UPPSALA
UNIVERSITET

UPTEC IT 21030
Examensarbete 30 hp
September 2021

Genomics in the Cloud

David Östlund

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Genomics in the Cloud

David Östlund

The continued cost reduction for sequencing genomics data is causing an exponential growth in the amount of data available. Moving both storage and calculation of this data to the cloud has been a common trend, but the way to do it is not always obvious. This report compares three different alternatives for doing ad-hoc queries in a cloud based setting: two solutions using data lakes and one solution using a relational database hosted in the cloud. The data lake solutions proved to be easy to set up and fully functional for querying genomics data. The relational database was more complicated to set up, but the queries were more time efficient and more cost efficient when performing more than 1200 queries per month on at least 100GB of data. To make the cloud computing possible for genomics data it had to be transformed into a file format supported by the cloud providers. For this purpose the Parquet file format was chosen, tested, and proven to work well.

Handledare: Max Block
Ämnesgranskare: Salman Toor
Examinator: Lars-Åke Nordén
UPTEC IT 21030
Tryckt av: Reprocentralen ITC

Contents

Acronyms

1KGP 100 Genomes Project. 2, 9

Amazon RDS Relational Database Service. 4, 15, 20, 21, 24, 29

Amazon S3 Amazon Simple Storage Service. 20

AWS Amazon Web Services. 4, 13, 16, 20, 21, 23–25, 35, 37

BAM Binary Alignment/MAP. 6–10

GATK GenomeAnalysisToolkig. 7

GCP Google Cloud Platform. 4, 7, 14, 16, 18, 22, 23, 25, 35, 37

GRC Genome Reference Consortium. 6

NGS next-generation sequencing. 6, 13

SAM Sequence Alignment/MAP. 6, 8, 9

SNPs single nucleotide polymorpisms. 5, 6, 13

VCF Variant Call Format. 7–11, 15–17, 22, 24, 25, 29, 31, 38

YOPD Young Onset Parkinson's Disease. 3

1 Introduction

The study of genomics is a field of research which has always generated large amounts of data in need of storage and analyzing. During the 20th century computation power and data storage capacity was generally not a problem for genomics researchers. This was because the increase in available data generally followed the increase in CPU power [Ste10]. With the turn of the century more advanced sequencing techniques were developed which drastically changed this situation. In 2003 the first mapping of the entire human DNA was completed, the project was called the human genome project [JM01] and took a total of 13 years to complete. Only five years later the 100 Genomes Project (1KGP) [The12] was announced with the goal to establish a detailed catalog of variations in the human DNA by sequencing over 1000 human genomes. This project generated an unprecedented amount of genomics data, but that was not all. During the years until its completion in 2012 the cost of sequencing a human genome decreased from $1 million to $1 thousand [Dav10]. With the 1KGP entirely available to the public, and with the continuously decreasing sequencing cost [NIH21] the amount of data in need of storage and processing has been increasing rapidly over the last decade. To handle this situation laboratories need to heavily invest in computer hardware able to handle the computations and skilled informatics support to configure everything. This is expensive as well as wasteful since the computers are only periodically used at max capacity.

Processing and storing genomics data in the cloud could be a solution to this problem. Today several companies offer the possibility to rent digital storage space for a fee based on the amount of data stored and the frequency at which it is accessed. Some of these companies also have the capability of running computations on the stored data through their data centres. This is known as cloud storage and cloud computing and may reduce the overall cost since you only pay for what you use. Cloud services allow for quick and easy scaling of data capacity based on the current needs which can help researchers focus on their research instead of infrastructure related topics.

This study was done from the perspective of a research group focusing on research about Young Onset Parkinson's Disease (YOPD). They are interested in finding out what genes have changed in YOPD patients and how they affect the body. For this purpose they have gathered large amounts of genomes from YOPD patients which will be compared with the goal of finding which gene mutations are causing of the disease. This leads to a large amount of ad-hoc queries that search through all available data for variations in certain genes. Today the data is stored in house, and in file formats not suited for effective analyses. Querying this data is therefore both complicated, expensive and time consuming.

The goal with this study was to find more effective ways of doing ad-hoc queries on genomics data in the cloud. The word ad-hoc is latin for "to this" and describes queries designed for a "particular purpose", in contrast to predefined queries that have the same output every time. The researchers using these queries are searching for brand new correlations between data containing a large amounts of parameters. Because the ad-hoc queries are constantly changing with the need of te user it becomes hard to enhance their processing beforehand. Instead the systems need to flexibly handle the new queries as they appear.

To reach this goal two different problems had to be solved. The first problem was regarding the data received from the client which was not in a file format supported by the cloud providers. A new file format had to be found which was both well suited for cloud computing and genomics data storage. The second problem was to find what kind of cloud storage solutions are effective for the chosen situation. Today cloud providers offer two main solutions which were both explored. The first is a simple to set up data lake solution where the data is able to be queried as soon as it is uploaded to the cloud storage, with minor preprocessing. The second option is to use a cloud hosted relational database. In this case a schema has to be created, and the data processed to follow said schema when uploaded to the cloud. When the processing is completed the database provides powerful and efficient tools for querying the data.

Comparisons between two file formats recommended by the cloud providers, Par-

quet and AVRO, were made to find which one was best suited for genomics data. The different cloud solution were then populated with the transformed data and tested for time and cost efficiency. First two data lake solutions were compared with each other, one using *Amazon Athena* and the other using *Google BigQuery*. Secondly the overall data lake performance was compared to a relational database developed and implemented with *PostgreSQL* using *Amazon's* Relational Database Service (Amazon RDS). These systems were populated with public genomics data and tested with queries trying to find information related to certain genes.

1.1 Delimitations

Because of time and cost restraints some delimitation had to be taken into account to make completion feasible.

The study used a smaller amount of data compared to what is generally used in real life scenarios. This delimitation has been made because of data availability and the costs connected with running the queries on larger amounts of data. The tests also assume that queries searching for all variations in certain genes make up the largest part of executed queries per 30 days.

This study will focus on implementing the solution on Amazon Web Services (AWS) and Google Cloud Platform (GCP). This is partly because these are two of the largest cloud services available today, but it is also because of availability. Limiting my work to two cloud services won't effect the study's credibility because many of the existing cloud services have similar capacities even though they do things in different ways. AWS and GCP were chosen because they are the two cloud services of which I have the most experience with and technical support.

2 Theory

This section will describe the theory related to this study.

2.1 From DNA to Digital Information

Before describing the processing of genomics information it is important to have a basic understanding of what our genome is and how DNA is converted from physical to digital information.

2.1.1 DNA, Genome and Variants

DNA is the hereditary material in humans and almost all other living organisms. A DNA molecule consists of two polynucleotide strands, each one made up of a long series of connected nucleotides. Each nucleotide contains a phosphate a sugar and a nitrogenous base. The phosphates and sugars are the same for each nucleotide, but the bases come in four different versions: adenine (A), guanine (G), cytosine (C) and thymine (T). The different combinations of these chemical bases in the DNA strand makes up the information which describes what each cell should do to make the organism what it is. Another important aspect of these bases is that they can attach to each other, but only in specific ways: A to T, G to C and vice versa. The second DNA strand, also known as the complementary strand, can therefore be seen as a mirrored version of the first one.

A DNA strand can be divided into a number of genes, where each gene encodes the synthesis of a gene product, either RNA or a protein. The genome is an organisms total set of genes. The human genome consists of around 20 000 genes made up of 3 billion nucleotide bases, 99.9% of which are exactly the same in all humans [Nat].

Because such a large percentage of the genome is the same for all humans the interesting parts to study becomes the areas which differ, these are called variants. There are three types of variants: single nucleotide polymorpisms (SNPs), insertions and

deletions. SNPs are the simplest type of variant and refers to the change in a single nucleotide, for example an A might have been switched to a T. Insertions refer to when one or several bases have been inserted into the gene, increasing the amount of bases. For example ATTA changing to ATCGCGCGTA is an insertion. Deletions refer to when one or several bases have been removed from the gene, decreasing the amount of bases. For example changing CCTTTTTTCC to CCCC is a deletion.

To make genome comparison possible the Genome Reference Consortium (GRC) released human reference genomes derived from several different humans. The latest reference, GRCh38, was released on 17th December 2013 but several researchers and tools still use the previous reference, GRCh37, released on 27th February 2009.

2.1.2 Sequencing of DNA

The process of turning a DNA string into human readable information is known as sequencing. Before the turn of the century DNA sequencing was a slow, tedious and expensive process. Over the last decade several new sequencing methods, collectively known as next-generation sequencing (NGS), have revolutionized the genomics sequencing possibilities lowering both the sequencing time and cost significantly. Even though the sequencing processes of the different systems vary a lot they all start with dividing the DNA strings into many smaller pieces known as short reads. They then go on to process each short read, and finally outputs the information digitally into the FASTQ format (Section 2.4.1).

At this state the output only shows a large amount of base sequences and how well they have been sequenced, but there is no information on how they correlate to each other. This is solved through the use of sequence alignment tools which use the reference genome to align the short reads into one cohesive genome. At this state we have all the information from the short reads together with new important information like where and how well every read matches into the human genome. The information is first outputted into the Sequence Alignment/MAP (SAM) format (Section 2.4.1), and then often converted into the Binary Alignment/MAP (BAM) format

to save space.

In many situations the only interesting parts of the BAM file are the variants. The process of comparing the BAM file to a reference genome and picking out variants is known as variant calling. There are several different variant calling tools available today, all with different strengths and weaknesses, but they emit their results into files of the Variant Call Format (VCF) (Section 2.4.2). The input data for this study is in the VCF file format.

2.2 Genome Workflows

The previously mentioned process of taking genomics information from the physical sequencer to the final digital output contains a lot of different steps. Depending on what the final output is there are also several different paths available. These different paths are generally formatted into something known as workflows or pipelines. Different companies structure workflows in different ways but in general they string together several different tools and tasks to be used on the input data to give the promised output. For example the GenomeAnalysisToolkig (GATK) have several different workflows for handling of genomics information which they have summarized in their "Best Practices Workflows"[1]. One such workflow is the 5$ genome analysis pipeline which takes an unmapped BAM file, which closely resembles a FASTQ file, as input and, with the help of different tools, outputs a CRAM (Compressed BAM) file together with some data. It is called the 5$ genome analysis pipeline because it has been optimized to run on a full genome on GCP for under 5$. Many of GATKs pipelines, and others similar to them, are optimized for running on cloud services which shows that the cloud is a necessary future for genomics.

[1] https://gatk.broadinstitute.org/hc/en-us/articles/360035894711-About-the-GATK-Best-Practices (2021-08-20)

2.3 Data Lakes and Relational Databases

Data lakes refers to a data repository that has the capacity to store large amounts of structured, semi-structured and unstructured data. Just like a real lake data flows into the data lake from different streams and are kept in their natural form. No data is turned away form the data lake. In its leaf level the data is stored in an untransformed state, but transformations and schemas may then be applied to the data to make make analysis possible. The use of data lakes offers an agile and flat data storage structure fit for large amounts of data.

