CS-422 Database Systems

Project II

Deadline: May 20th at Midnight

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 theta join operator and in the third a data streaming pipeline. All these tasks have to be implemented over **Apache Spark**. For the distributed execution of Spark we have set up a virtualized multi-node environment, using multiple Docker containers. You may find more information on how you can setup the virtualized cluster on your machine under `Project2/docker/README.md`.
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/Project2`. In order to compile and execute your code, check carefully the given readme files. 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

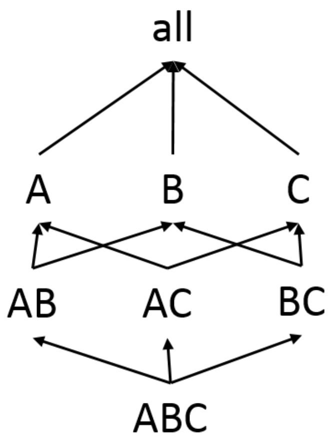

Figure 1: A search lattice for the cube operator.

same key and merging them into one pair. The shuffling step sorts the <key, value> pairs, 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.

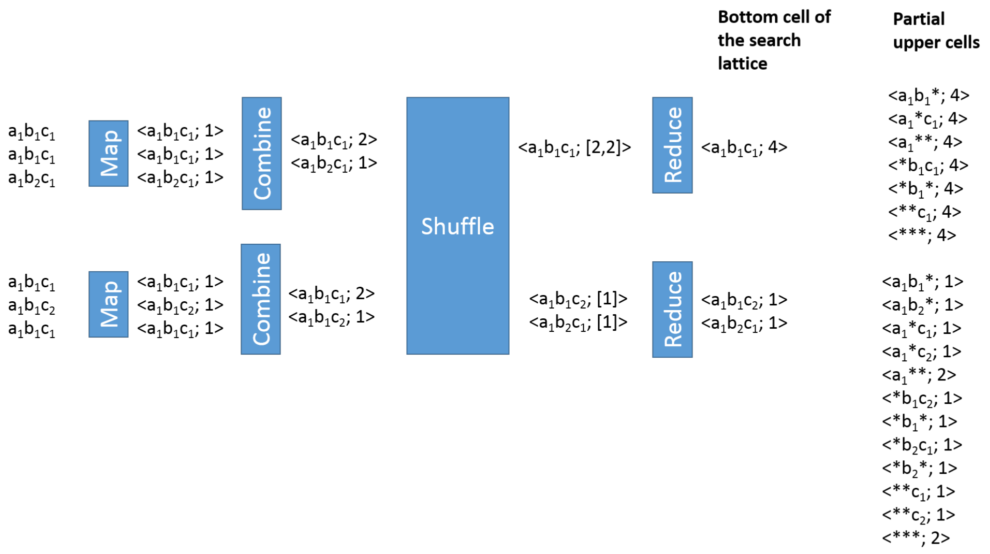

Figure 2: An example of the first phase corresponding to the lattice of Figure 1.

