Milestone 2 Squall/Storm project proposal: Theta joins in Squall

1 Introduction

The goal of this project is to implement a two-way theta-join operator in Squall/Storm. Squall/Storm is a distributed online processing system. You can find a detailed description of the system, as well as installation steps in the squallQuickStart document.

You will base your implementation on the paper "Processing Theta-Joins using MapReduce" [1], run experiments and compare it with the traditional approach. A theta-join is a join that returns all tuples from the carthesian product of two relations that satisfy a given join condition. The join condition can be an arbitrarily complex function - i.e. equijoins, inequality joins etc. are all subtypes of theta-join. The paper [1] is concerned with traditional, non-online MapReduce systems, so we will have to adapt it in order to use it in the online context. You will implement your code on top of Squall/Storm.

We model a join between two data sets S and T with a *join-matrix*, in which each cell represents a pair of tuples which might produce a result tuple, depending whether the join condition for that pair is satisfied or not. A join can be defined using a join condition and a join result function. The join can be an equijoin, inequality, band or cross-product join. Alternatively, the join condition can also be specified for each cell of the join matrix separately. The join result function takes two tuples as input and produces (if the join condition of the cell corresponding to the two tuples is satisfied) a result tuple. In your implementation, both the join condition and the join result function needs to be easily configurable and properly decoupled in the code.

The content of this document is structured as follows. Subsection 1.1 discusses the Theta-Join paper in more details (you will still have to read the original paper). Section 2 gives a detailed explanation of your task, and gives some hints about the places in the code you will have to modify in order to implement the ThetaJoinOperator. Finally, Section 3 presents optional requirements (coupling a SQL parser with the ThetaJoinOperator).

1.1 Summary of the paper "Processing Theta-Joins using MapReduce"

In the following subsection we summarize the main ideas of the paper "Processing Theta-Joins using MapReduce". We will use some terms of the paper without defining them here, so you will have to read the paper first.

Traditionally, the basic idea for computing equijoins in MapReduce is that the tuples from both relations with the same join key are sent to the same Reducer, and then each Reducer applies a variant of merge-join. As explained in the paper, there are two downsides of this approach. First, the number of different join condition values limits scalability. For example, in the following query:

```
SELECT R.A * S.C FROM R, S WHERE R.B=S.B
```

with the following schema:

```
R(A,B), S(B,C)
```

assume that R and S have only 3 different values for `B`. That means that we cannot use more than 3 Reducers, no matter how huge `R` and `S` are. Second, the data might be skewed. This means some values occur much more often in the same attribute than others. Consider for example the case where the value "a" occurs 1000 times under the join attribute `B` and the value "b" only 17 times. Then the completion time is determined by the Reducer with the most work, in this case the reducer that is assigned the value "a".

Cross-product joins. The authors presented a join algorithm which has a different policy for assigning work to Reducers. Namely, each cell (which produces a join result) is assigned to exactly one Reducer. A cell is defined by a row (representing a set of tuples from one relation) and a column (representing a set of tuples from the other relation). A Mapper is responsible for sending a tuple to each Reducer that has an intersection with the row (column) that the tuple belongs to. A Reducer receives tuples from both relations. It has to examine all the pairs of tuples, such that the first tuple belongs to the first relation, and the second tuple belongs to the second relation. For each pair, if the join condition is satisfied, the Reducer produces a join result tuple. A cell for which the join condition is satisfied is denoted as a *candidate cell*.

The paper's main observation is that near-optimal solutions for cross-product joins can be achieved by carefully grouping cells into regions, and assigning each region to a Reducer. Optimality in this context refers to minimizing both the maximum number of tuples a Reducer receives as input and the maximum number of tuples a Reducer sends as output. The former is denoted as `Max-Reducer-Input`, and the latter as `Max-Reducer-Output`. If these regions are squares, then we can achieve optimal solutions for both `Max-Reducer-Input` and `Max-Reducer-Output`. Intuitively, a square is a rectangle which covers the largest space with the smallest sum of two sides - in our case, each side is a part of a relation that the region is responsible for. This makes the solution `Max-Reducer-Input`-optimal. Since the squares are of equal size and we have cross-product as the join type, the work each Reducer has to do is also equal. This makes the solution `Max-Reducer-Output`-optimal. See Lemmas 1 and 2 from the paper for rigorous proofs of these claims.

