Assignment 4 - Report

Daniel Bin Schmid, Ermias Habtegabr, Kristiyan Sakalyan

December 2022

1 Azure Deployment and Testing

1.1 After you have finished with the Azure tutorial, measure the time it takes for the Assignment 3 query to run on Azure. What do you notice?

- There is an overhead for creating the container instances compared to running it on the local machine. The overhead increases the time before the containers can start processing the query.
- A single worker runs on a single CPU, hence has less computation power compared to typical multi-core CPUs.

1.2 Go to the Azure monitoring panel for your containers: explain what is the bottleneck that increases query execution time. Include screenshots if needed.

The network download rate is only 10 mbit/s for the worker. Since the task's query execution time strongly depends on the download rate, the low download rate becomes a bottleneck. A system with a faster network finishes the task significantly faster.

1.3 Let's get faster: Pre-upload the data partitions and fileList inside Azure blob storage. Adapt your solution to read from there. What is the speedup you observe? How is it explained?

To be expected would be an increased download rate compared to the 10mbit/s mentioned above. The faster download rate would be explained by the fact, that the Microsoft Azure backbone can be used which may optimize the network between the containers and the blob storage. However, after adapting the solution to read from the blob storage instead of downloading files via http requests, the download time is not reduced. This may be explained by a bad network setup inside of Microsoft Azure. A solution to resolve this problem would be

to upload the data partitions and fileList to Azure filesystem instead of blob storage. With that it would be possible to mount the filesystem inside of the container instances and would provide fast write and reading from files in scale of a local SSD.

2 Managing Shared State - Design Questions

2.1 Give a brief description of your solution.

Our solution works in the following way. First, the coordinator is spawned and waits for incoming connections. As soon as one worker connects, the coordinator assigns a task. There are two tasks that we have to perform and in order to differentiate between them we simply prepend 0 or 1 in the beginning of the message that is sent to the workers. **0** corresponds to the counting and calculating the partial aggregates. **1** corresponds to merging the partial aggregates and writing the results to files. In the end, when all the workers have done tasks 0 and 1 we close the connection between the coordinator and the workers and do the final merging of all intermediate results. After doing this the results are written into a CSV file.

2.2 What was the query execution time in Azure? Include screenshots, logs if needed.

Shared file storage	Environment	Number of sub-partitions	Task	Time(s)
Local	Local	1	0	22.223
Local	Local	10	0	24.465
Local	Local	200	0	23.275
Local	Local	1	1	7.928
Local	Local	10	1	1.151
Local	Local	200	1	1.153
Local	Local	1	**TOTAL**	30.157
Local	Local	10	**TOTAL**	25.73
Local	Local	200	**TOTAL**	25.73
Azure blobs	Local	1	0	129.531
Azure blobs	Local	10	0	144.801
Azure blobs	Local	200	0	255.237
Azure blobs	Local	1	1	47.706
Azure blobs	Local	10	1	32.71
Azure blobs	Local	200	1	173.743
Azure blobs	Local	1	**TOTAL**	177.718
Azure blobs	Local	10	**TOTAL**	173.758
Azure blobs	Local	200	**TOTAL**	460.184
Azure blobs	Azure	10	0	154.282
Azure blobs	Azure	10	1	70.946
Azure blobs	Azure	10	**TOTAL**	226.671

***The results were obtained by running the same program multiple times and averaging the obtained execution time.**

We can clearly see from the results that having only one sub-partition increases the execution time of the merging task significantly. Furthermore, we can conclude that an increase of 10 to 200 partitions does not improve the performance locally and when ran with Azure Blob Storage it actually worsens it. This is due to the fact that there is a bigger overhead for reading and writing to shared files. With a higher number of sub-partitions, the number of files to be transferred via network when using the Azure Blob Storage increases significantly.

When running the query entirely on Azure, it can be observed, that the execution time is similar to when running the query on the local machine with Azure Blobs as shared files. Please note, that benchmarks with the local machine as environment is run with 25 workers whereas the benchmark with Azure is run with 4 workers. Thus the comparison is not entirely fair. However, it can be observed, that running the query on Azure is slightly worse than when running it on a local machine. It would be interesting to compute more benchmarks with the same number of workers for better direct comparison.

2.3 Which partitioning method did you choose for the calculation of the partial aggregates? Why?

We used hash-based range-partitioning because with range-based partitioning only on a non-random primary key, hotspots would occur which would in turn lead to an in-balanced work distribution among workers. The hash-based range-partitioning scheme can be split into two phases:

1. First, a random *unsigned long* hash is computed from the domain. This is done by using *boost::hash* [1]. Since we found, that the first digit of the resulting hash is not uniformly distributed, we discard the first digit.

2. Second, we apply range-partitioning based on the digits of the resulting hash. Instead of interpreting the *unsigned long* hash as a single number, every digit of the hash is considered separately. The most significant digit is considered first.

2.4 What number of subpartitions did you choose, and why? If you choose to create 100 subpartitions for each partition, is it a good choice? Is there a scalability limit?

After experimenting and measuring the performance of our program, we decided to use 10 subpartitions. If the number of subpartitions is too small the merge sort complexity is significantly increased and results in worse performance. You can also see this in the results described in the table above in Subsection 2.2. On the other hand, if we can increase the number of subpartitions to 200, there is a network overhead when the program is ran on the cloud.

