Graph Databases for storing & analysis of genomes and pangenomes

Author: Markus Beicht

Co-authors: Julius Krämer, Thomas Rattei (supervisor), Fatih Sarigöl and Oleg Simakov

University of Vienna, Course ID: 2021W 053531

1. Abstract

In this project the viability of using graph databases, more accurately the graph database Neo4j, for storing and analysis of genomes and pangenomes was tested.

Therefore, performance tests between Neo4j and the relational database Microsoft SQL Server (MSSQL) measuring the used storage space as well as the query runtime two different queries, a simple query with many data records output and a more complex query requiring relationships between the data, were done. The Results of the performance tests show that Neo4j used up more storage space than MSSQL. The MSSQL database performed the simple query a lot faster than Neo4j; however, Neo4j was faster for the complex query.

Moreover, a workflow to implement sequence data into a Neo4j database instance was created. Starting from sequence data in FASTA format, which is processed by a python3 script, CSV files are produced, which are then loaded into Neo4j, in order to store the sequences in a graph database and perform queries on them. The sequences were successfully implemented in two different structures, as a De Bruijn graph using overlapping k-mers and a pangenome graph, in which identical sequence regions were condensed into a single node, in order to reduce the amount of required storage space.

This project found that Neo4j has a lot of potential as a replacement for relational databases in certain biological and bioinformatical topics, particularly when working with highly connected data and complex networks, for instance, biochemical pathways, metabolic interactions or genome structure data. However, some of the limitations of Neo4j were discovered as well, namely that CSV files larger than 10 megabytes could not be added to an already filled database instance. Therefore, Neo4j does not seem suitable for storing whole genome sequence, until this problem is solved.

Keywords: neo4j, pangenome graph, graph databases, relational databases, Microsoft SQL server, database performance

2. Introduction

Given the rapidly increasing amount of biological data due to modern sequencing and sequence analysis methods it is more important than ever to store all this data in an efficient and easily manageable way.

Large amounts of data can be stored in database systems (DBS). A DBS consists of a database, an organized collection of structured information, which is usually accessed electronically, and the database management system, which is a software used to store, manage and query the data in the database.

There are several different types of database systems, which can be differentiated based on how the data they contain is stored, connected and accessed. The most commonly used type of database are relational databases; however, storing the data in a network graph, as it is done in graph databases, might be more suitable for many biological applications, particularly if the data is highly connected.

2.1. Relational Databases

Relational databases store data in relational tables with rows and columns. Each column in a table specifies a certain attribute, while each row represents a data record. The data is connected by defining relationships between entire tables, which is achieved by using keys.

A single column or a certain combination of columns can be defined as a key in a table.
There are several different types of keys which are used to uniquely identify the data records, as well as distinctly specify the connections between the tables and consequently the data records.

The primary key is a column containing a unique value for each row in a table and its function is to uniquely identify each data record. Each table can only have one primary key; however, the primary key can consist of several different columns. While a primary key is not required to be a certain data type, integer values are generally the preferred choice.

Foreign keys are used to connect two tables, since they define a column which corresponds to the values of the primary key of a different table. In contrast to the primary key, the foreign key does not have to be unique for each row in the table.

The use of unique primary keys and corresponding foreign keys in a relational database defines a one-to-many relationship. This means that each row in a table A uniquely defined by a primary key can correspond to zero, one, or multiple rows in a table B with the corresponding foreign key; however, each row in table B can only correspond to a single row in table A.

The relationships between the tables in consequently the structure of the entire relational database can be described and visualized using a unified modelling language (UML) diagram. “The UML gives us the ability to model, in a single language, the business, application, database, and architecture of the system. By having one single language, everybody involved can communicate their thoughts, ideas, and requirements.” (S. Yin and I. Ray, 2005)

In order to optimise the performance of relational databases indices are used. Indexing improves the query time at the cost of requiring additional storage space. It does so by bringing the unordered data in a table into an ordered structure. This improves the runtime of queries because it limits the number of rows which must be searched through, reducing the effort from a linear to a logarithmic time complexity. More precisely, indexing sets up a B-tree, which is a tree data structure that maintains the data in an order and enables searches and other operations in logarithmic time.

