# 2AMD15 - Big Data Management

Calli Evers[1252194] Sander Cauberg[1008909] Filippo Daniotti[1950002] Horea Breazu[1229343]
Klára Tauchmanová[1955853] Filipe da Costa Quinta Goncalves Sobrinho[1890352]

## I. Question 2

*a) **SQL query**:* First, we perform two sequential `DataFrame.crossJoin()` operations, each of them followed by a filtering operation to remove repetitions from the combination: this way, we obtained all the triples of vectors $< X, Y, Z >$ in the dataset. Then, we compute the aggregate vectors for all the triples and compute the variance of those vectors. The chain of operations can easily be read from the following SQL query:

```
SELECT CONCAT_WS(',', t1._1, t2._2, t3._3) AS full_id,
       var_udf(agg_udf(ARRAY(t1.ARR, t2.ARR2, t3.ARR3))) AS var
FROM (
    SELECT * FROM (
        SELECT * FROM (
            SELECT _1, ARRAY(_2, ...) AS ARR FROM dataframe) t1
            CROSS JOIN (
                SELECT _1 AS _2, ARRAY(_2, ...) AS ARR2 FROM dataframe) t2
        WHERE t1._1 < t2._2) t12
        CROSS JOIN (
            SELECT _1 AS _3, ARRAY(_2, ...) AS ARR3 FROM dataframe) t3
    WHERE t12._2 < t3._3)
WHERE var < τ
```

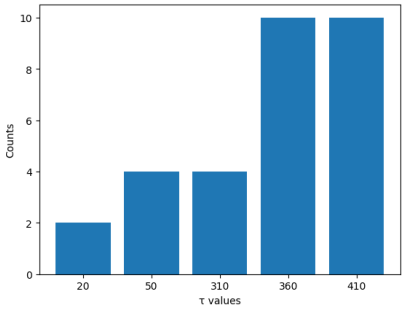


Figure 1: Triples with aggregate variance $\leq \tau$

The User Defined Function `var_udf` in the query retrieves the vector corresponding to each id from the broadcasted map, computes the aggregate vector (component-wise summation of three vectors) and then computes the variance of the aggregate vector (*aggregate variance*).

*b) **Results**:* Our query took **237.11 seconds** to run on the cluster and the results are summarized in Fig.1.

| Optimization | Execution time |
|---|---|
| All | 163.56s |
| No numpy | 443.19s |
| No repartition/coalesce | 2108.38s |
| No broadcast | ran out of heap |

Table I: Impact of optimizations tricks on the dataset of size 125x10000, ran locally.

| Partitions | Execution time |
|---|---|
| 32-64-128 | 237.11s |
| 16-32-64 | 246.13s |
| 64-128-256 | 295.03s |
| 64-64-64 | 323.13s |

Table II: Impact of partition combinations on the dataset of size 250x10000, ran on the server.

*c) **Optimization**:* The main indicator that we considered evaluating the performance of our SQL query was the *execution time*. We leveraged a variety of techniques to improve it; their impact on performances is summarized in Tab.I. In the following, we are presenting the most relevant ones.

- We broadcast the ID-values mapping to the worker nodes: this allows us to transfer only the IDs during the shuffle phases, severely reducing the network transfer speed bottleneck.
- We use the `.coalesce()` API to reduce the number of partitions after the two `.crossJoin()` operations; the number of partitions has been empirically fine-tuned (see Tab.II).
- We use `numpy.ndarrays` instead of Python's built-in data types, as they provide better performances on mathematical operations; moreover, we were able to further improve the performances by reducing the size of the integer to 16 bits integers - which is the minimum size we could use before overflow issues would occur.

## II. Question 3

*a) **System architecture**:* The diagram in Fig.2 summarizes our final system architecture for Question 3.