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.*

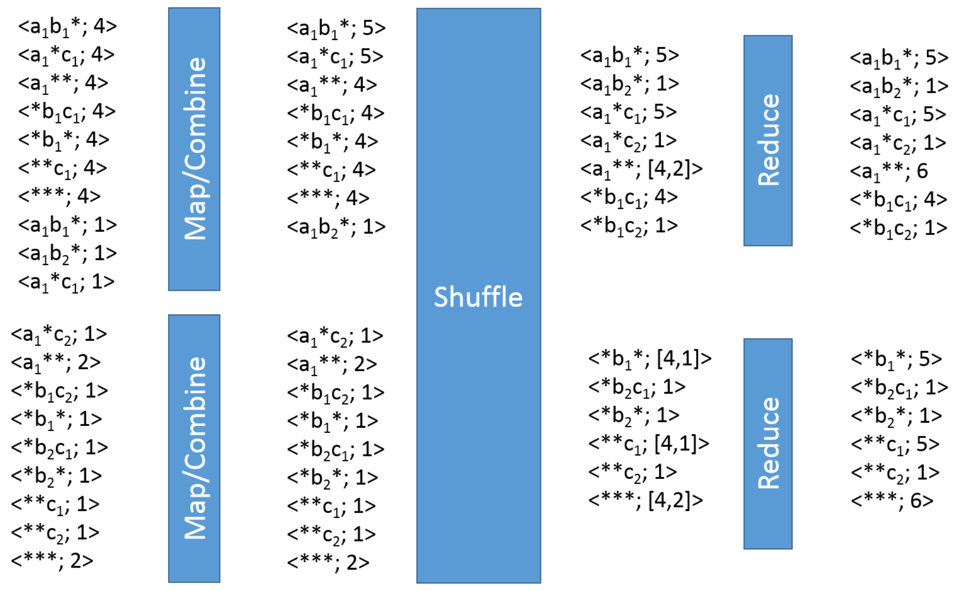

Figure 3: An example of the second phase corresponding to the lattice of Figure 1.

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 OLAP 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/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`

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 theta join operator

In this task you need to implement the M-Bucket-I algorithm described in Section 5 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 "$=$", "$<$", "$>$", "$<=$", "$>=$", "$\neq$" conditions. Since these operators (apart from "$=$") involve the computation of the cartesian product, the point of this task is to balance the partitioning of 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.
For the M-Bucket-I algorithm you need to implement the following:

(a) Compute the equi-depth histogram for R and S. Given values cr, cs, where $|S| = cs{*}\sqrt{|S||R|/r}$ and $|R| = cr{*}\sqrt{|S||R|/r}$, compute the horizontal and vertical boundaries of the matrix M. You can compute the boundaries by extracting a random sample of cr and cs values from the relations R and S respectively. Then, you need to compute and keep the number of values from the corresponding dataset that fall in each bucket.

(b) Implement the algorithm M-Bucket-I using the histogram of (a). The algorithm gets as input the number of reducers and the maxInput value which denotes the maximum size of each bucket. The algorithm tries iteratively to find an optimal assignment of buckets to reducers. Specifically, the algorithm starts looping over the number of rows of the dataset and in each loop i it tries to compute the buckets; the index i of the loop will stand as one dimension of the bucket and the other dimension will be maxInput$/i$. You may find as an example the first two iterations of the algorithm in Figures 4 and 5. In each iteration you must also keep a score which represents the number of cells that are candidates for satisfying the join condition (e.g., the blue cells of Figures 4, 5 are candidate cells for the equality condition). In order to compute this score you must consult your histogram. Then, after each loop of maxInput rows you must keep the buckets that have the maximum average score. This is useful in order to avoid having buckets which contain no candidate cells. For example in Figures 4, 5 we have:

```
1st iteration: b1 = (starti = 0, endi = 0, startj = 0, endj = 5)

               candidate cells for b1 = 4
               score = 4/1=4


2nd iteration: b1 = (starti = 0, endi = 1, startj = 0, endj = 2)
               b2 = (starti = 0, endi = 1, startj = 3, endj = 5)
               b3 = (starti = 0, endi = 1, startj = 6, endj = 8)

               candidate cells for b1 = 6
               candidate cells for b2 = 4
               candidate cells for b3 = 3
               score = 13/3=4.3

3rd iteration: b1 = (starti = 0, endi = 2, startj = 0, endj = 1)
               b2 = (starti = 0, endi = 2, startj = 2, endj = 3)
               b3 = (starti = 0, endi = 2, startj = 4, endj = 5)
               b4 = (starti = 0, endi = 2, startj = 6, endj = 7)
               b3 = (starti = 0, endi = 2, startj = 8, endj = 9)

               candidate cells for b1 = 6
               candidate cells for b2 = 6
               candidate cells for b3 = 4
               candidate cells for b4 = 4
               candidate cells for b5 = 2
               score = 22/4=5.5
4th iteration: ...
               score = 31/9= 3.44
...
```

Thus, the algorithm must select the assignment with the best score (third one) and continue to the next maxInput loop. For more details you may consult Algorithms 4, 5 and 6 of the paper. Regarding this step you are allowed to use any reasonable heuristic to reduce the number of loops (e.g., choose as rowsize values which are perfect divisors of maxInput).

