UPPSALA
UNIVERSITET

Large Datasets for Scientific Applications

Final Project: Finding GC content with Spark

Anna Hillver, Erik Zhivkoplias,
Jiahao Lu, Rinyarat Buakhao

Group 8

June 10, 2019

1 Background

The initial data chosen for this project was the data generated for the 1000 Genomes Project (CRAM files), which is a project where researchers aim to sequence at least 1000 human genomes from different populations (1; 2). The 1000 Genomes Project goal is to increase the understanding of the human genome and create a catalog of the variations in the human genome. Studying the composition of human genomes will give the researchers valuable insights on how genetics contribute to human health and risks of developing diseases.

Our project goal was to create a scalable data processing application that can read genomic files and then calculate the GC-content (guanine-cytosine content) of the input sequence.

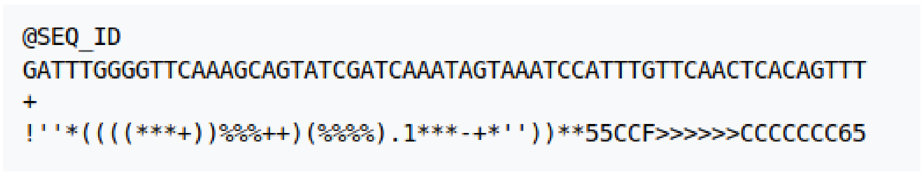
GC-content represents the percentage of cytosine and guanine (out of four different bases) in the DNA sequence. However, as we learned along with the project the overall GC-content for human genomes does not vary greatly and has the mean around 41% (3). That is why we decided to change dataset and to work with Human Microbiome Project data (FASTQ files), which is the genomic collection of all the microorganisms living in association with the human body (with 14 157 samples up to date).

Interestingly, that GC-content metric correlates with many important features of prokaryotic genomes: genome size(4), amino acid usage patterns(5), and the environment species live in(6). Therefore, microorganisms that share the same GC-content could participate in the same metabolical processes in the human body and could share the same function. Knowing new players in metabolic cross-talk will help us to unravel the origins of key metabolites and better understand how human microbiota make an impact on human health.

2 Data format

The data from Human Microbiome Project is available in the FASTQ format. FASTQ(7) is the file format that stores raw read sequencing data. FASTQ is a textfile containing sequences that are not aligned to a reference genome, and also contains some information about the sequence such as name of the sequences and the quality scores. The actual sequence is represented at every fourth row in the file, which makes it easy to read and filter using PySpark, see an example in Figure 1 below.

First, when we started coding the application we used a FASTQ file with the size 436 MB and then when we had everything working we loaded another FASTQ files with the size 4.56 GB and 10.52 GB to see how the application works with larger files and test the application scalability.



```
@SEQ_ID
GATTTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATCCATTTGTTCAACTCACAGTTT
+
!''*((((***+))%%%++)(%%%%).1***-+*''))**55CCF>>>>>>CCCCCCC65
```

Figure 1: FASTQ file format. The second row contains information about sequence.

3 Computational experiments

This section describes a virtual machine (VM) settings, the solution to calculate GC-content, the plan to experiment horizontal scalability, results and discussion.

3.1 System settings

A virtual machine with "ssc.large" flavour was created and added to "spark-cluster-client" security group for it to work correctly with Spark. A volume of 100 GB was also created and attached to the virtual machine. On the virtual machine, hadoop-2.7.7 and spark-2.4.2 were installed. All following solutions are implemented through Jupyter Notebook on the virtual machine.

