CS-422 Database Systems

Project I: Data Layouts and Execution Models

Deadline: March 23rd at 23:59.

In this project you will implement three different execution models based on three different storage layouts. Subsequently, you will execute queries in order to test different combinations of data layouts and execution models. We provide a skeleton codebase (`https://gitlab.epfl.ch/DIAS/COURSES/CS-422/2020/ta/project-1`), written in Scala, on which you will add your implementation. Your implementation will be hosted under `https://gitlab.epfl.ch/DIAS/COURSES/CS-422/2020/students/Project-1-<username>` where username is your GitLab username[1]. Test cases that will allow you to check the functionality of your code as well as measure the efficiency of your implementation are also provided. In what follows, we will go through the three tasks that you need to carry out.

Assumptions. In the context of this project you will store all data in-memory, thus removing the requirement for a buffer manager. In addition, you can assume that the records will consist of objects (*Any* class in Scala). You must use Java 10 or 11 for the project in order to be compatible with the Scala version used.

In the following we present the three different data layouts which are already implemented in the given skeleton. The three layouts are: (i) a row layout, (ii) a columnar layout, and (iii) PAX.

Row data layout (NSM). The class "ch.epfl.dias. cs422.helpers.store.RowStore" contains all the necessary data structures that enable storing relational tables in a row-oriented format. The class extends the existing class "Store". The most important method of this class is "getRow".

Columnar data layout (DSM). The class "ch.epfl. dias.cs422.helpers.store.ColumnStore" contains all the necessary data structures that enable storing relational tables in a column-oriented form. Similarly to class "RowStore", the class extends the existing class "Store". The most important method of this class is "getColumn".

[1] e.g. `https://gitlab.epfl.ch/DIAS/COURSES/CS-422/2020/students/Project-1-turing`

Partition Attributes Across data layout (PAX). The class "ch.epfl.dias.cs422.helpers.store.PAXStore" contains all the necessary data structures that enable storing relational tables using the PAX format. Similarly to classes "RowStore" and "ColumnStore", the class extends the existing class "Store". The most important method of this class is"getPAXPage".

Hint. Notice that the above classes use the types "Tuple", "Column" and "PAXPage".

Task 1. Implementation of execution models

You will now implement six basic operators (**Scan**, **Select**, **Project**, **Join**, **Aggregate** and **Sort**) in three different execution models. You will support both early (Subtasks 1.1, 1.2, 1.3) and late (Subtask 1.4) materialization.
You are free to implement the **Join** operator of your preference. The **Scan** operators need to support and distinguish among all underlying storage layouts. The rest of the operators need to be usable for all storage layouts (implementation should not be data layout dependent).

Important!!! Implement the operators based on the prototypes given in the skeleton code.

Subtask 1.1. Implement a volcano-style tuple-at-a-time engine. Implement the volcano-style operators by extending the provided trait/interface "ch.epfl.dias.cs422.helpers.rel.early.volcano.Operator". As described in class, each operator in a volcano-style engine requires the implementation of three methods:

- **open():** Initialize the operator's internal state.
- **next():** Process and return the next result tuple or null for <EOF>.
- **close():** Finalize the execution of the operator and clean up the allocated resources.

In this case the operators should process one tuple at a time.

Subtask 1.2. Implement an operator-at-a-time engine. Implement operators that process a whole column at-a-time by extending the provided trait/interface "ch.epfl.dias.cs422.helpers.rel.early.operatoratatime.Operator". Each operator requires the implementation of an **execute()** method.

Subtask 1.3. Implement a volcano-style vector-at-a-time engine. This task is similar to 1.1, with the sole distinction that each operator operates over a vector of tuples rather than a single tuple at a time. In other words, the *next()* method of each operator returns a vector of at most N tuples. The trait/interface for these vectorized operators is "ch.epfl.dias.cs422.helpers.rel.early.blockatatime.Operator".

Subtask 1.4. Implement a late materialization-based engine. In this task, you have to implement operators that process columns using late materialization. The operators use an operator-at-a-time execution model. The columns are stored as *virtual ids* and are reconstructed on-demand within each operator. The trait/interface for these vectorized operators is "ch.epfl.dias.cs422.helpers.rel.late.operatoratatime.Operator".

