2AMD15 - Big Data Management

Discovery of Combinations of Vectors with Low Aggregate Variance

Quartile 3 - 2022-2023

Full name	Student number	Study	Participation
Thiam Wai Chua	0666146	Data Science in Engineering	16.67%
Yang Yang	1697692	Computer Science and Engineering	16.67%
Xiaohan Wang	1840894	Data Science & A.I.	16.67%
Freddie Liu	1655124	Data Science & A.I.	16.67%
Zheyuan Su	1653776	Data Science & A.I.	16.67%
Guangyu Li	1779567	Data Science & A.I.	16.67%

Eindhoven, November 14, 2024

1 Dataset Description

The dataset in this assignment consists of 250 or 1000 vectors generated by a predefined generator java program. Each vector has size of 10000.

2 Data Import and Preprocessing (Question 1)

In essence, Question 1 requires us to import the dataset both as a DataFrame and an RDD. Prior to importing the data, we establish a connection to the `SparkContext` to enable the use of Spark capabilities. Once the data is read as a DataFrame, we rename the columns and convert the vector representation from a default String type to an array of Integer type. The same procedure is followed when creating the RDD.

3 Question 2

Listing 1: Optimized SQL query

```sql
SELECT id1, id2, id3, final_var
FROM (
    SELECT first(id1) as id1, first(id2) as id2, first(id3) as id3, sorted_id,
            SUM(var) + 2/10000*SUM(dot) - 2*SUM(mean_product) AS final_var
    FROM(
        SELECT t2.id1 as id1, t2.id2 as id2, t1.id as id3, t2.mean_product as mean_product,
            t1.var as var, t2.dot as dot,
            concat_ws('', array_sort(array(t2.id1, t2.id2, t1.id))) as sorted_id
        FROM (
            SELECT data.id as id,
                aggregate(data.int_list, cast(0 as long), (acc, x) -> acc + x*x)/size(data.
                    int_list) - POW(aggregate(data.int_list, 0, (acc, x) -> acc + x)/size(
                    data.int_list), 2) as var
            FROM data ) as t1
        CROSS JOIN (
            SELECT data1.id as id1, data2.id as id2,
                aggregate(zip_with(data1.int_list, data2.int_list, (x, y) -> x * y), 0, (acc
                    , x) -> acc + x) as dot,
                aggregate(data1.int_list, 0, (acc, x) -> acc + x)/size(data1.int_list) *
                    aggregate(data2.int_list, 0, (acc, x) -> acc + x)/size(data2.int_list)
                    as mean_product
            FROM data as data1, data as data2
            WHERE data1.id < data2.id
            ) as t2
        WHERE t1.id!= t2.id1 and t1.id!=t2.id2 )
    GROUP BY sorted_id )
WHERE final_var<=tau
```

3.1 SQL Query for Question 2

The SQL query used for question 2 is presented in Listing 1, and the results of the SQL execution are summarized in Figure 1. The query shows good performance on the large dataset `250×10000`, taking only an average of `58` seconds to process on a device with a 6-core and 12-thread CPU. The final optimization successfully identified all the vectors whose variance is at most $\tau = \{20, 50, 310, 360, 410\}$.

The SQL query is written based on following computation of aggregate variance of triplet.

$$\sigma^2 = (\frac{1}{l}\sum_{i=1}^{l}(X[i])^2) - (\sum_{j=1}^{l}\frac{X[j]}{l})^2 \tag{1}$$

$$= \frac{1}{l}\sum_{i=1}^{l}(a_i + b_i + c_i)^2 - \mu^2 \tag{2}$$

$$= \mathrm{var}(a) + \mathrm{var}(b) + \mathrm{var}(c) + \frac{1}{l}\sum_{i=1}^{l}(2a_ib_i + 2b_ic_i + 2a_ic_i) - 2(\overline{a} * \overline{b} + \overline{b} * \overline{c} + \overline{a} * \overline{c}) \tag{3}$$