3.2 Solution

The purpose of this project is to calculate GC-content among the four nitrogenous bases, A C G T, in FASTQ files containing DNA sequences. To reach this goal we developed a simple program for GC-content calculation using Spark and Python. Since the HDFS has been installed on the driver VM, the testing files were uploaded on the HDFS cluster that running alongside the Spark cluster. The algorithm to implement the program is as follows:

- After setting the spark_session and the spark_context, an RDD was created by reading dataset file from HDFS.
- RDD with its element indices was zipped so every RDD element was transformed to value-key pair (string, index), then switched their position to be key-value (index, string).
- Every fourth string in the RDD (the one that contains the DNA sequence) was stored, everything else is filtered out.
- Each DNA string was split into single letters, then the appearances of A, C, G, T were counted (Map-Reduce operation). Out of these counts the GC-content was calculated.
- Last but not least, time performance was measured.

The GC-content can be calculated using the following formula:

$$GC - content = \frac{G + C}{A + C + G + T} \times 100\%$$

This means that the script calculated both the number of G:s and the number of C:s as well as the total number of nucleotides to get the GC-content in percentage.

3.3 Scalability Experiments

Scalability in data management is defined as the ability of a system to efficiently handle the growing amount of workload. It is categorized into horizontal scalability and vertical scalability. Horizontal scalability or scaling out means that the system performs by adding more resources. For example, adding more storage nodes or by replication of system services. Vertical scalability or scaling up means the increase in the actual capacity of the available resources. For example, CPU, memory or network bandwidth. In simple words, one said:

" Scaling horizontally - Thousands of minions will do the work together for you. Scaling vertically - One big hulk will do all the work for you."(8)

The focus of this project is to evaluate the horizontal scaling performance of the analysis application. The application runs on the shared spark cluster, and there are 10 workers with 8 cores each. To obtain this, the spark session needs to be configured. The application performance will be evaluated by three different experiments.

3.3.1 Exeriment 1.

Use `spark.executor.cores`(9; 10) property to set the number of cores to use on each executor. The experiment will be performed by setting the number of cores on each executor to 1, 2, 3, 4, 5, 6, 7, 8 (max core in a worker), respectively. E.g.

```
.config("spark.executor.cores", 1)
```

Then test with two files with different sizes, this is 4.56 GB and 10.52 GB, the idea is to compare how well the horizontal scaling performance of the application is, how long the time is taken and the speedup between a large and a small dataset. Therefore we monitor how much resources the application is using on the spark master UI, as well as the UI of our application (on localhost:4040).

3.3.2 Exeriment 2.

Use `spark.cores.max`(9; 10) property to control the maximum amount of cores to request for the application from across the cluster (not from each machine). The experiment will be performed by setting the maximum number of cores to 1, 2, 4, 8, 16, 32, 64, respectively. E.g.

```
.config("spark.cores.max", 1)
```

and use the same test files as in experiment 1. Then monitor the performance.

3.3.3 Exeriment 3.

For the last experiment, use both properties `spark.executor.cores` and `spark.cores.max` to control the number of cores on each executor and the maximum amount of cores to request for the application from across the cluster, respectively. The experiment will be performed by setting `spark.executor.cores` to a fixed size, let's say 4, and specify the maximum number of cores `spark.cores.max` to 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, respectively. E.g.

```
.config("spark.cores.max", 4)
.config("spark.executor.cores", 4)
```

The expected result for this setting when monitoring should show only one executor on one worker being used, 4 cores on the executor. And the following setting

```
.config("spark.cores.max", 12)
.config("spark.executor.cores", 4)
```

should show only 3 workers being used, one executor per worker, 4 cores per each executor.

3.4 Results and discussion

The result of our scalability experiments is presented in tables and plots below. As described above, the application was run on the shared Spark cluster with 10 workers, 8 cores each. For each experiment, the application performance is shown on two datasets with different sizes (4.56 GB and 10.52 GB). There were 80 cores available on the cluster when the experiments were performed.