In a relational database all the incoming data has to be processed to follow the structure of the applied schema. The schema groups the data into several tables with relations between them that describe how their data relate to each other. The goal is to create a declarative method of specifying how the data is structured and accessed. This makes it clear for the user what the database contains and what they can get from it. Today there are several powerful relational database systems that provide algorithms and tools for effective data storage and retrieval.

The two forms of data storage both offer advantages in different categories. The data lakes provides flexibility for data access, storage and analysis with little needed preprocessing. The relational databases gives the opportunity to make data access and structure more effective for the chosen schema.

2.4 File Formats

This section describes all file formats mentioned in this report. Section 2.4.1 explains formats related to the initial part of the DNA sequencing process, FASTQ, SAM and BAM. Section 2.4.2 describes the VCF format which is the format of the input files to this study. Section 2.4.3 and 2.4.4 describes two formats which VCF files can be converted into to give it certain properties relevant for this study.

2.4.1 FASTQ, SAM and BAM

FASTQ [2] is a text-based file format used for storing the short reads from the DNA sequencing, and their corresponding quality score. The FASTQ files have four lines for each short read:

Line 1 Begins with a '@' character followed by a sequence identifier and an optional description.

Line 2 Holds the raw sequence letters.

Line 3 Begins with a '+' character and is optionally followed by the same sequence identifier again.

Line 4 Holds an encoded quality score.

SAM [3] is a TAB-delimited text format which takes the information from the FASTQ file and adds information about where each short read is positioned in the genome. BAM contains the same information as SAM but compressed into a machine readable binary version. The full genome of one human compressed into the BAM format takes about 80GB. Both SAM and BAM are two of the many important outcomes from the 1KGP project. Before 1KGP there was no standard in how sequenced genomes were digitally stored. This made it extremely hard to compare sequences from different research institutes. With the SAM format standard this has become much easier.

2.4.2 VCF

VCF [4] is a human-readable format in which the variant calling programs output their data. The VCF format has a standardized base structure where column 1 through 7 always shows the same kind of information independent of which tool was used to create the file (Table 1). The number of key-value pairs in the INFO field and their

[2] https://emea.support.illumina.com/bulletins/2016/04/fastq-files-explained.html (2021-05-10)
[3] https://samtools.github.io/hts-specs/SAMv1.pdf (2021-05-10)
[4] https://samtools.github.io/hts-specs/VCFv4.2.pdf (2021-05-10)

meaning is however completely up to the the creator, as long as they are correctly documented. The same goes for the SAMPLE columns and their corresponding description in the FORMAT column. There are even programs that takes VCF files as inputs, analyses them and emits their results to new columns in the inputted file. This means that even though VCF is the standard Variant Calling format you never know exactly what information the file will contain until you open it.

The format can be compared to CSV except that it has specifically defined columns. It is not optimized for size compression or data accessibility and one VCF file containing the variants of a full human genome takes about 2GB. When stored VCF files are often zipped with tools like gzip, having to zip and unzip files whenever they are used is not efficient for data processing. The VCF files also do not have any metadata to improve how its data is accessed making it a slow and tedious process. In summary the VCF file format has well defined columns for storing genomics data, but it is inefficient for any kind of processing.

Table 1: **VCF file data fields**. The VCF file has nine required fields. The first seven are fixed and describe the variant. The INFO field is en extensible list. The FORMAT and SAMPLE fields are optional and can be used to describe samples.

Column	Name	Description
1	CHROM	The name on the sequence on which the variation is being called
2	POS	The 1-based position of the variation on the given sequence
3	ID	The identifier of the variation, or "." if unknown.
4	REF	The reference base at the given position
5	ALT	The list of alternative bases at the given position.
6	QUAL	The same quality score as in the BAM file.
7	FILTER	A flag indicating which of a given set of filters the variation has passed
8	INFO	An extensible list of key-value pairs describing the variation.
9	FORMAT	An optional extensible list of fields for describing samples.
+	SAMPLEs	For each sample described in the file, values are given for the fields listed in FORMAT

2.4.3 Parquet

Parquet [5] is a machine-readable columnar based big-data storage file format created by Apache Spark. In contrast to the regular, human readable, row based formats like CSV, JSON and of course VCF, *Parquet* stores the data one column at a time. This makes it impossible for humans to read at first glance, but it opens up opportunities for both storage compression and boosts in query speed in certain scenarios.

When it comes to storage compression a key importance is the fact that the file knows that all the values that come after each other have the same data type. First of all this allows for the data to align better in memory which saves space. Secondly this allows for the use of compression schemas which further reduce the space taken.

When it comes to data accessibility *Parquet* files get a boost in query speed when querying for information in columns. When "regular" row based files need to access all the data in a single column they have to iterate through the entire file to get the information. In *Parquet* this is easily solved because the entire column is already lined up and can be accessed without touching anything else.

2.4.4 AVRO

AVRO [6] is a machine-readable row-oriented data serialization format created within Apache's Hadoop project. *AVRO* uses JSON to define its data-types and protocols which makes it easy to use if one has worked with JSON. It does however serialize the data into a compact binary format which is what makes it machine-readable but saves a lot of storage space.

One of *AVRO*'s selling points is the ability to define schemas for the data. These schemas gives the files features similar to relational database tables with requirements and instructions for how the interpret the data. This will for example make sure that only data of the correct type is written to each column protecting the file

[5] http://parquet.apache.org/documentation/latest/ (2021-05-10)
[6] http://avro.apache.org/docs/current/ (2021-05-10)

from becoming corrupted. The documentation fields also help in explaining how the schema should be used.

2.5 Client: NysnoBioMetriX

The client for this study is the research institution NysnoBioMetriX through the software engineering consultants at Data Ductus. NysnoBioMetriX is a non-profit organization which, in partnership with Aligning Science Across Parkinson's (ASAP) and The Michael J. Fox Foundation (MJFF) work to launch a novel, patent-driven data repository for Parkinson's data. The goal is to investigate relations between three known genetic causes of Parkinson's Disease to better understand it and further the work towards finding a cure.

3 Related Work

In his article from 2010 Lincoln D. Stein makes a strong case for cloud computing to become the future for genome informatics [Ste10]. He starts by describing the relation between the cost of generating genomics data and the cost for hard disk storage. These have followed each other ever since the beginning of the 1990s, but since the advent of NGS technologies this had stopped being the case. If this continues then storage capacity will become the new limiting factor in genome informatics. He then argues that the capacities and flexibilities of cloud based systems like AWS is a possible solution. He does also mention that there are still problems to be solved, like how to transfer large amounts of data effectively.

One project that has taken this advice is Rainbow, a cloud-based software package with tools to assist the users with automation of SNPs detection in whole-genome sequencing data [ZPS+13]. It uses a set of tools from AWS to make this possible. The problem with transferring large amounts of data is solved by using the *Amazon Import* service which allows the user to ship their hard drives to *Amazon* via *FedEx*. *Amazon* then directly uploads the data to *Amazon Simple Storage Service*. The data is transferred to *Amazon's EC2* (Elastic Cloud Compute) upon which several virtual machines can be fired to analyze the data in parallel. Their tests show that using *Amazon* Cloud for SNPs detection is both fast and cost effective, with the added benefit that no local infrastructure is necessary.

SNPs detection is one of the previously mentioned variant calling processes and could serve as input data for the tests done in this study. Because the preprocessing needed for this study has already proven to be effective in the cloud the assumption can be made that the cloud is a solution for other parts of the genomics field as well.

Another successfully attempt to move parts of the genomics field to the cloud is project Bionimbus. It is an open-source cloud-based infrastructure for managing, analyzing and sharing large amounts of genomics and phenotypic data [HGP+14]. Bionimbus has helped execute several multi-pipeline research projects in a contained and safe cloudbased environment.

The American National Cancer Institute have also created a cloud based platform called The ISB Cancer Genomics Cloud (ISB-CGC) [RML+17]. The goal with the project was to find a way to give users the possibility to access the entirety of The National Cancer Institute Cancer Genome Atlas (TCGA) and TARGETs (Therapeutically Applicable Research to Generate Effective Treatments) data through the cloud. The platform does many different things, but one of those is query-based analysis using SQL. Similarly to this project ISB-CGC uses GCPs Cloud Storage and *BigQuery* for this purpose.

Another project used the cloud to help in the calculation of the reciprocal smallest distance algorithm (RSD) [WKF+10]. The algorithm is used to find orthologs, genes in different species that evolved from a common ancestral gene, and is prone to scaling problems. They used *Amazon's EC2* and were able to run simulations in the same way as would be done on a local Linux compute farm for a manageable cost. From this study they drew the conclusion that the cloud represents an ideal platform for comparative genomics.

These projects refer to the increase in time and cost for processing genomics data because of its exponential growth [HGP+14] to be an immediate problem. They also look to solve this with the help of cloud based solutions. In the same way this study also tries to find effective cloud solutions, but for analyzing genomics variant calling data.

4 Data and Tools

This section describes the data and tools used to set up and compare the three systems: *Google Cloud Platform's BigQuery, Amazon Web Services' Athena* and the relational database designed in this study which was hosted on Amazon RDS. The systems will be compared by having them find all variants connected to certain genes, where a gene is represented by which chromosome it exists on and its start and end positions on that chromosome.

4.1 Input Data for Query Tests

The data used in this study was gathered from The Personal Genome Project which is a world spanning collaboration dedicated to creating public genome, health and trait data. It was founded at Harvard University in 2005 as a pilot project and has today been joined by Universities from Canada, United Kingdom Austria and China [Per].

As input data the following six files were downloaded from their public database [7] making up a total of 13GB of VCF files.:

File 1 56001801068863.filtered.snp.vcf

File 2 60820188474283.snp.vcf

File 3 ca8e4c6b-ba8a-4eda-8ee6-1e9b581d79d6.vcf

File 4 Genome-VCF-199709-1528098_bam-22Dec2017.vcf

File 5 Helix_ExomePlus_variants_1550345278.vcf

File 6 NB72462M.vcf

As previously mentioned VCF files always have the same information in the first seven columns, the other columns can look different depending on how they were

[7] https://my.pgp-hms.org/public_genetic_data?utf8=%E2%9C%93&data_type=other (2021-05-05)

created (Table 1). The optimal input data for this study would have the same information in the INFO, FORMAT and SAMPLE columns, this was however very hard to find. Instead these file were chosen because they all only had one SAMPLE column. The INFO, FORMAT and SAMPLE columns do not hold relevant information for the queries used in this study and could therefore be formatted as general strings.

