EECS4411: Project 2

By turning in this assignment, I declare that all of this is my own work.

Instructions

You will be writing Relational Algebra for SQL queries before and after they are optimized by the Catalyst, Spark's SQL optimizer.

1. Start a `spark-shell` session and load the Cities and Countries tables, as shown in `a2_starter.scala`. We suggest you copy-paste the loading code into your spark shell or you can also have the shell run all the commands in the file for you with `spark-shell -i a2_starter.scala`).
    - Run `SPARK_233_HOME/bin/spark-shell` from the `qep/` directory (where `SPARK_233_HOME` is the directory where you downloaded and unzipped Spark 2.3.3).
2. Examine `Cities.csv` and `Countries.csv`. Observe the output of `printSchema` on the dataframes representing each table (as in the starter code). `temp` indicates average temperature in Celsius and `pop` is the country's population in millions.
3. For each of the Problem sections below:
    (a) Think about what the given SQL query does.
    (b) Run the query in `spark-shell` and save the results to a dataframe.
    (c) Run `.show()` on the dataframe to inspect the output.
    (d) Run `.explain(true)` on the dataframe to see Spark's query plans.
    (e) Write Relational Algebra for the Analyzed Logical Plan and for the Optimized Logical Plan, in the space provided for each problem.
    (f) Write a brief explanation (1-3 sentences) describing why the optimized plan differs from the original plan, or, why they are both the same.

Use the Relational Algebra (RA) notation as introduced in Lecture 6 on Query Execution (and corresponding book chapters). The output of Spark's query plans does not necessarily map perfectly to our RA syntax. One of the tasks of this assignment is to think critically about the plans that Spark produces and how they should map to RA.
Below are a couple of examples of simplifying assumptions you can make. You are welcome to make other reasonable assumptions.

- The pound + number suffix of fields (e.g. the `#12` in `city#12`) in the query plans are used by Spark to uniquely determine references to fields. This is because a single SQL query can, for instance, have multiple fields named `city` (from aliasing or in subqueries). You should ignore the field number and just use the name in your RA expressions. E.g. treat `city#12` as just `city`.

- `cast(4 as double)` can be just `4.0`
- You can omit `isnotnull` from your select (σ) predicates.

NOTE: We have provided two example queries and their valid corresponding solutions below. Please examine them carefully, as they provide hints and guidance for solving the rest of the problems.

Example 1

```
SELECT city
FROM Cities
```

Analyzed Logical Plan

$\pi_{city}(Cities)$

Optimized Logical Plan

$\pi_{city}(Cities)$

Explanation

The analyzed and optimized plans are the exact same because there is no logical optimization for projecting a single column from a table.

Example 2

```
SELECT *
FROM Cities
WHERE temp < 5 OR true
```

Analyzed Logical Plan

$\pi_*(\sigma_{temp<5 \vee true}(Cities))$

Optimized Logical Plan

$\pi_*(Cities)$

Explanation

$temp < 5 \vee true = true$, so σ selects every row, which is the same as the relation `Cities` itself.
That is: $\pi_*(\sigma_{temp<5 \vee true}(Cities)) = \pi_*(\sigma_{true}(Cities)) = \pi_*(Cities)$

Problem 1

```
SELECT country, EU
FROM Countries
WHERE coastline = "yes"
```

Analyzed Logical Plan

$\pi_{country,EU}(\sigma_{Coastline=yes}(Countries))$

Optimized Logical Plan

$\pi_{country,EU}(\sigma_{Coastline=yes}(Countries))$

Explanation

The optimized plan and the original plan are the same because there is no opportunity for further optimization beyond what the original plan already expresses. Both plans select the "country" and "EU" attributes from the "Countries" relation where the "Coastline" attribute is equal to "yes".

Problem 2

```
SELECT city
FROM (
    SELECT city, temp
    FROM Cities
)
WHERE temp < 4
```

Analyzed Logical Plan

$\pi_{city}(\sigma_{temp<4}(\pi_{city,temp}(Cities)))$

Optimized Logical Plan

$\pi_{city}(\sigma_{temp<4}(Cities))$

Explanation

The original plan first projects the "city" and "temp" attributes from the "Cities" relation and then applies a selection condition to filter out the rows where "temp" is greater than or equal to 4. The optimized plan applies the selection condition directly on the "Cities" relation and then projects only the "city" attribute from the resulting relation.

Problem 3

```
SELECT *
FROM Cities, Countries
WHERE Cities.country = Countries.country
    AND Cities.temp < 4
    AND Countries.pop > 6
```

Analyzed Logical Plan

$\pi_*(\sigma_{(cities.country=countires.country \wedge Cities.temp<4) \wedge countries.pop>6}(Cities \bowtie Countries))$

Optimized Logical Plan

$\sigma_{Cities.country=Country.country}(\sigma_{temp<4}(Cities) \bowtie \sigma_{pop<4}(Countries))$

Explanation

The optimized plan is more efficient because it applies the selection conditions before joining the "Cities" and "Countries" relations. The optimized plan first applies the selection condition on "Cities" to filter out the rows where "temp" is greater than or equal to 4, and then applies the selection condition on "Countries" to filter out the rows where "pop" is less than or equal to 6, before performing the join. This reduces the size of the intermediate relations and improves performance.

