UNIVERSITY OF GRONINGEN
FACULTY OF SCIENCE AND ENGINEERING
DEPARTMENT OF COMPUTING SCIENCE

Information Systems

Group 07

AUTHORS
Frans Simanjuntak (S3038971)
Stefan Cretu(S3048438)

28 November 2017

Contents:

1. Problem statement

The general requirement for this assignment is to create a database schema, or more database schemas, that stores the amount of rainfall within a given area. An area can be represented by a matrix which contains data points with values for the amount of precipitations. The total number of considered data points should be: 100.000, 1 mln, 10 mln and 100 mln, respectively. For each case, there need to be analyzed the write performances of the corresponding number of data points to database. Also, there is needed to be analyze the read performances for an image and for a given data point, in each of the 4 cases.

2. Approach

The approach in solving the above mentioned problem is described in this section. Thus, in order to properly fulfill the requirements, we outline the input information, one by one, below. The presented input data contain the conclusions or assumptions we made while discussing the project during the lab session, as well as during our group meetings. Moreover, there is an explanation for each such conclusion or assumption.

- *The given area is represented by a 550x500 pixels matrix which contains values for the zones where rain occurs at a given moment, and zeros in rest, as it does not rain every where*

Conclusion 1: That means the matrices are sparse, so there is not needed to store the entire data in a matrix, but just the coordinates for the points where rain occurs, as well as the amount of rain for each such point. That said, the data is stored as a triple (x, y, value), where the pair (x, y) is used to index a pixel in the image, also named datapoint, and value is the measurement of rain contained in that datapoint.

- *Random data can be used to create the matrices.*

Assumption 1: Depending on the amount of data there is needed to be generated, there might exist duplicates, for precipitation values, fact that does not affect the correctness of the assignment's outcome. With respect to coordinates, we try to eliminate duplicates, in order to fulfill the requirement properly, but with the increasing amount of data points, this gets harder to do.

Assumption 2: Since random data can be used to create each matrix (every 5 minutes) the result data may not represent a big cloud, but just random points which have a given amount of rain, whose value is also randomly generated.

- *Every image is generated every 5 minutes.*

Conclusion 2: The data generated for the previous image as well as writing that data to database should take less than 5 minutes.

Conclusion 3: Every 5 minutes a time stamp can be generated, that is used to uniquely identify an image. Multiple data points corresponding to same image, will have same timestamp.

- *100.000 images are generated, in order to simulate the measurements for an entire year (as a consequence of the previous requirement).*

Assumption 3: The amount of clouds can remain constant for each generated image, in each 4 cases. Therefore, when writing different amount of data points, each generated image will contain a constant amount of data points, that is 1000. This number comes from the fact that

there are needed 100.000 images to cover an entire year and the maximum number of data points that have to be written is 100 mln.

Conclusion 4: Given the above assumption and taking into account that maximum number of data points per image is 275000, it means that in our case, the percentage of non zero values per image is also constant, namely 0.37%. That said, matrices will be really sparse fact which supports the approach presented in Conclusion 1.

Conclusion 5: By having the same amount of data points per image in each of the 4 cases, will help us in realistically comparing the reading performance per image, in each considered case, as there will be always retrieved the same amount of data, having the difference in time per query influenced only by the database schema.
That said, in Table 2.1 is presented the number of images needed to be generated for each analyzed case:

Total number of data points (writes to database)	Number of images
100 mln	100.000
10 mln	10.000
1 mln	1.000
100 000	100

Table 2.1 Number of generated images

Conclusion 6: Based on the previous conclusion, and on the above illustrated table, it means that the generated data points will not cover the measurements for one year, but for a shorter period of time. That said, since one image is generated every 5 minutes, it means there are generated 12 images per hour, which means 288 images per day. Therefore, the number of covered days per each case is shown in Table 2.2

Total number of data points (writes to database)	Number of images	Covered period (days)
100 mln	100.000	~ 348
10 mln	10.000	~34.8
1 mln	1.000	~3.48

100 000	100	~ 8.4 hours = ~ ⅓ of a day

Table 2.2 Measurement period covered in each case (in days)

Assumption 4: For each case when we have to generate a given amount of data points, this data does not have to cover an entire year, as long as the number of generated data points is respected, or is closer to the requirement.

Conclusion 7: Based on all previous conclusions and assumptions, the database schema needs to be adapted to the amount of data we have to store there. Since we know from the beginning the amount of data that has to be written to the database, it makes it a realistic approach.

3. Solution

All the assumptions and conclusions presented in Section helped us to shape a solution for the problem described in Section 1.

Firstly, it is needed to mention that the code development is done in Python. Furthermore, we take for analysis 2 different databases: a SQL based one, that is MySQL, and a non-SQL one, that is MongoDB.

Secondly, given the choice of our programming language, we present the data types used in the presented solution, in Table 3.1. Also, here are illustrated the restrictions over the values for this data, given by the input information for this assignment, or by measurements that have been made regarding to amount of rainfall. That said, according to [1], more than 50 mm per hour of rain means there is a massive rainfall. In the following we suppose the maximum value per hour shall be 60 mm, which means that every 5 minutes (frequency of measurement in our case) the maximum shall be 5 mm.

