Milestone 1

Group 6

1 Dataset description

1.1 wikipedia page visit frequency

The wiki dataset are obtained from the Massviews Analysis [1], an offical tool offered by wikipedia. It can import related pages and compare their pageviews. We use the keyword Europe and get more than 1000 related topics. The results can be exported as csv file and we use 1001 topics in our dataset. Some facts about our dataset are list in Table 1:

Table 1: Wikipedia page views dataset

size	32 MB
number of time series	1001
length of each time series	1305
total rows	1326326

1.2 Stocks

The Stocks dataset with a size of 3 GB contains time series of daily stock prices and volumes for 29821 distinct stocks for the period of 1 Jan 2016 – 31 Dec 2020. Each time series contains more than 1000 updates but it's not time-aligned due to missing values. In the following section, we will clean the data such that each time series has exactly 1000 readings and the values across the time series are time-aligned.

2 Data preprocessing step

During data preprocessiong, we read the data and clean them with the following operations:

1.load the dataset

2.remove duplication

3.filter out the date of weekends and the date out of range(2016/01/01-2020/12/31).

4.time-align the dataset, and search for the lost dates. Supplement the lost data by copying the data from the previous weekday.If the missing datas are continuous, then the stock will be dropped.

5.finally drop stocks with null value and complete the data cleaning step.

3 result of Part 2

In part 2, we use spark sql to compute the cosine similarity of each tuple.

3.1 τ value execution time

Because of the limited availability of cluster, we did not have sufficient opportunities to optimize the code and get meaningful results. Therefore, the results represented in this report are obtained locally. We tested 250 topics combined with 100 stocks. The τ value and execution time for part 2 are shown as follows:

Table 2: Part 2 result

metrics	τ value	#result	execution time(secs.)*
Avg	0.994122379	20	331.4
Min	0.995819854	20	336.3
Max	0.991408893	20	371.8

*Execution time run on PC(8 core 16g) with 100 stocks and 250 * 250 wikipedia topics.

3.2 SQL line

The steps are similar for avg, min and max. We only include the avg as example as followed:

- Line 9- 12: Self join the wiki table and filter the result as $\langle a, b\rangle$ and $\langle b, a\rangle$ are considered the same;
- Line 7-8: Add one column named average by calculating the average of y1 and y2;
- Line 3-6: Join the stock with what we get from step 1, and also remove duplication;
- Line 2- 3: Caculate the cosine similarity by calling the UDF simcos
- Line 1 and 13: Output Top 20 combinations based on average cosine similarity.

```
1 select stock, topic1, topic2, avg_cosine
2     from
3     (select stock, topic1, topic2, simcos
          (price, average) as avg_cosine
4         from
5         (select stock.stock, collect_list
              (price) as price from stock
              group by stock)
6         cross join
7         (select *, (transform(arrays_zip(
              visit1, visit2), x -> (x.
              visit1 + x.visit2) / 2)) as
              average
8             from
9             (select wiki1.topic as topic1
                  , wiki2.topic as topic2,
                  wiki1.visit as visit1,
                  wiki2.visit as visit2
10                from wiki wiki1
```

[1] https://pageviews.wmcloud.org/massviews/

```
11                cross join wiki wiki2
12                where wiki1.topic < wiki2
                      .topic)))
13    order by avg_cosine desc limit 20;
```

3.3 optimization trick

We used spark's dataframe by calling the *pyspark.sql.DataFrame* package, step by step query *wiki* and *stock* data sets and state the table we need, through *join*(), *groupBy*() and relevant library functions to get the results we want. After that, we rewritten these codes and modified them into the form of SQL query and finally integrate all steps into one SQL query. In this way, Spark SQL's query optimization engine can automatically choose an optimized physical plan according to the integrated SQL query. From the comparison of the execution time before and after, the execution efficiency has been greatly improved. In the optimization of part 2, we are satisfied with the following optimization methods:

3.3.1 Compressed the dataset

We decided to condense the dataset and integrate the visits according to the topic and the price as per stock since both datasets comprise 1305 time serials. As a result, we receive two compressed datasets and save a lot of storage space.

3.3.2 UDF(simcos) to compute similarity

As price and visit are both lists for stock and topic and they are time-aligned. Therefore, we can compute similarity by multiplying vectors, which should minimize computing time and storage space. We created a UDF that computed the similarity between two vectors. We put the function to the test using the example provided in the instructions.

3.3.3 Duplication removal during join

As $\langle X, agg\,\langle Y1, Y2\rangle\rangle$ and$\langle X, agg\,\langle Y2, Y1\rangle\rangle$ outputed the same result, it is necessary to reduce the duplicated combos after cross join. We use filter to archive this goal, which reduce the amount of combinations by more than half.

3.4 results and justifiability

Table 3: Top 5 Cosine Similarity Agg by min

similarity	stock	wiki page 1	wiki page 2
0.9965813	26933	Belgium	Romania
0.9965451	26933	Czech Republic	Romania
0.9962820	26933	Romania	Kazakhstan
0.9962632	26933	Caribbean	Oceania
0.9962296	26933	Saint Petersburg	French language

Analysis: the stock 26933: "$Futures - -Indices - -ETF\_ETF\_iShares - eb.rexx - R - Gov. - Germany -$ " dominates the top lists. By simply google it, it is the fund to track the performance of an index composed of German government bonds", which means it is a German related company. We can see top 5 wiki topics combos includes: Belgium, Romania, Czech Republic, Saint Petersburg, French and etc. Those are mostly European countries or cities and are closely related with German.