When running the *BigQuery* tests it was noted that some files needed significantly higher percentage of data to be scanned even though they had the same data compositions as the others. Because of this the files file 3, file 4 and file 5 were left out in the rerun of the *BigQuery* tests.

4.2 Choosing File Format for VCF Conversion

The VCF format is not suited for direct processing because of its simplicity, it is also not supported by either *Amazon Athena* or *Google BigQuery*. When choosing which file format to convert the VCF files into there are two main factors that need to be taken into account, size compression and data accessibility. File formats use different methods to optimize themselves depending on how they are meant to be used. In the goal to find an efficient method of doing ad-hoc queries on genomics data two file formats were compared: *Parquet* and *AVRO*.

Both the *Parquet* and *AVRO* formats are supported and recommended by the GCP and AWS platforms. In the end the structure of the data and in what way it is queried will be the deciding factor for which file format is best suited. Because of its columnar format *Parquet* is more optimized for tall data with few columns and many rows. It is also good for queries accessing data spread around the file. *AVRO* on the other hand is more optimized for wider data and when there is a large need for full-scans or queries accessing many rows in tandem.

A regular variant file has about 10-15 columns and 5 million rows making it very tall. The data needed to be accessed in one query can also be spread out in the entire file. This makes *Parquet* the better file format for this study.

4.3 VCF to Parquet with Tomatula

To convert the VCF files to *Parquet* format the Tomatula tool was used [BFvK+17]. The tool consists of a set of python scripts for handling VCF conversions and Allele frequency calling. In this study only the VCF conversion scripts were used. Converting supported file formats to *Parquet* is as simple as calling the write.parquet() function from the python library pyspark. Because the VCF format is not supported by Apache Spark it first needed to be converted from VCF to the JSON.

To make the script compatible with this study some changes had to be made to the Tomatula scripts. JSON, in contrast to VCF, has data types connected to each column. When converting from VCF to JSON these types have to be found and added in a correct way. For the first seven columns the Tomatula script only checks whether the values are integers or string. The values in the QUAL column are often decimal numbers and were therefore labeled as strings. To solve this a float check was added.

To get as much as possible out of *Parquet's* compression schemas Tomatula converts the INFO and FORMAT columns into dictionaries which makes Apache Spark see them as embedded tables. This part was disabled, instead labeling them as strings, to ensure that the different files were outputted with the same types in each column. The CHROM columns also followed different formatting styles. In some it had values like chr1, chr2, chrX and so on, while others had 1, 2, X and so on. Measures were taken such that, when converted to JSON, all CHROM columns followed the second formatting style.

4.4 The Test Query

The tests in this study focuses on queries of the following structure:

I'm interested in a specific gene, for example FBX07 aka PARK15 that has been linked to early-onset parkinsonian-pyramidal syndrome. For this gene I want to find all variants and which subjects have them. I also want to sort on the variant quality value.

The query takes as input four variables. The first is the chromosome on which the gene exists. The second and third are start and stop values that represents where on the chromosome the gene is. Finally an optional quality value may be added which describes how well that variant has been sequenced. The query outputs all found variants, represented by the actual base and the base of the reference genome. For each variant it is also shown from which file it was found. Each file represents one subject.

This type of query was chosen because it represents the majority of the queries done by NysnoBioMetriX each month. Focusing on a smaller range of queries opens the possibility for a more focused study which still contains high relevance. This was deemed to be the best way to conduct this study.

4.5 Google Cloud Platform

GCP is *Google's* platform for managing their cloud based services. GCP has many different products for storing and processing data on the cloud, the two that were used in this study are *Cloud Storage* and*BigQuery* .

4.5.1 Google Cloud Storage

Cloud storage is *Google's* main system for cloud based data storage. It allows for the creation of buckets that can be tailored to fit the needs of the user. When creating a bucket there are two main properties that have to be set, location type and storage class. *Cloud Storage* is billed based on two factors, storage size and data retrieval size, where the different bucket settings affect these prizes in different ways.

Google has storage facilities all over the world which they have divided into 25 regions. The location type describes how the data will be accessible through these regions. Dual-region is the most expensive option with low latency access over two chosen regions. Single-region is the cheapest alternative which offers the same thing as Dual-region but only in one region. In between these two is the Multi-region op-

Table 2: **Prize structure for storage classes in Google Cloud Storage**. *Cloud storage* offers four different storage class options. Standard has the highest storage prize but free data retrieval. Archive has cheap storage prize but expensive data retrieval. The other two fall in between the first two.

Name	Storage prize ($ per GB-month)	Retrieval prize ($ per GB-month)
Standard	0.0200	0.000
Nearline	0.0100	0.010
Coldline	0.0040	0.020
Archive	0.0012	0.050

tion which gives the highest availability across the largest area but with a higher latency.

Storage class describes how the data will be stored which affects the relation between the prize of the storage size and data retrieval size. *Cloud Storage* offers four alternatives and their different prizes can be seen in Table 2.

For this study the single region location type with region eu-north1 (Finland) was chosen because the data would only be accessed from Stockholm. The Standard storage class was chosen because it is the best option for when the data is readily accessed as is the case in this study.

4.5.2 Google BigQuery

BigQuery is *Google's* serverless, cloud based and ACID compliant data warehouse which enables scalable analysis over petabytes of relational structured data. It can be described as a hybrid between a SQL and NoSQL system. It uses SQL dialects and supports complex analytical SQL-based queries for big data. In addition one can use NoSQL techniques like denormalization and loading of data to improve performance.

The integrated connections between *Cloud storage* and *BigQuery* offers an easy way to create a data lake storage environment. Creating data sets in *BigQuery* and connecting them to storage buckets allows you to upload the unstructured data into ta-

bles. It is then possible to run SQL based queries on the data stored in the tables.

4.6 Amazon Web Services

AWS is *Amazon's* platform for managing their cloud based services. AWS declares themselves to be the largest cloud provider in the world, providing the most services and features under the fastest pace of innovation. In this study Amazon Simple Storage Service (Amazon S3) is used for cloud storage and *Amazon Athena* and Amazon RDS are used for the calculations.

4.6.1 Amazon Simple Storage Service

Amazon S3 is *Amazon's* service for cloud based storage. Similar to *Google's* version it works by setting up buckets that you connect to regions of your choice. Amazon S3 does not offer different storage classes, their archive storage is separated into another product. They do have bucket versioning which allows for the keeping of multiple variants of an object in the same bucket.

4.6.2 Amazon Athena

Amazon Athena is a serverless, ACID compliant infrastructure which allows for interactive querying of relational structured data stored in Amazon S3 using standard SQL. The only thing needed to run the queries is a schema for the data. The schema can be defined yourself, but it is also possible to run the integrated *AWS Glue* crawler which crawls the available data and constructs the schema based on it.

4.6.3 Amazon Relational Database Service

Amazon RDS is *Amazon's* service for setting up relational databases in a cloud based environment. It works with six different database engines: Amazon Aurora, *Post-*

greSQL , MySQL, MariaDB, Oracle Database and SQL Server. *PostgreSQL* was chosen for this study. Amazon RDS also offers a large range of different instance types that offer different types of functionalities and capacities. For this study the M6g instance type was chosen which is powered by Arm-based AWS Graviton2 processors with 64-bit Arm Neoverse cores. More specifically the M6g.large version was chosen which comes with two virtual CPUs, 8 GB Memory and a network performance of up to 10 Gb/s. This instance type was chosen because it is the latest and cheapest general purpose instance type available.

4.7 PostgreSQL

PostgreSQL is an open source object-relational database which uses and extends the SQL language. Being a object-relational database means that it allows for the definition of objects and table inheritance which enables the creation complicated data structures. *PostgreSQL* has grown in support over the last few years and is today used by large companies like *Instagram, Twitch* and *Skype* [Rom].

5 Implementation

This section describes how the tools were set up to analyze the three different systems. Firstly the preprocessing is described in Section 5.1, secondly the three setups are described in Sections 5.2, 5.3 and 5.4. Finally the setup of the tests are described in Section 5.6.

5.1 Data Preprocessing

As mentioned previously VCF is not an optimal file format for any kind of data processing. The files were therefore converted to *Parquet* format via JSON through the *Tomatula* tool using *Apache Spark's* pyspark library. When a file, for example called file_name.json, is converted to *Parquet* it will be partitioned into several ".parquet" files. All of these files will be stored in a folder called file_name.parquet. Henceforth when mentioning a *Parquet* file it is the file_name.parquet folder containing the partitions which is referred to.

After data preprocessing the three systems were set up.

5.2 GCP BigQuery Setup

To start setting up the test environment in *BigQuery* the *Parquet* files were uploaded to a bucket in the GCP *Cloud Storage*. The bucket was created with a single region location type located on europe-nort1 (Finland) with the Standard storage class. The files were uploaded by dragging and dropping them from the computer to GCP's storage bucket.

In *BigQuery* a dataset called *test* was created on the same project and with the same location type as the previously created bucket. A table was created and filled with the data from one *Parquet* file with the following command written in GCP's Active Cloud Shell:

```
bq load --source_format=PARQUET test.table_X
"gs://bucket_name/parquet_folder_name.parquet/*.parquet"
```

The X in test.table_X was replaced with a unique tag referring to that specific *Parquet* file. The bucket_name and Parquet_folder_name were also replaced with the actual bucket and folder names in GCP. This upload was done once for each *Parquet* file.

The Python script for initiating the tests can be seen in script 2 . It uses the google.cloud library to connect to a *BigQuery* client which executes queries read from a CSV file through *Pandas*, a python library for data analysis. The query used can be seen below where variables initiated by @ are defined by the current test and _TABLE_SUFFIX gives the name of the table which represents the *Parquet* file.

```
SELECT CHROM, POS, ALT, _TABLE_SUFFIX as table_name
FROM `GCP_project_name.test.*`
WHERE CHROM = @chrom AND POS > @start
AND POS < @end AND QUAL > @qual
```

To be able to connect to *BigQuery* using a Python script a service account key needs to bo created and downloaded. An environment variable called GOOGLE_APPLICATION_CREDENTIALS then has to be created pointing to the location of the service account key.

5.3 AWS Athena Setup

To start setting up the test environment in *Athena* the *Parquet* files were uploaded to an *S3* bucket in the AWS cloud storage. The bucket was created with standard settings in the eu-west-1 (Ireland) region. A crawler was then created using *AWS Glue* which used the *Parquet* storage bucket as source and an *AWS Glue* database as destination. When run, the crawler automatically found the schema of the *Parquet* files and created a table connecting them together and making it possible for *Athena* run queries on them. The created table is not a physical table as would have been created in a relational database, but instead a table of metadata connected to the *S3* bucket.

