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 cube operator, in the second a similarity join operator and in the third stratified sampling. 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.
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://github.com/CS-422/CS422-Project2`. In order to compile and execute your code, check carefully the given readme file. In what follows, we will go through the three tasks that you need to carry out.

Task 1 (20%) Implementation of a CUBE operator

In this task you need to implement a CUBE operator over Spark. Your CUBE 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). You will implement a two-phase MapReduce Algorithm, MRDataCube proposed by Suan Lee et al. [`http://ieeexplore.ieee.org/document/7072817/`], that exploits parent/child relations among the group-bys. Figure 1 shows a three attribute cube (ABC) and the options for computing a group-by from a group-by having one more attribute called its *parent*. For example, AB, AC and BC can be all computed from ABC.
The **first phase** of the algorithm computes the group-by corresponding to the bottom cell of the search lattice (ABC in the example of Figure 1) and generates partial results for the rest of the lattice cells from the raw input data. The map function emits a single <key, value> pair for each tuple, where the *key* is the combination of the values of all the $|D|$ attributes that are present in the cube (3 in the example of Figure 1) and the *value* is the result of the aggregate function applied on the tuple (e.g. for COUNT the value is set to 1). Then, a combine function performs mapper-side aggregation to decrease the load on the shuffle and reduce phases by aggregating the values of <key, value> pairs that have the same key and merging them into one pair. The shuffling step sorts the <key, value> pairs,

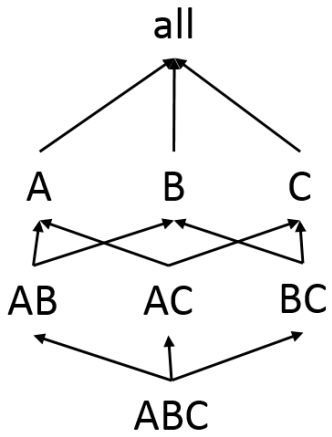

Figure 1: A search lattice for the cube operator.

merges <key, value> pairs with the same key into one pair, and sends these pairs to the reduce function. The reduce function calculates the result corresponding to the bottom cell of the search lattice and extracts from it partial results for the rest of the cells. These partial results are output in the form of <key, value> pairs, where the *key* denotes the cell and is the combination of the cell attribute values and the *value* is the extracted partial result. Figure 2 shows an example of this phase.

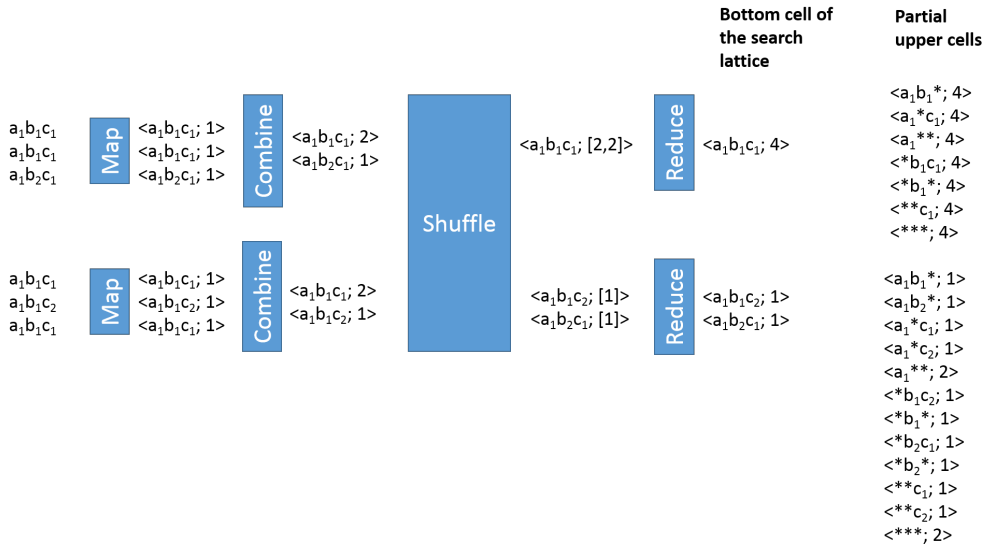

Figure 2: An example of the first phase corresponding to the lattice of Figure 1.

