CompSci 316 Fall 2018: Homework #4

100 points (8.75% of course grade) + 20 points extra credit
Assigned: Thursday, November 8
Due: Tuesday, December 4 (except Problems 5 & X2, which are due Thursday December 6)

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 Problems 1 and 5, you will need to use Gradiance. Access Gradiance via the "Gradiance" link on the course website. There is no need to turn in anything else for these problems; your scores will be tracked automatically.

For other problems, you will need to turn in the required files electronically. Please read the "Help → Submitting Non-Gradiance Work" section of the course website, and follow the submission instructions for each problem carefully.

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 (15 points)

Complete the Gradiance homework titled "Homework 4.1 (Indexes)."

Important note: These problems use a definition of "fan-out" that is different from what we discussed in class. When they say "fan-out $n = 3$" they mean that the maximum number of keys per node is 3, and the maximum number of pointers per node is 4 (i.e., "max fan-out is 4" in our terminology).

Problem 2 (20 points)

Consider a table R occupying 1,000,000 disk blocks. There is no index, and 501 memory blocks are available for query processing. Suppose you have the following two options to improve query performance:

1. Buy more memory to increase the number of available memory blocks to 1001.
2. Buy a faster disk to increase the speed of I/O by 20%.

(a) Suppose the objective is to speed up the query "`SELECT * FROM R ORDER BY A;`". Which option is more effective? Briefly justify your answer.
(b) Suppose the objective is to speed up the query "`SELECT * FROM R WHERE R.A > 100;`". Which option is more effective now? Briefly justify your answer.

Prepare a single PDF file for submission. Put your answers to subproblems on different pages.

Problem 3 (25 points)

Consider the following schema for an online bookstore:

Cust (*CustID*, *Name*, *Address*, *State*, *Zip*)
Book (*BookID*, *Title*, *Author*, *Price*, *Category*)
Order (*OrderID*, *CustID*, *BookID*, *ShipDate*)
Inventory (*BookID*, *Quantity*, *WarehouseID*, *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.

$\pi_{Title,Author}$

$\sigma_{(State="NC")}$ and $(Author\ LIKE\ "\%Kondo")$ and $(ShipDate>:today-60)$

$\sigma_{(Cust.CustID=Order.CustID)}$ and $(Book.BookID=Order.BookID)$

$(Cust \times Order \times Book)$.

Suppose we have the following statistics:

- $|Cust| = 3{,}000;\ |\pi_{State}Cust| = 50;$
- $|Book| = 1{,}000; |\pi_{Category}Book| = 10;$
- $|Order| = 60{,}000; |\pi_{BookID}Order| = 1{,}000; |\pi_{CustID}Order| = 3{,}000; |\pi_{ShipDate}Order| = 1{,}000;$
- $|Inventory| = 40{,}000; |\pi_{BookID}Inventory| = 1{,}000; |\pi_{WarehouseID}Inventory| = 50;$
- $|Warehouse| = 50; |\pi_{State}Warehouse| = 50.$

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. If you make different or additional assumptions, please state them explicitly.

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

Prepare a single PDF file for submission. Put your answers to subproblems on different pages.

Problem 4 (20 points)

Continuing with Problem 3, 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);
- 8 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.

Prepare a single PDF file for submission. Put your answers to subproblems on different pages.

Problem 5 (20 points)

Complete the Gradiance homework titled "Homework 4.5 (Concurrency Control and Recovery)."

Extra Credit Problem X1 (5 points)

Suppose we have a set of 2-dimensional points $P = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$. We say that point (x, y) *dominates* point (x', y') if $x \geq x'$ and $y \geq y'$. The *skyline* of the point set P is defined as those points in P that are not dominated by any other point in P, i.e., the subset

$$\{\, (x, y) \in P \mid \neg\exists (x', y') \in P : x' \geq x \wedge y' \geq y \wedge (x, y) \neq (x', y') \,\}.$$

Skyline queries have many applications in real life where objects are compared using multiple criteria. For example, suppose you are interested in cheap restaurants nearby, but it is hard to decide between two restaurants where one is closer than the other but also more expensive. In this case, it is useful to return the skyline of all restaurants (where x would be inverse price and y would be inverse distance).

Say you have a huge number of points stored compactly in a table $P(x, y)$ spanning N disk blocks, in no particular order. For simplicity, you may assume that all x values and y values are distinct. You have M blocks of memory available for processing, but $M \ll N$. Design an I/O-efficient algorithm to compute the skyline of P, and state its complexity in terms of N and M.

There is a naïve algorithm that uses only 3 blocks of memory and performs $O(N^2)$ I/Os. To receive any partial credit, your algorithm must perform no more than $O(N^2/M)$ I/Os. To receive full credit, the complexity of your algorithm must be lower than quadratic in N.

Prepare a single PDF file for submission.

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/dbcourses/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 two 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).

(b) 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 program `spark.py` parses the JSON files, loads the data into Spark, and answers the two 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). Your specific job for this problem is to implement (b) using the basic MapReduce method. You should modify only two functions in `spark.py`: `rdd_votes_with_excuses_map` and `rdd_votes_with_excuses_reduce`. An implementation of (b) using DataFrame has already been provided, and you can use its output as a reference to check your answer.

To run the program, type:

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

It will run (a) and (b) using the DataFrame method first, and then (a) and (b) again using the basic MapReduce method. You can ignore the harmless warning messages at the beginning. The outputs are delineated by "`=====...`". The output for (b) using the basic MapReduce method will come last, and obviously will be empty until you have provided your own implementation in place of the stub implementation.

Submit your modified `spark.py` electronically.