3.4.1 Experiment 1

For the first experiment, the `spark.executor.cores` property was set to 1, 2, 3, . . . , 8 respectively. When `spark.executor.cores` was explicitly set, each executor was specified to use the same number of cores that set. The number of executors of the application was launched across all workers and on the same workers if those workers had enough cores and memory. As monitored, the system automatically provided the number of executors per application which depends on the size of the dataset and on the available number of cores on the workers. This is shown clearly when a really small dataset was used ($<$ 1 GB). First, the system spread executors to all the available workers and then on some workers which had enough cores left. So it was possible that one worker had many executors. The following was two examples showing how the system partitioned the work during experimentation.

The examples experimented by setting the number of cores on each executor to 1, where 10 workers were available and using two small datasets.

```
Example 1: 2 2 1 2 1 1 1 2 1 1 (10 workers were used)
Example 2: 1 1 1 0 1 1 1 0 1 0 (only 7 workers were used)
```

Table 1. presents the result of the experiment using the dataset of size 4.56 GB, consisting of the time-consuming for each number of cores on each executor and the speedup.

Number of cores on each executor	**Time in HH:MM:SS**	**Time in seconds**	**Speedup**
1	0:03:21.494919	201.495	1.000
2	0:03:14.312858	194.313	1.037
3	0:04:01.765694	241.766	0.833
4	0:03:10.618193	190.618	1.057
5	0:03:41.810071	221.810	0.908
6	0:03:42.816633	222.817	0.904
7	0:05:04.204308	304.204	0.662
8	0:05:00.568450	300.568	0.670

Table 1: Experiment1 for 4.56GB file

Looking at the time-consuming, it was not much difference between each number of cores. Except when the number of cores on each executor was set to 7 or 8, in these cases the time-consuming was a little bit higher than the others.

Table 2. presents the result of the experiment using the dataset of size 10.52 GB (about 2 times larger than the first test dataset).

Number of cores on each executor	**Time in HH:MM:SS**	**Time in seconds**	**Speedup**
1	0:06:11.485800	371.486	1.000
2	0:05:46.956581	346.957	1.071
3	0:06:23.284633	383.285	0.969
4	0:05:44.645486	344.645	1.078
5	0:06:42.344938	402.345	0.923
6	0:06:26.555039	386.555	0.961
7	0:05:54.900093	354.900	1.047
8	0:05:48.935241	348.935	1.065

Table 2: Experiment1 for 10.52GB file

The time-consuming for this dataset was not much difference between each number of cores on each executor. For this size of dataset, the system consumed the highest number of cores that was available on the cluster (80 cores). The Speedup for both datasets were plotted in Figure 2.

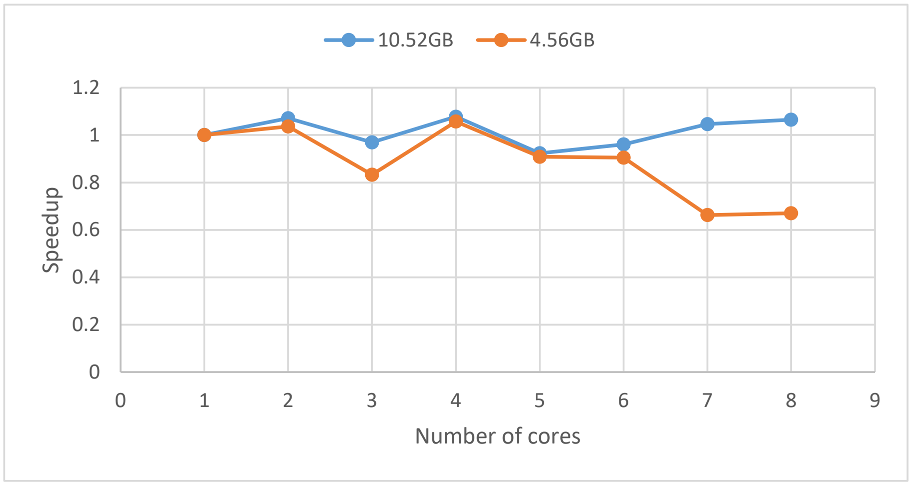


Figure 2: Speedup of 4.56 GB and 10.52 GB

The graph for 4.56 GB was slowing down after 6 number of cores on each executor while the graph for 10.52 GB was growing almost horizontally. The speedup of both datasets was approximately close to each other, and the best speedup for both datasets was when 4 cores on each executor were used.