The **second phase** of the algorithm takes as input the result of the first phase and merges the partial results to produce the final values for each cell. More specifically, <key,value> pairs with the same key are first reduced locally and then shuffled across partitions to generate the full data cube. Figure 3 shows an example of this phase. *Note: Be careful on how you compute the final result out of partial ones in the case of AVG.*

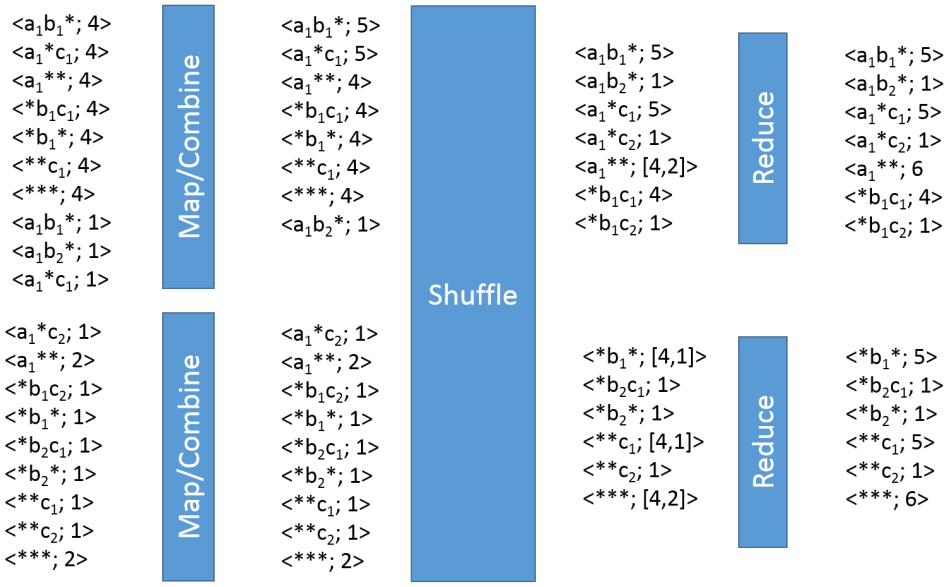

Figure 3: An example of the second phase corresponding to the lattice of Figure 1.

Put your implementation under the `cubeoperator.CubeOperator` class. The cube operator gets as input the attributes on which the group-bys are performed, the attribute to be aggregated and the aggregate function. Check the skeleton for the usage of the algorithm. Your algorithm must return the full materialized data cube. 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.
You also need to implement (and include in your submission under the CubeOperator class) the naive algorithm that is described in this paper by Nandi et al. (Algorithm 1) which computes all the required group-bys directly from the raw data and then compare the execution time of this naive algorithm with that of your optimized algorithm. More specifically, evaluate and compare the performance by varying the following three parameters and explain succinctly your results in the report (Project2.pdf):

1. Input size (number of tuples).
2. Number of CUBE attributes (the dimension D of the CUBE).
3. Number of reducers.

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. You will extract the schema of the LINEORDER table using SparkSQL (code provided in the skeleton). A test dataset is available under `src/main/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`

Example Query.

```
SELECT lo_suppkey, lo_shipmode, lo_orderdate, SUM (lo_supplycost)
FROM LINEORDER
CUBE BY lo_suppkey, lo_shipmode, lo_orderdate
```

Assumptions. In our tests the number of CUBE 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 (50%) Implementation of a similarity join operator

In this task you need to implement a scale-out similarity join operator which is optimized using a variation of the *clusterJoin* algorithm described in the paper `https://dl.acm.org/citation.cfm?doid=2732977.2732981`. You will implement a simplified version of the algorithm that focuses on pruning away dissimilar values.
Given a CSV file, the algorithm gets as input the required number of clusters, the distance threshold and executes a similarity self join on a given column. The algorithm returns pairs of values which are similar.