Creating 100 subpartitions for each file wouldn't be an optimal. Although it would reduce the mergesort complexity and execution time, it would result in higher network overhead when ran on the cloud. This can be proven using the results shown in the table that we previously mentioned as well.

When increasing the number of partitions, we would increase the I/O and network overhead and this could get too expensive compared to the benefit of reducing the merge-sort complexity. The scalability limit cannot be defined that simple because it could vary depending on the partitioning strategy that is used.

2.5 How does the coordinator know which partitions to send to the workers for the merging phase?

The coordinator knows exactly how many subpartitions there are. This allows him to send the index of each subpartition to a worker. Then the worker would merge all the subpartitions that have this index. Essentially, the coordinator sends a message that starts with 1 and identifies that the merge task has begun. It further appends a subpartition index that indentifies, which subpartitions

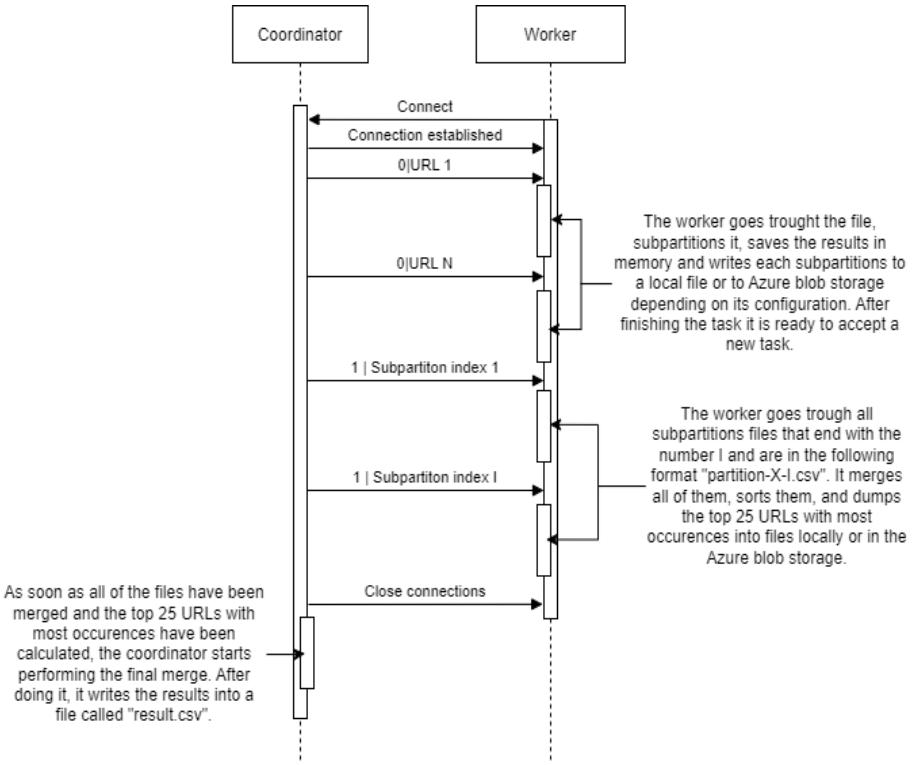
[1] `https://www.boost.org/doc/libs/1_63_0/doc/html/hash.html`

have to merged by the worker. For example, if we have 10 subpartitions and we send "1|5" to the worker, he would merge all files that have the following format "parition-X.5.csv".

2.6 How do workers differentiate from the two tasks they have to perform (partial aggregation, merging subpartitions)?

The worker differentiate between the two different tasks in the following way. The coordinator add a number in the beginning of each message that triggers different functions in the worker. In our case, if the message starts with 0, it means that the worker must calculate the partial aggregates and write them to appropriate files. If the message starts with 1, it means that the worker should merge the partial aggregates and write the top 25 URLs that it has found into other files.

2.7 Give a brief sequence diagram to show how the coordinator and workers communicate. Add some details about the communication format.



3 Notes

3.1 Testing

We have defined 3 tests that allow you to test our implementation. To do so you have to first download all the necessary files. To do so you can simply run:

```
python3 download_files.py
```

The consistency test checks if we get the right result using 10 workers. To execute it, run:

```
./testCorrectness.sh
```

The resilience tests if we get the right results even if we kill a worker, which is currently working. To execute it, run:

```
./testResilience.sh
```

The elasticity tests checks if we get the right results when new workers are spawned. Firstly, it spawns 10 workers, waits a second, and spawns 6 new workers. To execute it, run:

```
./testElasticity.sh
```

3.2 Configuration

Everything with regards to configuration can be found in the **config.h** header file.

To configure the number of initial partitions you can adjust the following line:

```
const int nInitialPartitions = 100;
```

To configure the number of subpartitions you can adjust the following line:

```
static const int nAggregates = 25;
```

To run the solution using the local file storage, you need to first configure your credentials in the following part of the code:

```
namespace credentials {
    // Name of the storage account
    static const std::string accountName = "";
    // Access token for Azure storage
    static const std::string accountToken = "";
}
```

Then you have to also adjust the following line:

```
const bool io_type = IO_TYPE::AZURE_BLOB;
```

to:

```
const bool io_type = IO_TYPE::LOCAL;
```