The query presented in Listing 1 uses an alternative computation method for variance, which is explained later. In the most inner query, a table `t1` is selected, which contains the vector IDs and their corresponding variances. Another table `t2` is also selected, which contains all possible pairs of two vectors (`a` and `b`), their dot product ($\sum_{i=1}^{l}(a_i b_i)$), and the product of their means ($\overline{a} * \overline{b}$) as shown in Equation 3. These two tables are then cross-joined to obtain a table that contains all possible combinations of triplets of vectors. The join operation is filtered by **data1.id** $<$ **data2.id and data2.id** to exclude duplicate tuples. Note that there are three rows for each triplet where the IDs are in a different order. This is necessary since the three rows contain the variance of each vector, the three different dot products, and the products of means of the three pairs of vectors in the same triplet. The IDs are then sorted so that all the outcomes of the same triplets can be aggregated to calculate the final variance of each aggregated vector triplet. Finally, in the outer `SELECT` statement, we select the triplets that have variances at most τ. The query exhibits good performance on a large dataset of `250`×`10000`. It took only an average of `58` seconds to process on a device with a 6-core and 12-thread CPU, with the final optimization to find all the vectors whose variance is at most $\tau = \{20, 50, 310, 360, 410\}$.

3.2 Performance Analysis

Regarding optimization, we partitioned the DataFrame with respect to the number of CPU cores to make the best use of the hardware, and this significantly reduced the execution time. We also cached the variance table on the `250`×`1000` dataset, which further reduced the computation time for the loop of tau from `118` seconds to `58` seconds. This demonstrates that both caching and partitioning can be beneficial optimizations.

The most crucial part of our optimization is summarized by the formulation shown in Equation 3. This equation calculates the total variance of the aggregated vectors without explicitly computing all the Cartesian products of the triplets and storing them. Instead, it only stores the Cartesian products of all the combinations of two vectors, which is considerably fewer than the number of triplets. Furthermore, it stores the variance and mean value of all vectors, the size of which is negligible compared to the triplet combinations. This optimization drastically reduces the execution time to less than a minute.

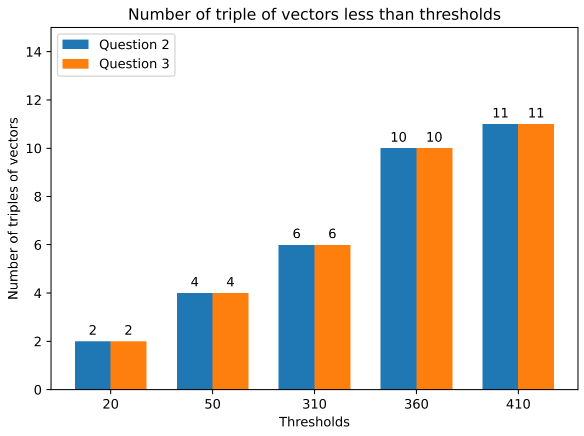

Figure 1: Number of triples of vectors whose aggregate variance at most τ in the dataset of 250×10000.

4 Question 3

4.1 System Architecture

The following steps outline a process for optimizing the calculation of aggregate variance for vectors in a distributed computing environment. The process involves obtaining unique combinations of keys, calculating variances and means for each vector, creating dictionaries to broadcast these values, calculating the aggregate variance for each unique triple of keys, and filtering the results based on a specified threshold value. This process can significantly reduce the execution time and resource requirements for calculating aggregate variance. The steps are shown below.

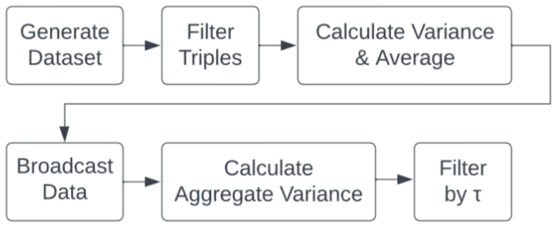

Figure 2: Process

$$rdd.keys() \xrightarrow{\text{cartesian}} keys2 \xrightarrow{\text{cartesian}} keys3$$
$$rdd \xrightarrow{\text{map}} var\_avg\_rdd \xrightarrow{\text{broadcast}} broadcast\_vectors$$
$$keys2 \xrightarrow{\text{map}} second\_term\_rdd \xrightarrow{\text{broadcast}} broadcast\_second\_term$$
$$var\_avg\_rdd \xrightarrow{\text{map}} var\_avg\_rdd2 \xrightarrow{\text{broadcast}} broadcast\_vectors$$
$$keys3 \xrightarrow{\text{map}} result \xrightarrow{\text{filter}} finalcount$$