The reason why it is not a good idea to use multiple small regions per Reducer is presented in Figure 1. We want to avoid sending a tuple (in this case with `S.A=7`) to many Reducers (in this case to three Reducers), because it will unnecessarily increase network traffic. In addition, in the same figure, a *duplicate output* situation might occur. That is, the same pair of tuples is sent

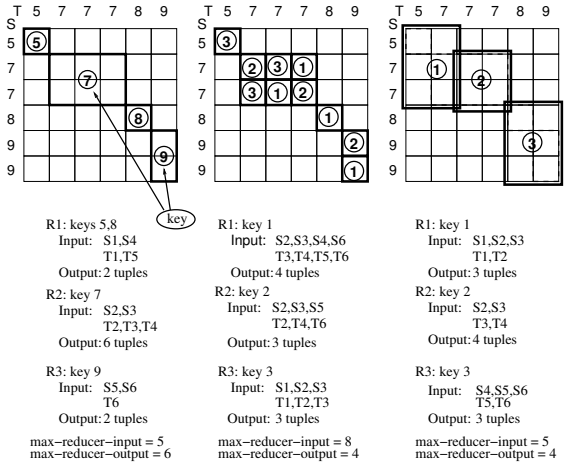

Figure 1: Matrix-to-reducer mappings for standard equi-join algorithm (left), random (center), and balanced (right) approach. (Figure 4 from the paper.)

to two reducers (in this case `S2` and `T2`) but they will provide the join result for different cells. To mitigate this problem, Mappers need to pass information to a Reducer about the join matrix tuples the Reducer is responsible for.

In order to implement this algorithm (the authors call it 1-Bucket-Theta), we need to know the cardinality of all the relations we want to join. The ideal region will be a square with area $\frac{|S||T|}{r}$, where `|S|` and `|T|` are the relation sizes and `r` is the number of Reducers. Thus, the optimal side length is $a = \sqrt{\frac{|S||T|}{r}}$, and this number is often not an integer. This side length guarantees both the optimal `Max-Reducer-Input` and `Max-Reducer-Output`.

The necessity for randomization. In a parallel system, it is hard to assign each tuple a unique serial number. The authors opted not to use a consensus algorithm. Instead, they resorted to a simple trick. Each tuple in a relation `S` is assigned a random number between `1` and `|S|` where `|S|` is the number of tuples in `S`. Similarly, each tuple in `T` is assigned a random number between `1` and `|T|`. The intersection of these two random numbers uniquely identifies the cell responsible for producing a result tuple out of these two tuples. As a side effect, we get one more benefit: load-balance of Reducer outputs in the presence of an arbitrary theta-join. For example, some regions might produce no join tuples, whereas others might produce many. By shuffling rows/columns of the join matrix, a skew is avoided. Thus, it is essential to have a good randomization function, which scatters tuples uniformly.

Sparse join matrices: discarding whole regions. Now, we will distinguish query types by the percent of candidate cells within a join matrix. We denote this percent as *selectivity*. A query is *input-dominated* if the sum of tuples from the two joined relations is bigger than the number of tuples in the result of the join. In the input-dominated example from the paper, the percentage was 0.0000003%. A query is *output-dominated* if the total number of tuples from the two joined relations is at least an order of magnitude smaller than the number

of tuples in the result of the join. In the output-dominated example from the paper, the percentage was 0.088%.

Thanks to randomization, it is practically impossible to beat the `Max-Reducer-Output` optimality of 1-Bucket-Theta algorithm. However, Mappers might send tuples to Reducers which produce no join result tuples. The authors showed in Section 4.2 that the cost cannot be degraded more than a $\frac{2}{\sqrt{x}}$, where `x` is the selectivity of the query. For example, for `x = 50%`, the cost is at most 3 times bigger than the optimal one. This is the worst-case which we might encounter when the join matrix is a row (or a column) of regions. If we have a grid of regions, with at least 2 regions per column and row, we have a guarantee that the cost is at most 2.12 times bigger than the optimal one. (For more information, please consult Section 4.2 of the paper).