The Python script for initiating the tests can be seen in script 1 on page 41. It uses the *boto3* Python library to connect to an *Athena* client which executes queries read from a CSV file through *Pandas*. The query used can be seen below where *database* and *table* refers to the items created by the crawler, and the other variables in brackets are defined by the current test. "split(split("$path", '/')[4], '.')[1]" extracts the path and formats it such that only the file name is returned, similarly to _TABLE_SUFFIX in *BigQuery*.

```
SELECT chrom, pos, alt,
split(split("$path", '/')[4], '.')[1] as patient
FROM "{database}"."{table}"
WHERE chrom = '{chrom}' and pos > {start}
AND pos < {end} AND qual > {qual}
```

To be able to connect to *Athena* using a Python script an AWS Access Key needs to be created. To connect the key to a computer The AWS Command Line Interface (awscli) needs to be installed. With awscli installed the aws configure command can be run which prompts for the Access Key information and allows for local access to AWS services.

5.4 PostgreSQL Database using AWS RDS

To start setting up the test environment a *PostgreSQL* m6g.large database was created with Amazon RDS [Ama]. The script shown in script 4 was used to define the database by the schema seen in figure 1. The design of the database schema is discussed in Section 5.5. It was populated using script 5 which uses *Pandas* to loop through the VCF files and adds the variants found inside the chosen genes. The chosen genes were the following: PARK1, PARK2, PARK6, PARK8, PARK9, PARK14, PARK15, PARK17, PARK18, PARK21, PARK23.

The tests were initiated through the Python script shown in script 3. The query used can be seen below where variables with %s are defined by the current test. The

database is connected to via username, password and by allowing the current IP on AWS.

```
SELECT uv.chrom, uv.pos, uv.alt, sv.file_name
FROM public.unique_variants uv
INNER JOIN public.specific_variants sv
on sv.variant_id = uv.variant_id
WHERE uv.chrom = %s and uv.pos > %s
AND uv.pos < %s AND sv.qual > %s
```

5.5 Designing the Relational Database

There are several paths which can be explored when trying to search through genomics data for interesting variants. A simple solution is to convert the VCF files into *Parquet* files which takes a lot less space and are searchable using SQL queries. The *Parquet* files are also columnar. This means that only the relevant columns needs to be searched through when doing queries. This solution requires a lower time and knowledge investment to set up and run which is what will be done in the GCP *BigQuery* and AWS *Athena* test cases. The amount of data which these queries have to search through does however grow quickly as more genomes are added. Finding a better scaling solution could therefore be interesting.

One such solution to this problem could be the use of a relational database. Just putting all the data in a database will however not be much better than the previously mentioned solution. Somehow the data needs to be reduced and scalability improved. One thing that could be taken advantage of is the fact that many of the variants found will be the same for different patients. This can be done by creating a table containing only the unique variants and connecting it with a table with the *Parquet* file names as shown in the schema in figure 1. This solution scales better than the first solution because the largest part of the query will be done on the Unique_Variant table which grows slower than the *Parquet* files.

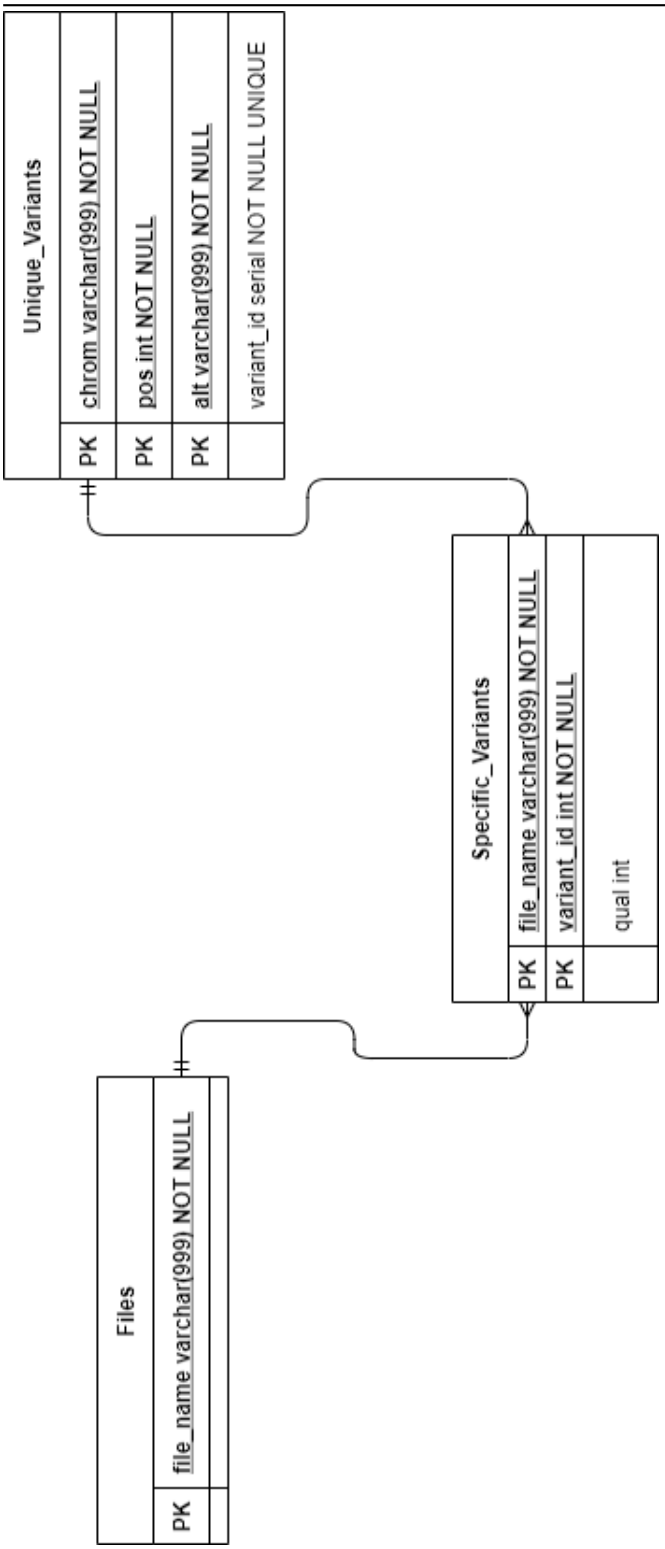

Figure 1 Schema for the relational database filtered on chosen genes.

An interesting factor to take into consideration is that many research institutes focus on certain parts of the human genome. For example Parkinson researchers have about 30 Parkinson related genes in which they are interested in. Compared to the total of 20 000 genes in the human genome, 30 genes is a much more manageable size. A final solution could therefore be to only add the variants of predetermined "interesting" genes to the database. This would significantly raise its scalability and lower the query compute cost while slightly restraining its flexibility. As long as the percentage of interesting genes is below 100% of the total available genes this solution would be strictly faster than the original database solution. The lower the percentage is pushed the more efficient it becomes. This database will be more complicated to set up, especially compared to the data lake solution, because all the *Parquet* files need to be filtered as they are uploaded to the database. If the majority of the executed queries are done on only a few genes then this will however improve the query to such a degree that significant time and money can be saved.

5.6 Test Structure

For each test the following nine queries were run on every system. Each query was run as described in each respective setup section shown previously.

- Find all variants in the PARK1 gene with quality score above 0
- Find all variants in the PARK1 gene with quality score above 0
- Find all variants in the PARK1 gene with quality score above 50
- Find all variants in the PARK1 gene with quality score above 50
- Find all variants in the PARK7 gene with quality score above 0
- Find all variants in the PARK7 gene with quality score above 50
- Find all variants in the PARK15 gene with quality score above 0
- Find all variants in the PARK15 gene with quality score above 50

- Find all variants in the PARK1 gene with quality score above 0

The test was run 10 times for each system, 5 times with 2.48 GB of data and 5 times with 4.96B of data. For each test the first query was initiated simultaneously on the three different systems. The second query was immediately executed as soon as the first one on the respective system completed. This continued until all nine queries had finished. For each query the time taken and data scanned was recorded.

The queries for the test were chosen for a few specific reasons. The first two sets of queries were the same to test cashe behavior. Having the final query being the same as the first one was done for the same reason. The changes in quality score is there to test the systems behavior with no quality filter (quality score 0) compared to an existing quality score filter (quality score 50). The systems were taken down between each test run to make sure that their cashes always behaved the same.

6 Results

The parquet file format proved to be an effective format for storing genomics data. It reduced the data size from 13GB to 2.48GB and made them able to be queried by *Athena* and *BigQuery*. For the data lake comparison *BigQuery* was 26% more expensive than *Athena* with the quality filter, but 29% cheaper without. The *BigQuery* queries were on average 45% faster.

The data lakes were more cost efficient compared to the relational database when the total data size was around 10GB (graph2). For a total data size of 100GB the database is more cost efficient when at least 1200 queries are done each 30 days (graph 3). When stored data reaches terabytes the *PostgreSQL* database quickly becomes the cheaper alternative (graph 4). All comparisons were made with the assumptions that the costs will scale linearly as the total amount of data is increased.

Time and cost were the metrics evaluated for each query. Time was measured with the Python time.time() function. Cost evaluation for *Athena* and *BigQuery* systems behave differently compared to the Amazon RDS *PostgreSQL* database. Both *Athena* and *BigQuery* are billed at $5 per terabyte of data scanned. The Amazon RDS database is on the other hand priced per hour with a rate depending on which database size is chosen. The one used in this experiment, db.m6g.large, is billed at $0.176 per hour which translates to $126.72 per 30 days.

The average time for one query with the *Athena* setup was approximately 2.98 seconds and required 20% of the total data to be searched. Adding the quality filter raised the searched percentage by 5.5% (Table 3). Doubling the total amount of data had no clear impact on either the percentage of data scanned or execution/query time indicating that both these parameters scale well as more data is added. The original tests using the *BigQuery* setup were slightly faster but significantly more expensive with each query taking about 1.8 seconds and needing to scan 80% to 113% of the data (Table 4). The *BigQuery* tests where VCF file 3, 4 and 5 had been removed showed results more similar to *Athena* with 26% data scanned with quality filter and 16% without (Table 5). These queries took a similar amount of time compared to the old

BigQuery tests and no significant difference was found when doubling the data. The *PostgreSQL* database was significantly faster taking about 0.2 seconds for each query.

Table 3: **Performance for Amazon Athena**. Data from the tests using *Amazon Athena* as described in Section 5.3. *Athena* consistently needed to scan 20% of the total data, adding the quality filter only raised this by about 5.5%. Doing one query took around 3 seconds, this did not change when the total data was doubled.