Task 2. Testing different execution models and data layouts

In this task you have to test the correctness and efficiency of the different combinations of data layouts and execution models.
Your system must support all test cases found under the "ch.epfl.dias.cs422.QueryTest" object. The tests evaluate queries that contain the operators Scan, Select, Project, Join, Aggregate, and Sort, which you implemented in Task 1. You may find the test queries under the directory "src/test/resources/tests". The tests operate over the datasets of the TPC-H benchmark, that are located under the "input" directory. You may find the schema of the datasets at the end of this project description. Using the tests, you can also evaluate the different execution models and storage layouts you have implemented.

You can test your implementation locally by executing the "ch.epfl.dias.cs422.QueryTest" object which contains the tests. You can also test your implementation on GitLab every time you push on your repository, by accessing the CI/CD tab on the left menu. If you click on the ✓/✗on the Stages column you can see the tests that passed, the tests that failed and the errors if any.

In this task you also need to evaluate the different combinations of storage layouts and execution models by reporting their response times. Your evaluation can be based on the provided test queries. Add the response times in Task2.pdf and add a short explanation on what you observe and why.

Deliverables

We will grade your last commit on your GitLab repository. Your implementation must be on the **master** branch. In the context of this project you only need to modify the "ch.epfl.dias.cs422.rel" package. **You do not submit anything on moodle**. Your repository must contain the **Task2.pdf** which is your answer to Task2.

Grading: Keep in mind that we will test your code automatically. We will harshly penalize implementations that change the original skeleton code and implementations which are specific to the given datasets. For the actual grading we will slightly modify the tests/datasets in the CI. Hence, your reported failures/successes in the existing tests only approximately determine your final grade. The new tests will be very similar however.

Datasets

All datasets are in CSV format where the fields of each tuple are separated by "|".

```
LINEITEM(L_ORDERKEY      INTEGER NOT NULL,
         L_PARTKEY       INTEGER NOT NULL,
         L_SUPPKEY       INTEGER NOT NULL,
         L_LINENUMBER    INTEGER,
         L_QUANTITY      INTEGER,
         L_EXTENDEDPRICE DOUBLE,
         L_DISCOUNT      DOUBLE,
         L_TAX           DOUBLE,
         L_RETURNFLAG    CHAR(1),
         L_LINESTATUS    CHAR(1),
         L_SHIPDATE      DATE,
         L_COMMITDATE    DATE,
         L_RECEIPTDATE   DATE,
         L_SHIPINSTRUCT  STRING,
         L_SHIPMODE      STRING,
         L_COMMENT       STRING)

ORDERS(O_ORDERKEY       INTEGER NOT NULL,
       O_CUSTKEY        INTEGER NOT NULL,
       O_ORDERSTATUS    CHAR(1) NOT NULL,
       O_TOTALPRICE     DOUBLE NOT NULL,
       O_ORDERDATE      DATE NOT NULL,
       O_ORDERPRIORITY  STRING NOT NULL,
       O_CLERK          STRING NOT NULL,
       O_SHIPPRIORITY   INTEGER NOT NULL,
       O_COMMENT        STRING NOT NULL)

CUSTOMER(C_CUSTKEY      INTEGER NOT NULL,
         C_NAME         STRING,
         C_ADDRESS      STRING,
         C_NATIONKEY    INTEGER NOT NULL,
         C_PHONE        STRING,
         C_ACCTBAL      DOUBLE,
         C_MKTSEGMENT   STRING,
         C_COMMENT      STRING)

NATION(N_NATIONKEY  INTEGER NOT NULL,
       N_NAME       STRING,
       N_REGIONKEY  INTEGER NOT NULL,
       N_COMMENT    STRING)
```

```
PART(P_PARTKEY      INTEGER NOT NULL,
     P_NAME         STRING,
     P_MFGR         STRING,
     P_BRAND        STRING,
     P_TYPE         STRING,
     P_SIZE         INTEGER,
     P_CONTAINER    STRING,
     P_RETAILPRICE  DOUBLE,
     P_COMMENT      STRING)

PARTSUPP(PS_PARTKEY     INTEGER NOT NULL,
         PS_SUPPKEY     INTEGER NOT NULL,
         PS_AVAILQTY    INTEGER,
         PS_SUPPLYCOST  DOUBLE,
         PS_COMMENT     STRING)

SUPPLIER (S_SUPPKEY     INTEGER NOT NULL,
          S_NAME        STRING,
          S_ADDRESS     STRING,
          S_NATIONKEY   INTEGER NOT NULL,
          S_PHONE       STRING,
          S_ACCTBAL     DOUBLE,
          S_COMMENT     STRING)

REGION (R_REGIONKEY INTEGER NOT NULL,
        R_NAME      STRING,
        R_COMMENT   STRING)
```