E6889 Large-Scale Stream Processing Homework 2 Report

Chenye Yang cy2540, Bingzhuo Wang bw2632, Zhuoyue Xing zx2269

1. Operator Reordering

1.1. Assumptions

1.1.1. Workload

The data used in this operator reordering optimization is exactly the same as that in homework 1, which is the epa-http data. The data has the following format:

```
141.243.1.172 [29:23:53:25] "GET /Software.html HTTP/1.0" 200 1497
query2.lycos.cs.cmu.edu [29:23:53:36] "GET /Consumer.html HTTP/1.0" 200 1325
tanuki.twics.com [29:23:53:53] "GET /News.html HTTP/1.0" 200 1014
wpbfl2-45.gate.net [29:23:54:15] "GET / HTTP/1.0" 200 4889
wpbfl2-45.gate.net [29:23:54:16] "GET /icons/circle_logo_small.gif HTTP/1.0" 200 2624
wpbfl2-45.gate.net [29:23:54:18] "GET /logos/small_gopher.gif HTTP/1.0" 200 935
140.112.68.165 [29:23:54:19] "GET /logos/us-flag.gif HTTP/1.0" 200 2788
wpbfl2-45.gate.net [29:23:54:19] "GET /logos/small_ftp.gif HTTP/1.0" 200 124
```

This epa-http data is stored in a text file with the suffix ‘txt’. We use the built-in function of spark to stream it.

```
conf = pyspark.SparkConf().setAppName('Operator_Reordering').setMaster('local[*]')
sc = pyspark.SparkContext.getOrCreate(conf=conf)  # create a spark context object
log_lines = sc.textFile('epa-http.txt')  # read file line by line to create RDDs
```

The “getOrCreate()” function in “SparkContext()” allows the program to create a new spark context object at first time and re-use it later. Without repeated creation of the spark context object, the program runs faster and avoids the spark context object conflict. The function “textFile()” reads the text file line by line to create spark RDDs as the input of Operators. This input stream has 47748 tuples in total.

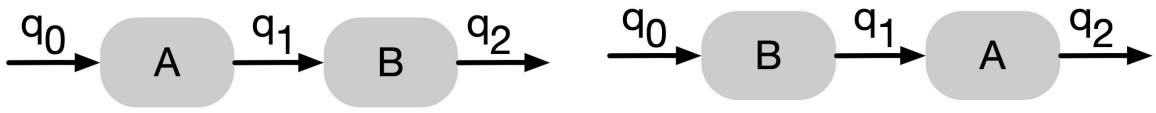
1.1.2. Measurement of performance

The performance of the system will be measured by “throughput”, which means the data volume being processed by the system per second. The throughput is usually measured in “bit/s”, while in this case it can be measured in “tuple/s”.

The reason for using throughput is that the lecture and paper both use this as the measurement of the stream processing system and have the figure of throughput versus other variables.

1.2. Experiment

In the experiment, we designed two stream processing systems as the following Figure 1.



(a) Not reordered (b) Reordered

Figure 1 Operator Reordering

The operator A and B simply conduct the “filter” function, however with a different selectivity rate. The selectivity of A is fixed at 0.5, while B has a variable selectivity rate between intervals $(0, 1)$. A selectivity rate equals 0.3 means that only 30% of the input data would go through the operator as the output, and the other 70% will be dropped. Moreover, these two operators A and B have equal cost, which means the computation resources used for processing each tuple are the same.

To fulfill the selectivity requirement and the cost requirement and then measure the operator’s performance, we have the following code:

```
input_A = log_lines.count()
# stream pass through operator A
start = process_time()
```

```
log_lines_A = log_lines.filter(lambda x: random.random() < selectRateA) # only pass through some tuple
# we add a small constant number to the cost of processing each tuple, to avoid the arithmetic overflow
for i in range(input_A):
  sleep(1e-6)
end = process_time()
time_A = end - start
cost_A = time_A / input_A # the computation cost (per tuple) of operator A
```

The lambda expression in the "filter()" function makes sure that only selectRateA% of the input tuples can pass through this operator. However all the input tuples have to be processed by it.