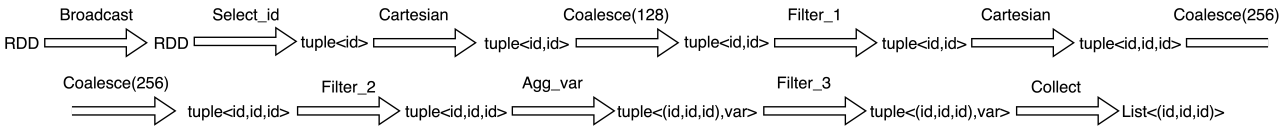


Figure 2: Brief description of system architecture for Q3

- `Broadcast`: broadcasts a $< \text{id}, \text{vector} >$ map over the worker nodes.
- `Select_id`: filters the RDD such that the `.cartesian()` will be computed only on ids.
- `Cartesian`: performs the cartesian product of the RDD with itself; outputs a new RDD of length `len(rdd)`$^2$.
- `Coalesce(number)`: returns a new RDD with at most `numPartitions` number of partitions.
- `Filter_1`: filters out all tuples where the first element is greater than or equal to the second, preventing duplicates.
- `Filter_2`: filters out all tuples where the second element is greater than or equal to the third, preventing duplicates.
- `Agg_var`: gets the vectors associated with the keys in the tuples from the broadcast variable, calculates the aggregate variance, adds 1 to the accumulator if it's less than or equal to 410, and returns a pair with the keys tuple and the variance.
- `Filter_3`: filters out all variances greater than 20.
- `Collect`: returns a Python list with all the elements in the RDD.

*b) **Performances and optimizations**:* The execution time of Q3 on the cluster was **910.05 seconds**. In the following, we summarize the most important optimization tricks we leveraged to obtain our results. The impact of each optimization tricks is summarized in Tab.III and Tab.IV.

- We use `numpy.ndarray`s of 16 bits of precision to represent the values of the vectors (see Sec.I).
- We broadcast the vector map to reduce the network transfer during the shuffle phases (see Sec.I).
- We used the `.coalesce()` API to reduce the number of partitions after the `.cartesian()` operations (see Sec.I).
- We used the `.Accumulator()` API during the variance computation: this allowed us to filter out the $\tau \leq 20$ triples while still retaining in the accumulator the amount of triples with $\tau \leq 410$.

| Optimization | Execution time |
|---|---|
| All | 30.59s |
| No numpy | 2940.19s |
| No broadcast | 185.30s |
| No accumulator | 36.44s |
| No coalesce | 131.46s |

Table III: Impact of optimizations tricks on the dataset of size 250x10000, ran locally.

| Partitions | Execution time |
|---|---|
| 64-128-256 | 910.05s |
| 128-256-512 | 946.79s |
| 125-250-500 | 969.51s |
| 64-64-128 | 1005.65s |
| 25-50-100 | 1097.70s |
| 64-64-64 | 1122.74s |
| 128-128-128 | 1162.01s |

Table IV: Impact of partition combinations on the dataset of size 1000x10000, ran on the server.

Besides the aforementioned, we have tried a variety of optimization strategies; even though they were discarded, some of them are still worth reporting, as a considerable amount of effort was put into them and they can still be useful in other scenarios.

- We tried to pre-process the RDD, to obtain the mean of each row before the variance computation and then used this mean as input for the `statistics.pvariance()` function. This allowed us to save some time for all the aggregate variance computation. We discarded this technique in favor of the usage of `numpy.var()`.
- A thing we tried instead of using an accumulator, was filtering out all triples with $\tau \leq 410$ and caching these. Using this cache we could then both count these triples and get the triples with $\tau \leq 20$ by filtering again. It turned out that using the accumulator was slightly faster.

*c) **Discussion on Q3 vs Q2**:* We ran the code for *Question 3* on the dataset generated for *Question 2* and we found out that the results are the same (they are summarized in Tab.V).