	Gene	Quality	Total Data (GB)	Data scanned (GB)	# tests	Time (s)
0	PARK1(SNCA)	0	2.48	0.51	15	2.957
1	PARK1(SNCA)	50	2.48	0.54	10	2.819
2	PARK7(DJ1)	0	2.48	0.51	5	2.515
3	PARK7(DJ1)	50	2.48	0.55	5	2.945
4	PARK15(FBX07)	0	2.48	0.51	5	3.119
5	PARK15(FBX07)	50	2.48	0.55	5	2.809
6	PARK1(SNCA)	0	4.96	1.03	15	3.069
7	PARK1(SNCA)	50	4.96	1.09	10	2.801
8	PARK7(DJ1)	0	4.96	1.03	5	2.927
9	PARK7(DJ1)	50	4.96	1.09	5	2.949
10	PARK15(FBX07)	0	4.96	1.03	5	3.137
11	PARK15(FBX07)	50	4.96	1.10	5	3.920

Table 4: **Performance for Google BigQuery using all data**. Data from the tests using Google *BigQuery* as described in Section 5.2. *BigQuery* had a significant difference between the queries with and without quality filters where the queries without filter needed to scan about 80% of the data, compared to 113% when the filter was added. Doing one query took around 1.8 seconds, this did not change when the total data was doubled.

	Gene	Quality	Total Data (GB)	Data scanned (GB)	# tests	Time (s)
0	PARK1(SNCA)	0	2.48	1.96	15	1.991
1	PARK1(SNCA)	50	2.48	2.93	10	2.362
2	PARK7(DJ1)	0	2.48	1.96	5	1.922
3	PARK7(DJ1)	50	2.48	2.93	5	1.877
4	PARK15(FBX07)	0	2.48	1.96	5	1.697
5	PARK15(FBX07)	50	2.48	2.93	5	1.405
6	PARK1(SNCA)	0	4.96	3.68	15	1.830
7	PARK1(SNCA)	50	4.96	5.51	10	1.321
8	PARK7(DJ1)	0	4.96	3.68	5	1.617
9	PARK7(DJ1)	50	4.96	5.51	5	1.429
10	PARK15(FBX07)	0	4.96	3.68	5	1.759
11	PARK15(FBX07)	50	4.96	5.51	5	1.314

Table 5: **Performance for Google BigQuery without faulty data**. Data from the tests using Google *BigQuery* as described in Section 5.2, but where some VCF have been left out. Compared to Table 4 this gave a much lower percentage of data scanned at only 26% with the quality filter, and 16% without. Doing one query took around 1.6 seconds, this did not change when the total data was doubled.

	Gene	Quality	Total Data (GB)	Data scanned (GB)	# tests	Time (s)
0	PARK1(SNCA)	0	2.02	0.33	12	1.360
1	PARK1(SNCA)	50	2.02	0.52	8	1.311
2	PARK7(DJ1)	0	2.02	0.33	4	1.289
3	PARK7(DJ1)	50	2.02	0.52	4	1.650
4	PARK15(FBX07)	0	2.02	0.33	4	1.298
5	PARK15(FBX07)	50	2.02	0.52	4	1.168

Table 6: **Performance for relational database**. Data from the tests using a relational *PostgreSQL* database using *Amazon* RDS as described in Section 5.4. The queries were consistently very fast with no significant change when the data was doubled.

	Gene	Quality	Total Data (GB)	# tests	Time (s)
0	PARK1(SNCA)	0	2.48	6	0.212
1	PARK1(SNCA)	50	2.48	4	0.132
2	PARK7(DJ1)	0	2.48	2	0.138
3	PARK7(DJ1)	50	2.48	2	0.131
4	PARK15(FBX07)	0	2.48	2	0.134
5	PARK15(FBX07)	50	2.48	2	0.126
6	PARK1(SNCA)	0	4.96	12	0.362
7	PARK1(SNCA)	50	4.96	8	0.140
8	PARK7(DJ1)	0	4.96	4	0.182
9	PARK7(DJ1)	50	4.96	4	0.139
10	PARK15(FBX07)	0	4.96	4	0.191
11	PARK15(FBX07)	50	4.96	4	0.139

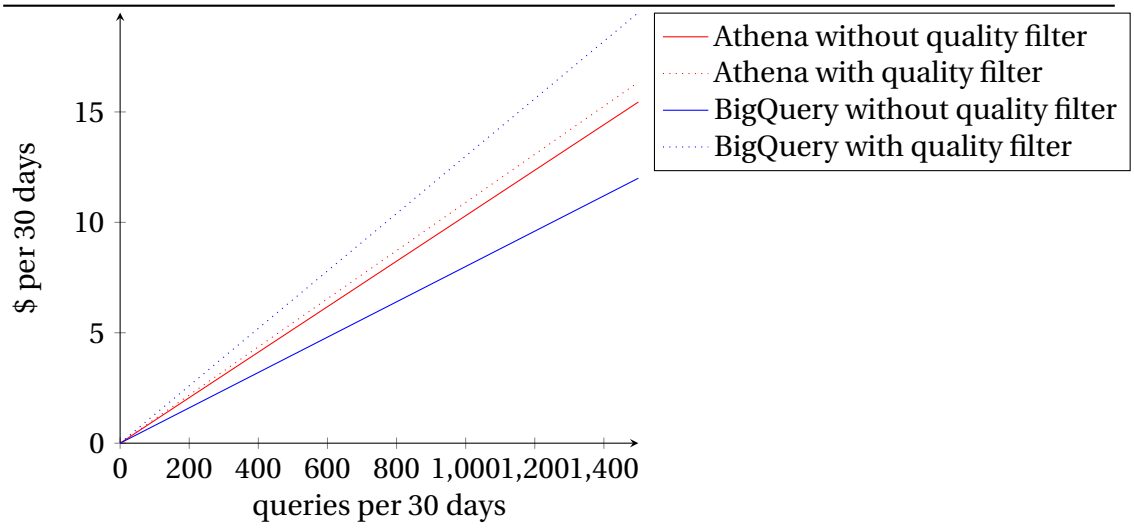

Figure 2 An extrapolation of the test data showing the costs of the systems with 10GB of genomics data in *Parquet* files. For the *BigQuery* data from Table 5 was used. At this data size the *PostgreSQL* database costs a lot more than the other systems, at $126.72 per month. *Athena* and *BigQuery* scale similarly.

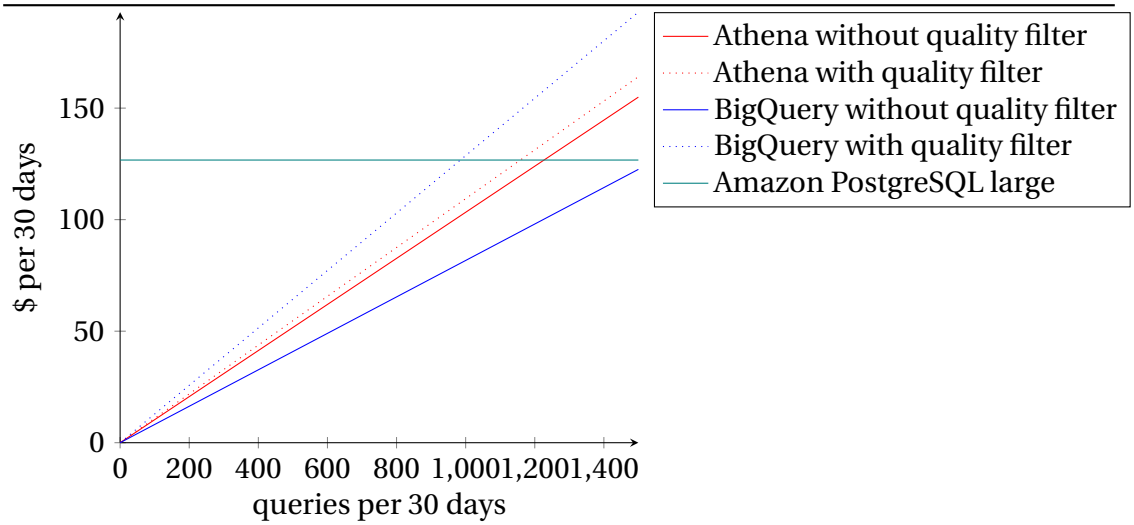

Figure 3 An extrapolation of the test data showing the costs of the systems with 100GB of genomics data in *Parquet* files. For the *BigQuery* data from Table 5 was used. At this data size the *PostgreSQL* database becomes more cost efficient when more than around 1200 queries are done per 30 days. *Athena* and *BigQuery* scale similarly.

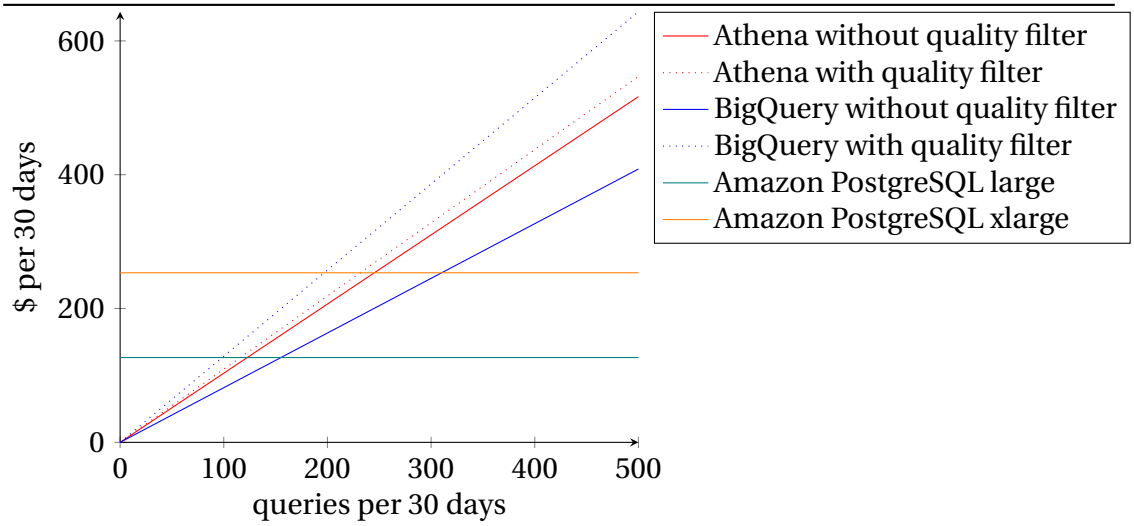

Figure 4 An extrapolation of the test data showing the costs of the systems with 1TB of genomics data in *Parquet* files. For the *BigQuery* data from Table 5 was used. At this data size the *PostgreSQL* database quickly becomes more cost efficient than both *BigQuery* and *Athena*. For large enough query sizes the database can even be upgraded and still cost less. *Athena* and *BigQuery* scale similarly.

7 Discussion

For this study there are two important comparisons that needs to be made, *Athena* versus *BigQuery* and data lake solutions versus a relational database in the cloud. These will be compared mainly on cost and time efficiency, but also on ease of use.

