CompSci 316 Fall 2021: Homework 4

100 points (6.25% of course grade) + 25 points extra credit
Assigned: Tuesday, November 16
Due: Tuesday, November 30 (except X2, which is due Tuesday, December 7)

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.

Problems 1-3 and X1 are Gradescope online assignments.

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 (35 points)
Consider tables Drinker (name, address) and Frequents (drinker, bar, times_a_week) from our familiar beer drinker database. Suppose that:

- \(|\text{Drinker}| = 10^5\) and \(|\text{Frequents}| = 5 \times 10^5\).
- Each block can hold 20 rows (from any table), and all rows in each table are stored as compactly as possible.

Answer the following questions. Note that one question may build on another; check each step carefully so you don’t get subsequent questions wrong because of a previous mistake.

(a) How many blocks does Drinker take?
(b) How many blocks does Frequents take?

Suppose that we have 101 blocks of memory to sort Frequents using the external merge sort algorithm described in lecture. If the algorithm already finished in Pass \(k\), you should answer “N/A” for the number of level-\(k'\) runs for any \(k' > k\).

(c) How many level-0 runs does the algorithm produce?
(d) How many level-1 runs does the algorithm produce?
(e) How many level-2 runs does the algorithm produce?
(f) How many level-3 runs does the algorithm produce?
(g) How many passes does the algorithm take? (Note that Pass 0 counts as one pass too.)

We want to use the two-pass hash join algorithm to compute \(\text{Drinker name} \bowtie \text{Frequents}\). Further suppose that the number of bars frequented by each drinker is roughly constant, and that we have a perfect hash function.

(h) What is \(M\), the minimum of number of memory blocks required for the two-pass hash join algorithm to work?
(i) Running under the above configuration with $M$ memory blocks, what is the number of partitions per table the partitioning phase creates?
(j) Running under the above configuration, what is the size of a Drinker partition (in blocks)?
(k) Running under the above configuration, what is the size of a Frequent partition (in blocks)?

Problem 2 (35 points)
Consider tables $Serves (\text{bar, beer})$ and $Likes (\text{drinker, beer})$ from our familiar beer drinker database. Suppose that:
- $|Serves| = 4 \times 10^4; |\pi_{\text{beer}}Serves| = 2000; |\pi_{\text{bar}}Serves| = 10^4$.
- $|Likes| = 2 \times 10^5; |\pi_{\text{beer}}Likes| = 10^5; |\pi_{\text{drinker}}Likes| = 500$.

Using the cardinality estimation techniques described in lecture, estimate the number of rows returned by the following queries.

(a) $\sigma_{\text{beer='Corona'}} Likes$.
(b) $\sigma_{\text{bar='The Library'}} Serves$.
(c) $Likes \bowtie Serves$. Assume containment of value sets for $\text{beer}$.
(d) $\sigma_{\text{bar='The Library'}} (Likes \bowtie Serves)$. Start with your answer for (c) and assume preservation of value sets for $\text{bar}$.
(e) Estimate the size of $\sigma_{\text{bar='The Library'}} (Likes \bowtie Serves)$ again. But now push the selection down, and start with your answer for (b); then estimate join size by assuming containment of value sets for $\text{beer}$.
(f) You may notice that you got different answers for (d) and (e). Discuss which one you feel to be more realistic.

Problem 3 (30 points)
Continuing with the last problem with the same schema and data distribution, we want to find a query plan for executing $\sigma_{\text{bar='The Library'}} (Likes \bowtie Serves)$. Assume the following:
- Each disk/memory block can hold up to 20 rows (from any table).
- All records in the tables are stored compactly on disk (20 rows per block).
- We have only 3 memory blocks for query processing. You need to write the final query result to disk.

First, assume that the records are unsorted and there are no indexes available.

