PROJECT 2: UNDERSTANDING COST ESTIMATION IN RDBMS

SC3020 DATABASE SYSTEM PRINCIPLES

TOTAL MARKS: 100

Due Date: April 21, 2024; 11:59 PM

The relational query execution engine implements a set of physical operators. An operator takes as input one or more data streams and produces an output data stream. Some examples of physical operators are sequential scan, index scan, and hash join. These physical operators are the building blocks for the execution of SQL queries. An abstract representation of such an execution is a physical operator tree (operator tree for brevity), denoted as $T = (N, E)$, where N is a set of nodes representing the operators and E is a set of edges representing data flow among the physical operators. The physical operator tree is the abstract representation of a **query execution plan** (QEP). The query execution engine is responsible for the execution of the QEP to generate results of a SQL query.

A QEP is associated with an estimated cost as computed by the underlying relational query engine. The nodes/subtrees in the operator tree are also associated with the estimated cost of executing the operators represented by them. You can view a QEP of a given query using the *EXPLAIN* feature of *PostgreSQL*.

The goal of your project is to **design and implement a software to *explain* the computation of the estimated cost associated with a QEP for a given input SQL query by exploiting the knowledge of various cost estimation you have learnt in the course**. Specifically, the input/output of your program is as follows:

- **Input:** An SQL query (you should be able to handle any SQL query)
- **Output:** Explanation of the computation of various cost in the QEP of the query. In particular, the explanation should focus on the followings: (a) how did the DBMS compute the cost of various nodes/subtrees in the QEP? That is, how did it get the values? (b) In the case, you are unable to find the details of computation of the cost (i.e., the estimated cost in the QEP is different from your analysis) then explain the differences. Note that all explanations need to be undertaken automatically for a given QEP (i.e., query).

Hint: To address this problem, you will need statistical information related to the relations in your database. This can be retrieved from the database catalog. Explore how you can do this. You can also access information related to blocks accessed by a query by issuing a set of queries (as discussed in the Example Classes). If you have strong foundation in coding and database concepts, you may also explore the codebase of the relational query processor to get information related to cost estimation (and map it

to your software). Note that PostgreSQL codebase is publicly available (e.g., https://github.com/postgres/postgres).

Specific tasks required for your project are as follows:

- Design and implement an algorithm that takes as input an SQL query and generates the explanation for various costs associated with the QEP. 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 explanation should be **concise** without sacrificing important information related to the estimated cost. The better is the algorithm design for the task, the more credit you will receive.
- A user-friendly, graphical user interface (GUI) to enable the aforementioned goal. Note that a working GUI to show your results easily should suffice. It is not necessary to generate a fancy one as the course is not on GUI development.

You should use **Python** as the host language on **Windows** platform for your project (We choose it to be consistent with PCs in all labs). For students using **Mac** platform, you can **install Windows on your Mac**. Alternatively, you may develop in one of the Windows machines in the Project Lab. 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? How should you explain the estimated cost?) 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.
- If you are using any **Large Language Model** (e.g., ChatGPT) for your project then you must acknowledge its usage appropriately and clearly specify where you have used it (in the report). Failure to do so may result in an F grade for the project.
- 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. Details of submissions will be released nearer to the submission date.

Note: Late submission will be penalized (10 marks per day). No submission is allowed if you are late by more than 3 days.

Appendix

I. Creating TPC-H database in PostgreSQL

Follow the following steps to generate the TPC-H data (this step may differ slightly due to different versions of TPC-H). So you should use this as just a general guideline. Please modify the steps as deemed necessary.

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;
```