While the two implementations produce the same results, the execution time of the code of Q3 is significantly smaller than the execution time of Q2. It is not trivial to understand the reason behind such behavior since the overall approach is comparable in its core steps and all the main optimization techniques were employed in both implementations.

| | Q2 code | Q3 code |
|---|---|---|
| Results $\tau = 20$ | 2 | 10 |
| Results $\tau = 420$ | 2 | 10 |
| Execution time | 237.11 | 42.35 |

Table V: Comparison between Q2 code and Q3 code on the same dataset (250 vectors).

One possible reason can be the different usage of `.coalesce()` and `.repartition()`. In the code of Q2, we employ one `.repartition()` on the dataframe with only the keys, before then doubling the partition count after each `.crossJoin()`; meanwhile, in Q3 we also use `.coalesce()` twice, once after every `.cartesian()` call (after the first we reduce the partitions to 128, and after the second to 256), while the repartition is inherited from the initial RDD creation. It may be the case that the configuration of partitions is suboptimal and, subsequently, our implementation for Q2 is not parallelizing enough to leverage the computing power of the cluster, or conversely, we are parallelizing too much. However, we have not found a better combination of partition counts: such values for the partitions are not trivial for us to determine and require a lot of trial and error and empirical evaluation; hence, we still believe that our solution for Q2 is valuable.

One other reason could be that our implementation of Q3 only queries 2 values ($\tau = 20$ and $\tau = 410$), while the solution for Q2 queries 5 values. However, since most of the time is taken by the combinations and aggregate variances computations, it is unlikely that this is the reason behind the lower performances of Q2.

Ultimately, the `DataFrame` API of SparkSQL might be inherently slower than the usage of plain Spark, or at least in our use-case, since the former is a library built on top of the Spark Core. In particular, the combination of a large user-defined function with SparkSQL might slow the overall execution down.

## III. Question 4

*a)* ***Sketch usage, validity and theoretical results****:* We used *Count-min (CM) sketch* (Cormode and Muthukrishnan [2005]) for representing the original vectors. Specifically, given parameters $(\varepsilon, \delta)$, we create a *CM sketch* with width $w = \lceil \frac{e}{\varepsilon} \rceil$ and depth $d = \lceil \ln \frac{1}{\delta} \rceil$ for each vector $\mathbf{x} = (x_1, \ldots, x_{10000})$. The *CM sketch* is created as follows: the $i$-th element of vector $\mathbf{x}$ is added to the counts on position $[j, h_j(i)]$ in row $j$. Formally,

$$\text{count}_{\mathbf{x}}[j, h_j(i)] = \text{count}[j, h_j(i)] + x_i, \quad \forall i, 1 \leq i \leq 10000 \text{ and } \forall j, 1 \leq j \leq d \tag{1}$$

Since every single *CM sketch* is constructed with identical hash functions and parameters, function $h_j$ hash the $i$-th element of vector $\mathbf{x}$ to the same position as the $i$-th element of vector $\mathbf{y}$. Therefore, we can sum the counters of vector triplet $\mathbf{x}, \mathbf{y}, \mathbf{z}$ pairwise to get a *CM sketch* approximating the aggregate vector $\mathbf{a} = \mathbf{x} + \mathbf{y} + \mathbf{z}$ of that triplet. Formally,

$$\text{count}_{\mathbf{a}}[j, i] = \text{count}_{\mathbf{x}}[j, i] + \text{count}_{\mathbf{y}}[j, i] + \text{count}_{\mathbf{z}}[j, i], \quad \forall \mathbf{x} \neq \mathbf{y} \neq \mathbf{z} \text{ and } \forall j, 1 \leq j \leq d \text{ and } \forall j, 1 \leq i \leq w \tag{2}$$

Clearly, $||\mathbf{a}||_1 = ||\mathbf{x}||_1 + ||\mathbf{y}||_1 + ||\mathbf{z}||_1$. The aggregate variance $\sigma^2 = \left(\frac{1}{10000} \sum_{i=1}^{10000} a_i^2\right) - \mu_{\mathbf{a}}^2$ can be approximated by the inner product of the *CM sketch* approximating the aggregate vector $\mathbf{a}$. Specifically, we used $\widehat{(\mathbf{a} \odot \mathbf{a})}$ from the Cormode and Muthukrishnan [2005] paper to approximate the sum of the aggregate vector squared. The aggregate variance can be expressed as follows,

$$\widehat{\sigma}^2 = \left(\frac{1}{10000} \widehat{(\mathbf{a} \odot \mathbf{a})}\right) - \mu_{\mathbf{a}}^2 \tag{3}$$