We assume that the computation cost of each tuple is the same. However, in practice, this is almost impossible. Therefore, we add a small constant cost (1e-6 second) to each tuple. This small constant cost has two functions: to avoid the arithmetic overflow (some processing time is extremely small) and to smooth the difference between the cost of processing each tuple (the constant is 10 times larger than the actual cost). Then after the average over each input tuple, we could get the cost of processing each tuple.

Let $c(A)$ be the compute cost of operator A (per tuple), $c(B)$ be the compute cost of operator B (per tuple), $s(A)$ be the selectivity of operator A, $s(B)$ be the selectivity of operator B. According to the lecture, the compute cost of the not reordered system in Figure 1(a) is $c(A) + s(A)c(B)$, the compute cost of the reordered system in Figure 1(b) is $c(B) + s(B)c(A)$. Thus we have the following code:

```
# the computation cost (per tuple) without reordering
cost_A_B = cost_A + selectRateA * cost_B
# the computation cost (per tuple) with reordering
cost_B_A = cost_B + selectRateB * cost_A
```

The throughput is calculated by 1 dividing the cost:

```
# throughput: how many tuples processed per second
throughput_A_B = list(map(lambda x: 1 / x, throughput_A_B))
throughput_B_A = list(map(lambda x: 1 / x, throughput_B_A))
```

Also, to get a smooth throughput curve, we run the systems several times and average them:

```
# average the throughput to smooth the curve
throughput_A_B_mean = np.mean(np.array(throughput_A_Bs), axis=0).tolist()
throughput_B_A_mean = np.mean(np.array(throughput_B_As), axis=0).tolist()
```

Then we use the maximum throughput of the not reordered system to do normalization:

```
# normalization
maxInAB = max(throughput_A_B_mean)
minInAB = 0
throughput_A_B_mean = list(map(lambda x: (x - minInAB) / (maxInAB - minInAB),
throughput_A_B_mean))
throughput_B_A_mean = list(map(lambda x: (x - minInAB) / (maxInAB - minInAB),
throughput_B_A_mean))
```

After calculation, the throughput is visualized.

1.3. Results

The result in the paper which we are trying to reproduce is Figure 2.

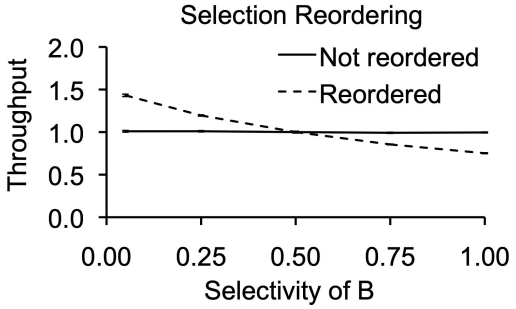


Figure 2 Result in paper

The result we reproduced is Figure 3. It is similar to the original result.

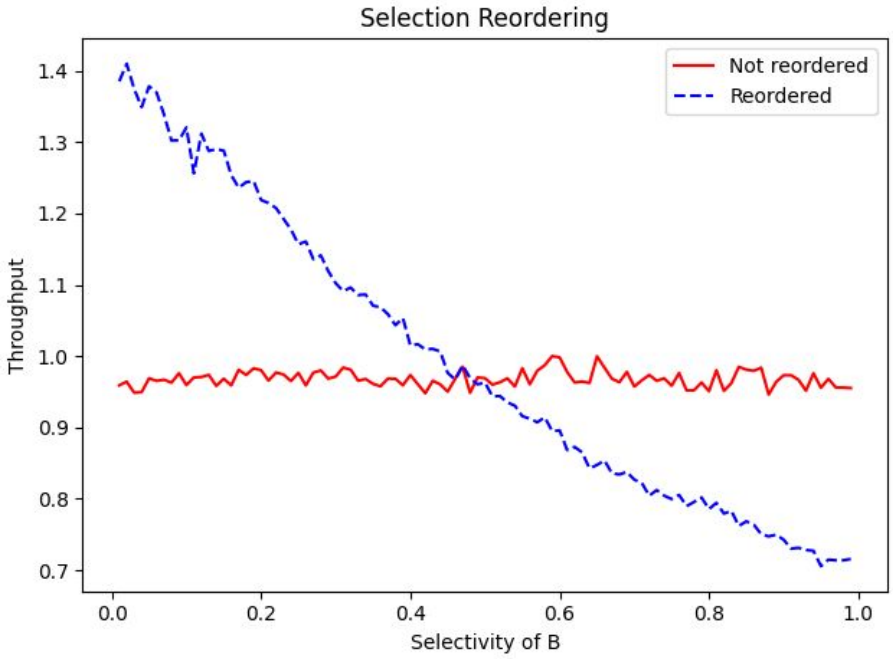


Figure 3 Reproduced result

The following Figure 4 and Figure 5 reveal the importance of small constant cost and several runs of the system. Without these settings, the result is way more different from the theory.