Data	Data type
Timestamp = time when an image was generated	datetime
X = coordinate on X axis of a data point, in a given image	int: [0, 550)
Y = coordinate on Y axis of a data point, in a given image	int: [0, 500)
Precipitation value in the considered data point	float: (0, 5] mm/hour

Table 3.1 Data types and values restriction for data stored in database

Thirdly, as mentioned in Conclusion 7, the database schema is adapted to the number of inserted values. That said, for 100.000 data points there will be generated 100 images, covering almost ⅓ of a day. All this data shall be stored in one single table/collection. For 1 mln data points, there will be generated 1000 images, covering almost 3.5 days. Like in the previous case, all data is stored in same table/collection.
For 10 mln data points, there will be generated 10000 images, covering almost 35 days. In this case, the database will be spread in multiple tables, one table storing the data measured for a week which means there are in total 5 tables. For the last case, an entire year is covered, so the database contains 12 tables, one for each month. In these 2 cases, the queries per image should end in overheads, as looking for an image means looking for a timestamp. This

timestamp contains the day and the month, data that can be retrieved from it and used for addressing a specific table. The performance suffers when querying for a specific datapoint, as the query has to look into multiple tables.

In the following we present our experiment and the results of these experiments conducted on 3 different databases: MongoDB (document based No-SQL), Cassandra (column based, No-SQL) and MySQL (SQL). For these experiments, in order to simulate write and reading performance, we divide the total number of inserted rows or read from database by the total durations of the query/write as given in the following formula.

$$\text{Performance} = \frac{Total\ (Inserted/Retrieved\ Rows)}{Computation\ time}$$

3.1 MongoDB

MongoDB is a document based No-SQL database that stores data in JSON formatted documents. Such a document can be seen as the equivalent of a row in SQL, as it represents an insertion in a collection. A collection contains more documents, thus it can be seen as the equivalent of a table in a SQL database. A database is a physical container for a group of collections and it gets its own files on the file system. [2]

3.1.1 Collection structure discussion

Our approach aims at creating different schemas for structuring a collection, for the two different types of read queries that need to be made, as they are described below:

1. **Best collection schema for reading a certain data point:** A new document with same timestamp is generated for a new datapoint that belongs to a given image. This happens even though the maximum size for a document is 16 MB [3] which would suffice to store 1000 data points, as each one would require 12 Bytes (4 (int) + 4 (int) + 4 (float)). In this way, reading a given data point is expected to be faster, as one document shall be retrieved per matching case. It can be noticed that a document has two fields: a timestamp and a nested document holding data for a datapoint as a triple (x, y, value).
2. **Best collection schema for reading an image:** However, when querying for an image, it would be better if all data points would be stored in the same document, as all the data for the given document would be retrieved. Thus, the structure of a document will contain 1000 datapoint “fields”.

By applying these approaches, for both types of queries there will be retrieved an entire document, each with different sizes. Since in the case of reading an image, the amount of data is ~1000 bigger than when reading a single datapoint, the former is expected to be slower than the latter, but faster than in the situation when one document per data point has to be retrieved for a single image (1000 document in total). Nevertheless, in both scenarios we aim at testing the performances of both queries

3.1.2 Document structure discussion

In order to achieve good read and write operations performances, a proper document structure should be used. That said, firstly it is presented the case when for each datapoint is created a new document, which means that one write or one read of a data point is the equivalent of one write or read of a document. Furthermore, once we know this, it remained to be determined how the data illustrated in table 3.1 should be organized, that is, whether each generated document should contain nested documents, how many and how they should be structured. In the following, we aim at answering to these questions.

In total, we proposed 4 schemas, whose performances are illustrated in Table 3.5 for 3 types of queries: writing 100000 data points, data point reading and image reading. As explained at the beginning of chapter 3, in this case all data is stored in one collection. Firstly, we present the results, in the table below, then each schema is illustrated alongside a discussion of the results, by comparing the performance values between them and, finally, a conclusion regarding which should be better to use is made. On top of this, we tried to find arguments that prove or deny the assumptions we made in section 3.1.1, when we proposed different schemas for a collection structure.

Performance/Schema	Schema 1 (Figure 3.1)	Schema 2 (Figure 3.2)	Schema 3 (Figure 3.3)	Schema 4 (Figure 3.4)
Write performance	Total time: 24.613 sec Writes/sec: 4062.9	Total time: 24.163 sec Writes/sec: 4138.6	Total time: 24.665 sec Writes/sec: 4054.4	Total time: 25.465 sec Writes/sec: 3926.9
Read performance image	Total time: 0.047 sec and returned 1000 data points (all data points for that image) Retrieved documents/sec: 21276.6	Total time: 0.047 sec and returned 1000 data points (all data points for that image) Retrieved documents/sec: 21276.6	Total time: 0.047 sec and returned 1000 data points (all data points for that image) Retrieved documents/sec: 21276.6	Total time: 0.048 sec and returned 1000 data points (all data points for that image) Retrieved documents/sec: 20833.4
Read performance datapoint	Total time: 0.047 sec and returned 100 data points (one data point per image)	Total time: 0.044 sec and returned 100 data points (one data point per image)	Total time: 0.046 sec and returned 100 data points (one data point per image)	Total time: 0.045 sec and returned 100 data points (one data point per image)

	Retrieved documents/sec: 2127.7	Retrieved documents/sec: 2272.8	Retrieved documents/sec: 2173.9	Retrieved documents/sec: 2222.2

Table 3.5 The performance of MongoDB when storing one data point in a document with different schemas

The queries used for getting the results presented in the above table, were performed in the following manner:

- Writing operation: was done using "insert" method, which is called for each newly generated document (data point in this case).
- Reading a data point: it was counted how many data points were found, that matched certain values for x and y. The precipitation value and the timestamp value were not taken into account in the query (see the query below).

```
collection.find({"datapoint.x": 518, "datapoint.y": 219 }).count()
```

However, we tried also to query for only one data point, by specifying a certain timestamp in the query and it was noticed the performance was similar, even though only one document was read (see the query below).

```
collection.find({"timestamp": datetime.datetime(2017, 1, 1, 00, 00, 00), "datapoint.x": 518, "datapoint.y": 219 }).count()
```

- Reading an image: it was searched for all data points matching a given timestamp. The timestamp value was chosen to be closer to the end of the table. Example of query below:

```
collection.find({"timestamp": datetime.datetime(2017, 1, 1, 8, 05, 00)}).count()
```

3.1.2.1 Document schema 1

The first proposed structure of a document is presented below:

```
document = {"timestamp": timestamp_value,
            "datapoint": {"x":x_value,
                          "y": y_value,
                          "value": rainfall_value
                          }
            }
```

Figure 3.1 First document schema

It can be noticed that the reading performance for an image is better than for a datapoint, even though when reading an image there are retrieved 1000 documents (or data points), leading to the following assumptions:

Assumption 5: Queries that access data in nested documents take longer than the queries that only refer to the data stored in the main document.
Assumption 6: Reading performance is better when consecutive documents are retrieved, in comparison to the case when random documents are retrieved. Nevertheless, in the presented case, the 100 retrieved data points are not that random, as all 100 can be found at equal distances (900 documents distance).

3.1.2.2 Document schema 2

Thus, the structure of a document is presented below:

```
document = {"timestamp": timestamp_value,
            "x":x_value,
            "y": y_value,
            "value": rainfall_value
            }
```

Figure 3.2 Second document schema

The reading time for an image stayed the same (1000 consecutive data points), whereas the reading time for 100 data points was better when compared to Schema 1 performances. This fact contradicts Assumption 6. Also, it can be noticed that reading 100 data points in this case was faster than when Schema 1 was used, fact that supports Assumption 5. Also, the performance prove to be better for the writing process. Therefore, these facts lead to the following conclusions:
Conclusion 8: Reading consecutive documents is not always faster than when reading random documents, as it depends if their data is organized in nested documents or not.
Conclusion 9: Writing documents that have nested documents hampers performance.
Assumption 8: An assumption drawn from the above conclusion can be that the writing performance depends on the number of data stored in the nested documents, as well as on the number of data stored in the main document.

3.1.2.3 Document schema 3

The 3rd schema which is supposed to be a mix between the 1st two, by storing only the coordinates in a nested document, obtaining a balance between the data stored in the nested document and in the main document. It is illustrated in Figure 3.3, below.

```
document = {"timestamp": timestamp_value,
            "value": rainfall_value,
            "coordinates":{ "x":x_value,
                            "y": y_value,
                            }
            }
```

Figure 3.3 Third database schema

It can be noticed that writing performance is worse when compared to previous 2 schemas performances, which supports Conclusion 9 and proves the Assumption 8. Also, the reading

time for 100 images proved to be better than in 1st case but slightly lower than in the 2nd case.

3.1.2.4 Document schema 4

The last proposed schema preserves the "coordinates" nested document and groups the other 2 values in another nested document, call "measurement". The schema is shown below:

```
document = {"measurement": {"timestamp": new_matrix.timestamp,
                            "value": new_matrix.val[j]
                            },
           "coordinates": {"x": new_matrix.x[j],
                           "y": new_matrix.y[j]
                          }
           }
```

Figure 3.4 Fourth database schema

The writing performance as well as the image reading got the worst performance from all 4 schemas, which proves Assumption 5. However, when it comes to reading random data points (100 in total) this schema had the second best time.

It can be noticed that in none of the cases, any of the stored values or a combination of those, was not used as an ID. The reason for that is that none of them are unique, as same timestamp is shared by all document corresponding to an image, same coordinates and precipitation value can be found in multiple images.

Thus, wrapping up the conclusions and the assumptions made in the entire section 3.1, the following conclusions can be made:
Conclusion 10: Writing performance in MongoDB is hampered when the documents inserted in a collection have nested documents, especially when all the value sin a document are structured entirely in nested documents.
Conclusion 11: Queries that address data stored in nested documents take longer than queries that address only data stored in main document.

These scenarios contradict the expectations we had when we proposed Schema 1, as it was shown that retrieving data points corresponding to one image (1000), might take longer or approximately equal times than when retrieving one document per matching case (100 in total), even though in the first case more documents are retrieved. However, only the scenario involving schema 4 seem to meet our initial expectations.
Furthermore, these conclusions show that we were wrong when we assumed that storing all data points afferent to an image as nested document would lead to faster reads, that is the best collection schema for reading an image. Actually, based on the above experiment it is expected to score quite worse when retrieving a data point per image. Since we aim at finding a schema that has good performance for both queries, we shall not test this case as the conclusions above do not seem to support it.

The measurements shown in Table 3.5 indicate the best reading performance for 100 data points (one for each image) is achieved with Schema 2, whereas the worst performance is given when using Schema 1. Another observation is that the reading time for 1 image (that is 1000 consecutive documents) stayed the same in all cases. Since Schema 2 scored the best for reading data points and performed the best for writing 100000, it shall be the one we use for the remaining 3 cases as well. The results are shown in the Table 3.6 below. For the first case, the values were copied from the Table 3.5.