1. Obtain all unique couples (keys2) and triples (keys3) of keys from the input RDD.
2. Calculate the variance and mean for each vector in the input RDD and create a dictionary of vectors, variances, and means to broadcast.
3. Calculate the second term required for the aggregate variance calculation and create a dictionary of the second term values ($\frac{1}{l}\sum_{i=1}^{l}(2a_ib_i + 2b_ic_i + 2a_ic_i)$ from Equation 3) to broadcast.
4. Create a new RDD with the variance and mean for each vector and broadcast a dictionary of these values.
5. Calculate the aggregate variance for each unique triple of keys using the broadcasted dictionaries.
6. Filter the results based on the τ values to keep only triples with an aggregate variance of at most $\tau = \{20, 410\}$.

4.2 Optimization

In distributed computing, optimizing data processing across worker nodes in a cluster is crucial to improve performance. In this regard, the program described in this context utilizes various optimization techniques to process large-scale data efficiently. The optimizations are discussion as follows.

- Partitioning: To enhance the performance of processing data across worker nodes in a cluster, the input RDD is partitioned into 160 partitions by setting the `NumPartition` variable. Through experimentation, we discovered that changes in the number of partitions to 240 or 320 partitions would double the execution time. We hypothesize that the running time is sensitive to the homogeneity of the size of each partition.
- Broadcasting: The primary optimization technique used in the program is broadcasting. Instead of sending the entire RDD to each node, the vectors are preprocessed to compute the required information such as variances, means, and second terms ($\frac{1}{l}\sum_{i=1}^{l}(2a_ib_i + 2b_ic_i + 2a_ic_i)$ from Equation 3), which are stored in dictionaries and broadcasted to all nodes in the cluster. This approach reduces the amount of data to be processed, thereby improving the overall performance.
- Filtering: After computing the aggregate variance for each unique triple of keys, the program filters the results and retains only the triples with an aggregate variance that does not exceed the specified threshold value τ.

4.3 Result

The outcomes obtained from Question 2 and Question 3 for identical τ values are identical, as demonstrated in Figure 1.

4.4 Performance

Table 1 illustrates the performance comparison between Q2 and Q3 for different dataset sizes, measured in execution time. It is evident that Q3 outperforms Q2, especially for larger datasets. The reason behind this performance difference is attributed to the use of different abstractions in the two approaches. Specifically, Q2 utilizes Spark SQL, a higher-level abstraction that employs SQL queries involving multiple joins and aggregations. These operations can become complex and inefficient as the dataset size grows. In contrast, Q3 uses RDDs, a lower-level abstraction that performs map and filter operations, which can be easily parallelized and optimized by Spark. Additionally, Q3 preprocesses data into dictionaries and broadcasts them,

Table 1: Execution time(s).

Method	Dataset						
	250×10	250×100	250×1000	250×10000	500×10000	750×10000	1000×10000
Q2	33 s	32 s	37 s	72 s	timeout	timeout	timeout
Q3	17 s	17 s	19 s	28 s	2 min	4 min	7 min

distributing common data to all worker nodes, which reduces the need for data shuffling and communication overhead. Both Q2 and Q3 employ caching and repartitioning strategies, but Q3 leverages RDDs more effectively, which is likely why it outperforms Q2.

5 Question 4

For this part of the assignment, count-min sketch (CMS) was utilized to estimate the aggregate variance. CMS is a popular probabilistic data structure used for estimating the frequency of items in a dataset. It is particularly suitable for handling large datasets efficiently. The reason for selecting CMS is its space efficiency, as it uses a fixed amount of memory irrespective of the number of items in the dataset. This makes it a practical option for storing and processing large datasets. The working principle of CMS is based on hashing. It comprises a fixed number of counters organized in a two-dimensional array. Each row in the array corresponds to a hash function, while each column represents a counter for a specific item. By leveraging CMS, the program estimates the aggregate variance as outlined in the Algorithm 1 presented in Appendix A.

