP to **annotate** the corresponding SQL query to **explain** how different components of the query are executed by the underlying query processor and why the operators are chosen among other alternatives. Note that to explain the why question, it is important to retrieve representative AQPs associated with a SQL query.

PROJECT DESCRIPTION

```
select *
from   customer   C,
orders O
where  C.c_custkey  =
O.o_custkey
```

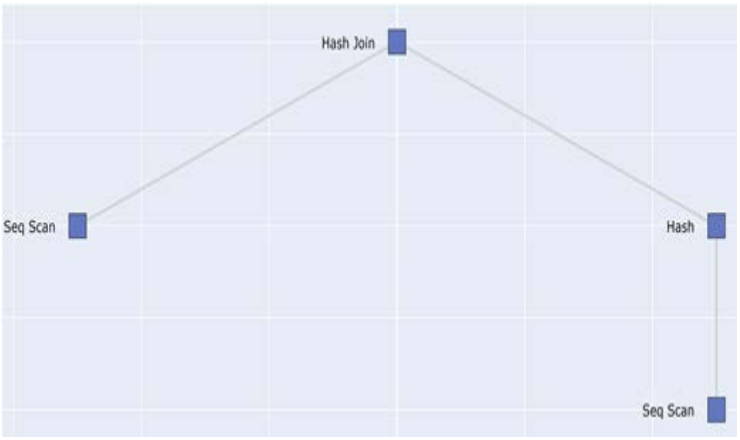

Figure 1

The goal of the project can be described with an example. Consider the SQL query in Figure 1. The simplified view of the QEP for execution of this query in PostgreSQL is given on the right. Observe that by simply reading the SQL query a user or a query writer does not know how different components of the query are executed. For instance, how the *customer* and *orders* relations are read from the database? What kind of join operator is used to implement the equi-join? Why this join operator is chosen among other alternatives? The goal of this project is to automatically annotate the different components of the query by processing the QEP, representative AQP, and the input query. An example output of the annotated query can be as follows:

```
select *
from customer C, orders O
where C.c_custkey = O.o_custkey
```

Tables are read using sequential scan. This is because no index is created on the tables.

This join is implemented using hash join operator as NL joins and merge join increase the estimated cost by at least 10 and 7 times, respectively.

The benefit of such annotation is that a query writer can easily view how different components he/she wrote are executed in the underlying DBMS.

(a) **Design and implement an efficient algorithm** that takes as input an SQL query, retrieves its query execution plan and representative AQPs, and returns as output an **annotated** SQL query that describes the execution of various components of the query. Note that the structure of annotation is open-ended. It can be in the form of natural language or in a more structured form. *You do not need any knowledge about NLP to address this problem.*
Your goal is to ensure generality of the solution (i.e., it can handle a wide variety of queries and should be independent of the underlying database schema). Hence, your software should work on any database schema and for a wide variety of SQL queries on it. In other words, you should not be hardcoding SQL queries or schema-specific information.

(b) **Design and implement a user-friendly graphical user interface (GUI)** to visualize your results. You can imagine that through it you can choose the database schema (TPC-H in this case), specify the query in the *Query panel* and upon execution of your algorithm the *Query panel* is updated with annotations. You may also selectively visualize the corresponding QEP in another panel.

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 and queries 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 deliberately open-ended (e.g., what will be the content of the annotations? how the GUI should look like? What are the functionalities we should support?) 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 **four** program files: *interface.py*, *annotation.py*, *preprocessing.py*, and *project.py*. The file *interface.py* contains the code for the GUI. The *annotation.py* contains code for generating the annotations. The *preprocessing.py* file contains any code for reading inputs and any preprocessing necessary to make your algorithm work. Lastly, the *project.py* is the main file that invokes all the necessary procedures from these three files. **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;
```