Report - Project 0

This short report describes the peculiarities of my code. In particular, I will focus on three classes among the six I had to implement, namely:

- Aggregate.scala
- Join.scala
- Sort.scala

This is because the implementation of Scan, Filter, and Project is pretty standard and straightforward. If you are interested in all my code's small details, note that every file is properly commented, and each "block" of logic is described.

Note: I avoided creating functions, and there are chunks of my code that are repeated. Why? Because I suppose that calling a function and returning from it takes more time than sequentially executing the code (fewer jumps). So, is my code ugly? Yes, but if that lets me save some precious milliseconds, I am ok with it.

Aggregate

Goal: implement SQL aggregation, this means implementing the logic behind all the aggregation operators (`COUNT`, `AVG`, `MIN`, `MAX` ...) as well as the `GROUP BY` clause. Please note that I build the entire input table in the `open()` method in my implementation. To do so, I use a simple while loop. Another idea could be to convert the input using its `toIndexedSeq()` method. Apparently, however, this is way less efficient than a simple loop because of some background operations that scan the table many times instead of one. Note also that all my logic is inside the `open()` method: here, I build the result, which will be returned, row by row, by the `next()` method. By doing so, I separate the two logics of building the final result and returning it.

When the initial table is built, there are three situations I have to deal with:

1. **The table is empty** In this case, for each aggregation operator, I have to return its default empty value. I loop over all the operators I have in input, calling the related method to get their empty values, appending them to a tuple that will be the only tuple in the final table.

   Logic: For each operator --> take its empty value and append it to a tuple. Then, once this tuple is built, append it to the final table.

2. **The table is not empty, and I have no** `GROUP BY` In this case, for each aggregation operator, starting from the input table, I build a temporary table containing, for each row, only the field(s) related to the operator itself. An interesting line of code, here is `intermediateResult :+= consideredValues.reduce(agg.reduce)`: basically `intermediateResult` is the tuple that contains the result for each operator (once completed, it will be the only row of the output table), `consideredValues` is the temporary table mentioned before, and I call the Scala `reduce` function on it, that expects a two parameters function that returns a single output of the same type of the two inputs. This is exactly what the `reduce` method of each aggregation operator does. So, with this simple line, I get the result of the aggregation. After the entire `intermediateResult` is built, I append it to the final table, and that's it.

   Logic: For each operator --> build the temporary table and then reduce it, appending the result to `intermediateResult`. Then, once this tuple is built, append it to the final table.

3. **The table is not empty, and I have to deal with a** `GROUP BY` The logic is the same as the one described before. The only thing that changes is that now I have to create the groups first and then apply that logic to each group. To make the groups, I apply the `groupBy()` Scala built in method to my input table, using the fields in the `groupSet` as keys. This returns a hash map on which I then iterate.

   Logic: Create the map containing the groups. For each group --> For each operator --> build the temporary table and then reduce it, appending the result to `intermediateResult`. Then, once this tuple is assembled, append it to the final table.

   So my final table will contain, as expected, a row for each group.

Join

Goal: Implement an SQL `JOIN` operation, considering it is always an *equijoin*.

Here, I use the implementation of a simple Hash Join, faster than two nested loops that check the condition on each row of the left table, paired with all the right table rows. The algorithm implemented is described here: https://rosettacode.org/wiki/Hash_join#Scala

So, in the `open()` method of the class, I build the two tables (left and right), and then I join them with the algorithm mentioned before. Of course, I return the result, row by row, in the `next()` method.

Sort

Goal: Implement SQL `ORDER BY`, `FETCH` and `OFFSET` clauses.

Regarding the `ORDER BY` clause, this can involve more than one field, and the order can be either `DESCENDING` or `ASCENDING` for each of the fields. First of all, I check that, in case the query contains a `FETCH`, this is not 0. If it is, I don't need to do any of the operations I am going to mention, so I return from the `open()` method, with an empty result table (why sorting the input table, or even creating it, if I am not going to return anything?). If this condition is false, I create the input table, as always, with a while loop. Then I revert the `collation` input list so that the last field to sort (the "least important") is the first in the list. After that, for each element of `collation` I apply the built in `sortWith` Scala function to the table, checking if the order is descending or ascending. So basically, the schema is: sort for the "least important" field, then for the second "least important", then for the third... and so on. Once the table is sorted, I need to deal with `FETCH` and `OFFSET`. The logic is the following:

- `OFFSET` I have a pointer in my code (`resultIndex`) that I use to return the next row of the resulting table in the `next()` method. If I have an offset, I start returning from its index, not from the first row.
- `FETCH` I have another integer variable in my code (`resultFetch`) containing the fetch's input value, if present, or -1 as default. When there's a `FETCH` SQL clause in the query, every time I return a tuple in the `next()` method, I decrement this `resultFetch` counter, so that, when it goes to 0, even if the `resultIndex` is not equal to the size of the output table, I know I have to stop returning tuples from the table, returning `NilTuple` instead.

Conclusion

My code should be pretty easy to understand. I could have implemented some fancier algorithms, for example to perform the `SORT` operation better. Still, when I submitted my code (commit `#b6ce0cfa`), I achieved a 1.09x improvement over the baseline that, at the time I am writing this short report (06/03/2021), is still the best improvement in the platform. I don't think that different algorithms would help me a lot in achieving a better score.

Giorgio Mannarini