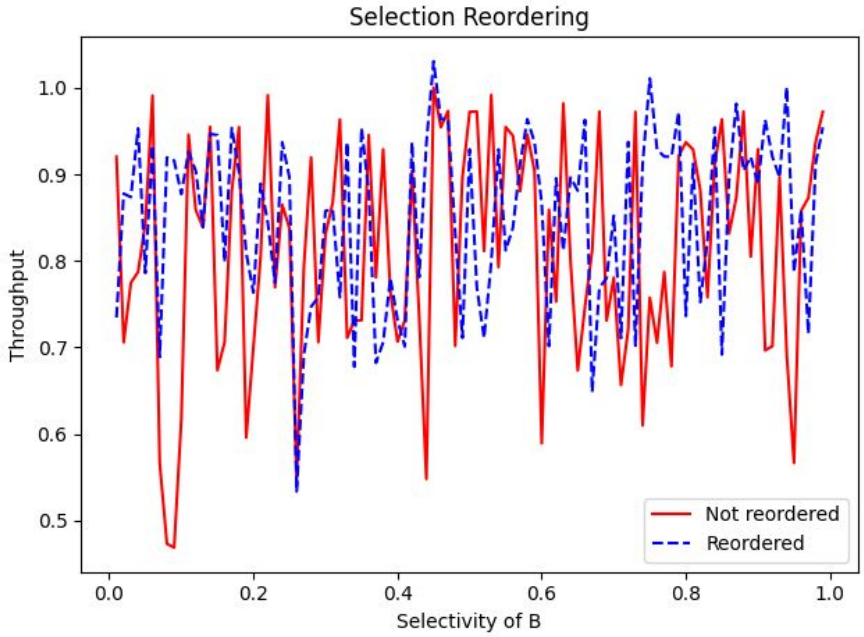


Figure 4 Without small constant cost

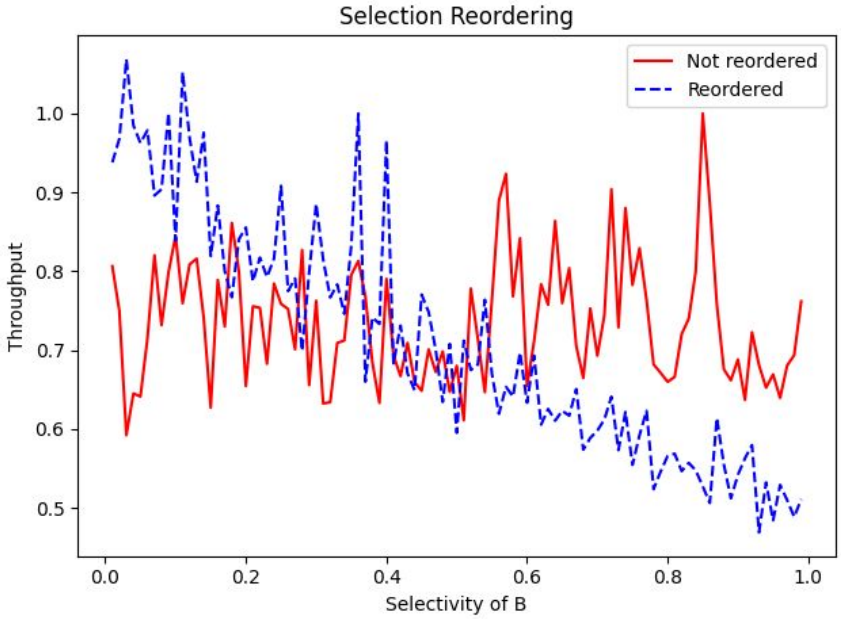


Figure 5 Run the system only once

2. Load Shedding

2.1. Assumptions

2.1.1. Workload

The data used in load shedding optimization is the same as operator reordering, which is still the epa-http data. In this particular task, we want to distribute every instance into different 'buckets' so that we can make comparisons between the original and sampling ones. Therefore, we are extracting the 'Bytes' term in each record and divide 'buckets' by setting the interval of 10000 bytes.
To stream it, we follow the same step as Section 1 does.