Your schema/DB	**Scenario 1: 100.000 (1 collection)**	**Scenario 2: 1mln (1 collection)**	**Scenario 3: 10mln (5 collections)**	**Scenario 4: 100mln (12 collections)**
Write performance	Total time: 2.045 sec => 0.02 sec per image Written documents/sec: 48899.8	Total time: 18.661 sec for writing 1000 images => 0.018 sec per image Written documents/sec: 53587.7	Total time (writing to 5 collections): 3 min and 7.012 sec = 187,012 sec Writes/sec: 53472.5	
Read performance image	Total time: 0.047 sec Returned data points: 1000 Retrieved documents/sec: 21276.6	Total time: 0.428 sec Returned data points: 1000 Retrieved documents/sec: 2336.5	Total time: 0.887 sec Returned data points: 1000 Retrieved documents/sec: 1127.4	
Read performance datapoint	Total time: 0.044 sec Returned data points: 100 Retrieved documents/sec: 2272.8	Total time: 0.438 sec Returned data points: 1000 Retrieved documents/sec: 2283.1	Total time: 4.538 sec Returned data points: 10000 Retrieved documents/sec: 2203.8	

Table 3.6 The performance of MongoDB when storing one data point in a document

It can be seen that in the first scenario, the performance for reading 100 random data points is slightly better than querying an image (retrieving 1000 consecutive data points). Initially, the writing performance was the following: total time = 24.163 sec => 4138.6 writes/sec.

In Scenario 2, the performance for querying an image (retrieving 1000 consecutive data points) is better than when 1000 random data points were retrieved. An explanation for this can be that it is easier to retrieve consecutive documents than random distributed ones, as this operation is directly affected by the operating system's pagination mechanism, as the random data points might be stored in more pages than the consecutives ones and there are read more pages from the file system.
In addition, we can observe the writing time increased almost linearly (10 times), when compared to initial writing performance of Scenario 1, with a total time of 25.753 sec => 3762.9 writes/sec. At this point we asked if we can do better and a possible answer to this question might be represented by the usage of insert_many() operation, that inserts multiple documents with one instruction and allows specifying if the documents have to be inserted in the given order[4][5]. Thus, we created an array of documents, storing all the document corresponding to an image and then bulk write all these documents. In other words, we did not insert each data point, but each image, and the result was significantly better (14 times faster, shown with green in the above table), which actually is by far a more realistic approach when compared to the first one.
Thereafter, we repeated the process for Scenario 1, with the result also written in green noticing an improvement in execution time. For the last two scenarios, the bulk write approach is exclusively used.

In Scenario 3, we created one collection corresponding to one week of measurements, that resulted in 5 collections, with the last one having less data than the other 4 (80 images less = 80 000 data points or documents). By using more collections for the last 2 schemas, we aimed at improving reading performance for an image, as we already know that each query should contain the timestamp, we can obtain the corresponding collection by parsing the input timestamp, this way avoiding searching through all collection. As it was expected, reading an image was quite fast as the time doubled in comparison to Scenario 2 as only one table was addressed and the size of a table almost doubled. Also, searching for a datapoint involves searching through all collections and the total time is 10 times bigger than in scenario 2 (time varied linearly).

For scenario 4, writing to 12 collections failed during data was written for 8th collection, with a memory error while generating data. Thus, for writing almost ⅔ of the entire data it took approximately 20 minutes. Based on this fact, we can make the assumption that the total time would have been around 30 minutes, which makes it 10 times slower than in Scenario 3. A solution to improve the performance might be storing all data in the same collection and then use sharding [6]. However, this would not be enough without sufficient RAM memory, so a server with enough RAM memory would be the best option for performance, and it seems doable. Holding data on multiple physical machines would make sense only in the case when passive redundancy is desired, for high availability. This comes from the fact that it is clear it is needed a strategy for splitting the amount of data in a convenient way.

3.1.3 Improving performance

In the previous section are already presented some solutions that help in improving writing and reading performance simultaneously, that are:

1. Insert one image per operation using mongo API bulk write
2. Data stored in documents is not organized in embedded/nested documents
3. Given the idea behind the measured data, there was computed the period covered by the measurement and a database schema that presents data in a logical way (as it is divided by weeks/months) was adopted, for Scenario 3 and 4. Furthermore, this aims at improving reading performance, with a bit of overhead for writing process, but given the fact that on a database there are performed more read than writes, this is a tradeoff that can be accepted.

However, for each of the points presented above, there can be adopted further solutions, but necessarily better from all perspectives, as follows:

1. There can be inserted more images using a single operations, by grouping more images together (storing in an array) based on a given logic, for example all images afferent to the measurements done in a day. This approach is suitable when data that is already measured and stored somewhere needs to be stored in a MongoDB. Surely, this approach is not inline with real time measurement and in the experiments presented in the previous section, we took into account only the real-time measurement scenario.
2. Since there is few data to be stored in a document, as we presented it in the previous section, the lack of embedded documents does not represent a problem for data readability. On top of this, there could be chosen as document id (equivalent of primary key) a combination of timestamp and both coordinates, thus eliminated the necessity of storing a separate id. However, this would have made the reading queries harder, as the id would have been needed to be parsed and checked for each document, most probably after retrieving all of them, so to see the values for coordinates and timestamp, which would not be a feasible approach.. In the solution proposed in the previous section, we let the id to be generated automatically and it became the 5th field in a document.
3. For Scenario 3 we firstly wrote all data points in the same collection and it was approximately 10 times slower than in Scenario 2, which proves again that performance decreases linearly with the increasing number of data points. However, the writing time slightly decreased by almost 3.5 seconds, when compared to the solution which involves 5 collections, with a total time (writing to same collection) of 3 min and 3.503 sec (183,503 sec), which means 54495 writes/sec. The performance for reading an image was (a bit more than) 5 times slower, lasting for almost 4.4 seconds. The performance for reading 10000 data points was faster by 1.5 sec, with a total time of 4.39 sec, which indicates that searching in more collections than in a

single one does not cause an overhead. All in all, it can be said that splitting data in multiple collection was a better approach..

3.2 Cassandra

Apache Cassandra is a distributed database for managing large amounts of structured data across many commodity servers, while providing highly available service and no single point of failure. Apache Cassandra offers capabilities that relational databases and other NoSQL databases simply cannot match such as: continuous availability, linear scale performance, operational simplicity and easy data distribution across multiple data centers and cloud availability zones [1]. One of Cassandra's "claim-to-fames" is the incredible amount of write volume. It is able to handle such a large volume of writes by first writing to an in-memory data structure, then to an append-only log. These data-structures are then "flushed" to a more permanent and read-optimized file at a later time. However, Cassandra does not perform well when joining rows from different tables and it has limited support for aggregations with a single partition. Normally data aggregation is implemented on application level because the cost of doing it on database level is very high.

3.2.1 Keyspace (Schema) and Table Structure

In this assignment, we use single node cluster of Cassandra database in order to asses the performance of read and write query. The structure of the table is described in table 3.1. The primary keys of this table are timestamp, X, and Y. Since the first primary key is timestamp, therefore it will set up the partition of the data in database memory. Then, the value of the rain on each partition will be ordered by the second and the third primary key, X and Y respectively. Defining these columns as composite primary keys will improve the reading performance of Cassandra.

The structure of table given in Table 3.1 is similar for all cases (100000, 1 million, 10 million, and 100 million data points). However the schema or so-called keyspace of database for 10 and 100 million data points is slightly different compared to two other cases. For 10 million data points, we created 5 different tables containing weekly images per each that makes 2016000 data points stored on each table.

Moreover, for 100 million data points, we created 12 different tables which contains monthly images that makes the total number of 8640000 data points per each table. For the rest of the cases (100000 and 1 million), each keyspace contains 1 table only. We come up with this solution in order to decrease the computation time and cost when performing writing and reading query. In theory, if we store too many data points in a single table, the performance of reading and writing will be decreasing since Cassandra database has to check whether each

[1] https://academy.datastax.com/planet-cassandra/what-is-apache-cassandra

row meets the condition of the given query. If we store 100 million records in one table, the computation time and cost will be very heavy thus decreasing performance.

The structure of the keyspace on each case is described in figure 1, figure 2, figure 3, and figure 4 respectively.