If a region contains no candidate cells, it can safely be discarded, without being assigned to any Reducer. For example, in an inequality join in which about 50% of cells do not contribute to the final result, many regions can be discarded. The phenomenon that a tuple is sent to many more Reducer than necessary is denoted as *tuple duplication*. However, in an online system we have no guarantees over the order in which tuples arrive, and we do not want to block until tuples are sorted. Thus, it makes it practically impossible to find relatively large areas with no candidate cells. The randomization further complicates things. In order to discard some regions, randomization must be turned off and an additional MapReduce phase for collecting statistics has to be employed, which is not feasible in an online systems. So we refrain from using M-Bucket-I and M-Bucket-O, and use only 1-Bucket-Theta algorithm.

To conclude: assigning regions with no candidate cells to Reducers certainly slows down input-dominant queries. To mitigate this problem, an additional MapReduce phase for collecting statistics is required, which is not feasible in an online system. Nevertheless, in most cases we are interested in Output-dominated queries.

Insights about equijoins. Although the authors of the paper claim their algorithm can be used to avoid skew in equijoins, we will see that this is not completely true. When performing an equijoin using the cross-product technique, as shown in Figure 5 from the paper, `Max-Reducer-Input` is 8 and `Max-Reducer-Output` is 4, which is exactly the same as in the random matrix-to-reducer mappings from Figure 1. This emphasizes once more that tuple duplication is the main problem of the theta-join algorithm. However, the theta-join algorithm attains a better `Max-Reducer-Output` value.

To conclude: the traditional equijoin implementation achieves better performance than the corresponding implementation using the theta-join algorithm when:

1. the query is input-dominated, and
2. data is not strongly skewed.

2 Your Task

It is your task to implement a theta-join algorithm (only 1-Bucket-Theta) on top of Squall/Storm. You should adopt the ideas from the paper discussed

above. You are advised to use the general architecture that is layer out in Subsection 2.1. You need to test your implementation using the experiments that are described in Subsection 2.2. Finally you are asked to produce some deliverables that are described in Subsection 2.3.

An important part of your task is proper construction of the regions (mapping from a join matrix to Reducers). As explained in the paper, each cell should belong to exactly one Reducer. As we already saw, the authors presented a formula for obtaining the optimal square side length `a`. However, you cannot always map a join matrix in a grid of squares with the optimal side length (`a` might not be an integer). To preserve load-balancing at the output, you have to to keep region areas nearly the same as much as possible. On the other hand, `Max-Reducer-Input` should be kept as close to optimal value (`2*a`) as possible.

The algorithm for region construction should work as follows:

1. each region is a square with $a = \sqrt{\frac{|S||T|}{r}}$,
2. if this is not possible, each region is a square with the side close to optimal,
3. if this is not possible, each region is a square-like rectangle with the sides close to optimal,

Close to optimal means that no region side is two times bigger than the optimal side length. Sometimes it might not be possible that all the region are of the same area. In a grid of regions, it is allowed that the regions close to the borders of the join matrix slightly differ in area from the other regions. A more formal explanation of how to build these regions can be found in Theorem 2 and 3 in the paper.

For example, assume `|S| = 10.000`, `|T| = 30.000` and `r = 5`. As $\frac{|T|}{r} < |S|$, Theorem 3 applies, and the optimal square side would be approximately 7746. However, the closest to optimal are regions such that the join matrix is partitioned in a single row of 5 rectangles, each of `|S|*|T| = 10.000 * 6.000`. In the `|S|` dimension, the side is `1.29 * optimal_side_legth`. In the `|T|` dimension the side is `0.77 * optimal_side_length`. For a given `a`, theoretically `Max-Reducer-Input = 7746 + 7746 = 15.492` would be the best. However, in this particular example, we cannot achieve this. Actually, our solution has `Max-Reducer-Input = 16.000`, so it is very close to the theoretically optimal solution. Indeed, it is the best one for this example, being optimal for `Max-Output-Reducer`, and only 3.2% worse than the theoretically optimal `Max-Input-Reducer`. You can try different regions to see other solutions. Thus, by having regions of equal size, we achieve the optimal
`Max-Reducer-Output`, since each Reducer has exactly the same number of join result tuples to produce. In terms of `Max-Reducer-Input`, we are as close to the optimal solution as the regions are more square-like (side lengths differ slightly).

Interestingly, the paper didn't anticipate side lengths less than optimal in Theorem 3 in the paper. In Theorem 2 from the paper, the authors showed that single row blocking is the optimum, but for different conditions.