1. We create a 2-dimensional array CMS with a number of rows $d = \lceil \ln(\frac{1}{\delta}) \rceil$ and a number of columns $w = \lceil \frac{e}{\epsilon} \rceil$. Each vector is hashed into the CMS by inputting the index (0 to 9999) into the respective hash function to determine the location of the value to be hashed in each row. The value in a cell of CMS is accumulated after hashing with multiple values.
2. We add the three CMSs for the vectors in a triple to obtain the aggregate CMS (CMS_agg).
3. To compute $\frac{1}{l}\sum_{i=1}^{l}(X[i])^2$, we perform the inner product for the aggregate CMS with itself, select the minimum value, and then divide it by the length of the vector which is 10000, i.e., min(CMS_agg $\odot$ CMS_agg)/10000.
4. To compute $\mu_X = \sum_{j=1}^{l} \frac{X[j]}{l}$, we sum up the first row in CMS_agg and then divide it by the length of the vector which is 10000, i.e., sum(CMS_agg[0])/10000.
5. We estimate the aggregate variance using $\hat{\sigma}^2 = (\frac{1}{l}\sum_{i=1}^{l}(X[i])^2) - (\sum_{j=1}^{l} \frac{X[j]}{l})^2$.
6. Next, we compute the relative error part using $tauError = \tau + \epsilon \times \frac{\|sum(CMS_{agg}[0])\|_1 \|sum(CMS_{agg}[0])\|_1}{10000}$.
7. Lastly, we estimate the number of triples of vectors by comparing the estimated $\hat{\sigma}^2$ and $tauError$.

We briefly discuss our Spark solution for identifying triple vectors that meet the defined aggregate variance thresholds. The diagram illustrating our solution is shown in Figure 3. Firstly, we generated the dataset and created a Resilient Distributed Dataset (RDD) for the vector IDs and their elements, distributing them across all CPU cores. Next, we performed a mapping operation to generate a CMS for each vector and collected these CMSs back to the driver. These CMSs were translated into dictionaries and broadcasted to all workers. In addition, we created distinct triples of vector IDs in the RDD, which were also distributed across all CPU cores. These triples of vectors found their respective CMSs from the broadcasted CMS dictionary, and the CMSs were used to compute the aggregate variance and relative error as explained above and in Algorithm 1 in Appendix A. Subsequently, we filtered all triples of vectors to remove those that did not meet the required threshold including relative error, for example, kept only the triples of vectors with an aggregate variance lower than the threshold of 400. Finally, we counted the number of triples of vectors from all workers and sent them back to the driver to output the results.

We utilized and slightly modified Theorem 3 from Cormode and Muthukrishnan [1] to compute the aggregate variance. Our theorem states that $\mathbf{a} \odot \mathbf{a} \leq \widehat{\mathbf{a} \odot \mathbf{a}}$, and with probability $1 - \delta$, $\widehat{\mathbf{a} \odot \mathbf{a}} \leq \mathbf{a} \odot \mathbf{a} + \frac{\epsilon \|\mathbf{a}\|_1 \|\mathbf{a}\|_1}{l}$, where l is the length of the vector. The proof of this theorem follows that of Theorem 3 from Cormode and Muthukrishnan [1].

Proof. $(\widehat{\mathbf{a} \odot \mathbf{a}})_j = \sum_{i=1}^{n} a_i a_i + \sum_{p \neq q, h_j(p) = h_j(q)} a_p a_q$. Clearly, $\mathbf{a} \odot \mathbf{a} \leq \widehat{\mathbf{a} \odot \mathbf{a}}$ for non-negative vectors. By pairwise independence of h,

$\mathbb{E}(\widehat{\mathbf{a} \odot \mathbf{a}}_j - \mathbf{a} \odot \mathbf{a}) = \frac{1}{l} \sum_{p \neq q} P[h_j(p) = h_j(q)] a_p a_q \leq \frac{1}{l} \sum_{p \neq q} \frac{\epsilon a_p b_q}{e} \leq \frac{1}{l} \frac{\epsilon \|\mathbf{a}\|_1 \|\mathbf{a}\|_1}{e}$.