7.1 Athena vs BigQuery

BigQuery and *Athena* behaved similarly when the second set of*BigQuery* tests were used. The *Athena* queries took on average 45% more time to execute than the *BigQuery* queries. The average time for each *Athena* query, at 2.98 seconds, is still inside what I would call a short query time. The execution times of both *BigQuery* and *Athena* did not change when doubling the amount of available data. This shows that the query time for both systems scale well with relation to available data.

When it comes to cost comparison things did not look good for *BigQuery* after the initial tests. With the quality filter active *BigQuery* somehow had to search through more than 100% of the data which is extremely bad, especially compared to *Athena's* 21.5%. After looking into this anomaly I noticed that for half the files *BigQuery* needed to scan 8-10 times more data compared to the other half. This is extremely interesting as the files have the exact same file structure and very similar content. This problem could have risen from the fact that the files come from different sources and have different information in their FILTER, INFO, FORMAT and SAMPLE columns (Table 1). All these columns are however only formatted as strings, and are never touched in any of the queries which means that their differences should not effect the queries. To find out more about this problem new tests would need to be made on more consistent data.

If hyper expensive files continues to be a problem in *BigQuery*, then *Athena* will be significantly cheaper. If however this problem is solved then they will have very similar costs. Another interesting subject notice is that the amount of data scanned in the *Athena* queries only raised by 1,5% when the quality filter was added, compared

to 62% for *BigQuery*. This shows that *Athena* and *BigQuery* have very different ways of handling their queries, meaning that the difference in data scanned for more advanced queries could be significant. The great thing however is that both of these systems have simple setups and my recommendation would be to set up a test scenario in both systems, run a test query and see how much your query costs in either system.

When it comes to usability I personally believe that GCP has a more user friendly interface. AWS tends to show the user all of their options, which is good but might be a bit confusing at first glance. GCP's cleaner interface is naturally navigated and their guides easier to follow. The code for the *BigQuery* tests was also easier to write. This was partially because it only needed 60% as many lines as the *Athena* tests, but also because of the more intuitive way they are handling the SQL queries. These are however minor differences which are quickly overshadowed if you have someone available who is informed on either of the systems.

7.2 Data Lake Solution vs PostgreSQL Database

The data lake solutions and PostgreSQL database are both ACID compliant systems which means that their query behavior is expected to be similar. There are still some differences that can make them tricky to compare. One such difference is how they are priced where *Athena* and *BigQuery* are billed per data amount scanned and the database is billed per active hour. This means that the database requires the user to be consistently active for it to be financially worth it. If I assume that the three systems continue to behave in the same way when adding more data then we have two scenarios for when the database becomes more cost efficient. If above 1200 queries are done each 30 days then 100GB of stored data is needed. If however 1TB of data is stored, then the database costs less when over 200 queries are done each 30 days. There are steps which can be taken to reduce the costs of the database. It can for example be turned off, stopping the hourly billing, if it is known that it will not be used for a period of time. *Amazon* also offers burstable database instances at a re-

duced cost. These database instances accumulate CPU credits every hour which can be used for running queries and can be stored for an 24h period. In summary these databases are cheaper when a regular amount of queries are done every day at specific times each day. These options do however add another layer of complexity into the database usage. When it comes to the time taken for each query then the database was clearly the fastest alternative taking only 15.4% of the *BigQuery* time and 6.7% of the *Athena* time.

Another important point of comparison is the ease of use and maintainability of the two different kinds of systems. *Athena* and *BigQuery* have sometimes been referred to as the simple solutions in this study, and that is not for nothing. Both of these systems are extremely easy to set up and almost only require the user to upload the data in a supported file format, and then it will be ready to be queried. The database on the other hand requires more maintenance. First a schema hast to be constructed for the users specific scenario. Then the database has to be populated based on the chosen interesting genes, and if new genes are deemed interesting then the files need to processed again to populate the database with this information. These are of course procedures that can be well managed with a proper infrastructure, but they still make the database more complicated than the data lake solutions.

8 Conclusion

In summary the Parquet file format worked well for holding genomics variant data, and the data lake solution and *PostgreSQL* database were both good solutions usable for different scenarios. If a quick to set up solution is needed or the queries done are infrequent or inconsistent then the a data lake solution is a good option. If however a large amount of queries are done each month and a low query time is very important then setting up a relational database is most likely the correct answer. If the extrapolations from the current test data are correct, then having amounts of data in at least the terabyte ranges would also make the database a better option. More tests with larger amounts of data are however needed to fully support that statement.

When it comes to AWS vs GCP then the choice will not make much difference and the platform with highest available support for the users current situation should be used. This is by making the assumption that the spikes in *BigQuery* costs comes from an error because of the inconsistency of input data. If this is not the case then the user needs to be aware of the data scanned by *BigQuery* or their queries could become very expensive. The easiest solution is for the user to set up their query in both AWS and GCP and compare the cost difference.

9 Future work

For future work related to this study there are two main areas to discuss. Firstly tests need to be done on larger amounts of data using the setups created in this study. Many of the conclusions are drawn from extrapolations of the given test data. Doing tests with larger amounts would help support the aforementioned extrapolations. Secondly, doing tests with a more concise source of VCF files would help to find out whether the spike in data scanned by *BigQuery* was a bug, or a system problem.

References

[Ama] Amazon Web Services. Amazon rds instance types. Accessed 2021-04-21. [Online]. Tillgänglig: https://aws.amazon.com/rds/instance-types/

[BFvK+17] A. Boufea, R. Finkers, M. van Kaauwen, M. Kramer, and I. N. Athanasiadis, "Managing variant calling files the big data way: Using hdfs and apache parquet," p. 219–226, 2017. [Online]. Tillgänglig: https://doi.org/10.1145/3148055.3148060

[Dav10] K. Davies, "The $1,000 genome: the revolution in dna sequencing and the new era of personalized medicine," *Free Press, New York*, 2010.

[HGP+14] A. P. Heath, M. Greenway, R. Powell, J. Spring, R. Suarez, D. Hanley, C. Bandlamudi, M. E. McNerney, K. P. White, and R. L. Grossman, "Bionimbus: a cloud for managing, analyzing and sharing large genomics datasets," *Journal of the American Medical Informatics Association*, vol. 21, no. 6, pp. 969–975, 01 2014. [Online]. Tillgänglig: https://doi.org/10.1136/amiajnl-2013-002155

[JM01] H. J. Maurer, "Human genome project: German perspective," *ScienceDirect*, 2001.

[Nat] National HUman Genome Research Institute. Genetics vs. genomics fact sheet. Accessed 2021-05-06. [Online]. Tillgänglig: https://www.genome.gov/about-genomics/fact-sheets/Genetics-vs-Genomics#:~:text=All%20human%20beings%20are%2099.9,about%20the%20causes%20of%20diseases.

[NIH21] NIH. (2021, Jan.) The cost of sequencing a human genome. Accessed 2021-01-26. [Online]. Tillgänglig: https://www.genome.gov/about-genomics/fact-sheets/Sequencing-Human-Genome-cost

[Per] Personal Genome Project. Personal genome project. Accessed 2021-05-07. [Online]. Tillgänglig: https://www.personalgenomes.org

[RML+17] S. M. Reynolds, M. Miller, P. Lee, K. Leinonen, S. M. Paquette, Z. Rodebaugh, A. Hahn, D. L. Gibbs, J. Slagel, W. J. Longabaugh, V. Dhankani, M. Reyes, T. Pihl, M. Backus, M. Bookman, N. Deflaux, J. Bingham, D. Pot, and I. Shmulevich, "The isb cancer genomics cloud: A flexible cloud-based platform for cancer genomics research," *Cancer Research*, vol. 77, no. 21, pp. e7–e10, 2017. [Online]. Tillgänglig: https://cancerres.aacrjournals.org/content/77/21/e7

[Rom] J. Romanowski. The most popular databases in 2020. Accessed 2021-05-11. [Online]. Tillgänglig: https://learnsql.com/blog/most-popular-sql-databases-2020/

[Ste10] L. D. Stein, "The case for cloud computing in genome informatics," *Genome Biology*, vol. 11, no. 5, p. 207, May 2010. [Online]. Tillgänglig: https://doi.org/10.1186/gb-2010-11-5-207

[The12] The 1000 Genomes Project Consortium, "An integrated map of genetic variation from 1,092 human genomes," *Nature*, 2012.

[WKF+10] D. P. Wall, P. Kudtarkar, V. A. Fusaro, R. Pivovarov, and P. J. Patil, Prasad Tonellato, "Cloud computing for comparative genomics," *BMC Bioinformatics*, no. 259, 2010. [Online]. Tillgänglig: https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-11-259#citeas

[ZPS+13] S. Zhao, K. Prenger, L. Smith, T. Messina, H. Fan, E. Jaeger, and S. Stephens, "Rainbow: a tool for large-scale whole-genome sequencing data analysis using cloud computing," *BMC Genomics*, vol. 14, no. 1, p. 425, Jun 2013. [Online]. Tillgänglig: https://doi.org/10.1186/1471-2164-14-425