Relational databases are particularly suitable when storing and analysing large numbers of data records. They are easy to manage and maintain, given that the data structure is not required to change. Some of the most popular examples of relational databases are Microsoft SQL server, MySQL and Oracle Database.
However, some of the limitations of relational databases are that they have a rigid database structure, which makes expanding an established data model difficult. Furthermore, complex queries containing data from many different tables requires connecting the data at query time with lots of joins between the tables, resulting in very complicated queries and drastically increasing runtimes.

2.2. Graph Databases

On the other hand, graph databases store data as a network graph containing nodes, edges and properties. All the individual data records are stores as nodes, which contain direct pointers, relationships, to all nodes they are connected to. The relationships between nodes can be bidirectional; however, Neo4j only supports unidirectional relationships. Storing data as a network graph allows for efficient many-to-many relationships, meaning that datapoints of the type A can correspond to multiple datapoints of type B and vice versa. Additionally, the type of relationship is defined in the connection between the data points.

The big advantage of graph databases compared to relational databases is that they do not have to compute the relationships between the data at query time, because the relationships are already defined in the way the data is stored. Therefore, complex queries with highly connected data could be significantly faster than relational databases.

2.2.1. Neo4j

One such graph database is Neo4j, which was used in this project. In Neo4j all nodes require an ID, a unique identifier, similarly to the column used as a primary key in a relational database table. Secondly, each node requires a label, which defines the node-type of the node. Additionally, nodes can have multiple properties, additional information belonging to the node, comparable to further columns in a relational table.

All relationships require a start-ID, end-ID and a type-value in Neo4j. The start-ID and end-ID specify between which nodes a relationship is as well as its direction. The type of the relationship also has to be defined in order to describe the kind of connection between the nodes. Moreover, a relationship can also have several properties, which contain additional information associated to the relationship.

The query language used in Neo4j is Cypher, a declarative graph query language which was developed specifically for Neo4j. It was inspired by SQL but optimized for querying graph databases, which requires a focus on the relationships between node types.

Neo4j can be used per command line by using the cypher shell, a command line tool which is part of the default installation of Neo4j and contains all functions for performing queries or administrative work on the database instance. However, the Neo4j browser, a free graphical user interface, additionally allows the user to visualize the data as a graph. The functions and tools for Neo4j are described in detail in the resources, user-manuals and tutorials at “neo4j.com”.

2.2.2. Genome graphs

There are several different types of graphs, which could be used when working with genome sequences. For instance, De Bruijn graphs are used in bioinformatics in the assembly of genomes based on shorter sequence reads. This is done by splitting the sequences into smaller pieces of a certain length, their k-mer length, evaluating the overlap between the individual sequence pieces and joining them together accordingly. A sample De Bruijn graph is visualized in figure 1 below.

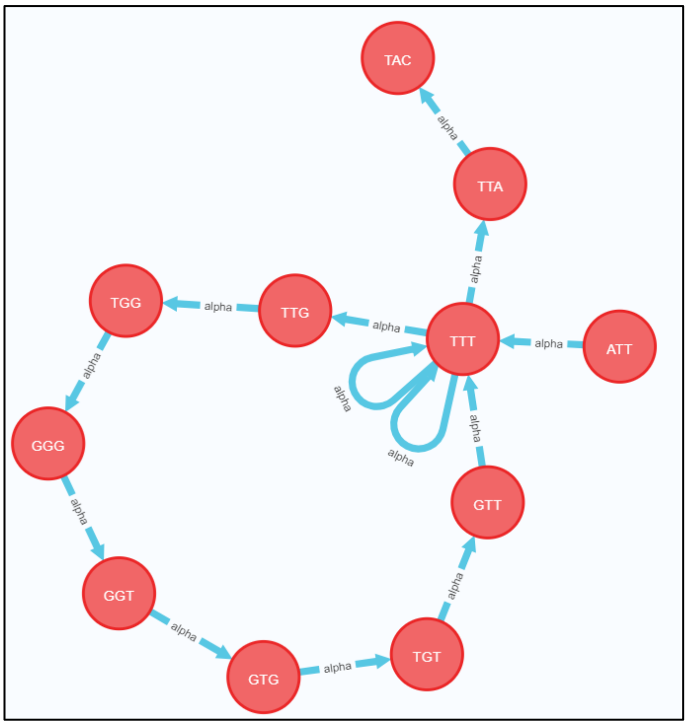

Figure 1: De Bruijn graph containing nucleotide sequences with k-mer length 3 visualized in the Neo4j browser.