**Theorem 1.** *$\sigma^2 \leq \widehat{\sigma^2}$ and, with probability $1 - \delta$, we have that $\widehat{\sigma^2} \leq \sigma^2 + \frac{1}{10000}\varepsilon||\mathbf{a}||_1^2$.*

*Proof.* From the **Theorem 3** from the Cormode and Muthukrishnan [2005] paper, $(\mathbf{a} \odot \mathbf{a}) \leq \widehat{(\mathbf{a} \odot \mathbf{a})}$ for non-negative vectors. By the definition of $\widehat{\sigma^2}$ and $\sigma^2$,

$$\sigma^2 = \frac{1}{10000}(\mathbf{a} \odot \mathbf{a}) - \mu_{\mathbf{a}}^2 \leq \frac{1}{10000}\widehat{(\mathbf{a} \odot \mathbf{a})} - \mu_{\mathbf{a}}^2 = \widehat{\sigma^2} \tag{4}$$

Therefore, $\sigma^2 \leq \widehat{\sigma^2}$ for non-negative vectors. Recalling the definition of $\widehat{\sigma^2}$ and $\sigma^2$,

$$\widehat{\sigma^2} - \sigma^2 = \left(\frac{1}{10000}\widehat{(\mathbf{a} \odot \mathbf{a})} - \mu_{\mathbf{a}}^2\right) - \left(\frac{1}{10000}(\mathbf{a} \odot \mathbf{a}) - \mu_{\mathbf{a}}^2\right) = \frac{1}{10000}\left(\widehat{(\mathbf{a} \odot \mathbf{a})} - (\mathbf{a} \odot \mathbf{a})\right) \tag{5}$$

Therefore,

$$\mathbb{P}\left[\widehat{\sigma^2} - \sigma^2 \geq \frac{1}{10000}\varepsilon||\mathbf{a}||_1^2\right] = \mathbb{P}\left[\frac{1}{10000}\left(\widehat{(\mathbf{a} \odot \mathbf{a})} - (\mathbf{a} \odot \mathbf{a})\right) \geq \frac{\varepsilon||\mathbf{a}||_1^2}{10000}\right] = \mathbb{P}\left[\left(\widehat{(\mathbf{a} \odot \mathbf{a})} - (\mathbf{a} \odot \mathbf{a})\right) \geq \varepsilon||\mathbf{a}||_1^2\right] \tag{6}$$

From the **Theorem 3** from the Cormode and Muthukrishnan [2005] paper, $\mathbb{P}\left[\left(\widehat{(\mathbf{a} \odot \mathbf{a})} - (\mathbf{a} \odot \mathbf{a})\right) \geq \varepsilon||\mathbf{a}||_1^2\right] \leq \delta$.
So, $\mathbb{P}\left[\widehat{\sigma^2} - \sigma^2 \geq \frac{1}{10000}\varepsilon||\mathbf{a}||_1^2\right] \leq \delta$, as required. *QED*

**Corollary 1.** *Recall that width $w = \lceil \frac{e}{\varepsilon} \rceil$ and depth $d = \lceil \ln \frac{1}{\delta} \rceil$; that means that the $w$ and $d$ parameters of the sketch can be tweaked in order to achieve the desired $\varepsilon$ and $\delta$ guarantees. It is important to specify that the final size of the sketch will be $w \cdot d$; this size could be bigger than the size of the original problem if $\varepsilon$ and $\delta$ are too strict.*

The pseudocode in Alg.1 and Alg.2 illustrates how we approximate the aggregate variance with CM sketches. Note that Alg.1 refers on how to create one single sketch for one single vector: this means that such operation can be easily performed in a distributed fashion inside a `.map()` procedure, and the same holds for Alg.2. They can leverage all the capabilities of Spark, such as variable broadcasting and repartition/coalesce.