So, by the Markov inequality, $P[\widehat{\mathbf{a} \odot \mathbf{a}}_j - \mathbf{a} \odot \mathbf{a} > \frac{\epsilon \|\mathbf{a}\|_1 \|\mathbf{a}\|_1}{l}] \leq \delta$, as required. □

Using the theorem mentioned above, we can obtain the following probabilistic guarantee: $P[\widehat{\mathbf{a} \odot \mathbf{a}}_j - \mathbf{a} \odot \mathbf{a} > \frac{\epsilon \|\mathbf{a}\|_1 \|\mathbf{a}\|_1}{l}] \leq \delta$. Given the values of $\|\mathbf{a}\|_1$ and l, the user can select appropriate values for ϵ and δ to achieve ϵ and δ guarantees. According to theorem, this theoretical result shows that as ϵ increases, the relative error increases. Therefore, we expect the identified triple of vectors increases as ϵ increases.

We implemented the solution using PySpark on the provided data, and the results are presented in Table 2. Initially, we observed that when $\epsilon = 0.0001$, the width of the count min sketch was calculated as $w = \lceil \frac{e}{\epsilon} \rceil = \lceil \frac{e}{0.0001} \rceil = 27183$, which is larger than the length of the vector (10000). This implies that CMS uses a larger memory to store the information of vectors. Therefore, $\epsilon = 0.0001$ is not a meaningful or appropriate error setting. Additionally, we observed that the execution time for $\epsilon = 0.0001$ is 25 minutes, which is due to the slow implementation of PySpark in aggregate variance computation. We expect that the execution time would be drastically reduced if implemented in Spark's Java version.

Before further discussion, we implemented our PySpark algorithm for question 3 using two different thresholds, namely > 200000 and at least > 1000000. We found that there is at least one triplet for the threshold of > 200000, while there is no triplet for the threshold of > 1000000.

According to Table 2, for the threshold < 400, almost all triples of vectors were identified as being lower than the threshold < 400. This is due to the low signal-to-noise ratio in the sketches, resulting in significant noise when identifying the number of triplets of vectors at this threshold. However, for the higher thresholds > 200000 and > 1000000, which contain higher signal, the number of triplets is relatively closer to the actual results of 1 and 0, respectively. The lower number of triplets identified for the thresholds > 200000 and > 1000000 indicates a low signal-to-noise ratio for these CMSes, making it difficult for them to accurately estimate the actual number of triplets of vectors. As ϵ increases from 0.0001 to 0.01, the error increases, and the identified number of triplets of vectors increases from 1 to 742. This meets the theoretical result and expectation as described above.

In discussing the usefulness of the sketch for optimizing the functionalities and parameters mentioned above, we found that the F1-score for almost all cases in Table 2 is 0.0 or very close to 0.0, except $\epsilon = 0.0001$ and threshold > 200000 which F1-score equals to 1.0. This indicates that CMS is not useful in optimizing these parameters. While $\epsilon = 0.0001$ and $\delta = 0.1$ with threshold $\tau > 200000$ is the closest to the exact result, $w = 27183$ for $\epsilon = 0.0001$ is larger than the length of the vector (10000), resulting in the CMS using more memory to store vector information. Therefore, $\epsilon = 0.0001$ is not a meaningful or appropriate error setting. We believe that if we computed the exact result without using the sketch, a threshold configuration of $\tau > 200000$ would be appropriate. Our PySpark implementation from question 3 supports this claim, as we identified a single triple of vectors with an aggregate variance exceeding 200000.

Next, we discuss the tightness of the bound of the CMS when estimating the variance of an aggregated vector with non-negative integer elements. For an aggregated vector a with length l, the range of variance is given by $[0, \frac{||a||^2}{l} - \frac{||a||^2}{l^2}]$. The estimation error arises from the inner product of vector a, and its range is $[0, \frac{\epsilon ||a||^2}{l}]$. Thus, for vectors with very low variance (i.e., $\tau = 400$), the ratio of estimation error to true variance is $O(\frac{\epsilon ||a||^2}{l})$, while for vectors with very high variance (i.e., $\tau = 200000$), the ratio of estimation error to true variance is $O(\epsilon)$. Therefore, the CMS bound is not tight when querying vectors with small variance, but it becomes tighter when querying vectors with larger variance.

