CompSci 316 Fall 2019: Homework 4

100 points (6.25% of course grade) + 20 points extra credit
Assigned: Wednesday, November 6
Due: Monday, December 2 (except X2, which is due Wednesday, December 4)

This homework should be done in parts as soon as relevant topics are covered in lectures. If you wait until the last minute, you might be overwhelmed.

For each problem below except X2, please submit one PDF file. You may prepare your answers electronically or on paper (handwritten); in the latter case, you need to scan or photograph the pages for submission. Please read the "Help → Submitting Non-Gradiance Work" section of the course website for instructions. If a problem has multiple parts, please put your solution for each part on a separate page.

Problem X2 must be completed on your course VM. Before you start, make sure you refresh your VM, by logging into your VM and issuing the following command:

```
        /opt/dbcourse/sync.sh
```

Problem 1 (30 points)

A table $R(\underline{K}, A, \ldots)$ with $100{,}000$ rows is stored in $10{,}000$ disk blocks. The rows are sorted by R's primary key K, but not by A. There is a dense, secondary B⁺-tree index on $R(A)$, which has 3 levels and 500 leaves.

Suppose we want to sort R by A. We have 101 memory blocks at our disposal. Method 1 performs an external-memory merge sort using all memory available. Method 2 takes advantage of the fact that the values of A are already sorted in the B⁺-tree index on $R(A)$: It simply scans the leaves of the index to retrieve and output R rows in order.

How many disk I/O's do these two methods require? Which one is the winner?

Problem 2 (40 points)

Consider the following schema for an online bookstore:

Cust (*<u>CustID</u>*, *Name*, *Address*, *State*, *Zip*)
Book (*<u>BookID</u>*, *Title*, *Author*, *Price*, *Category*)
Order (*<u>OrderID</u>*, *CustID*, *BookID*, *ShipDate*)
Inventory (*<u>BookID</u>*, *<u>Quantity</u>*, *<u>WarehouseID</u>*, *ShelfLocation*)
Warehouse (*WarehouseID*, *State*)

Cust and *Book* represent customers and books, respectively. When a customer buys a book, a tuple is entered into *Order*. *Inventory* records the quantity and shelf location of each book for every warehouse. *Warehouse* records the state where each warehouse is located in. *Price* is numeric. *ShipDate* is an integer representation of a date. In the following, `:today` is a constant denoting the integer representation of today's date.

(a) Transform the following query into an equivalent query that 1) contains no cross products, and 2) performs projections and selections as early as possible. Represent your result as a relational algebra expression tree.

$$\pi_{Title}$$
$$\quad \sigma_{(Author\ LIKE\ "\%Rowling\%")\text{ and }(State="NC")\text{ and }(ShipDate \geq :today-60)}$$
$$\qquad \sigma_{(Cust.CustID=Order.CustID)\text{ and }(Book.BookID=Order.BookID)}$$
$$\qquad\quad (Cust \times Order \times Book).$$

Suppose we have the following statistics:

- $|Cust| = 1{,}000$; $|\pi_{State} Cust| = 50$;
- $|Book| = 600$; $|\pi_{Category} Book| = 10$;
- $|Order| = 60{,}000$; $|\pi_{BookID} Order| = 600$; $|\pi_{CustID} Order| = 1{,}000$; $|\pi_{ShipDate} Order| = 2{,}000$;
- $|Inventory| = 30{,}000$; $|\pi_{BookID} Inventory| = 600$; $|\pi_{WarehouseID} Inventory| = 50$;
- $|Warehouse| = 50$; $|\pi_{State} Warehouse| = 50$.

For each of the following relational algebra expressions, estimate the number of tuples it produces. Note that each estimation may build on the previous ones. You may make the same assumptions as in the lecture on query optimization. You may also make different or additional assumptions, but please state them explicitly.

(b) $\sigma_{ShipDate > :today-60} Order$.
(c) $\pi_{CustID}\left(\sigma_{ShipDate > :today-60} Order\right)$.
(d) $\sigma_{State="NC"} Customer$.
(e) $(\sigma_{State="NC"} Customer) \bowtie \left(\sigma_{ShipDate > :today-60} Order\right)$.

Problem 3 (30 points)

Continuing with Problem 2, but further suppose that:

- Each disk/memory block can hold up to 10 rows (from any table);
- All tables are stored compactly on disk (10 rows per block);
- 5 memory blocks are available for query processing.