A Scripts

Listing 1: Python code for running tests on Amazon Athena

```
# Imports
import sys
import pandas as pd
import ntpath
import boto3
import botocore
import time
import os
import logging
from tabulate import tabulate

test_path = sys.argv[1]

tests = pd.read_csv(test_path)
test_list = []

for index, row in tests.iterrows():
    test_dict = {
        "NAME": row["NAME"],
        "CHROM": row["CHROM"],
        "START": row["START"],
        "END": row["END"],
        "QUAL": row["QUAL"]
    }
    test_list.append(test_dict)

master_start_time = time.time()

logging.basicConfig(filename='transfer.log', filemode='a',
                    format='%(name)s - %(levelname)s - %(message)s',
                    level=logging.INFO)

# Starting connection with athena through boto3
try:
    client = boto3.client('athena')

except botocore.exceptions.ClientError as error:
    print("Failed connection to athena:", error)
    sys.exit()

# Setting headers for the Latex output dictionary
test_data = [{"TEST": "Test",
              "PTIME": "Time (Python)",
              "ETIME": "Engine Execution Time",
              "TTIME": "Total Execution Time",
              "N_RESULTS": "# results",
              "DATA_SCANNED": "Scanned Data (bytes)"
              }]

# Setting database,table and output bucket for queries
database = "thesis-parquetschema"
table = "thesisdatascience_parquetbucketd7bb2fe9_m3fwnlu10gkx"
output_bucket = 's3://thesisdatascience-queryresult/test1/'

# Looping over the different tests
for i in range(len(test_list)):
```

```
    test = test_list[i]

    # Two different strings are built depending on QUAL
    try:
        ts = time.time()
        if test["QUAL"] > 0:
            query_string = """
                SELECT chrom, pos, alt, split(split("$path", '/')[4], '.')[1] as patient
                FROM "{database}"."{table}"
                WHERE chrom = '{chrom}' and pos > {start} AND pos < {end} AND qual > {qual}
                """.format(database=database, table=table, chrom=str(test["CHROM"]),
                           start=test["START"], end=test["END"], qual=test["QUAL"])

        else:
            query_string = """SELECT chrom, pos, alt, split(split("$path", '/')[4], '.')[1] as patient
                        FROM "{database}"."{table}"
                        WHERE chrom = '{chrom}' and pos > {start} AND pos < {end}
                        """.format(database=database, table=table, chrom=str(test["CHROM"]),
                                   start=test["START"], end=test["END"])

        print("query: \n", query_string)

        # Query execution
        query_id = client.start_query_execution(
            QueryString=query_string,
            QueryExecutionContext={
                'Database': database
            },
            ResultConfiguration={
                'OutputLocation': output_bucket
            }
        )['QueryExecutionId']

        # Wait until execution completion
        iterations = 100
        while iterations > 0:
            iterations = iterations - 1

            query_status = client.get_query_execution(QueryExecutionId=query_id)
            query_execution_status = query_status['QueryExecution']['Status']['State']
            print("Query ", query_id, " status: ", query_execution_status, " iteration: ",
                  100-iterations)

            if (query_execution_status == 'FAILED') or (query_execution_status == 'CANCELLED'):
                print("test: ", test, " FAILED in execution")
                break
            elif query_execution_status == 'SUCCEEDED':

                # Get query results
                results_paginator = client.get_paginator('get_query_results')
                results_iter = results_paginator.paginate(
                    QueryExecutionId=query_id,
                    PaginationConfig={
                        'PageSize': 1000
                    }
                )
                data_list = []
                for results_page in results_iter:
                    for row in results_page['ResultSet']['Rows']:
                        data_list.append(row['Data'])

                number_of_variants = len(data_list)-1
```

```
                query_statistics = query_status['QueryExecution']['Statistics']
                engine_execution_time = query_statistics['EngineExecutionTimeInMillis']
                total_execution_time = query_statistics['TotalExecutionTimeInMillis']
                data_scanned = query_statistics['DataScannedInBytes']

                te = time.time()
                print("Test ", test, "\nCompleted successfully in ", te - ts, "seconds")
                print("Number of variants found: ", number_of_variants, "\n")

                test_data.append({
                    "TEST": "Gene: " + test["NAME"] + ", Quality: " + str(test["QUAL"]),
                    "PTIME": te - ts,
                    "ETIME": engine_execution_time,
                    "TTIME": total_execution_time,
                    "N_RESULTS": number_of_variants,
                    "DATA_SCANNED": data_scanned,
                })

                write_to_file_path = os.path.join("test_data/athena/" + str(i) + "/run_"
                                                  + str(master_start_time) + ".txt")
                with open(write_to_file_path, "w") as file_handle:
                    for row in data_list:
                        file_handle.write("{row}\n".format(row=row))

                break
            else:
                time.sleep(0.2)

    except botocore.exceptions.ClientError as error:
        print("Failed to execute test ", test, " with error: \n", error)


# Writing result data to csv
write_to_file_path = os.path.join("tests/csv", str(master_start_time) + "_Athena.txt")
with open(write_to_file_path, "w") as file_handle:
    for list_item in test_data:
        file_handle.write("{item}\n".format(item=list_item))

with open("tests/csv/athena_latest.txt", "w") as file_handle:
    for list_item in test_data:
        file_handle.write("{item}\n".format(item=list_item))

# Writing data as latex
write_to_file_path = os.path.join("tests/latex", str(master_start_time) + "_Athena.txt")
with open(write_to_file_path, "w") as file_handle:
    file_handle.write(tabulate(test_data, headers="firstrow", showindex="always", tablefmt="latex"))

with open("tests/latex/athena_latest.txt", "w") as file_handle:
    file_handle.write(tabulate(test_data, headers="firstrow", showindex="always", tablefmt="latex"))

master_end_time = time.time()

print("Successfully ran all Athena tests")
print("Total time for all tests: ", master_end_time - master_start_time)
```

Listing 2: Python code for running tests on Google BigQuery

```
## Imports
import sys
import pandas as pd
```

```
import time
import os
import logging
from tabulate import tabulate
from google.cloud import bigquery

test_path = sys.argv[1]

tests = pd.read_csv(test_path)
test_list = []

for index, row in tests.iterrows():
    test_dict = {
        "NAME": row["NAME"],
        "CHROM": row["CHROM"],
        "START": row["START"],
        "END": row["END"],
        "QUAL": row["QUAL"]
    }
    test_list.append(test_dict)

master_start_time = time.time()

test_data = [{"TEST": "Test",
              "TIME": "Time",
              "N_RESULTS": "# results",
              "BYTES_PROCESSED": "Scanned data (bytes)"
              }]


client = bigquery.Client()
print("client succesfully created")

for i in range(len(test_list)):
    test = test_list[i]

    ts = time.time()

    if test["QUAL"] > 0:
        query = """
            SELECT CHROM, POS, ALT, _TABLE_SUFFIX as table_name
                FROM `parquet-testing-305610.test.*`
                WHERE CHROM = @chrom AND POS > @start AND POS < @end AND QUAL > @qual
            """

        job_config = bigquery.QueryJobConfig(
            query_parameters=[
                bigquery.ScalarQueryParameter("chrom", "STRING", str(test["CHROM"])),
                bigquery.ScalarQueryParameter("start", "INT64", test["START"]),
                bigquery.ScalarQueryParameter("end", "INT64", test["END"]),
                bigquery.ScalarQueryParameter("qual", "FLOAT", test["QUAL"]),
            ]
        )
    else:
        query = """
            SELECT CHROM, POS, ALT, _TABLE_SUFFIX as table_name
                FROM `parquet-testing-305610.test.*`
                WHERE CHROM = @chrom AND POS > @start AND POS < @end
            """

        job_config = bigquery.QueryJobConfig(
            query_parameters=[
```

```
                bigquery.ScalarQueryParameter("chrom", "STRING", str(test["CHROM"])),
                bigquery.ScalarQueryParameter("start", "INT64", test["START"]),
                bigquery.ScalarQueryParameter("end", "INT64", test["END"]),
            ]
        )

    print("running test: ", test)
    query_job = client.query(query, job_config=job_config)
    results = query_job.result()  # Waits for job to complete.

    te = time.time()

    while(query_job.done() != True):
        print("Wait")
        time.sleep(1)

    print("test_query took: ", te - ts, "seconds")
    print("variants found: ", results.total_rows)
    print("Data scanned: ", query_job.total_bytes_processed/1000000, "MB")

    test_data.append({
        "TEST": "Gene: " + test["NAME"] + ", Quality: " + str(test["QUAL"]),
        "TIME": te - ts,
        "N_RESULTS": results.total_rows,
        "BYTES_PROCESSED": query_job.total_bytes_processed
    })

    write_to_file_path = os.path.join("test_data/bigQuery/" + str(i) + "/run_" + str(master_start_time)
    with open(write_to_file_path, "w") as file_handle:
        for row in results:
            file_handle.write("{item}\n".format(item=row))

# Writing result data as csv
write_to_file_path = os.path.join("tests/csv", str(master_start_time) + "_BigQuery.txt")
with open(write_to_file_path, "w") as file_handle:
    for list_item in test_data:
        file_handle.write("{item}\n".format(item=list_item))

with open("tests/csv/BigQuery_latest.txt", "w") as file_handle:
    for list_item in test_data:
        file_handle.write("{item}\n".format(item=list_item))

# Writing data as latex
write_to_file_path = os.path.join("tests/latex", str(master_start_time) + "_BigQuery.txt")
with open(write_to_file_path, "w") as file_handle:
    file_handle.write(tabulate(test_data, headers="firstrow", showindex="always", tablefmt="latex"))

with open("tests/latex/BigQuery_latest.txt", "w") as file_handle:
    file_handle.write(tabulate(test_data, headers="firstrow", showindex="always", tablefmt="latex"))

master_end_time = time.time()

print("Successfully ran all bigQuery tests")
print("Total time for all tests: ", master_end_time - master_start_time)
```

Listing 3: Python code for running tests on the PostgreSQL database through Amazon RDS

```
## Imports
import sys
```

```
import pandas as pd
import ntpath
import psycopg2
import time
import os
import logging
from tabulate import tabulate

test_path = sys.argv[1]

tests = pd.read_csv(test_path)
test_list = []

for index, row in tests.iterrows():
    test_dict = {
        "NAME": row["NAME"],
        "CHROM": row["CHROM"],
        "START": row["START"],
        "END": row["END"],
        "QUAL": row["QUAL"]
    }
    test_list.append(test_dict)

master_start_time = time.time()

logging.basicConfig(filename='transfer.log', filemode='a', format='%(name)s - %(levelname)s - %(message)
                    level=logging.INFO)

# Starting connection with postgres database through psycopg2
try:
    connection = psycopg2.connect(user="postgres",
                                  password="e_kW=Z8.loP062-xVM07H8lg3y3Eg5",
                                  host="tp1crio6pixfyuf.cf1yoyudwvis.eu-west-1.rds.amazonaws.com",
                                  port="5432",
                                  database="postgres")
    cursor = connection.cursor()

except (Exception, psycopg2.Error) as error:
    print("Failed connection to database:", error)
    sys.exit()

test_data = [{"TEST": "Test",
              "TIME": "Time",
              "N_RESULTS": "# results"
              }]

# running test query
for test in test_list:

    try:
        ts = time.time()
        if test["QUAL"] > 0:
            postgres_test_query = """SELECT uv.chrom, uv.pos, uv.alt, sv.file_name
                        FROM public.unique_variants uv
                        INNER JOIN public.specific_variants sv on sv.variant_id = uv.variant_id
                        WHERE uv.chrom = %s and uv.pos > %s AND uv.pos < %s AND sv.qual > %s"""
            record_to_test = (str(test["CHROM"]), test["START"], test["END"], test["QUAL"])

        else:
            postgres_test_query = """SELECT uv.chrom, uv.pos, uv.alt, sv.file_name
            FROM public.unique_variants uv
            INNER JOIN public.specific_variants sv on sv.variant_id = uv.variant_id
```

```
            WHERE uv.chrom = %s and uv.pos > %s AND uv.pos < %s"""
            record_to_test = (str(test["CHROM"]), test["START"], test["END"])

        cursor.execute(postgres_test_query, record_to_test)

        connection.commit()

        result = cursor.fetchall()
        te = time.time()
        print("Test ", test, "\nCompleted successfully in ", te - ts, "seconds")
        print("Number of variants found: ", len(result), "\n")

        test_data.append({
            "TEST": "Gene: " + test["NAME"] + ", Quality: " + str(test["QUAL"]),
            "TIME": te - ts,
            "N_RESULTS": len(result),
        })

    except (Exception, psycopg2.Error) as error:
        print("Failed to execute test " + test, error)

# Writing result data as csv
write_to_file_path = os.path.join("tests/csv", str(master_start_time) + "_Postgres.txt")
with open(write_to_file_path, "w") as file_handle:
    for list_item in test_data:
        file_handle.write("{item}\n".format(item=list_item))

with open("tests/csv/postgres_latest.txt", "w") as file_handle:
    for list_item in test_data:
        file_handle.write("{item}\n".format(item=list_item))

# Writing data as latex
write_to_file_path = os.path.join("tests/latex", str(master_start_time) + "_Postgres.txt")
with open(write_to_file_path, "w") as file_handle:
    file_handle.write(tabulate(test_data, headers="firstrow", showindex="always", tablefmt="latex"))

with open("tests/latex/postgres_latest.txt", "w") as file_handle:
    file_handle.write(tabulate(test_data, headers="firstrow", showindex="always", tablefmt="latex"))


master_end_time = time.time()

print("Successfully ran all tests")
print("Total time for all tests: ", master_end_time - master_start_time)

# closing database connection.
if connection:
    cursor.close()
    connection.close()
    print("PostgreSQL connection is closed")
```