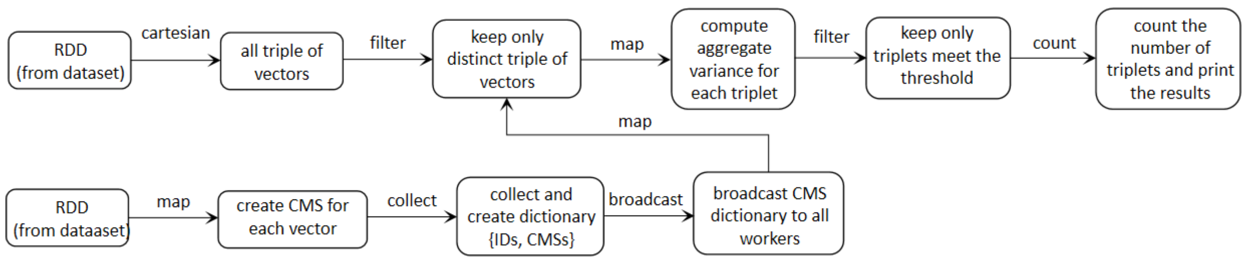
RDD (from dataset)
cartesian
all triple of vectors
filter
keep only distinct triple of vectors
map
compute aggregate variance for each triplet
filter
keep only triplets meet the threshold
count
count the number of triplets and print the results
map
RDD (from dataaset)
map
create CMS for each vector
collect
collect and create dictionary {IDs, CMSs}
broadcast
broadcast CMS dictionary to all workers

Figure 3: Diagram to find the triple of vectors which meet the requirement of thresholds using PySpark.

Table 2: Number of triples and execution time for Question 4.

ϵ	δ	threshold (τ)	Number of triples	Execution time	precision ($\frac{TP}{TP+FP}$)	recall ($\frac{TP}{TP+FN}$)	F1-score ($\frac{2 \times recall \times precision}{recall + precision}$)
0.001	0.1	< 400	2572258	3 min	$\frac{11}{11+2572247} = 4.276 \times 10^{-6}$	$\frac{11}{11+0} = 1.0$	8.552×10^{-6}
0.01	0.1	< 400	2562349	50 s	$\frac{11}{11+2562338} = 4.293 \times 10^{-6}$	$\frac{11}{11+0} = 1.0$	8.586×10^{-6}
0.0001	0.1	> 200000	1	25 min	$\frac{1}{1+0} = 1.0$	$\frac{11}{11+0} = 1.0$	1.0
0.0001	0.1	> 1000000	0	25 min	0.0	0.0	0.0
0.001	0.1	> 200000	687	3 min	$\frac{1}{1+686} = 1.456 \times 10^{-3}$	$\frac{1}{1+0} = 1.0$	2.907×10^{-3}
0.001	0.1	> 1000000	0	3 min	0.0	0.0	0.0
0.002	0.1	> 200000	742	1 min 30 s	$\frac{1}{1+741} = 1.347 \times 10^{-3}$	$\frac{1}{1+0} = 1.0$	2.690×10^{-3}
0.002	0.1	> 1000000	1	1 min 30 s	$\frac{0}{0+1} = 0.0$	0.0	0.0
0.01	0.1	> 200000	742	50 s	$\frac{1}{1+741} = 1.347 \times 10^{-3}$	$\frac{1}{1+0} = 1.0$	2.690×10^{-3}
0.01	0.1	> 1000000	742	50 s	$\frac{0}{0+742} = 0.0$	0.0	0.0

Note: The PySpark algorithm from question 3 was utilized, and it was found that there exists one triplet with a threshold of at least 200000, while there are no triplets with a threshold of at least 1000000 on the dataset 250×10000.

6 Contribution

The contributions of each team member is tabulated in Table 3.

Table 3: Contribution for each team member.