By observing the performance, this scalability experiment indicates that the application has ability to scale horizontally across the workers. However, this experiment setting consumed all available cores on the cluster.

3.4.2 Experiment 2

The second scalability experiment was planned to use `spark.cores.max` property. The maximum number of cores for the application across the cluster was set to 1, 2, 4, 8, 16, 32, 64 respectively. As monitored, the system did not spread the executors in the same way as in the experiment 1, and it seemed to provide only one executor for each worker. That is, when the maximum amount of cores to request for the application from across the cluster set to 1, 2, 4, 8 respectively, the system used only one executor on one worker by increasing the number of cores on that executor from 1, 2, 4, and 8 respectively. But when the maximum amount of cores to request for the application set to 16, 32, 64 respectively, the executors were spread across workers. For example:

```
max 16 cores: 0 8 8 0 0 0 0 0 0 0
(2 workers, one executor each, and 8 cores each executor)

max 32 cores: 8 8 0 8 0 8 0 0 0 0
(4 workers, one executor each, and 8 cores each executor)
```

For the maximum amount of cores set 64, the expected result was

```
max 64 cores: 8 8 8 0 8 8 8 8 0 8
(8 workers, one executor each, and 8 cores each executor)
```

but the actual result was

```
max 64 cores: 8 8 8 0 8 8 8 4 4 8
(9 workers, one executor each, and 8 cores each executor,
except 2 workers that was 4 cores on each executor)
```

Another observation is that when the dataset was small, the system did not use the cores up to the maximum number of cores set, it provided only the actual cores the application needed. Also as observed, this experiment 2 was slower than the experiment 1, comparing time-consuming for the same test datasets.

The time-consuming and the speedup for dataset 4.56 GB is shown in Table 3. and for dataset 10.52 GB in Table 4.

Maximum Number of cores	**Time in HH:MM:SS**	**Time in seconds**	**Speedup**
1	1:03:42.015692	3822.016	1.000
2	0:46:00.861681	2760.862	1.384
4	0:24:32.347101	1472.347	2.596
8	0:22:21.865033	1341.865	2.848
16	0:08:46.299956	526.300	7.262
32	0:05:19.575565	319.576	11.960
64	0:04:56.512024	296.512	12.890

Table 3: Experiment 2 for 4.56GB file

Maximum Number of cores	**Time in HH:MM:SS**	**Time in seconds**	**Speedup**
1	3:29:30.731665	12570.732	1.000
2	1:46:40.402333	6400.402	1.964
4	0:57:30.999221	3450.999	3.643
8	0:51:52.577576	3112.578	4.039
16	0:17:38.061006	1058.061	11.881
32	0:10:30.357869	630.358	19.942
64	0:05:55.378041	355.378	35.373

Table 4: Experiment 2 for 10.52GB file

The time-consuming of both datasets were reduced when the maximum number of cores was increased, resulting in the increased speedup. The speedup for both datasets is plotted in Figure 3.