2.1.2. Measurement of performance

The performance of the system will be measured by both throughput and accuracy. Throughput is the metric which measures the data volume processed per second, while accuracy is the metric measuring the similarity between the sampling data and original ones. Specifically, we define accuracy as (1-error), where error is the Wasserstein distance between the distribution of original data and reduced ones, and define throughput as processed time per record. To better illustrate the

relationship between both metrics, we use normalized throughput which scales from 0 to 1. For this task, we are controlling selectivity from 0.01 to 0.99, which means we are randomly sampling from the whole data from 1% to 99%.

2.2. Experiment

To begin with, we clean the data by removing the instances where bytes terms are missing. Then, we categorize data to different buckets with intervals of 10000 bytes.

```
cleaned_lines = log_lines.filter(lambda x: (x[-1] != "-"))
bytes_per_line = cleaned_lines.map(lambda x: (int(x.split(" ")[-1])//10000, 1))
```

To compute the accuracy, we need to compare the original data distribution among all buckets and each reduced distribution among all buckets with a certain portion of selectivity. To compute the throughput, we need to calculate each processed time for any selectivity. The backbone of the logistic can be described below while more detailed implementation can be found in Python codes.

```
for select in selectivity:
        # compute delay time
        bytes_per_line_sampled = sampling(bytes_per_line, select)
        start = process_time()
        reduced_list = aggregation(bytes_per_line_sampled)
        for i in range(bytes_per_line_sampled.count()):
                sleep(1e-6)
        end = process_time()
        t_time = end - start
        throughput_time.append(t_time/bytes_per_line_sampled.count())
        # compute accuracy
        base_list = aggregation(bytes_per_line)
        error.append(distance(base_list, reduced_list))
```

With error (distribution distance) known, we can easily calculate accuracy by using the formula “accuracy = 1 - error”. With processed time per record known, we can normalize the data into the interval from 0 to 1. At the end of the day, we draw the accuracy and throughput plots of load shedding.

2.3. Results

The result we produced is shown in Figure 6. It is clear that as the selectivity grows, the accuracy is gradually closer 1 while the throughput drops towards 0, which is

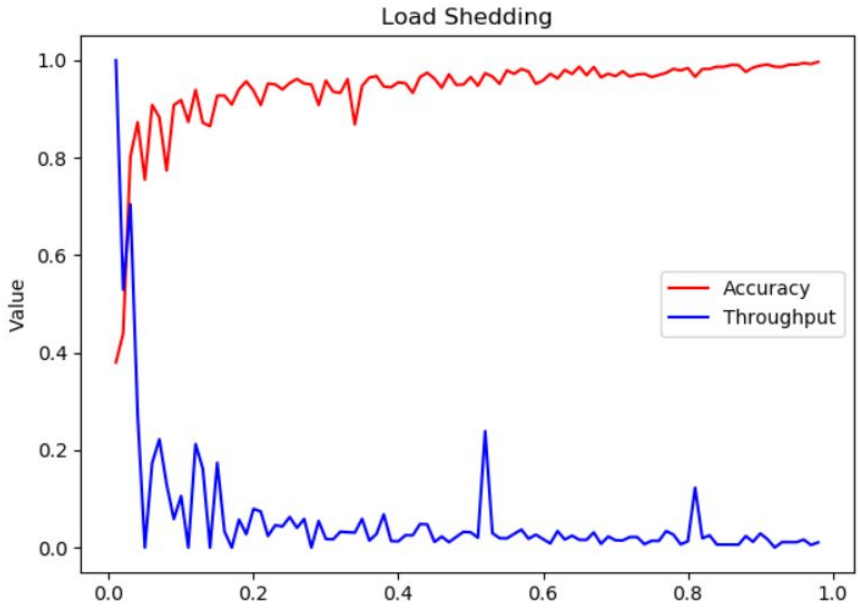


Figure 6 Reproduced result

3. Fusion

3.1. Assumptions

3.1.1. Workload

We choose fusion as bonus optimization and reproduce its graph.