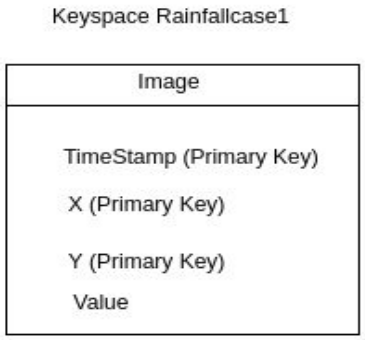

Figure 1. Cassandra keyspace for 100000 data points

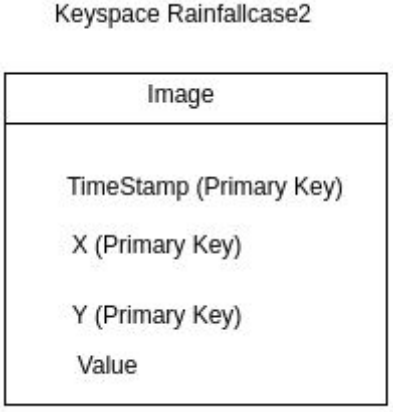

Figure 2. Cassandra keyspace for 1 million data points

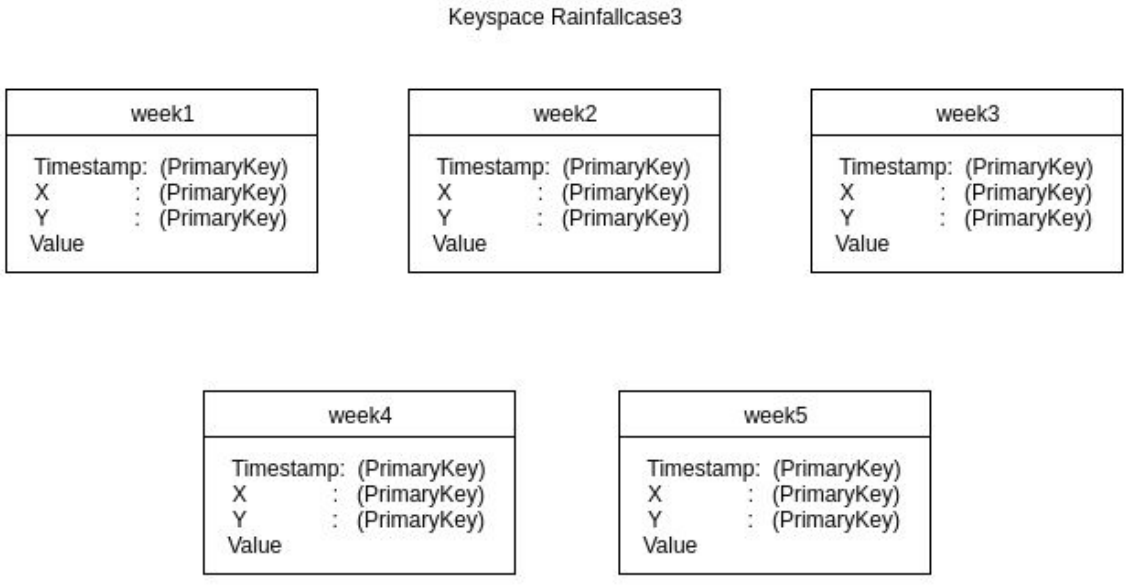

Figure 3. Cassandra keyspace for 10 million data points

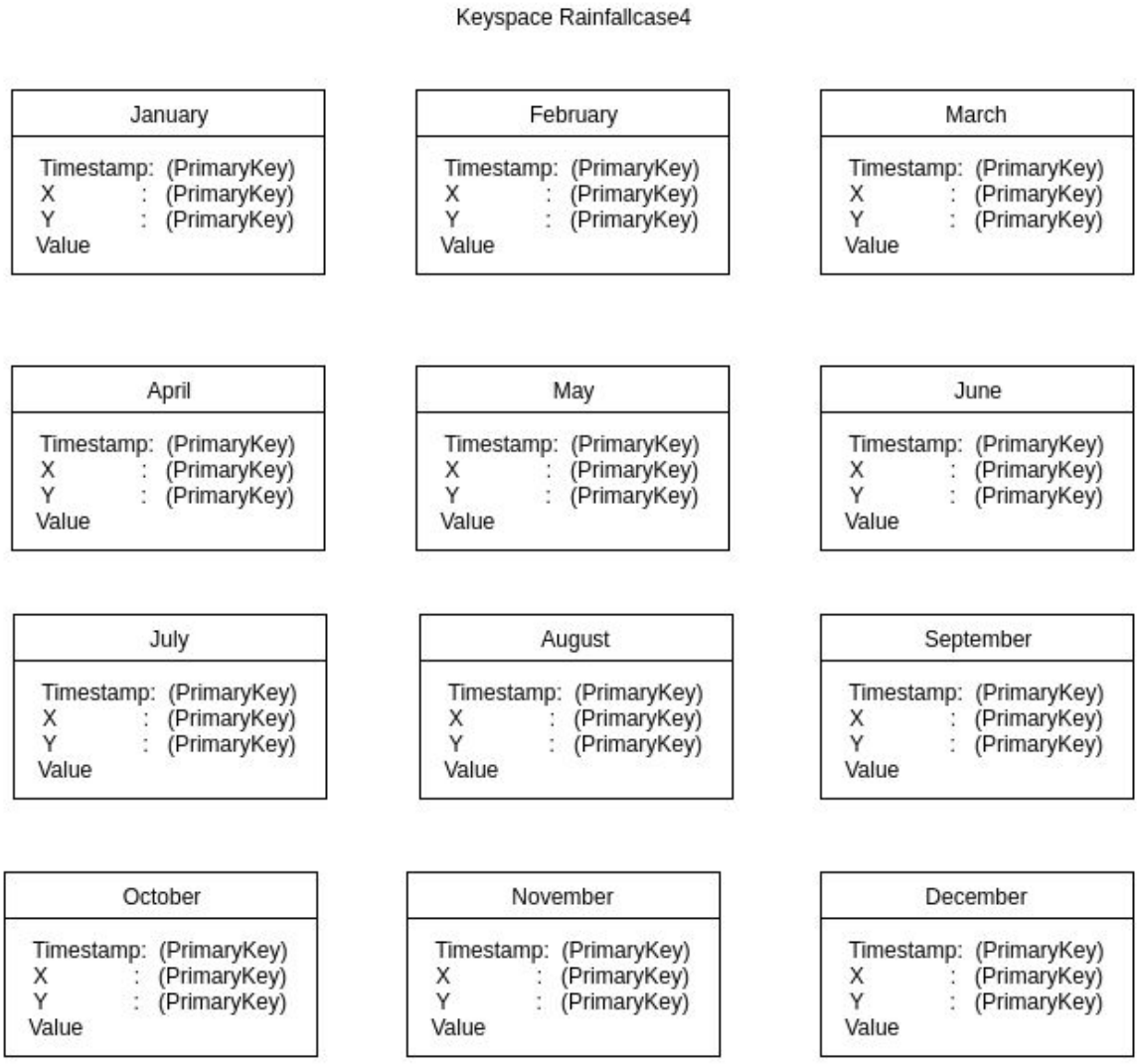

Figure 4. Cassandra keyspace for 100 million data points

3.2.2 Reading and Writing Images

The strategy of writing images to a database is almost similar on all cases. For the case of 100000 and 1 million data points, all data points will be written to a single table in the keyspace of each case. However, for the other cases such as 10 million, data will be written to a new table after reaching at 2016000 th iteration and at 8640000 th iteration for the case of 100 millions.