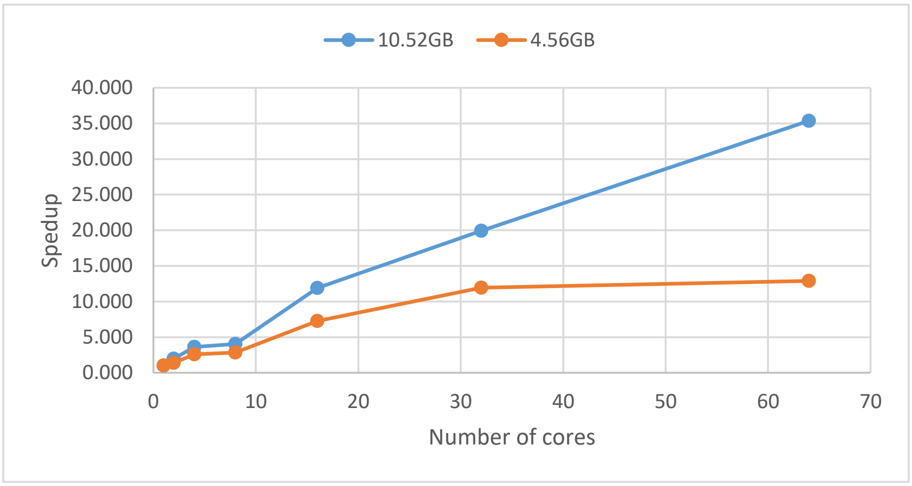


Figure 3: Show the speedup of 4,56 GB and 10.52 GB

The reason why speedup for the 4.56 GB file was barely increasing when we used more than 32 cores is that when the performance of the application had reached a certain point, it started to slow down even a higher number of cores had been used. One of the reasons was due to the mismatch between the size of the dataset and the number of cores. In this case, the dataset probably might be a little bit too small for using 64 cores. The speedup for the 10.52 GB file was increasing almost linearly.

Overall, the scalability experiment 2 indicates that the application has the ability to scale horizontally across workers.

3.4.3 Experiment 3

For the third experiment, both `spark.executor.cores` and `spark.cores.max` properties were used to control the number of cores on each executor and the maximum amount of cores to request for the application from the cluster, respectively. According to the description in section 3.3.3, the `spark.executor.cores` property was set to a fixed size, i.e. 4, throughout the experiment. And `spark.cores.max` was set to 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, respectively.

As for the observation, the experiment resulted as expected. When the maximum amount of cores was set to 4, one executor (4 cores) on one worker was used, at the same time when the maximum amount of cores was set to 8, 12, 16, . . . , 40, the number of workers was increased to 2, 3, 4, . . . , 10, respectively. The number of executor per worker remained the same which was 1 with 4 cores. The examples from the result.

```
max 4 cores: 0 0 4 0 0 0 0 0 0 0
(1 worker, 1 executor with 4 cores)

max 16 cores: 4 0 4 4 0 4 0 0 0 0
```

```
    (4 workers, 1 executor with 4 cores each)

    max 40 cores: 4 4 4 4 4 4 4 4 4 4
    (10 workers, 1 executor with 4 cores each)
```

Therefore it indicates that the application was scaling out.

The result of time-consuming and speedup for the datasets 4.56 GB and 10.52 GB were presented in Table 5. and 6., respectively.

Number of workers	**Maximum Number of cores**	**Time in HH:MM:SS**	**Time in seconds**	**Speedup**
1	4	0:24:31.181749	1471.182	1.000
2	8	0:13:24.537183	804.537	1.829
3	12	0:08:24.384099	504.384	2.917
4	16	0:06:15.363797	375.364	3.919
5	20	0:05:50.955132	350.955	4.192
6	24	0:05:20.323463	320.323	4.593
7	28	0:03:58.681712	238.682	6.164
8	32	0:03:49.992353	229.992	6.397
9	36	0:03:32.346293	212.346	6.928

Table 5: 4 cores on each executor, one executor each worker. File size 4.56 GB

Number of workers	**Maximum Number of cores**	**Time in HH:MM:SS**	**Time in seconds**	**Speedup**
1	4	0:57:22.478353	3442.478	1.000
2	8	0:30:12.076985	1812.077	1.900
3	12	0:17:53.939024	1073.939	3.205
4	16	0:14:11.430407	851.430	4.043
5	20	0:11:19.267304	679.267	5.068
6	24	0:09:49.155799	589.156	5.843
7	28	0:08:50.795151	530.795	6.486
8	32	0:08:13.167836	493.168	6.980
9	36	0:06:26.000937	386.001	8.918
10	40	0:06:26.642415	386.642	8.904

Table 6: 4 cores on each executor, one executor each worker. File size 10.52 GB

The third experiment showed better time performance than the second experiment when compare runs with the same maximum number of cores used. The speedup for both datasets is plotted in Figure 4.