Listing 4: Python code for setting up the PostgreSQL database through Amazon RDS

```
## Imports
import sys
import pandas as pd
import ntpath
import psycopg2
import time
import os
import logging
```

```
9
10 # Starting connection with postgres database through psycopg2
11 try:
12     connection = psycopg2.connect(user="postgres",
13                                   password="e_kW=Z8.loP062-xVM07H8lg3y3Eg5",
14                                   host="tp1crio6pixfyuf.cf1yoyudwvis.eu-west-1.rds.amazonaws.com",
15                                   port="5432",
16                                   database="postgres")
17     cursor = connection.cursor()
18 except (Exception, psycopg2.Error) as error:
19     print("Failed connection to database:", error)
20     sys.exit()
21
22
23 # Creating files table
24 try:
25     postgres_insert_query = """
26     CREATE TABLE public.files
27     (
28         file_name text COLLATE pg_catalog."default" NOT NULL,
29         CONSTRAINT files_pkey PRIMARY KEY (file_name)
30     )
31     """
32     cursor.execute(postgres_insert_query)
33
34     print("Successfully crated table: public.files")
35
36 except (Exception, psycopg2.Error) as error:
37     print("Failed to crate table: public.files", error)
38
39 try:
40
41     cursor.execute("""CREATE SEQUENCE unique_variants_variant_id_seq""")
42
43     postgres_insert_query = """
44     CREATE TABLE public.unique_variants
45     (
46         chrom text COLLATE pg_catalog."default" NOT NULL,
47         pos integer NOT NULL,
48         alt text COLLATE pg_catalog."default" NOT NULL,
49         variant_id integer NOT NULL DEFAULT nextval('unique_variants_variant_id_seq'::regclass),
50         CONSTRAINT unique_variants_key PRIMARY KEY (chrom, pos, alt),
51         CONSTRAINT unique_variants_variant_id_key UNIQUE (variant_id)
52     )
53     """
54     cursor.execute(postgres_insert_query)
55
56     print("Successfully crated table: public.unique_variants")
57
58 except (Exception, psycopg2.Error) as error:
59     print("Failed create table: public.unique_variants: ", error)
60
61 try:
62     postgres_insert_query = """
63     CREATE TABLE public.specific_variants
64     (
65         file_name text COLLATE pg_catalog."default" NOT NULL,
66         variant_id integer NOT NULL,
67         qual double precision,
68         CONSTRAINT specific_variants_pkey PRIMARY KEY (file_name, variant_id),
69         CONSTRAINT file_name FOREIGN KEY (file_name)
70             REFERENCES public.files (file_name) MATCH SIMPLE
```

```
            ON UPDATE NO ACTION
            ON DELETE NO ACTION,
        CONSTRAINT variant_id FOREIGN KEY (variant_id)
            REFERENCES public.unique_variants (variant_id) MATCH SIMPLE
            ON UPDATE NO ACTION
            ON DELETE NO ACTION
    )
    """
    cursor.execute(postgres_insert_query)

    print("Successfully crated table: public.specific_variants")

except (Exception, psycopg2.Error) as error:
    print("Failed create table: public.specific_variants: ", error)

connection.commit()
# closing database connection.
if connection:
    cursor.close()
    connection.close()
    print("PostgreSQL connection is closed")
```

Listing 5: Python code for filling the PostgreSQL database with variants filtered based on input

```
## Imports
import sys
import pandas as pd
import ntpath
import psycopg2
import time
import os
import logging

# function for writing data from parquet_file_path to postgres database filtering on genes
def parquetTOpostgres(parquet_file_path):
    # fetching the data from the parquet file based on file_path
    df = pd.read_parquet(parquet_file_path, columns=["CHROM", "POS", "ALT", "QUAL"])

    try:
        ts = time.time()

        unique_variants_insert_query = """
        INSERT INTO public.unique_variants(chrom, pos, alt) VALUES (%s, %s, %s) ON CONFLICT DO NOTHING
        """
        specific_variants_insert_query = """
        INSERT INTO public.specific_variants(file_name, variant_id, qual) VALUES (%s, %s, %s) ON CONFLIC
        """
        variant_id_select_query = """
        SELECT variant_id from public.unique_variants WHERE chrom = %s AND pos = %s AND alt = %s
        """

        chromTag = df.iloc[0]["CHROM"]
        if str(chromTag)[0] == "c":
            chromPrefix = "chr"
        else:
            chromPrefix = ""
        current_chrom = ""
        current_chrom_genes = []

```

```
36          for index, row in df.iterrows():
37
38              if (index % PRINT_EVERY_N) == 0:
39                  print("Current row: ", index)
40
41              if str(row["CHROM"]) != current_chrom:
42                  print("changing to chromosome " + str(row["CHROM"]))
43                  current_chrom = str(row["CHROM"])
44                  current_chrom_genes.clear()
45                  for gene in gene_list:
46                      if chromPrefix + gene["CHROM"] == str(row["CHROM"]):
47                          current_chrom_genes.append(gene)
48                  print("genes in chromosme: ")
49                  print(current_chrom_genes)
50
51              for gene in current_chrom_genes:
52                  if str(row["CHROM"]) == chromPrefix + gene["CHROM"]:
53                      if gene["START"] <= row["POS"] < gene["END"]:
54                          print(gene["NAME"] + " found a variant inside chromosome " + str(row["CHROM"]))
55                          record_to_insert = (gene["CHROM"], row["POS"], row["ALT"])
56                          cursor.execute(unique_variants_insert_query, record_to_insert)
57
58                          cursor.execute(variant_id_select_query, record_to_insert)
59                          variant_id = cursor.fetchone()
60
61                          if variant_id:
62                              variant_id = variant_id[0]
63                          else:
64                              print("SELECT query did not work properly")
65                              sys.exit()
66
67                          if row["QUAL"] == ".":
68                              print("changing QUAL from ", row["QUAL"], "to 0")
69                              qual = 0
70                          else:
71                              qual = row["QUAL"]
72
73                          record_to_insert = (parquet_folder_name, variant_id, qual)
74                          cursor.execute(specific_variants_insert_query, record_to_insert)
75
76          connection.commit()
77          te = time.time()
78          print("Time for file ", parquet_file_path, ":", te - ts)
79          print("Total time: ", te - master_start_time)
80
81      except (Exception, psycopg2.Error) as error:
82          print("Failed to insert record successfully into table uv or sv table: ", error)
83          sys.exit()
84
85      return True
86
87
88  # Input:
89  # python parquetTOpostgres.py parquet_file_path gene_list_file_path skip_file_insert start_from_file
90
91
92
93  # Getting the path to the parquet file from input
94  origin_path = sys.argv[1]
95  gene_path = sys.argv[2]
96  if (sys.argv[3]) == "True":
97      skip_file_insert = True
```

```
else:
    skip_file_insert = False
start_from_file = int(sys.argv[4]) + 1


genes = pd.read_csv(gene_path)
gene_list = []

for index, row in genes.iterrows():
    gene_dict = {
        "NAME": row["NAME"],
        "CHROM": str(row["CHROM"]),
        "START": row["START"],
        "END": row["END"],
    }
    gene_list.append(gene_dict)

gene_list = sorted(gene_list, key=lambda i: (i["CHROM"], i["START"]))

print("Input gene list:")
for gene in gene_list:
    print(gene)

PRINT_EVERY_N = 100000

master_start_time = time.time()

logging.basicConfig(filename='transfer.log', filemode='a', format='%(name)s - %(levelname)s - %(message)

# Starting connection with postgres database through psycopg2
try:
    connection = psycopg2.connect(user="postgres",
                                  password="e_kW=Z8.loP062-xVM07H8lg3y3Eg5",
                                  host="tp1crio6pixfyuf.cf1yoyudwvis.eu-west-1.rds.amazonaws.com",
                                  port="5432",
                                  database="postgres")
    cursor = connection.cursor()
except (Exception, psycopg2.Error) as error:
    print("Failed connection to database:", error)
    sys.exit()

origin_directory = os.fsencode(origin_path)

# Looping over the parquet directories
for parquet_folder in os.listdir(origin_directory):
    folder_timer_start = time.time()
    parquet_folder_name = os.fsdecode(parquet_folder)

    # adding parquet_folder_name to files table
    if not skip_file_insert:
        try:
            postgres_insert_query = """INSERT INTO public.files(file_name) VALUES (%s)"""
            record_to_insert = (parquet_folder_name,)
            cursor.execute(postgres_insert_query, record_to_insert)

            connection.commit()
            print("Filename", parquet_folder_name, " inserted successfully into table: files")

        except (Exception, psycopg2.Error) as error:
            print("Failed to insert record successfully into table: files", error)

    parquet_directory = os.fsencode(origin_path + "\\" + parquet_folder_name)
```

```
160     print("Parquet folder: ", parquet_folder_name)
161 
162     # Looping over the files making up the total parquet file
163     for file in os.listdir(parquet_directory):
164 
165         filename = os.fsdecode(file)
166         if filename.endswith(".parquet"):
167             print("Starting file: ", filename)
168             parquetTOpostgres(origin_path + "\\" + parquet_folder_name + "\\" + filename)
169             logging.info(filename)
170         else:
171             print(filename, " is not a parquet file.")
172 
173     folder_timer_end = time.time()
174 
175     print("Successfully inserted all variants from: ", parquet_folder_name)
176     print("Total folder time: ", folder_timer_end - folder_timer_start, "s")
177 
178 master_end_time = time.time()
179 print("Finished with all parquet folders")
180 print("Total time for all folders: ", master_end_time - master_start_time)
181 
182 
183 # closing database connection.
184 if connection:
185     cursor.close()
186     connection.close()
187     print("PostgreSQL connection is closed")
```