The data used in this operator reordering optimization is exactly the same as that in homework 1, which is the epa-http.txt data.

This epa-http data is stored in a text file with the suffix ‘txt’. We use the built-in function of spark to stream it. The function “textFile()” reads the text file line by line to create spark RDDs as the input of Operators. This input stream has 47748 tuples in total.

3.1.2. Measurement of performance

The measurement of performance we use for Fusion optimization is throughout over the ratio that is operator cost / communication cost. Throughput is the metric which measures the data volume processed per second, while accuracy is the metric measuring the similarity between the sampling data and original ones. Operator cost / communication cost shows the communication cost by operator cost with which we can calculate communication cost from operator cost.

3.2. Experiment

In the experiment, we designed two stream processing systems as the following Figure 7.

Figure 7 Fusion

During the experiment, we change the operator cost / communication cost ratio and calculate the throughout. At the same time we fix the select rate of operator A and the select rate of operator B. We set selectRateA = 0.5 and selectRateB = 0.5.

In this Fusion optimization, we need to consider communication cost while calculating the cost. With the cost we calculated, we can calculate the throughout

Let $c(A)$ be the compute cost of operator A (per tuple), $c(B)$ be the compute cost of operator B (per tuple), $s(A)$ be the selectivity of operator A. According to the lecture, the compute cost of the not reordered system in Figure 7(a) is $c(A) + s(A)c(B) + communication\ cost$, the compute cost of the reordered system in Figure 7(b) is $c(A) + c(B) + communication\ cost$. Thus we have the following code:

```
# Not Fused
cost_A_B = cost_A + selectRateA * cost_B*(1+1/commu_cost)
# Fused
cost_AB = cost_A +  cost_B
```

The other part of Fusion is similar with Operator Reordering.

3.3. Results

The result we reproduced is Figure 8. We can see that as shown in the paper, when the number operator cost / communication cost is small which means communication cost is dominant, the throughput with Fusion optimization is larger, which shows the result is better.

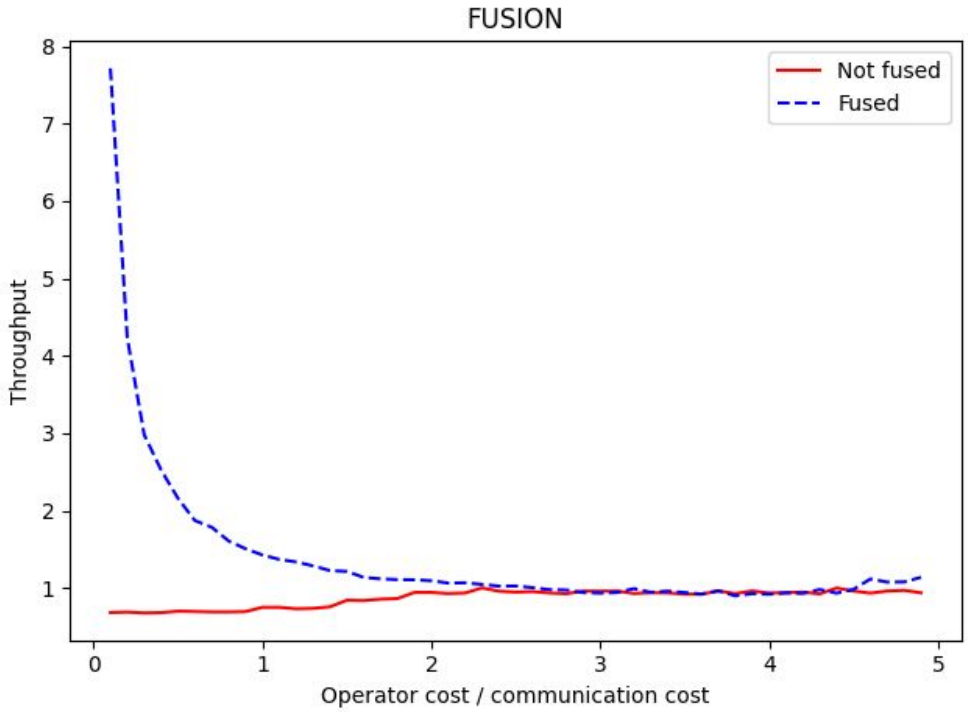


Figure 8 Reproduced result of Fusion