(a) Consider a (very inefficient) query plan where we compute $Likes \bowtie Serves$ using the block-based nested-loop join algorithm; whenever the algorithm produces a join result row, we check the condition $\text{bar} = 'The Library'$. What is the total number of block I/O’s this query plan uses (not counting the cost of writing the final query result to disk)?
(b) Consider an improve query plan that pushes the selection $\text{bar} = 'The Library'$ down when scanning $Serves$. For each block of “qualified” $Serves$ rows, we scan $Likes$ to find join result rows. What is the expected total number of block I/O’s this query plan uses (again not counting the cost of writing result to disk)?
For the last part of this problem, assume instead that Serves is stored in a (primary) B+tree index on (bar, beer); Likes is stored in a (primary) B+tree index on (drinker, beer). No other indexes are available. Each of these indexes has 4 levels.

(c) Describe a further improved query plan that beats the performance of the query in (b). Calculate the expected number of block I/O’s this query plan uses (again not counting the cost of writing result to disk).

Extra Credit Problem X1 (10 points)
Suppose we are given three relations \( R(X, Y), T(Y, Z), S(Z, W) \) with sizes \( N_1, N_2, N_3 \) respectively. Suppose that there are no duplicate tuples within each relation. To compute \( R \bowtie T \bowtie S \), we would traditionally consider two possible query execution plans, with different join orders: \( (R \bowtie T) \bowtie S \) and \( R \bowtie (T \bowtie S) \). However, for some practical database instances, both plans could perform badly. Interestingly, there is a better algorithm that exploits data characteristics more carefully and uses different strategies for processing different parts of the input.

Here are some high-level hints to get you started. Suppose we have \( M \) memory blocks for query processing, and each block can hold \( B \) tuples (from any relation). We set a threshold \( \tau = B \cdot \left( \frac{M}{2} - 1 \right) \). Let us call a value \( y \) for attribute \( Y \) “heavy” if \( |\sigma_{Y=y}R| > \tau \), or “light” otherwise. Now:

1. Sort both \( R \) and \( T \) by attribute \( Y \).
2. For each heavy value \( y \), we compute \( \sigma_{Y=y}(R \bowtie T \bowtie S) \).
3. For light values in \( Y \), we pack them into groups \( Y_1, Y_2, \ldots \) such that each group includes almost \( \tau \) tuples in \( R \), i.e., \( |\sigma_{Y \in Y_i}R| \leq \tau \), and no two groups have any value in common. Assume the light parts of \( R \) and \( T \) are sorted by the group id of their (light) \( Y \) value. Then, for each group \( Y_i \), we compute \( \sigma_{Y \in Y_i}(R \bowtie T \bowtie S) \).

(a) Give a detailed query execution plan for computing \( \sigma_{Y=y}(R \bowtie T \bowtie S) \) in Step 2 above.
(b) Give a detailed query execution plan for computing \( \sigma_{Y \in Y_i}(R \bowtie T \bowtie S) \) in Step 3 above.
(c) Provide a worst-case big-\( O \) cost analysis of this algorithm (given your execution plans above).

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):

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

We are interested in the following 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 state and party. Each output tuple should have four components: state, party, and count. Order the output by state (ascending), and then party (ascending). Return up to the top 20 results.

(c) For each vote, find which legislators provided explanations. 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.

(d) For each vote, find how many legislators provided explanations, and among these, how many provided “Ambiguous or no reason” explanations, and how many provided “Voted incorrectly” explanations. Each output tuple should contain the following components: the ProPublica URI of the vote (which uniquely identifies it), vote description, and the three counts (renamed as `count_total`, `count_ambiguous`, `count_incorrect`, respectively). Note that the counts may be 0. Sort the output by the ProPublica URI (ascending) and return only the first 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), and the DataFrame method for (c). Their implementations can be found in `a_dataframe.py`, `a_mapreduce.py`, and `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`;
- Implement (d) using the DataFrame method in `d_dataframe.py`;
- Implement (d) using the basic MapReduce method in `d_mapreduce.py`.

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

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

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

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

Page 4 of 4