Name	Work	Percentage
Thiam Wai Chua	question 1, question 3, question 4, report, poster	16.67%
Yang Yang	question 1, question 3, question 4, report, poster	16.67%
Xiaohan Wang	question 2, question 3, report, poster	16.67%
Freddie Liu	question 1, question 2, report, poster	16.67%
Zheyuan Su	question 1, question 2, question 3, report	16.67%
Guangyu Li	question 4	16.67%

References

[1] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, 2005.

A Appendix

Algorithm 1 Estimate number of triple of vectors with aggregate variance

Input: RDD, threshold τ, epsilon ϵ, delta δ

Output: Number of triple of vectors lower than or higher than a threshold

```
0: function CMS($vector, \epsilon, \delta$)
1: initialize 2D array count min sketch (CMS) with size $\lceil \ln(\frac{1}{\delta}) \rceil \times \lceil \frac{e}{\epsilon} \rceil$ fill with zero
2: for index $i$ in vector do
3:    for row $k$ in CMS do
4:       column $j \leftarrow k^{th}$ hash function($i$)
5:       CMS[$k$][$j$] $\leftarrow$ CMS[$k$][$j$] + vector value at $i$
6:    end for
7: end for
8: return CMS
8: end function

8: function AGGREGATE_VARIANCE(CMS_A, CMS_B, CMS_C, $\tau, \epsilon$)
9: CMS_agg $\leftarrow$ CMS_A + CMS_B + CMS_C
10: CMS_aggInner $\leftarrow$ inner_product(CMS_agg, CMS_agg)
11: aggregate variance $\leftarrow$ min(CMS_aggInner)/10000 - mean(CMS_agg[0])$^2$/10000
12: error $\leftarrow \tau + \epsilon \times \|$CMS_agg[0]$\|_1 \times \|$CMS_agg[0]$\|_1$/10000
13: return aggregate variance, error
13: end function

14: extract vector IDS: $RDD\_IDs \leftarrow$ get vector IDs
15: triples of vectors with cartesian: $RDD\_IDs\_cart \leftarrow RDD\_IDs$.cartesian($RDD\_IDs$).cartesian($RDD\_IDs$)

16: keep distinct triples of vectors: $RDD\_cart \leftarrow RDD\_IDs\_cart[0] < RDD\_IDs\_cart[1] < RDD\_IDs\_cart[2]$
17: create count-min sketch for each vector: $RDD.map$(CMS($vector, \epsilon, \delta$))
18: broadcast all count min sketches to all workers
19: filter the RDD_cart based on the threshold: Number of triple of vectors $\leftarrow$ RDD_cart.filter(compare aggregate variance and relative error return from AGGREGATE_VARIANCE(CMS_v1, CMS_v2, CMS_v3, $\tau$, $\epsilon$))

20: return  Number of triple of vectors =0
```