**Algorithm 1:** Creates CM sketch from vector.

**Input:** vector[], hash_functions, $\varepsilon$, $\delta$
**Output:** sketch[]

$w = \lceil e/\varepsilon \rceil$, $d = \lceil \ln 1/\delta \rceil$
sketch[] = new Table(size = ($w$, $d$), fill = 0)
**for** $i = 0$ *to* 10000 **do**
  **for** $j = 0$ *to* $d$ **do**
    index = hash_functions $[j](i)$ mod $w$
    sketch[$j$, index] $\leftarrow$ sketch[$j$, index] + vector[$i$]
**return** sketch

**Algorithm 2:** Computes aggregate variance.

**Input:** sketch_1, sketch_2, sketch_3, $w$
**Output:** approx_agg_var

approx_agg_var = sketch_1 + sketch_2 + sketch_3
inner_product = $\sum_{j=1}^{w}$ (approx_agg_var$[j]^2$)
**return** $\frac{1}{10000}$min(inner_product) $- \mu^2_{\text{inner\_product}}$

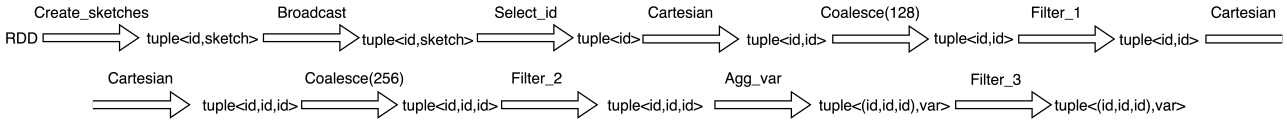


Figure 3: Brief description of system architecture for Q4

The diagram in Fig.3 summarizes our final system architecture for Question 4.

- `Create_sketch`: creates a CM sketch for given $\varepsilon$ and $\delta$ for each vector in the dataset.
- `Broadcast`: broadcast a $<\text{id}, \text{sketch}>$ map over the worker nodes.
- `Select_id`: filter the RDD such that the `.cartesian()` will be computed only on ids.
- `Cartesian`: performs the cartesian product between all the elements in two RDDs; outputs a new RDD of length `len(rdd_1)` · `len(rdd_2)`.
- `Coalesce(number)`: return a new RDD with number of partitions `numPartitions`.
- `Filter_1`: filters out all tuples where the first element is greater than or equal to the second.
- `Filter_2`: filters out all tuples where the second element is greater than or equal to the third.
- `Agg_var`: gets the sketches associated with the keys in the tuples from the broadcast variable, estimates the aggregate variance according to Alg.2 and returns a pair with the keys tuple and the variance.
- `Filter_3`: filters out all variances either greater or smaller than a certain threshold (depending on the functionality).

*b)* ***Results and observations****:* It took **164.33 seconds** to execute the Q4 on the cluster. The precision/recall values of the results retrieved by our code for Q4 are summarized in Tab.VI.

| | $\epsilon = 0.0001$ | $\epsilon = 0.001$ | $\epsilon = 0.002$ | $\epsilon = 0.01$ |
|---|---|---|---|---|
| $\tau \leq 400$ | - | $P = 1^{\dagger}_{,} R = 0/10$ | - | $P = 1^{\dagger}_{,} R = 0/10$ |
| $\tau \geq 200000$ | $P = 1, R = 1$ | $P = 1/26819, R = 1$ | $P = 1/91885, R = 1$ | $P = 1/2573000, R = 1$ |
| $\tau \geq 1000000$ | $P = 1^{\dagger}_{,} R = 1^{2}$ | $P = 1^{\dagger}_{,} R = 1^{2}$ | $P = 0/1, R = 1^{2}$ | $P = 0/91885, R = 1^{2}$ |

[1] Number of TPs is 0, number of FTs is also 0 (eg. 0/0)
[2] Number of relevant triplets is 0, number of retrieved triplets is also 0 (eg. 0/0)

Table VI: Precision/recall values of the Q4 results.