A De Bruijn graph can also be created from a whole genome sequence by splitting the sequence into overlapping pieces. In Neo4j nucleotide sequences can be stored as a De Bruijn graph by storing all unique k-mers as nodes and defining the path through the graph by using indexed relationships between the nodes.

However, this is very inefficient for storing whole genome data, since the nature of using overlapping k-mers inevitably requires the storage of more data than the initial sequence contained, as well as the definition of the relationship to the next k-mer node in order to replicate the genome sequence.

A more efficient way of storing genome sequence data in a graph database is a directed pangenome graph approach, which stores the matching nucleotides between several sequences once and only the differing nucleotides multiple times. Consequently, multiple, ideally relatively similar whole genomes can be stored far more economically. A version of this approach is visualized in figure 2 below, as described in previous studies. (Bhanu Gandham, 2020)

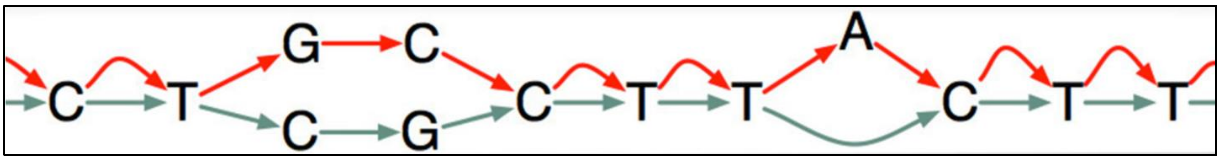

Figure 2: Pangenome graph visualization, which reduces the number of required nodes for storing two similar sequences by joining matching nucleotides into a shared node. (Bhanu Gandham, 2020)

While this directed pan genome graph implementation significantly reduces the number of nucleotides, which must be stored when aligning two or more genomes, it requires the creation of as many relationships as the whole sequence length when storing it in a graph database. Consequently, the number of nodes and edges in a resulting network graph describing several whole genome sequences would be at least as high as the length of the longest genome. Therefore, an additional adaptation can be made to allow a more efficient graph structure in terms of its used storage space. Consecutive matches, as well as consecutive mismatches, between the sequences can be grouped together in order to reduce the number of nodes and relationships significantly, as can be seen in figure 3 for multiple sequences. (Eizenga JM, 2020)

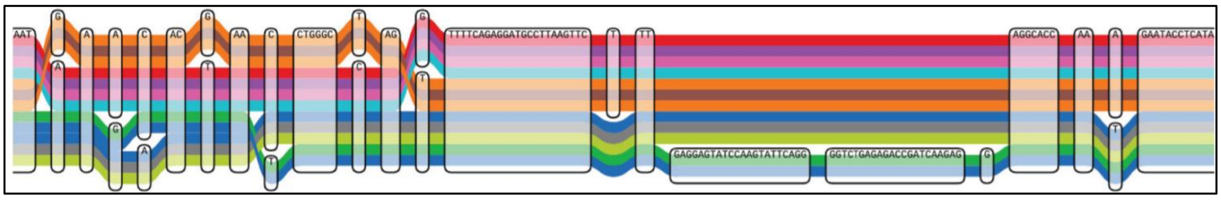

Figure 3: Advanced pangenome graph visualization of multiple sequences, which additionally groups continuously matching or mismatching nucleotides into a single node. Visualized using Tube Map. (Eizenga JM, 2020)

3. Performance Tests

In order to evaluate the performance of graph databases compared to relational databases several performance tests were conducted on the graph database Neo4j and a Microsoft SQL server (MSSQL) as a sample relational database.