Listing 2: Physical plan of the SparkSQL query

```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- Filter (isnotnull(final_var#52) AND (final_var#52 <= 410.0))
   +- HashAggregate(keys=[sorted_id#51], functions=[sum(var#40), sum(dot#43), sum(mean_product#44)])
      +- Exchange hashpartitioning(sorted_id#51, 200), ENSURE_REQUIREMENTS, [plan_id=881]
         +- HashAggregate(keys=[sorted_id#51], functions=[partial_sum(var#40), partial_sum(dot#43),
              partial_sum(mean_product#44)])
            +- Project [mean_product#44, var#40, dot#43, concat_ws(, array_sort(array(id1#41, id2#42, id
                 #21), lambdafunction(if ((isnull(lambda left#78) AND isnull(lambda right#79))) 0 else if
                  (isnull(lambda left#78)) 1 else if (isnull(lambda right#79)) -1 else if ((lambda left
                 #78 < lambda right#79)) -1 else if ((lambda left#78 > lambda right#79)) 1 else 0, lambda
                  left#78, lambda right#79, false), false)) AS sorted_id#51]
               +- BroadcastNestedLoopJoin BuildLeft, Cross, (NOT (id#21 = id1#41) AND NOT (id#21 = id2
                    #42))
                  :- BroadcastExchange IdentityBroadcastMode, [plan_id=876]
                  :  +- Project [id#21, ((cast(aggregate(int_list#32, 0, lambdafunction((lambda acc#58L +
                       cast((lambda x#59 * lambda x#59) as bigint)), lambda acc#58L, lambda x#59, false)
                      , lambdafunction(lambda id#60L, lambda id#60L, false)) as double) / cast(size(
                      int_list#32, true) as double)) - pow((cast(aggregate(int_list#32, 0,
                      lambdafunction((lambda acc#61 + lambda x#62), lambda acc#61, lambda x#62, false),
                      lambdafunction(lambda id#63, lambda id#63, false)) as double) / cast(size(int_list
                      #32, true) as double)), 2.0)) AS var#40]
                  :     +- Project [_c0#17 AS id#21, cast(split(_c1#18, ;, -1) as array<int>) AS int_list
                      #32]
                  :        +- Exchange RoundRobinPartitioning(24), REPARTITION_BY_NUM, [plan_id=858]
                  :           +- Filter isnotnull(_c0#17)
                  :              +- FileScan csv [_c0#17,_c1#18] Batched: false, DataFilters: [isnotnull(
                      _c0#17)], Format: CSV, Location: InMemoryFileIndex(1 paths)[file:/E:/PyProject/
                      sublib/Bigdata/vectors.csv], PartitionFilters: [], PushedFilters: [IsNotNull(_c0)
                      ], ReadSchema: struct<_c0:string,_c1:string>
                  +- Project [id#21 AS id1#41, id#66 AS id2#42, aggregate(zip_with(int_list#32, int_list
                      #57, lambdafunction((lambda x#67 * lambda y#68), lambda x#67, lambda y#68, false))
                      , 0, lambdafunction((lambda acc#75 + lambda x#76), lambda acc#75, lambda x#76,
                      false), lambdafunction(lambda id#77, lambda id#77, false)) AS dot#43, (((cast(
                      aggregate(int_list#32, 0, lambdafunction((lambda acc#69 + lambda x#70), lambda acc
                      #69, lambda x#70, false), lambdafunction(lambda id#71, lambda id#71, false)) as
                      double) / cast(size(int_list#32, true) as double)) * cast(aggregate(int_list#57,
                      0, lambdafunction((lambda acc#72 + lambda x#73), lambda acc#72, lambda x#73, false
                      ), lambdafunction(lambda id#74, lambda id#74, false)) as double)) / cast(size(
                      int_list#57, true) as double)) AS mean_product#44]
                     +- BroadcastNestedLoopJoin BuildRight, Inner, (id#21 < id#66)
                        :- Project [_c0#53 AS id#21, cast(split(_c1#54, ;, -1) as array<int>) AS int_list
                             #32]
                        :  +- Exchange RoundRobinPartitioning(24), REPARTITION_BY_NUM, [plan_id=861]
                        :     +- Filter isnotnull(_c0#53)
                        :        +- FileScan csv [_c0#53,_c1#54] Batched: false, DataFilters: [isnotnull(
                             _c0#53)], Format: CSV, Location: InMemoryFileIndex(1 paths)[file:/E:/
                             PyProject/sublib/Bigdata/vectors.csv], PartitionFilters: [], PushedFilters:
                             [IsNotNull(_c0)], ReadSchema: struct<_c0:string,_c1:string>
                        +- BroadcastExchange IdentityBroadcastMode, [plan_id=872]
                           +- Project [_c0#55 AS id#66, cast(split(_c1#56, ;, -1) as array<int>) AS
                                int_list#57]
                              +- Exchange RoundRobinPartitioning(24), REPARTITION_BY_NUM, [plan_id=863]
                                 +- Filter isnotnull(_c0#55)
                                    +- FileScan csv [_c0#55,_c1#56] Batched: false, DataFilters: [
                                         isnotnull(_c0#55)], Format: CSV, Location: InMemoryFileIndex(1
                                         paths)[file:/E:/PyProject/sublib/Bigdata/vectors.csv],
                                         PartitionFilters: [], PushedFilters: [IsNotNull(_c0)],
                                         ReadSchema: struct<_c0:string,_c1:string>
```