When reading data either single image or xy rain, we query the image and the xy rain from a single table which is applied on the case of 100000 and 1 million data points only. However, for 10 million data points, we execute 5 parallel queries (one per each table) and combine the final results on application level.

For the case of 100 million data points, the technique is similar to the case of 10 million in terms of reading the amount of rain based on xy position. However, when it comes to reading an image, the technique is slightly different. Firstly we filter the parameter for retrieving a single image on application level, which is in this case the timestamp. Since one timestamp represents one image, therefore we extract the name of the month from the timestamp which indicates the name of the table that stores the image. For instance, if we want to query an image on 1 January 2016 00:00:05, at first glance, we already know that this is a January data. Therefore, we extract the month as identifier and read the image from the January table. In this occasion, we don't have to select from all tables since the image is definitely in January table. Hence, the query will be faster.

3.2.3 Simulating reading and writing performance

The performance of Cassandra database are shown in table 3.7.

Schema/DB	**100.000 (1 table)**	**1mln (1 table)**	**10 mln (5 tables)**	**100 mln (12 tables)**
Write performance	2293 writes/sec Total Duration: 43.61 secs	2393 writes/sec Total Duration: 6.96 minutes	2231 writes/sec Total Duration: 1.245 hours	2222 writes/sec Total Duration: 12.50 hours
Read performance Image (Total Rows: 1000)	532481 reads/sec Total Duration: 0.0018 sec	242777 reads/sec Total Duration: 0.0041 sec	107665 reads/sec Total Duration: 0.0092 sec	588928 reads/sec Total Duration: 0.0016 sec
Read performance xy rain	9009 reads/sec Total retrieved Rows: 38 Total Duration: 0,0042 sec	15635 reads/sec Total retrieved Rows: 378 Total Duration: 0,024 sec	1168 reads/sec Total retrieved Rows:3774 Total Duration: 3,2 secs	898 reads/sec Total retrieved Rows: 37736 Total Duration: 42.02 secs

Table 3.7 The performance of Cassandra database on four different cases

The results of writing performance given in table 3.7 are almost similar on each case no matter how many data points need to be inserted in database. However, the reading performance for querying the image is decreasing depending on the total number of data

points in the keyspace except for the case of 100 million. Since we already implemented a strategy that filters the destination table on application level, the performance of the query in the case of 100 million data points is increasing. Thus, the performance is almost similar with the case of 100000 data points. This is such a good news since in may cases, the performance of a query normally linear with the size of table. The most realistic reason is because a database has to check each row of the corresponding table whether they meet the condition of query.

Moreover, the reading performance of xy rain is decreasing compared to reading performance of the image. The performance of querying an image is at least three times higher than the performance of querying xy rain in all cases. The more data points, the lower the reading performance when querying xy rain. Interestingly, the reading performance between 10 million and 100 million is almost equal. The strategy that we used by separating data points on different tables in the case of 100 million seem work effectively on Cassandra.

Considering the computation times of reading an image in all cases, the performances of the query in all cases are pretty good and acceptable since the computation time less than 1 second. Hence, we don't need any further improvement in this part. However, the computation time of reading xy rain in the case of 100 millions data points needs to be improved since it takes almost one minute. It happens because Cassandra database has to check each row of the table whether they meet the condition of query given x and y value. We will elaborate this more with a solution in section 3.2.4.

From the performance given in table 3.7, we can draw a conclusion that the more data we have in Cassandra database, the reading performance will be decreasing. Conversely, the writing performance seems constant no matter how many data points we store.

3.2.4 Improving reading performance

There are eight factors that influence the read performance of Cassandra: index interval, bloom filter false positive, consistency level, read repair chance, caching, compaction, data modelling, and clustering. Referring to the keyspace of Cassandra described in section 3.2.1, we already implemented several techniques namely data modelling and index interval. The data modelling can be seen from the structure of the table while the index interval is identified by three primary keys in a table (timestamp, X, and Y). Timestamp divides data points into partition which subsequently ordered by X and Y.

One out of seven remaining techniques that suitable to our Cassandra architecture is caching, since the other techniques belong to cassandra cluster which is not the case. There are two types of caching namely key and row cache. The key cache is a cache of the primary key index for a Cassandra table which is enabled by default while the row cache is similar to a

traditional cache like memcached which is disabled by default. The row cache holds the entire row in memory so reads can be satisfied without using disk.

With proper tuning, hit rates of 85% or better are possible with Cassandra, and each hit on a key cache can save one disk seek per SSTable. Row caching, when feasible, can save the system from performing any disk seeks at all when fetching a cached row. Then, whenever growth in the read load begins to impact your hit rates, we can add capacity to quickly restore optimal levels of caching [7].

In order to improve the reading performance when querying xy rain for 100 million data points, we implemented caching which can done by enabling the row cache when creating a table. Below syntax describes how to create row caching in cassandra table.

```
CREATE TABLE january (
   TimeStamp timestamp,
   X int,
   Y int,
   Amount float,
   PRIMARY KEY(TimeStamp, X, Y)
)
WITH caching = { 'keys' : 'ALL', 'rows_per_partition' : '1000' };
```

After we applied caching on all tables on keyspace rainfallcase4, the performance of reading xy rain is increasing by 22 %. The computation time before and after applying caching is shown in table 3.8.