2.1 Architecture of theta-join inside Squall/Storm

The previous subsection promoted using theta-joins. However, the take-away point was the algorithm for efficiently constructing regions from a join matrix.

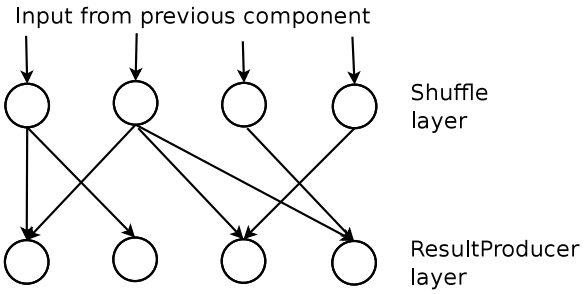

Figure 2: ThetaJoinComponent layers: the arrow(s) between a Shuffle node and a ResultProducer node(s) represent a possible shuffling for a single tuple.

The global architecture of a Theta-join component of Squall/Storm is presented in Figure 2. The input stream from a previous component is uniformly partitioned across a layer of nodes that we call the *Shuffle layer*, independently of a tuple's join condition key. The Shuffle layer is responsible for sending a tuple to a ResultProducer node(s). The Shuffle and ResultProducer layers work analogously to Map and Reduce phases. The `r` constant from the paper refers to the number of nodes in ResultProducer level here. Keep in mind that you are responsible for specifying a connection not only between the Shuffle and ResultProducer layers, but also between the previous query component and the Shuffle layer, as well as between the ResultProducer layer and the next query component. For more information about interconnecting components, please refer to the `Stream groupings` section of `https://github.com/nathanmarz/storm/wiki/Concepts`. You might also need `CustomStreamGrouping`, so please consult `http://nathanmarz.github.com/storm/doc-0.7.0/index.html`. You will need to make necessary changes in the `components` and `stormComponents` package. The former defines a user interface for a component (this is where you define the type and parameters of a query). The latter specifies a bridge between Squall and Storm. Namely, it translates user-defined join characteristics into Storm-understandable settings.

Inside each ResultProducer node, an arbitrary join strategy can be used. You can adopt the strategy from JoinComponent. When a new tuple arrives, you can perform a join with the full content of the opposite relation. You can apply an optimization which allows you to not scan over the whole opposite relation. However, keep in mind that this optimization is not generic, rather it depends on the query. A user of your class is not supposed to change the code inside, but only to instantiate the class with the appropriate parameters.

Scalability. The authors of the paper argued that higher `r` we use, the less skew we might have (Section 4.1.1). Our cluster has 176 nodes, out of which 17 are Ackers, and some of them are also dedicated to other operators in the query plan. Keep in mind that the network traffic might cause your operator to not scale after some `r`. You have to report this threshold value and show it on graphs. In addition, our online processing system is a main-memory system, so keep in mind that the maximum heap size for Squall Java processes is constrained (on cluster it is around 1GB). When the memory used is close to this amount, the system slows down drastically, and when it reaches the bound it does not report

an `OutOfMemoryException`, but rather the topology fails some tuples, so that you have to kill your topology. Memory aware solutions are not feasible in the online context, since some tuples need to be batched.

A merge-layer optimization. You might also merge the Shuffle layer with the previous component. This optimization brings you several advantages:

1. The nodes previously used by Shuffle layer can be used in some other layers.
2. The number of layers is reduced by one, which slightly decreases average tuple latency. Tuple latency is defined as a time interval between the arrival of a tuple in system, and its departure from the last component of the query plan in the form of a final result tuple.
3. Consequently, the total execution time of a query is slightly decreased.

However, keep in mind that in that case you have to modify all the other existing components in order to connect them directly to/from ResultProducer layer. Also, in the future, we might add more of them, so if you decide to implement this optimization, make sure it is robust enough.

2.2 Experiments

You will experiment with TPCH queries and DBGen-erated databases. You can do all the experiments by slightly modifying the `HyracksPlan`. By specifying different join condition, you can make the query Input- or Output-dominant. You will also have to use skew data, and now we will explain how you can generate it.

