CS-422 Database Systems

Project II

Deadline: 19th of May 23:59

The aim of this project is to expose you to a variety of programming paradigms that are featured in modern scaleout data processing frameworks. The project contains three tasks; in the first you have to implement a rollup operator, in the second a theta join operator, and in the third a near neighbor algorithm. All these tasks have to be implemented over **Apache Spark**. For the distributed execution of Spark we have set up a 10-node cluster on IC Cluster. We will give you access and usage details during the third week of your project. A skeleton codebase on which you will add your implementation is provided along with some test cases that will allow you to check the functionality of your code. You may find the skeleton code in `https://gitlab.epfl.ch/DIAS/COURSES/CS-422/2020/Project-2-group-<ID>`. In what follows, we will go through the three tasks that you need to carry out.

Task 1 (20%) Implementation of a ROLLUP operator

In this task you need to implement a ROLLUP operator over Spark. The ROLLUP operator is an extension of the GROUP BY, it produces the super-aggregates for a given set of attributes, having subtotals and grand total in the output. It is defined as follows:

```
(v1 ,v2 ,...,vn, f()),
(v1 ,v2 ,...,ALL, f()),
...
(v1 ,ALL,...,ALL, f()),
(ALL,ALL,...,ALL, f()).
```

For example, consider the inventory data in table 1. The following SQL query calculate total quantity of units, grouped by supplier and product as the output is shown in table 2.

```
SELECT
    supplier, product, SUM(units)
FROM
    inventory
GROUP BY
    supplier, product
```

supplier	product	color	units
Digitec	iPhone	black	20
Digitec	iPhone	white	30
Digitec	Samsung Note	black	25
Digitec	Samsung Note	white	32
Amazon	iPhone	black	15
Amazon	iPhone	white	50
Amazon	Samsung Note	black	18
Amazon	Samsung Note	white	28

Table 1: Inventory Table

supplier	product	SUM(units)
Digitec	iPhone	50
Digitec	Samsung Note	57
Amazon	iPhone	65
Amazon	Samsung Note	46

Table 2: Output of Group-By SQL Query

Using ROLLUP operator for the same grouping-set, the output also contains sub-totals and grand-totals, that is, super-aggregates for the grouping set.

```
SELECT
    supplier, product, SUM(units)
FROM
    inventory
GROUP BY
    ROLLUP(supplier, product)
```

supplier	product	SUM(units)
Digitec	iPhone	50
Digitec	Samsung Note	57
Digitec	ALL	107
Amazon	iPhone	65
Amazon	Samsung Note	46
Amazon	ALL	111
ALL	ALL	218

Table 3: Output with ROLLUP operator

In this task, the implementation of your ROLLUP operator should support the following aggregate functions: *COUNT*, *SUM*, *MIN*, *MAX*, *AVG*. Note that *COUNT*, *SUM*, *MIN* and *MAX* are distributive functions, that is they allow the input set to be partitioned into disjoint sets that can be aggregated separately and later combined, and *AVG* is an algebraic function that can be expressed in terms of other distributive functions (*SUM* and *COUNT*).

Tasks. You need to complete the following tasks in the `rollup.RollupOperator` class.

- In `rollup.RollupOperator.rollup_naive` method, given a set of input attributes, compute the ROLLUP results, that is, aggregates and super-aggregates.
- In `rollup.RollupOperator.rollup` method, optimize your naive implementation by exploiting parent/child relations among the group-bys. For example, the grouping AB, can be computed from ABC. Therefore, each group-by computation must exploit the already computed group-bys.
- Compare the naive and optimized implementation of the ROLLUP operator. Specifically, evaluate and compare the performance by varying the following parameters and explain your results succintly in the report (Project2.pdf):
    1. Input size (number of tuples).
    2. Number of ROLLUP attributes.

The ROLLUP operator gets as input the dataset, the grouping attributes, the aggregate attribute and the aggregate function. Check the code skeleton for the interfaces, input and output types. Your algorithm must return the full materialized ROLLUP results.
Your implementation must rely on the basic Spark RDD API (you are not allowed to use SparkSQL or Dataframes). You can only use SparkSQL for the extraction of the schema (given in the skeleton). However, you can use the multi-dimensional aggregate operators of the Spark SQL API to test the correctness of your implementation.

Dataset. "lineorder.tbl", CSV format where the fields of each tuple are separated by "|". LINEORDER has 17 attributes: orderkey, linenumber, custkey, partkey, suppkey, orderdate, orderpriority, shippriority, quantity, extendedprice, ordertotalprice, discount, revenue, supplycost, tax, commitdate and shipmode. A test dataset is available under `src/test/resources/lineorder_small.tbl` and you can download two bigger datasets through the following link:
`http://diaswww.epfl.ch/courses/cs422/2018/project2/input/lineorder_[medium|big].tbl`

Assumptions. In our tests the number of ROLLUP dimensions $|D|$ will not be very large. Your implementation has to work for the small and medium dataset, whereas the big one is optional.