(a) Suppose that there are no indexes available at all, and records are stored in no particular order. What is the best execution plan (in terms of number of I/O's performed) you can come up with for the query $\sigma_{ShipDate=:today}(Order \bowtie Inventory)$? Describe your plan and show the calculation of its I/O cost.
(b) Suppose there is a B$^+$-tree primary index on *Order*(*OrderID*) and a B$^+$-tree primary index on *Inventory*(*BookID*, *WarehouseID*), but no other indexes are available. Furthermore, assume that both B$^+$-trees have a maximum fan-out of 100 for non-leaf nodes; each leaf stores 10 rows; and all nodes in both B$^+$-trees are at maximum capacity except the two roots. What is the best plan for the same query in (a)? Again, describe your plan and show the calculation of its I/O cost.

Extra Credit Problem X1 (5 points)

Consider a table R with 100,000 rows, which are stored compactly on disk in exactly 10,000 blocks in no particular order. There are only 20,000 distinct rows in R (other rows are duplicates of these). We wish to compute the following query Q: `SELECT DISTINCT * FROM R;`

(a) What is the minimum amount of memory (in blocks) required to compute Q in one pass (i.e., using only 10,000 I/O's excluding the cost of writing the result)?

(b) Suppose that we only have 1,001 blocks of memory available. Devise a strategy that can compute Q in no more than 20,000 I/O's in the worst case (again, excluding the cost of writing the result).

Extra Credit Problem X2 (15 points)

For this problem, your task is to get Spark to analyze some data about recent votes cast by the U.S. Congress as well as documented explanations (or excuses) provided by legislators for failing to cast some votes (or even casting the wrong votes!). The data came from an API offered by ProPublica, a non-profit newsroom that provides investigative reporting in the public interest. You can find the JSON response files in

```
/opt/dbcourse/examples/congress/propublica/
```

These files were obtained by using the following API endpoints (you do not need to be concerned with all the details, but here is the documentation in case you are interested):

```
https://projects.propublica.org/api-docs/congress-api/votes/#get-recent-votes
https://projects.propublica.org/api-docs/congress-api/votes/#get-recent-personal-explanations
```

To get started, copy the template code to a subdirectory in your workspace in the VM and check that everything is in order (you may replace `~/shared/hw4-x2/` below with any other appropriate path):

```
mkdir -p ~/shared/hw4-x2
cp -r /opt/dbcourse/assignments/hw4-x2/. ~/shared/hw4-x2/
cd ~/shared/hw4-x2/
```

We are interested in three queries:

(a) Count the number of recent explanations by category. Each output tuple should have two components, category and count. Order the output by count (descending) and then category (ascending). Return up to the top 20 results.

(b) Count the number of recent explanations by person. Each output tuple should have four components: name, state, party, and count. Order the output by count (descending) and then name (ascending). Return up to the top 20 results.

(c) For each vote, find which legislators provided explanations for failing to vote or voting incorrectly. Each output tuple should contain the following components: the ProPublica URI of the vote (which uniquely identifies it), date, time, vote question, vote description, vote result, count of how many legislators provided explanations for the vote (which could be 0), and names of these legislators (in a list, which could be empty). Sort the output by count, date, and then time, all in descending order, and return only the top 20 results.

The script `spark.py` parses the JSON files, loads the data into Spark, and answers the queries above using two alternative methods. One method uses the full power of Spark's DataFrame; the other uses only the most basic MapReduce support provided by Spark's RDD. In lecture, we have already gone over the two methods for (a). Their implementations can be found in `a_dataframe.py` and `a_mapreduce.py`. We have also implemented the DataFrame method for (c) in `c_dataframe.py`. Your specific job for this problem is to:

- Implement (b) using the DataFrame method in `b_dataframe.py`;
- Implement (b) using the basic MapReduce method in `b_mapreduce.py`;
- Implement (c) using the basic MapReduce method in `c_mapreduce.py`.

You should modify and submit ***only these three files above***. Read the comments therein to see which specific functions you need to implement.

To run a particular query (`a`, `b`, or `c`) with a particular implementation method (`dataframe` or `mapreduce`), use the following command (with `a` and `dataframe` for example):

```
./spark.py /opt/dbcourse/examples/congress/propublica/ a dataframe
```

You can ignore the harmless warning messages at the beginning. The outputs are delineated by "=====…".