CS-422 Database Systems

Project I: Data Layouts and Execution Models

Deadline: March 25th at 23:59.

In this project you will first implement three different data layouts and three different execution models. Subsequently, you will execute queries in order to test different combinations of data layouts and execution models. A skeleton codebase on which you will add your implementation is provided in `https://github.com/CS-422/CS422-Project1`. 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 have a fixed size, therefore you need to perform a simple memory allocation strategy, no slotted pages. Finally, you don't have to implement update or delete operations.

Task 1 (33%) Implementation of data layouts

In this first task you have to implement three different data layouts: (i) a row layout, (ii) a columnar layout, and (iii) PAX.

Subtask 1.1. Implement a row data layout (NSM). Implement the class "ch.epfl.dias.store.row.RowStore" which contains all the necessary data structures that enable storing relational tables in a row-oriented format. The class has to extend the existing class "Store" and thus must implement the corresponding methods. In particular, you have to implement the methods "load" and "getRow".

Subtask 1.2. Implement columnar data layout (DSM). Implement the class "ch.epfl.dias.store.column.ColumnStore" which contains all the necessary data structures that enable storing relational tables in a column-oriented form. Similarly to class "RowStore", the class has to extend the existing class "Store" and thus must implement the corresponding methods. In particular, you have to implement the methods "load" and "getColumns".

Subtask 1.3. Implement Partition Attributes Across data layout (PAX). Implement the class "ch.epfl.dias.store.pax.PAXStore" which contains all the necessary data

structures that enable storing relational tables using the PAX format. Similarly to classes "RowStore" and "ColumnStore", the class has to extend the existing class "Store" and thus must implement the corresponding methods. In particular, you have to implement the methods "load" and "getRow". The "PAXStore" should be used interchangeably with "RowStore".

Hint. To implement the above classes you will need to define the classes "DBTuple", "DBColumn" and "DBPAXpage". We provide the "DBTuple" class as a guideline.

Task 2 (33%) Implementation of execution models

You will now implement five basic operators (**Scan**, **Select**, **Project**, **Join** and **ProjectAggregate**) in three different execution models. You are free to implement the join operator of your preference.
`Important!!!` Implement the operators based on the prototypes given in the skeleton code.

Subtask 2.1. Implement a volcano-style tuple-at-a-time engine. Implement the volcano-style operators by extending the provided interface "ch.epfl.dias.ops.volcano.VolcanoOperator". 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 <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. This query engine has to be usable over the row and PAX data layouts that you implemented in Task 1.

Hint. You have to create a hierarchy of operators by giving to each operator a reference to its child. The plan is initialized by calling *open()* on the root operator. Check the usage of the operators in the tests.

Subtask 2.2. Implement a column-at-a-time engine. Implement operators that process a whole column at-a-time by extending the provided interface "ch.epfl.dias.ops.columnar.ColumnarOperator". The column-at-a-time query engine should be usable over the columnar data layout that you implemented in Task 1. When implementing column-at-a-time, enable your system to execute both early and late materialization in the implementation of the operators.

Subtask 2.3. Implement a volcano-style vector-at-a-time engine. This task is similar to 2.1, 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 interface for these vectorized operators is "ch.epfl.dias.ops.vector.VectorOperator". The vector-at-a-time query engine should be usable over the columnar data layout that you implemented in Task 1. For simplicity when executing vector-at-a-time, use early materialization in the implementation of the operators.

Task 3 (33%). Execute queries over different execution models and data layouts

In this task you have to experiment with different combinations of data layouts and execution models and observe the impact on query processing performance. More specifically, you need to test the following four combinations:

		Execution model		
		Tuple-at-a-time	Column-at-a-time	Vector-at-a-time
Data layout	NSM	✓	✗	✗
	DSM	✗	✓	✓
	PAX	✓	✗	✗

Your system should support any query that uses any combination of the three operators that you implemented in Task 2. That is, you should support select, project, join and aggregate queries of the following form:

```
SELECT attr1, attr2, ...
FROM lineitem
WHERE attri op value

SELECT c.attri, o.attrj, ...
FROM lineitem l, orders o
WHERE l.l_orderkey=o.o_orderkey

SELECT AGGR(attri)
FROM lineitem
GROUP BY attr
```

where `op` can be one of the following operators: "=", "≠", "<", ">", "<=", ">=" and `value` is a value to which we compare an attribute, e.g., l_orderkey = 2 (we constraint comparisons to integers). For each of the above query types, state which pair of data layout-execution model performs best and explain why. Due to the garbage collector of java the results may not be obvious. In this case, respond using your insight and argue based on theory.

Datasets

"lineitem.csv" (find link on moodle page), CSV format where the fields of each tuple are separated by "|". The relation LINEITEM has the following schema:

```
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.csv" (find link on moodle page), CSV format where the fields of each tuple are separated by "|". The relation ORDERS has the following schema:

```
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)
```

`L_ORDERKEY` and `O_ORDERKEY` are primary keys and `O_ORDERKEY` is a foreign key referencing `L_ORDERKEY`.

Deliverables

- Project1.zip: A self-contained zip file with your project.
- Task3.pdf: A short report containing your answer to Task 3.

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.