(c) Assign the values of each dataset to the corresponding bucket by taking into account the new boundaries now. In order to assign the values to the new buckets, you need to also keep a sorted version of your dataset.

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

(e) Perform the theta-join operation locally in each partition.

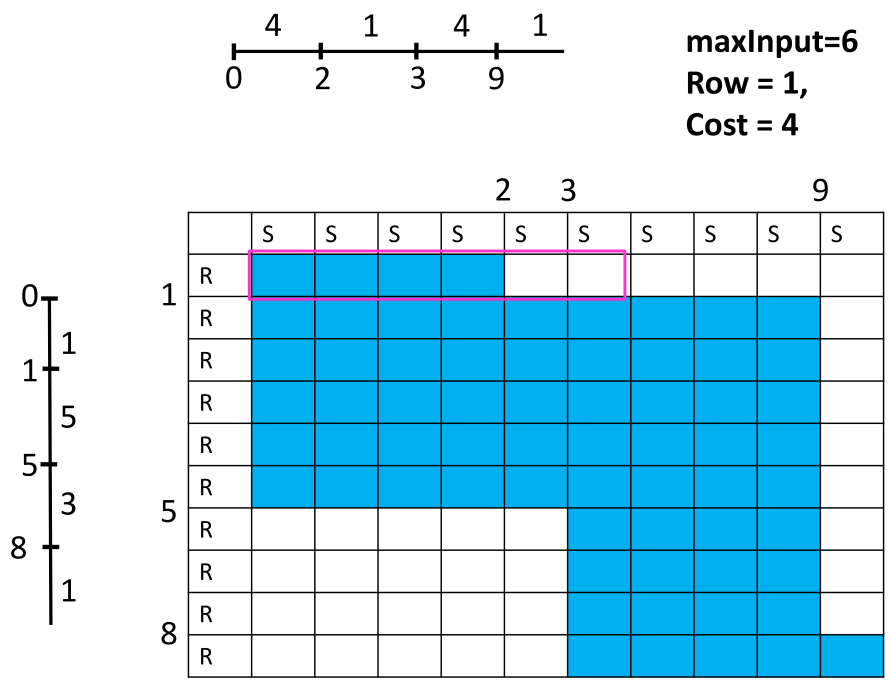

Figure 4: First iteration

Put your implementation under the `thetajoin.ThetaJoin` class. The theta join operator will get as input the join keys and the join condition. Check the skeleton and the provided tests for the usage of the algorithm. 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 schema (given in the skeleton).
Experiment with several combinations of maxInput/number of reducers and succinctly mention the performance results in Project2.pdf.

Assumptions. Assume that |S| and |R| are multiples of $\sqrt{|S||R|/r}$. 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. Also, for the M-bucket-I algorithm you don't need to find the optimal placement for the given number of reducers; it is sufficient to produce a set of buckets for the maxInput value which is given as input even though the resulting number of buckets you produce is larger than the number of reducers. Finally, you can assume that the buckets have rectangular shape and the buckets you produce in each iteration have fixed dimensions.
You can test your implementation using the test datasets which are located in `src/test/resources/input[1|2]_*K.csv`. These datasets contain two attributes `(id, num)` where the `num` is a random number created using the zipf distribution. The size of the datasets is 1K, 2K, 3K and 4K rows respectively. Your implementation must work over the small datasets (1K, 2K), whereas the bigger ones are optional.