Problem 4

```
SELECT city, pop
FROM Cities, Countries
WHERE Cities.country = Countries.country
    AND Countries.pop > 6
```

Analyzed Logical Plan

$\pi_{city,pop}(\sigma_{(cities.country=countires.country \wedge countries.pop>6}(Cities \bowtie Countries))$

Optimized Logical Plan

$\pi_{city,pop}(\sigma_{cities.country=countires.country}(Cities \bowtie \sigma_{pop>6}(Countries)))$

Explanation

The optimized plan is more efficient because it applies the selection condition on "Countries" before joining the "Cities" and "Countries" relations. The optimized plan first applies the selection condition on "Countries" to filter out the rows where "pop" is less than or equal to 6, and then performs the join on the "country" attribute to obtain the desired result, and finally projects the "city" and "pop" attributes. This reduces the size of the intermediate relation and improves performance.

Problem 5

```
SELECT *
FROM Countries
WHERE country LIKE "%e%d"
```

Analyzed Logical Plan

$\pi_*(\sigma_{country LIKE "\%e\%d"}(Countries))$

Optimized Logical Plan

$\sigma_{country LIKE "\%e\%d"}(Countries)$

Explanation

The optimized plan is simpler and more efficient because it avoids unnecessary projections. The original plan first selects the "country" attribute from the "Countries" relation and then applies a selection condition to filter out the rows where "country" does not match the pattern "%e%d". The optimized plan applies the selection condition directly on the "Countries" relation without projecting any attributes first.

Problem 6

```
SELECT *
FROM Countries
WHERE country LIKE "%ia"
```

Analyzed Logical Plan

$\pi_*(\sigma_{country LIKE "\%ia"}(Countries))$

Optimized Logical Plan

$\sigma_{country LIKE "\%ia"}(Countries)$

Explanation

The optimized plan is the simplest form of the query that selects all attributes from the "Countries" relation where the "country" attribute matches the pattern "%ia". Since no projections are necessary, the optimized plan is already the most efficient expression of this query.

Problem 7

```
SELECT t1 + 1 as t2
FROM (
    SELECT cast(temp as int) + 1 as t1
    FROM Cities
)
```

Analyzed Logical Plan

$\pi_{t1+1ast2}(\pi_{cast(tempasint)+1ast1}(Cities))$

Optimized Logical Plan

$\pi_{cast(tempasint)+2}(Cities)$

Explanation

the optimized plan is simpler and more efficient because it avoids an unnecessary projection and uses a constant instead of a computed value. The original plan first projects "temp" as an integer and adds 1 to it, and then again add 1. The optimized plan simply casts "temp" as an integer and adds 2 to it. it is unnecessary and can be simplified to just adding 2 to the integer cast of "temp" instead of 1.

Problem 8

```
SELECT t1 + 1 as t2
FROM (
    SELECT temp + 1 as t1
    FROM Cities
)
```

Analyzed Logical Plan

$\pi_{t1+1.0ast2}(\pi_{temp+1.0ast1}(Cities))$

Optimized Logical Plan

$\pi_{temp+2.0}(Cities)$

Explanation

the optimized plan is simpler and more efficient because it uses arithmetic optimization to simplify the expression. The original plan first selects the "temp" attribute from the "Cities" relation, adds 1.0 to it twice. The optimized plan simplifies the expression by combining the arithmetic operations and directly selects the "temp" attribute, adds 2.0 to it.

Problem 9

Write your own interesting (non-trivial) SQL query that involves self-joins and joins.

```
SELECT A.country, B.pop, C.city, C.temp
FROM Countries as A, Countries as B
WHERE A.country = B.country
Cities as C
ON
C.country = A.country
AND
C.country = B.country
```

Analyzed Logical Plan

$\pi_{country,pop,city,temp}()$

Optimized Logical Plan

Explanation

Problem 10

Write your own interesting SQL query that involves aggregation.

```
SELECT Cities.country, MIN(Cities.temp)
FROM Cities, Countries
WHERE Cities.country = Countries.country
```

Analyzed Logical Plan

$\pi_{Cities.country,MIN(Cities.temp)}(\sigma_{Cities.country=Countries.country}(Cities \bowtie Countries))$

Optimized Logical Plan

$\pi_{Cities.country,MIN(Cities.temp)}(\sigma_{Cities.country=Countries.country}(Cities \bowtie Countries))$

Explanation

The optimized plan is the same as the original plan because there are no additional optimizations that can be made to improve the query's performance or reduce its complexity. The original plan already uses the appropriate relational algebra operators to join the Cities and Countries tables, apply a selection to filter the data based on matching country values, and calculate the average temperature for each country using the AVG() function. Therefore, the optimized plan retains the same operators and expressions as the original plan.