4 result of Part 3

4.1 comparison

The results of part 3 are the same as the results of part 2.

4.2 execution time

Same as the part 2, following are the result we obtained from the local test. We measured time usage of three similarity in a sequential order, therefore the preprocessing time is included in the execution time of Avg.

Table 4: Part 3 result

metrics	τ value	#result	execution time(secs.)
Avg	0.994122379	20	1217.8
Min	0.995819854	20	1.5
Max	0.991408893	20	1.4

4.3 system architecture

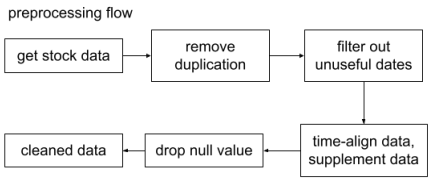

Figure 1: data processing flow

The system architecture consists of two parts. The first part(Figure 1) refers to data preprocessing. The second part(Figure 2) illustrates how data is combined and calculated step by step. In terms of implementation, all the operations are executed by using RDD.

4.4 optimization trick

The optimization of RDD execution mainly involves four aspects:

4.4.1 Broadcast join and partition

These approaches aim at reducing shuffling. Comparing to *join()*, which involve a shuffle of all partitions, the broadcast join enables the smaller table to be broadcasted to all workers and stored in the main memory. The join operation can be done

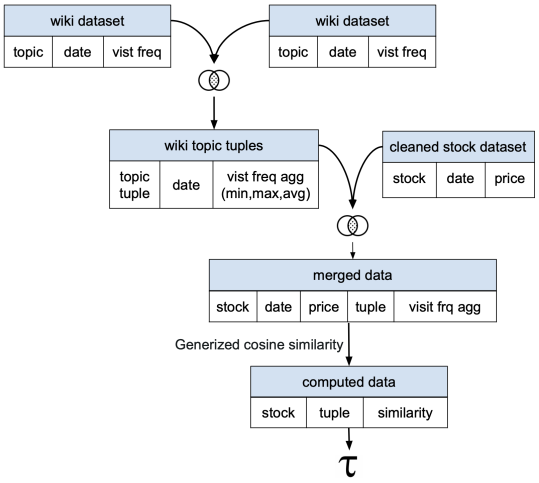

Figure 2: system architecture

without shuffling other partitions when the dataset is sufficient small. As for *reduceByKey()*, we use *partitionBy()* to partition the RDDs on the key in advance. Here, key is a combination of two topics with a stock. Each key links to 1305 records. We tune the number of partitions such that the records with the same key are stored in the same partition. As a result, *reduceByKey()* only involves aggregation of data within the same partition rather than shuffling to gathering data with the same key. As our wiki dataset and the final table are small, we benefits a lot from this optimization: The time to process 70 stocks with 100 wiki topics on a local machine reduces from 13 minutes to 3 minutes.

4.4.2 Cache and persistence

As the final table which is used to calculate the three similarity values is relatively small and repeatedly used, we apply *persist()* to its RDD such that it would not be generated multiple times.

4.4.3 Pushdown projection and selection

To join two tables, the larger the tables are, the higher the overhead is. Therefore, we always use *map()* to resize or adjust the RDDs before apply *broadcast()* or *reduceByKey()* on them.

4.4.4 Encoding

The long stock name increases the size of intermediate results, especially when we combine stock name with topic names in one column. Therefore, we encode these attribute values in integer numbers to reduce space usage.

4.5 discussion

We know that the query optimization engine in Spark SQL converts each SQL query into a logical plan, then into many physical plans, and finally it chooses the most optimized physical plan. In part 2, we finally let the spark kernel execute an integrated SQL query. We saw a significant improvement by comparing the efficiency of executing the spark dataframe functions step by step. Through the plan tree printed by the EXPLAIN operator, we can observe the optimization of the entire processing process by Spark SQL in detail. From the plan tree, it is not difficult to find that Broadcast plays a very important role in the execution of the plan.

In comparison with using SparkSQL, data processing with RDD is a more low-level approach. A compact SQL query can be optimized by Spark engine as a whole, while RDD requires more manual optimizations to generate a better physical execution plan. As mentioned above, brocast join is a very common operation in SQL physical plan, but if we use RDD we must call it explicitly. It is obvious that not all optimizations can be covered by manual realization. Therefore, our part 3 consumes longer time than part 2. The Spark web UI also indicates that part 3 involves more stages than part 2. Moreover, there are also some limitation in the aspect of implementation. According to logs from the cluster, we find that the most time-consuming step in part 3 is always the last step, i.t. get the top 20 results with the highest similarity value. We use the library function *top()* to implement this functionality, which is however not very satisfying. Besides, it is a non-trivial work to find the optimal partition number that fit cluster best, because the information in the logs are limited, and there are many other factors resulting in a timeout in online testing.

5 contribution

Name	%	Work
Tingyuan Zhang	16.6	code for part 1, part 3; cluster test; code optimization
Mingzhe Shi	16.6	code for part 2, report
Ruichen Hu	16.6	code for part 3, report, video
Siyue Chen	16.6	code for part 2, report, video, poster
Simin Sun	16.6	code for part 2, report, poster
Jing Wu	16.6	code for part 2, report, poster