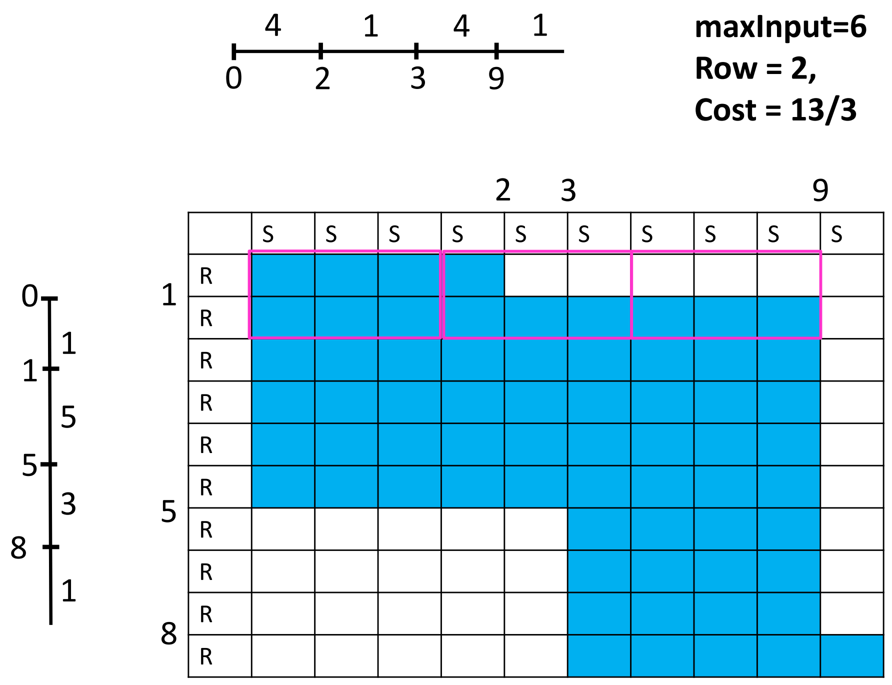

Figure 5: Second iteration

Task 3 (30%). Implementation of a data streaming pipeline

In this exercise, you will implement a data streaming pipeline. Our pilot application is network monitoring. Internet service providers and large companies frequently deploy network monitoring algorithms for protecting their network from malicious attacks and unlawful actions, and for identifying possible mis-configurations and faults. These algorithms are installed at the edge routers of the corporate network, for keeping track of all incoming and outgoing networks.
Specifically, in the context of this exercise, you have to solve the problem of calculating "heavy-hitters"; given a stream of IP pairs, you want to find the pairs that occur most frequently. As routers often have limited computational capabilities and storage size, you will implement two approaches (i) precise calculation, (ii) approximate calculation. To implement your approaches you will have to use the Spark Streaming interface.

StreamGenerator To imitate the existence of a stream, we provide you with a java application for generating a stream, The java application you can use is in the repository under `CS422-Project2/docker/cs422-pr2/jars/streamgenerator.jar`
It takes 5 arguments:
(i) HDFS uri, (ii) input file, (iii) output directory, (iv) number of rows per batch, (v) seconds between batches
To execute in the docker run:

```
java -jar ./jars/streamgenerator.jar \
hdfs://master:9000 /samples/stream.tsv \
/stream_input 50 10
```

Code Skeleton We provide you with the skeleton for your solution. The skeleton takes at least 4 attributes (you will add more):
(i) input directory, (ii) number of seconds per window, (iii) Top-K, (iv) execution type ("precise" or "approx")
To execute in the docker run:

```
./spark-2.2.1-bin-hadoop2.7/bin/spark-submit \
--class streaming.Main ./jars/cs422-project2_2.11-0.1.0.jar \
/stream_input 5 5 precise
```

Output For each of the batches you will compute (each RDD of a batch) you will print out two lines one for the current batch and one for the global stream each line will have the following format: [(number of appearances, (IP1, IP2)), (...)] e.g.:

```
This batch: [(6,(111.37.249.138,3.249.158.125)),(5,(43.9.189.51,201.241.252.94))]
Global: [(15,(43.9.189.51,201.241.252.94)),(13,(43.139.101.116,1.103.139.4))]
```

Input We provide you with a test input under:
`CS422-Project2/docker/cs422-pr2/workloads/samples/stream.tsv`
and it is automatically loaded to HDFS in the docker.

(a) Precise approach:
In this first part of the task you will have to calculate the partial and global top K heavy-hitters precisely for every execution of the streaming context.

(b) Approximate data structures:
In this part of the task you will have to develop a set of structures that will reduce the storage requirements of storing the partial and global information of the stream. Specifically, you will have to implement a Count-Min sketch and use it to calculate the partial and global top-K IP pairs. In order to set up the cm-sketch correctly you will have to pass more variables as arguments to the program.

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.