- Implement the algorithm through the following main steps:
  (a) Extract a random sample of anchor points; the number of anchors *numAnchors* will be given as input to the algorithm. Given the *numAnchors* value you will compute the probability p_A based on which you will randomly sample the anchor points from the dataset. Thus, the *numAnchors* value represents the approximate size of the anchor set.
  (b) Assign each element to the closest anchor point and construct the partitions. In this step you need to also take into consideration the outer partitions of each element. The outer partitions of an element Q are the ones whose anchor point C satisfies the formula: $c \leq x + 2 * dthreshold$, where c is the distance between C and Q, x is the distance between Q and the closest anchor, and *dthreshold* is the distance threshold. Since in this step you might assign a point to multiple partitions, you need to apply a minimal number of similarity checks, i.e., you need to avoid checking the similarity of two elements multiple times.
  (c) Execute the similarity join locally within each cluster.
- Experiment by varying the number of anchor points, as well as the size of the dataset, and describe what you observe.

Assumptions. Assume that the column on which the similarity join is applied contains only string values. Assume edit distance as a distance function.

Add your code under the `simjoin.SimilarityJoin` class. You can test your implementation using the test datasets which are located under `src/main/resources` folder. These datasets contain a single column with author names. Specifically, we provide a smaller version of the dataset (`dblp_small.csv`) and a bigger one (`dblp_10K.csv`). For the experiments you need to construct subsets of `dblp_10K.csv`, e.g., `dblp_1K.csv`, `dblp_5K.csv`, etc., optionally up to `dblp_10K.csv`. You can construct the subsets, by taking the first 1K, 5K, etc. rows.

Task 3 (30%). Approximate query processing

In this exercise, you have to utilize approximate query processing to speed-up the execution of TPC-H queries `https://github.com/electrum/tpch-dbgen/tree/master/queries`. There are 3 steps in this task

1. Implement stratified sampling. For the implementation of the stratified sampling, follow the algorithm described here `http://proceedings.mlr.press/v28/meng13a.pdf`.

2. Calculate the set of samples usable by the provided query set. To calculate all the usable samples utilize the theoretical background provided by BlinkDB.

3. Write an algorithm which given a storage budget and accuracy requirements will choose the best set of samples to create and use. The set should minimize the total execution time of the queries. To decide on the optimal set of samples you are free to implement the algorithm of your choice.

Implementation. Under `sampling/Sampler`, you need to implement the algorithm for deciding which samples to create as well as compute the stratified samples. The generated samples should be persistent and should not be recomputed during query execution. Sampler's `sample` function should return a pair consisted of 1) the list of generated samples and 2) an object of your choice, which you can use to pass information regarding the samples-queries correspondence from the sampler to the executor.
Under `sampling/Executor`, you need to implement the missing functions to compute approximate answers to the TPC-H queries. Each query response should contain all the groups contained in the exact solution and the value in each block should not exceed $\pm e\%$ with a $ci\%$ confidence level, where $e\%$ and $ci\%$ will be provided to you at runtime as part of the `desc` argument and will be the same across queries and the same with the ones provided to the sampler. The `params` list will contain the values of the query arguments. `desc`'s `sampleDescription` field contains the second value of the pair returned by the sampler.
If you can not provide the requested accuracy using your samples, you are allowed to answer queries using the original tables. The executor should be able to answer TPC-H queries, with the exception of TPC-H queries Q2, Q4, Q8, Q13-16, Q21, Q22. Each query will be submitted with the same frequency.

Deliverables

- Project2.zip: A self-contained zip file with your project.
- Project2.pdf: A short report.

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