Information System
Lab 1

T. Back s3218147

R. Bwana s3232352

15th January, 2018

1 Introduction

For this assignment we implemented 2 different storage systems to test and compare between them regarding the reading and writing performance.

We chose to implement the system using a Linux-based file system (ext4) and MongoDB run in a docker container on the same machine. While the solution may not show the best optimized results for each implementation, we felt it was sufficient to compare the two solutions.

The code to perform all the necessary data generation, insertion, and reads was written in- and run on- Python v3.6.

In order to clarify the understanding of our code and the results we start by stating the assumptions we made when answering the assignment.

2 Assumptions

As the assignment is not clear in every point, the following assumptions have been made.

- One image represents the rain of the complete Netherlands in opposite to just a tile or cutout.
- We use a sparsity of 10% rain per image, leading to approximately 4000 images with 100.000.000 data points (at 550x500 data points per image).

- Instead of saving specific timestamp data we label data from 0 to 4000 with the understanding that they are in chronological order.
- The query gets optimized to return as fast as possible all data points at one location (one pixel in the image) for all images.
- An image is not represented by an advanced format with compression, but simply by a bitmap
- The used database do not store the data in memory, but might have indexes in memory.
- The used format for comparison is JSON.

3 Schema

We treated the image and the XY coordinates as separate data structures in order to optimize the queries for each type of read request.

For images the structure of the data we created was a bitmap-type image where 0s and 1s represent the absence and presence of rain respectively on that pixel of the 550x500 image. These were all stored in a single array where the array index represents the position in the image. In MongoDB the images were saved as:

$$\{"image" : image\_number, "type" : "blob", "data" : the\_blob\_array\}$$

In the File system images were saved as:

The image number was used as the file name and was saved with the .blob extension. Inside the file was a simple array of the 1s and 0s as generated from the code.

For the XY coordinates we converted the 550x500 matrix of 1s and 0s to a linear array indexed from 0 to 275,000. In MongoDB the XY coordinates were saved as:

$$\{"image" : image\_number, "type" : "json", "data" : the\_json\_array\}$$

In the File System the XY coordinates were saved as:

The image number was used as the file name and was saved with the .json extension. Inside the file was a json array with the structure: $"index" : value$.

4 Results

This section shows the results for MongoDB and the File System for inserting and reading the data in two different ways.

Inserting is measured by inserting both data schemes into the database (image and XY coordinate).

The databases are emptied before every run and the full data is stored in the databases before read performance is measured.

The testing data set was created once and therefore is comparable between the different systems.

4.1 MongoDB

For MongoDB, we arrived at the following results:

MongoDB	**100.000**	**1.000.000**	**10.000.000**	**100.000.000**
Write Performance	70.000 inserts/sec	80.000 inserts/sec	80.000 inserts/sec	80.000 inserts/sec
Read Performance Image	1.000.000 reads/sec	3.300.000 reads/sec	30.000.000 reads/sec	10.000.000 reads/sec
Read Performance XY	550.000 reads/sec	1.600.000 reads/sec	150.000 reads/sec	1.500 reads/sec (extrapolated)

Table 1: MongoDB Performance

4.2 File System

For the file system, we arrived at the following results:

File System	**100.000**	**1.000.000**	**10.000.000**	**100.000.000**
Write Performance	10.000.000 inserts/sec	10.000.000 inserts/sec	303.000 inserts/sec	280.000 inserts/sec
Read Performance Image	5.000.000 reads/sec	2.800.000 reads/sec	3.000.000 reads/sec	3.000.000 reads/sec
Read Performance XY	150.000 reads/sec	160.000 reads/sec	160.000 reads/sec	155.000 reads/sec

Table 2: File System Performance

4.3 Comparison and Deliberations

Inserting data is into the file system is very cheap. It is up to 13x faster than inserting the same amount of data into MongoDB. This may be due to the fact that the generated data is already in file form. Where we to generate and store the data at the same time this may lead to different results but then we would not be sure whether the performance is due to the random generation of data or the actual storing of the information.

To our surprise, reading the data out of the file system is slower than writing it (for small chunks of data). We expect that a disk cache is first filled up during write, which leads to these results.

But when it comes to reading, MongoDB outperforms the File System when reading a bigger amount of data. The extra time taken during the insert makes up for it - if the data is read multiple times.

The read performance of the image is always better than the read performance of the XY coordinate. As we are querying the complete data, this was expected. The results could be different if the objective of the query would be different.

5 Improvements

To optimize the queries even further, several optimization's can be made. Those might only have a better result if more data is used, where parallelizing can improve the speed.

5.1 In-Memory

The used databases store all their data on disk. Loading data from disk is always slower than having the data already available in-memory.

MongoDB and most other databases have the indexes for the data in memory, so that it can find the data faster.

Redis even goes a step further by storing all data only in-memory. But since memory is limited, is not appropriate for the amount of data we are dealing with as the amount of data points goes beyond 10.000.000 or the sparsity is lower than 10%. In that case, we would recommend a hybrid architecture, where the latest data is in Redis and can be retrieved fast. Older or historical data should be stored in another database that can handle big amounts of data like MongoDB or Cassandra. Since format of the data format is consistent, a traditional SQL database can also be used.

5.2 Schema

Assuming that the data is very sparse, it might be interesting to only store the positions of the 1s instead of the complete sequence of 1s or 0s. If for example the place located at (1, 10) has rain, only the number 510 (1x500+10) is stored.

The idea is to reduce the amount of data that is saved. It can be seen as a compression, since the client can recreate the original data.

A reduced amount of data leads to faster data transfer between the server and client, which can improve the overall performance depending on the time taken to de-compress vs. use the raw data.

5.3 Sharding

Sharding can help with distributing the data over multiple machines and splitting it up into mangeable chunks. The used sharding key (Partition Key in Cassandra) would be the first level of the Schema. Currently that would be *image: image_number* field.

5.4 Client-side vs Server-side

Another alternative would be to use less complex queries and rather return larger amounts of information to the client which can they filter and extract the information it needs from the returned data. Such an approach would only be reasonable however when we can be certain that the client side computer can handle the more complex task.