Schema/DB	**100 mln (Before caching)**	**100 mln (After caching)**
Read performance xy rain	898 reads/sec Total retrieved Rows: 37736 Total Duration: 42.02 secs	1143 reads/sec Total retrieved Rows: 37736 Total Duration: 33 secs

Table 3.8 The performance of Cassandra database before and after caching tables

From table 3.8, we can see that caching decreases the computation time by 10 seconds faster than before caching. Even though the reading performance is not significantly increasing, however this technique seems to be a good way to start improving the performance.

3.3 MySQL

MySQL is an open-source relational database management system which developed, distributed, and supported by Oracle Corporation. The database structures are organized into physical files and its schema consists of the combination of tables and the relationship between them. Each table contains column and row and it has unique identifier called primary key.

In this assignment, we tried to simulate reading and writing performance of MySQL. However, we could not manage to finish 10 million and 100 million data points since it takes longer than what we expected. Instead, we just made a mathematical calculation in terms of writing performance based on the performance we got from 100.000 and 1 million data points. For the reading performance, we could not do the calculation since the schema between 1 million and 10 million is totally different and it's the same for 100 million data points as well.

3.3.1 Schema and Table Structure

The schema of MySQL database is similar to the keyspace of Cassandra. The structures of the tables are also similar as shown in figure 1, figure 2, figure 3, and figure 4 respectively but instead of using keyspace, we call the keyspace as schema in MySQL. We come up with this solution in relation to the facts that the more data points we have, the heavier the computation time as mentioned before in section Mongodb and Cassandra. So, we want to reduce the computation time, thus increasing the performance.

3.3.2 Reading and Writing Images

The reading and writing images techniques are similar to Cassandra database (Please refer to section 3.2.2). For the case of 10 and 100 million data points, joining rows from different tables (table week1 - week5 for 10 million and table january - december for 100 million), we execute them on application level.

3.3.3 Simulating and Writing Images and XY rain

he performance of Cassandra database are shown in table 3.9.

Schema/DB	**100.000 (1 table)**	**1mln (1 table)**	**10 mln (5 tables)**	**100 mln (12 tables)**
Write performance	20 writes/sec Total Duration:	19 writes/sec Total Duration:	19 writes/sec Total Duration:	19 writes/sec Total Duration:

	1.38 hours	14.3 hours	143 hours	1430 hours
Read performance Image (Total Rows: 1000)	199800 reads/sec Total Duration: 0.005005 sec	194818 reads/sec Total Duration: 0.005133 sec	*	*
Read performance xy rain	1788 reads/sec Total retrieved Rows: 38 Total Duration: 0.021248 sec	14698 reads/sec Total retrieved Rows: 378 Total Duration: 0.025717 sec	*	*

Table 3.9 The performance of MySQL database on four different cases

Note: * = Can't perform the simulation since data does not exist

Table 3.9 gives the description of reading and writing performance of MySQL database on several cases. It can be seen from the table that the writing performance seems constant. Even though we could not simulate the writing performance of 10 and 100 millions, however based on the calculation obtained from 100.000 and 1 million data points, the duration for 10 million is 143 hours and 1430 hours for 100 million data points. If we compare this the performance with MongoDB and Cassandra, we could say that it is the worst one.

Moreover, the performance of reading image from 100.000 and 1 million data points is more or less equal, it is around 0.005 second. However, the reading performance of xy rain is decreasing. In general, it takes 0.02 seconds to obtain xy rain no matter how many rows does the query returned. For 10 and 100 million data points, we could not simulate the reading performance either reading an image or xy rain since we don't have data points in our schema due to heavily computation time.

3.3.4 Improving writing performance

One solution to improve writing performance of MySQL database is by configuring the innodb variables properly inside the file mysqld.cnf. The goal is to stress the double write buffer so the load has to be write intensive. Below are the new configurations of mysqld.cnf which was suggested on internet[8].

```
innodb_read_io_threads=4
innodb_write_io_threads=8  #To stress the double write buffer
innodb_buffer_pool_size=20G
innodb_buffer_pool_load_at_startup=ON
innodb_log_file_size = 32M #Small log files, more page flush
innodb_log_files_in_group=2
innodb_file_per_table=1
innodb_log_buffer_size=8M
innodb_flush_method=O_DIRECT
innodb_flush_log_at_trx_commit=0
skip-innodb_doublewrite  #commented or not depending on test
```

After applying the new configuration above, the writing performance of MySQL is increasing significantly as shown in table 3.10

Schema/DB	**100.000 (1 table)**	**1mln (1 table)**	**10 mln (5 tables)**	**100 mln (12 tables)**
Write performance	7246 writes/sec Total Duration: 13.80 seconds	7246 writes/sec Total Duration: 2.3 minutes	7246 writes/sec Total Duration: 23 minutes	7246 writes/sec Total Duration: 3.8 hours

Table 3.10 The writing performance of MySQL database on four different cases after configuring innodb properly

References

[1] https://scool.larc.nasa.gov/lesson_plans/RainfallRatesWaterVolume.pdf
[2] https://docs.mongodb.com/manual/core/document/
[3] https://docs.mongodb.com/manual/core/databases-and-collections/
[4] https://docs.mongodb.com/manual/reference/method/db.collection.insertMany/
[5] http://api.mongodb.com/python/current/api/pymongo/collection.html
[6] https://docs.mongodb.com/manual/sharding/
[7] http://shareitexploreit.blogspot.nl/2012/09/cassandra-read-performance.html
[8]https://www.percona.com/blog/2014/05/23/improve-innodb-performance-write-bound-loads/