CS 245 Winter 2020 Assignment 2 – Part I

By turning in this assignment, I agree to the Stanford honor code and 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. (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 `part1/` 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 explantation (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. 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 examples of simplifying assumptions you can make. You are welcome to make other reasonable assumptions (if you're not sure, feel free to ask during OH or post on piazza).

- 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

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

Optimized Logical Plan

$Cities$

Explanation

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

Problem 1

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

Analyzed Logical Plan

$\prod_{country,EU}(\sigma_{coastline=\text{``}yes\text{''}}(Countries))$

Optimized Logical Plan

$\prod_{country,EU}(\sigma_{isnotnull(coastline)\wedge coastline=\text{``}yes\text{''}}(Countries))$

Explanation

The optimizer adds $isnotnull(coastline)$ to the predicate. The *null* check filters out rows that have *coastline* column set to *null*. *null* checks are faster to test compared to string equality checks i.e. $coastline = \text{``}yes\text{''}$. The underlying Data Source also can avoid disk IOs if it uses Run Length Encoding or other compression algorithms that allow it to skip rows with certain values without reading the whole block. We will see this benefit when we run the query over tables that have a lot of rows with *null* values in the *coastline* column.

Problem 2

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

Analyzed Logical Plan

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

Optimized Logical Plan

$\prod_{city}(\sigma_{isnotnull(temp)\wedge((int)temp<4)}(Cities))$

Explanation

The query uses a sub-query to project *city* and *temp* columns from the *Cities* table and later filters rows $temp < 4$. Finally the query only projects the *city* column. The optimizer makes 3 changes to the query:

1. The optimizer removes the sub-query as it is redundant. The sub-query unnecessarily forces Spark to read *all* blocks from disk and materialize rows in memory only to later find that those rows may not match the filter criteria $temp < 4$. Removing the sub-query does not change the outcome of the result and avoids unnecessary IOs.

2. The optimizer introduces the $isnotnull(temp)$ check *before* casting it to *int*. Performing this condition check filters rows where the *temp* column is *null*. This can benefit tables that have a large number of *null*s. The underlying DataSource plugin can efficiently skip large number of rows if it uses *RunLengthEncoding* or similar compression algorithm where it doesn't need to read the actual block to filter on the value.

3. The optimizer only projects the *city* column. Depending on the underlying Data Source, we can entirely avoid the cost of reading any columns except *city* and *temp*.

Problem 3

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

Analyzed Logical Plan

$\prod(\sigma_{(((Countries.country=Cities.country)\wedge((int)temp<4))\wedge((int)pop>6))}(Cities \bowtie Countries))$

Optimized Logical Plan

$\sigma_{((isnotnull(temp)\wedge((int)temp<4))\wedge isnotnull(country))}(Cities) \bowtie$
$\sigma_{((isnotnull(pop)\wedge((int)pop>6))\wedge isnotnull(country))}(Countries)$

Explanation

This query is performing a join between *Cities* and *Countries* table on the *country* column. Later it only selects the rows that have $temp < 4 \wedge pop > 6$. In this case the analyzed logical plan first performs a natural join on the *country* column and then applies the predicate to filter the rows. This can be inefficient so the optimizer performs 3 optimizations:

1. The optimizer eliminates the explicit $\prod$ operator. We're reading all columns anyway.
2. The optimizer pushes down the predicate that is applicable for each table. This reduces the overall number of IOs and rows that Spark needs to join. For example, $\sigma_{(int)temp<4)}(Cities)$ may produce a much smaller result set assuming the predicate is highly selectivity.
3. The optimizer introduces *isnotnull* check to both *temp* and *pop* column allowing the underlying Data Source to skip rows that have null values in those columns.
4. Finally, the optimizer adds *isnotnull*(*country*) to the predicates to filter out records that have *null* values in the *country* column. Since this column is used for joining the two tables it has to be non *null*.

By adding *isnotnull* checks and pushing down predicates the optimizer can greatly reduce the number of records that need to be read and joined. This saves on IO, CPU and Memory cost.

Problem 4

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

Analyzed Logical Plan

$\prod_{Cities.city,Countries.pop}(\sigma_{Cities.country=Countries.country \wedge (int)Countries.pop>6}(Cities \bowtie Countries))$

Optimized Logical Plan

$\prod_{Cities.city,Countries.pop}((\prod_{city,country}(\sigma_{isnotnull(country)}(Cities))) \bowtie$
$(\prod_{country,pop}(\sigma_{(isnotnull(pop) \wedge ((int)pop>6)) \wedge isnotnull(country)}(Country))))$

Explanation

In this join query, the optimizer chooses to push down the predicates, employs *null* checks and projects only the *city*, *country*, *pop* columns from their respective tables.