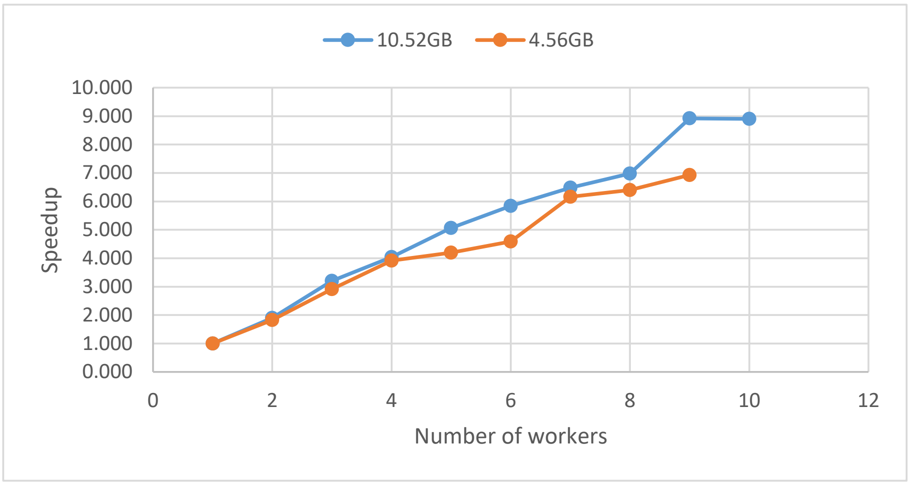


Figure 4: The speedup of 4,56 GB and 10.52 GB

Looking at the graph for the 4.56 GB file, we conclude that the speedup grows almost ideally for 4 - 16 cores (1 - 4 workers), and then starts to slow down a little bit for the higher number of cores. For the 10.52 GB file, the speedup grows almost ideally for 4 - 24 cores (1 - 6 workers), and then starts to slow down after 36 cores.

Nonetheless, the scalability experiment 3 also indicates that the application has the ability to scale horizontally across the workers.

Note that the property `spark.deploy.spreadOut`(9) had also been experimented and set to `False` while experimenting horizontal scalability. According to the observation, the difference between setting it to `False` or `True` was hard to discover. We then left it as default.

Moreover, the property like `spark.dynamicAllocation.maxExecutors` (10) had also been experimented. The property is used to set upper bound for the number of executors. The combination of `spark.dynamicAllocation.maxExecutors` and `spark.executor.cores` can specify the maximum number of executors and the number of cores on each executor for the application, e.g.

```
.config("spark.dynamicAllocation.maxExecutors", 4)
.config("spark.executor.cores", 5)
```

As monitored, 4 workers had been used, 1 executor on each worker and 5 cores on each executor.

There is also the property like `spark.dynamicAllocation.minExecutors` (10) which is used to set a lower bound for the number of executors.

3.4.4 GC-content

The appearances of four nitrogenous bases in the dataset 4.56 GB were [('C', 1714440), ('T', 1752065), ('A', 1808603), ('G', 1032089)] and the GC-content was 43.55%. For the dataset 10.52 GB, the appearances of four nitrogenous bases were [('C', 15860848), ('T', 12763095), ('A', 17572447), ('G', 6420838)] and the GC-content was 42.35%.

4 Discussion and future work

4.1 Discussion

The objective of the project was to implement an application to calculate GC-content of DNA in the large dataset and then test the horizontally scalability of the application.