2.2.1 Generating skew data

In order to perform experiments with skewed data, we provide you a modified TPCH databases where the columns have non-uniform (skewed) data distributions. These datasets are generated using a modified version of DBGen program which can be found in `INSTALL_DIR/auxiliary/tpch_skew`. The program is capable of generating data from a Zipfian distribution, where the Zipf value (z), which controls the degree of skew in the data, is a parameter that can be specified to the program. For example, the parameter z=0 generates a uniform distribution for each column in the database, whereas z=4 generates a highly-skewed distribution (i.e., a few values occur very frequently) for each column.

Compilation. The modified DBGen is pre-compiled for Mac and Solaris. If you want to compile it for Linux, you have to run the following:

```
cd $INSTALL_DIR/auxiliary/tpch_skew
cp makefile_linux makefile_linux
make clean
make
```

If you want, you can also recompile it for other operating systems, by using `makefile_MacSolaris`:

```
cd $INSTALL_DIR/auxiliary/tpch_skew
cp makefile_MacSolaris makefile
make clean
make
```

Running. In Local Mode, You will need to create a skewed database and refer to it properly from your config files. In Cluster Mode, you can use a set of predefined databases available at `/export/home/avitorovic/queries/skew_tpch`. We generated datasets with different skew values (1,2,3,4) for scaling factors 1 and 10.

The following commands generates a TPC-H database with the scaling factor 1 where data in each column is created from a Zipfian distribution where the degree of skew is 2:

```
cd $INSTALL_DIR/auxiliary/tpch_skew
./dbgen -s 1 -z 2
```

2.3 Deliverables

From your experiments, you need to produce some graphs that present the following:

1. The performance difference between the traditional equijoin implementation (available in the `JoinComponent`) and the theta-join implementation of equijoin, both with and without skew
2. The performance difference between Input- and Output-dominant inequality joins.
3. The final result for each query you were running - to make sure the results are correct, you can compare with a traditional database, such as MySQL.

You should submit:

1. Detailed documentation of the code with the required graphs. If you had to change some of the existing classes, you have to explain exactly what you changed and why it was necessary.
2. Source code. Separate changed from new classes.

These are set of mandatory SQL queries you need to experiment with (you can write more if you want):

```
1. Hyracks (you already have query plan for this query)
2. An example of Input-dominated query
   (around 134000 tuples our of 600.000x150.000 matrix):
   SELECT SUM(LINEITEM.EXTENDEDPRICE*(1-LINEITEM.DISCOUNT))
   FROM LINEITEM, ORDERS
   WHERE LINEITEM.ORDERKEY = ORDERS.ORDERKEY AND
         ORDERS.TOTALPRICE > 10*LINEITEM.EXTENDEDPRICE
3. An example of Output-dominated query is a cross product:
   SELECT SUM(SUPPLIER.SUPPKEY)
   FROM SUPPLIER, NATION
```

4. An example of a query containing multiple join operators:

```
SELECT SUM(LINEITEM.EXTENDEDPRICE*(1-LINEITEM.DISCOUNT))
FROM LINEITEM, ORDERS, NATION, SUPPLIER, PARTSUPP
WHERE LINEITEM.ORDERKEY = ORDERS.ORDERKEY AND
      ORDERS.TOTALPRICE > 10*LINEITEM.EXTENDEDPRICE AND
      SUPPLIER.SUPPKEY = PARTSUPP.SUPPKEY AND
      PARTSUPP.PARTKEY = LINEITEM.PARTKEY AND
      PARTSUPP.SUPPKEY = LINEITEM.SUPPKEY AND
      PARTSUPP.AVAILQTY > 9990
```

In `INSTALL_DIR/dip/Squall/src/queryPlans` you can find examples of query plans. Beside Hyracks, we implemented TPCH3, 4, 5, 7 and 8.

3 An optional task: coupling a SQL parser with the ThetaJoinOperator

Optionally, you can try to couple the ThetaJoinOperator and our SQLplugin. SQLPlugin is a SQL parser which translates SQL directly to Squall query plans, and then executes them. You will have to modify SQLPlugin such that it properly recognizes theta-joins in SQL and instantiates your ThetaJoinOperator with valid arguments. You can find more information about SQLPlugin in SquallQuickStart document.

References

[1] A. Okcan and M. Riedewald, "Processing theta-joins using mapreduce," in *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, (New York, NY, USA), pp. 949–960, ACM, 2011.