1. As noted in previous problems the predicate pushdown and *null* check can result in big IO savings as the underlying Data Source can skip a lot of blocks if it uses Run Length Encoding or similar compression schemes where it doesn't need to read the block to skip certain values.

2. After the predicate is applied we apply $\prod_{city,country}$ and $\prod_{country,pop}$ to the respective tables. If the underlying Data Source is columnar then it can avoid reading and materializing other columns in memory. Therefore the projections can improve the performance.

3. Finally, the top level projection $\prod_{Cities.city,Countries.pop}$ only projects the columns that are requested.

Problem 5

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

Analyzed Logical Plan

$\prod_{country,pop,EU,coastline}(\sigma_{country LIKE \text{“}\%e\%d\text{”}}(Countries))$

Optimized Logical Plan

$\sigma_{isnotnull(country) \wedge country LIKE \text{“}\%e\%d\text{”}}(Countries)$

Explanation

Here the optimizer eliminates the projection as we're projecting all columns. It also adds $isnotnull(country)$ to the predicate. The *null* check can significantly reduce the IOs to filter out records especially if the underlying Data Source plugin is using Run Length Encoding or similar compression. This check doesn't help reduce IOs if the column is always populated with a non-*null* value.

Problem 6

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

Analyzed Logical Plan

$\prod_{country,pop,EU,coastline}(\sigma_{countryLIKE``\%i\%a"}(Countries))$

Optimized Logical Plan

$\sigma_{isnotnull(country) \wedge EndsWith(country,``ia")}(Countries)$

Explanation

1. Here the optimizer eliminates the projection as we're projecting all columns.
2. It also adds $isnotnull(country)$ to the predicate. The $null$ check can significantly reduce the IOs to filter out records especially if the underlying Data Source plugin is using Run Length Encoding or similar compression. This check doesn't help reduce IOs if the column is always populated with a non-$null$ value.
3. The optimizer also prefers the specialized $EndsWith$ over the $LIKE$ clause. The $LIKE$ clause results into a full regular expression evaluation. Comparatively, prefix and postfix matches can be implemented cheaply.

Problem 7

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

Analyzed Logical Plan

$\prod_{t1+1 \rightarrow t2}(\prod_{(int)temp+1 \rightarrow t1}(Cities))$

Optimized Logical Plan

$\prod_{(int)temp+2 \rightarrow t2}(Cities)$

Explanation

The given query has a sub-query which casts the *temp* column to an Integer and adds 1. The query subsequently reads this column value and adds one to the resulting sum. In this case the optimizer performs the following optimizations leading to a different Optimized Logical Plan:

1. Eliminates the sub-query as it doesn't add any value.
2. Eliminating the sub-query leads to an algebraic expression of the form $temp + 1 + 1$ which can be simplified to $temp + 2$.

This can be accomplished because the original query casts the *temp* column to an Integer. Integer sum is associative and commutative.

Problem 8 (Extra Credit – purely optional)

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

Analyzed Logical Plan

$\prod_{t1+(double)1 \to t2}(\prod_{(double)temp+(double)1 \to t1}(Cities))$

Optimized Logical Plan

$\prod_{(((double)temp+1.0)+1.0) \to t2}(Cities)$

Explanation

This query should be optimized similar to the one in Problem 7 but it isn't.

1. The key difference is that Problem 7's query treats the *temp* column as an Integer while in Problem 8, Spark's Type Inference identifies it as a Double (Floating point value).
2. As we all know floating point arithmetic is not necessarily associative[1]. Therefore the optimizer cannot simplify this expression. This is confirmed by the fact that the optimizer's ReorderAssociativeOperator[2] code only optimizes Integral types[3].

Of note is the following quote from "What Every Computer Scientist Should Know About Floating-Point Arithmetic (Appendix D)"[1].

> Due to roundoff errors, the associative laws of algebra do not necessarily hold for floating-point numbers. For example, the expression (x+y)+z has a totally different answer than $x + (y + z)$ when $x = 10^{30}$, $y = -10^{30}$ and $z = 1$ (it is 1 in the former case, 0 in the latter). *The importance of preserving parentheses cannot be overemphasized.* [emphasis added]

As you can see, Spark's optimizer does not even simplify the expression to remove the parentheses from $(((double)temp + 1.0) + 1.0)$.

[1] https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html
[2] https://github.com/apache/spark/blob/v2.3.3/sql/catalyst/src/main/scala/org/apache/spark/sql/catalyst/optimizer/expressions.scala#L156
[3] https://github.com/apache/spark/blob/v2.3.3/sql/catalyst/src/main/scala/org/apache/spark/sql/catalyst/optimizer/expressions.scala#L188