Three scalability experiments were designed for the experimentation. The results of the experiments indicate that the application has ability to scale horizontally across the workers. The first experiment showed good time performance but it consumed all available resources on the cluster, which is not a problem only if just one application runs at a time. However, to allow multiple applications running, the maximum number of resources that each application will use should be set. While doing the second and the third experiment we controlled the maximum number of resources for the application. However, the second experiment could not control the number of cores on each executor while the third experiment could which is an advantage if there are other applications that need to be run in parallel on the same cluster.

In conclusion, in order to achieve the best performance, Spark properties need to be chosen wisely to meet the needs and have to be configured correctly and separately for each application.

4.2 Future work

Below is a list of things which can be used to improve the application in the future.

- Experiment more Spark properties that might be useful for the project, e.g.
  - `spark.dynamicAllocation.initialExecutors` (10). Initial number of executors to run if dynamic allocation is enabled. The default is the value of `spark.dynamicAllocation.minExecutors`.
  - `spark.dynamicAllocation.executorAllocationRatio` (10). This setting allows to set a ratio that will be used to reduce the number of executors w.r.t. full parallelism.
  - etc.
- Test the vertical scalability.
- Build simple ML model that could cluster human microbiota based on their GC-content.

References

[1] Service, R. F. ”GENE SEQUENCING: The Race For The $1000 Genome”. Science, vol 311, no. 5767, 2006, pp. 1544-1546. American Association For The Advancement Of Science (AAAS), doi:10.1126/science.311.5767.1544.

[2] Clarke, Laura et al. ”The 1000 Genomes Project: Data Management And Community Access”. Nature Methods, vol 9, no. 5, 2012, pp. 459-462. Springer Nature, doi:10.1038/nmeth.1974.

[3] Almpanis, Apostolos et al. ”Correlation Between Bacterial G+C Content, Genome Size And The G+C Content Of Associated Plasmids And Bacteriophages”. Microbial Genomics, vol 4, no. 4, 2018. Microbiology Society, doi:10.1099/mgen.0.000168.

[4] Almpanis, Apostolos et al. ”Correlation Between Bacterial G+C Content, Genome Size And The G+C Content Of Associated Plasmids And Bacteriophages”. Microbial Genomics, vol 4, no. 4, 2018. Microbiology Society, doi:10.1099/mgen.0.000168.

[5] Zhou, Hui-Qi et al. ”Analysis Of The Relationship Between Genomic GC Content And Patterns Of Base Usage, Codon Usage And Amino Acid Usage In Prokaryotes: Similar GC Content Adopts Similar Compositional Frequencies Regardless Of The Phylogenetic Lineages”. Plos ONE, vol 9, no. 9, 2014, p. e107319. Public Library Of Science (Plos), doi:10.1371/journal.pone.0107319.

[6] Reichenberger, Erin R. et al. ”Prokaryotic Nucleotide Composition Is Shaped By Both Phylogeny And The Environment”. Genome Biology And Evolution, vol 7, no. 5, 2015, pp. 1380-1389. Oxford University Press (OUP), doi:10.1093/gbe/evv063.

[7] Cock, Peter J. A. et al. ”The Sanger FASTQ File Format For Sequences With Quality Scores, And The Solexa/Illumina FASTQ Variants”. Nucleic Acids Research, vol 38, no. 6, 2009, pp. 1767-1771. Oxford University Press (OUP), doi:10.1093/nar/gkp1137.

[8] ”Difference Between Scaling Horizontally And Vertically For Databases”. Stack Overflow, 2019, https://stackoverflow.com/questions/11707879/difference-between-scaling-horizontally-and-vertically-for-databases. Accessed 9 June 2019.

[9] ”Spark Standalone Mode - Spark 2.4.2 Documentation”. Spark.Apache.Org, 2019, https://spark.apache.org/docs/2.4.2/spark-standalone.html. Accessed 9 June 2019.

[10] ”Configuration - Spark 2.4.2 Documentation”. Spark.Apache.Org, 2019, https://spark.apache.org/docs/2.4.2/configuration.html. Accessed 9 June 2019.