Task 2 (40%) Implementation of a theta join operator

In this task you need to implement the 1-Bucket-Theta algorithm described in Section 4 of the paper `https://dl.acm.org/citation.cfm?id=1989423`. Specifically, you need to implement a theta-join operator over Spark RDD that supports "<", ">" conditions. Since these operators involve the computation of the cartesian product, the point of this task is to partition the cartesian product across the reducers. Given relations S and R and r reducers,

the cartesian product is a matrix M, where the rows represent the values of the join attribute of relation R and the columns represent the values of the join attribute of relation S.
You need to implement a variation of the 1-Bucket-Theta algorithm (Algorithm 1 of the paper) that also exploits statistics to prune unnecessary comparisons:

(a) Compute the number and the dimension length of the horizontal and vertical rectangles. The number of horizontal and vertical blocks is $c_R = |R|/\sqrt{|S||R|/r}$ and $c_S = |S|/\sqrt{|S||R|/r}$ respectively. The size of the blocks will be $|S||R|/r$.

(b) Compute the horizontal and vertical boundaries of the blocks. Given the values c_R, c_S computed in (a), compute the boundaries by extracting a random sample from each relation. The size of the sample will be $c_R - 1$ and $c_S - 1$ respectively. Sort the resulting values and constrcut the boundaries based on them.

(c) For each incoming R-tuple, you need to choose a row in matrix M. Consult the bucket boundaries and randomly pick a row belonging to the boundary range of the tuple. Then, based on the algorithm, identify the partition to which that row belongs. Afterwards, create an output tuple for each partition that intersects with this row. Similarly, for each incoming S-tuple you need to randomly choose a column in M within the boundary range of the tuple. Then, map the S-tuple to all the partitions that intersect with that column.

(d) Partition the datasets using as key the corresponding bucket number that you have computed in step (c).

(e) Perform the theta-join operation locally in each partition. Optimize your implementation in order to (i) prune non-qualifying partitions and (ii) prune the number of pair comparisons within a partition.

Put your implementation under the `thetajoin.ThetaJoin` class. The theta join operator will get as input the two datasets, the join keys, and the join condition. Check the skeleton and the provided usage example. Your algorithm must return the pairs of values that satisfy the given condition. Your results must satisfy the SQL semantics, that is your result must be exactly the same as the one of the cartesian product followed by the condition. Your implementation for this task must be at the RDD level. Spark SQL can **only** be used for the extraction of the csv dataset (already given in the skeleton).
Evaluate your implementation by modifying the number of reducers and succinctly mention the performance results in Project2.pdf.

Assumptions. Assume that |S|, |R| are multiples of $\sqrt{|S||R|/r}$ and also $|S| \cdot |R|$ is a multiple of r. Thus, the cartesian product can be divided into r square blocks of equal size $(\sqrt{|S||R|/r} \times \sqrt{|S||R|/r})$. In addition, for this exercise you assume that the join condition is over integer values, thus you don't need to take into consideration any other datatypes.
You can test your implementation using the test datasets which are located in `src/main/resources/tax[A|B]4K.csv`. These datasets contain four attributes `(name, areaCode, salary, tax)` representing the tax information of employees. Extract subsets of 1K, 2K, 3K and 4K rows to better evaluate your implementation.

Task 3 (40%). Near Neighbor query processing

In this exercise, you are given a subset of the IMDb dataset that contains a list of keywords for each movie. You are interested in processing near-neighbor queries to find movies with similar movie sets based on Jaccard similarity. You need to evaluate i) the runtime performance of approximate LSH-based queries compared to exact solutions, and ii) the impact of different data distribution techniques.

Dataset. You are given different data files "lsh-corpus-*.csv" and query files "lsh-query-*.csv". All files follow the same format: each line corresponds to a movie and consists of the name of the movie followed by a pipe-separated ('|') list of keywords. You need to build suitable data structures by using the data files, and then process near-neighbor queries for each movie in the query files. Run query files 0-2 with the small dataset, queries 3-5 with the medium dataset and queries 6 and 7 with the large dataset.

Tasks. You need to complete the following steps for this tasks:

1. Implement a naive near-neighbor algorithm in the class **ExactNN**. The naive algorithm computes the distance of each query point against all the available data points and returns the set of movies with Jaccard similarity above a threshold.

2. Implement LSH for a single $(a_1, a_2, 1 - a_1, 1 - a_2)$-sensitive hash function in the class **BaseConstruction**. First, you need to produce a consistent perturbation of the keywords for computing *MinHash*. Then, you should use the *MinHash* values to index the data points. Finally, you need to use to perturbation to compute the *MinHash* of each query point and retrieve matching movies as approximate near-neighbors.

3. Implement a broadcast-based LSH scheme in the class **BaseConstructionBroadcast**. Assume that the data points fit in the memory of each executor. Write an alternative implementation of LSH that reduces query point shuffles.

4. Implement the composition of different AND and OR constructions. In the class **ANDConstruction**, you need to compute the AND construction of other LSH functions, simple or composite. In the class **ORConstruction**, you have to do the same for the OR construction. We provide an average precision/recall requirement for query files 0-2 (in the methods of Main). Find a construction that satisfies the given requirements. **[Note: Due to randomized perturbations, the assertions might fail even for correct solutions. We are interested in your way of thinking rather than an accuracy guarantee]**

5. Evaluate the performance and accuracy of the partitioned and broadcast-based implementations of LSH compared to pairwise comparisons. Plot the execution time for different implementations. In addition, measure the average distance of each query point from each nearest neighbors and report the difference between the exact and approximate solutions. When is each method preferable?

Deliverables

We will grade your last commit on your GitLab repository. Your implementation must be on the **master** branch. **You do not submit anything on moodle**. Your repository must contain the **Project2.pdf**.

Grading: Keep in mind that we will test your code automatically.