The key to properly reading through the precision/recall values is that, since we are using CM sketches, we are *overestimating* the variance. The impact of the approximation depends on the functionality:

- *functionality 1*: if we are filtering the variance lower than a threshold, we underestimate the counts (since we overestimate the variance), but all the counts we detect will be TPs, hence we expect precision to be 1;
- *functionality 2*: if we are filtering the variance higher than a threshold, we overestimate the counts, hence we will get FPs but no FNs, which provides us with a recall value of 1.

Similarly, we can determine how the parameters $\varepsilon$ and $\delta$ affect the precision and recall values.

- Smaller values of $\varepsilon$ imply larger values of width $w$ for the sketch, which ensure fewer collisions and therefore a better approximation of the aggregate variance, resulting in higher recall for *functionality 1* and higher precision for *functionality 2*; the results shown in Tab.VI confirm indeed this observation.
- Similarly, smaller values $\delta$ imply a larger number $d$ of hash functions, which also mitigates the overestimation of the variance and ultimately ensures higher recall for *functionality 1* and higher precision for *functionality 2*.

The main thing to consider when talking about the usefulness of a sketch is the dimension of the sketches. The configuration $\varepsilon = 0.0001$ for *functionality 2* produces a *CM sketch* with over 27.000 columns, which is larger than the size of the original vectors. Even though the results for such configuration are $P = 1$ and $R = 1$, the computation would be much slower than the computation of the exact solution and it still wouldn't ensure the correct results (because of the possibility of collisions). However, $\varepsilon = 0.01$ is not good either, as it produces a very high number of FPs. Using the two intermediate values $\varepsilon = 0.001$ and $\varepsilon = 0.002$ make sense, as they ensure a reduced size of the dataset while retaining a reasonable approximation of the results. Moreover, if the number of FPs is small enough, one could also compute the variance of each of the results and manually discard the FPs; this may take less time than computing an exact solution.

It is not advisable to use this sketch for *functionality 1*: from our experiments, we could see that $\varepsilon = 0.001$ is not enough for obtaining any results. In order to improve the approximation we would need to reduce the value of $\varepsilon$, but this might result in a sketch larger than the original dataset.

Ultimately, we are presenting some concluding remarks on the tightness of the bounds.

- **Lower bound**. The lower bound is tight, as the estimated value is never less than the actual value.
- **Upper bound**. From Theorem 1, we can see that the upper bound depends on $\varepsilon$, $\delta$ and $||\mathbf{a}||_1^2$; the latter dependence implies that the upper bound will be different for each aggregate vector and that the error is relative to the first norm squared. This is unfortunate, as it means that we can fine-tune $\varepsilon$ and $\delta$, but the tightness will inherently have some variability introduced by the varying sum of the elements of the aggregate vectors.

## Appendix A: Contribution

Tab.VII is detailing the contribution of each team member.

| Name | Percentage | Tasks |
|---|---|---|
| Calli Evers | 22% | Q2 definition and implementation, Q3 optimization, server debugging |
| Sander Cauberg | 6% | poster writing, report maintenance and reviewing |
| Filippo Daniotti | 22% | Q2/Q4 optimization, submission scripts, report writing and reviewing |
| Horea Breazu | 6% | poster writing, report maintenance and reviewing |
| Klára Tauchmanová | 22% | Q2 optimization, Q4 definition and theoretical results, report writing and reviewing |
| Filipe Sobrinho | 22% | Q3 definition and implementation, Q4 definition and implementation |

Table VII: Contribution

## References

Graham Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, apr 2005. ISSN 0196-6774. doi: 10.1016/j.jalgor.2003.12.001. URL https://doi.org/10.1016/j.jalgor.2003.12.001.