The used storage space was determined, as well as the runtime for queries. Two different queries, a simple and a more complex one, were performed in order to determine how the query complexity effects the efficiency of the databases.

3.1. Data used in Performance tests

The data used in the performance tests was gene and chromosome data from two cephalopod species: Doryteuthis paleii (DORY) and Euprymna scolopes (EUP). The data was provided by Fatih Sarigol in the form of four csv files (table 1-4), which were generated with the software orthofinder.

The two files “DORYgeneChr” and “EUPgeneChr” contain the chromosomes, genes and gene positions of the respective species. Each row in the CSV file represents a gene and additionally contains information about the species it belongs to, on which chromosome the gene is and where it is positioned on its chromosome with start- and end-coordinates.
A few sample data records can be seen in the tables 1 and 2.

Table 1: Excerpt from the “DORYgeneChr” file, which contains the Doryteuthis paleii genes and their chromosome, start- and end-coordinates.

Species	**Chromosome**	**Gene**	**CoordinateStart**	**CoordinateEnd**
Doryteuthis	Dpe14	Dopeav2048957m	28997973	28998412
Doryteuthis	Dpe15	Dopeav2053769m	66093848	66105677

Table 2: Excerpt from the “EUPgeneChr” file, which contains the Euprymna scolopes genes and their chromosome, start- and end-coordinates.

Species	**Chromosome**	**Gene**	**CoordinateStart**	**CoordinateEnd**
Euprymna	Lachesis_group11	cluster_27022	39094430	39095758
Euprymna	Lachesis_group11	cluster_23308	68834427	68894052

The other two files contain information about the chromosome homology (“dory_eup_orth_chrs”) and the gene orthology (“dory_eup_orth_genes”). Each row in these files contains a Doryteuthis chromosome/gene and an Euprymna chromosome/gene which is orthologous to it. A few sample data records can be seen in the tables 3-4 below.

Table 3: Excerpt from the "dory_eup_orth_chrs" file, which contains the homologous chromosomes of Doryteuthis paleii (left) and Euprymna scolopes (right).

DoryCHR	**EupCHR**
Dpe11	Lachesis_group5
Dpe02	Lachesis_group0

Table 4: Excerpt from the "dory_eup_orth_genes" file, which contains the orthologous genes of Doryteuthis paleii (left) and Euprymna scolopes (right).

DoryGENE	**EupGENE**
Dopeav2057982m	cluster_8033
Dopeav2073714m	cluster_15613

3.2. Setting up the databases

For the Neo4j database instance the version "AuraDB Free" was used. It is a free version of Neo4j suitable for small development projects and learning to work with graph databases. The data was then implemented using the LOAD CSV command, which supports the upload of CSV files up to a size of 10MB and allows defining nodes, relationships and properties from the files.

Each Gene was stored as a node with five properties: the gene name, the species it belongs to, the chromosome the gene is on and the start and end-coordinate on the chromosome. Additionally, all chromosomes are stored as nodes containing the chromosome name and the species name as properties. Moreover, three relationship types were defined: a gene is positioned on a chromosome, a Euprymna chromosome can be homologous to Doryteuthis chromosomes and a Doryteuthis gene can be orthologous to Euprymna genes. These relationships have to be defined unidirectional, because Neo4j does not allow bidirectional relationships; however, this does not cause a problem, since the filter in the queries can be adjusted accordingly.

For the Microsoft SQL server the free developer version was used to provide a fair comparison to Neo4j. Before the data was implemented into the database it was pre-processed using a python3 script in order to add unique "identCode"-columns and corresponding "GeneID"- and "ChrID"-columns, which enable the distinctive connection of data records between the four tables in the database. This is necessary because a gene can be orthologous to more than one other gene.

The processed CSV files are then loaded into the MSSQL server, and the "IdentCode"-column of each table is defined as the primary key of that table. The UML (unified modelling language) diagram, which describes the structure of the MSSQL database instance and particularly how the different tables are connected, can be seen in figure 4 below.

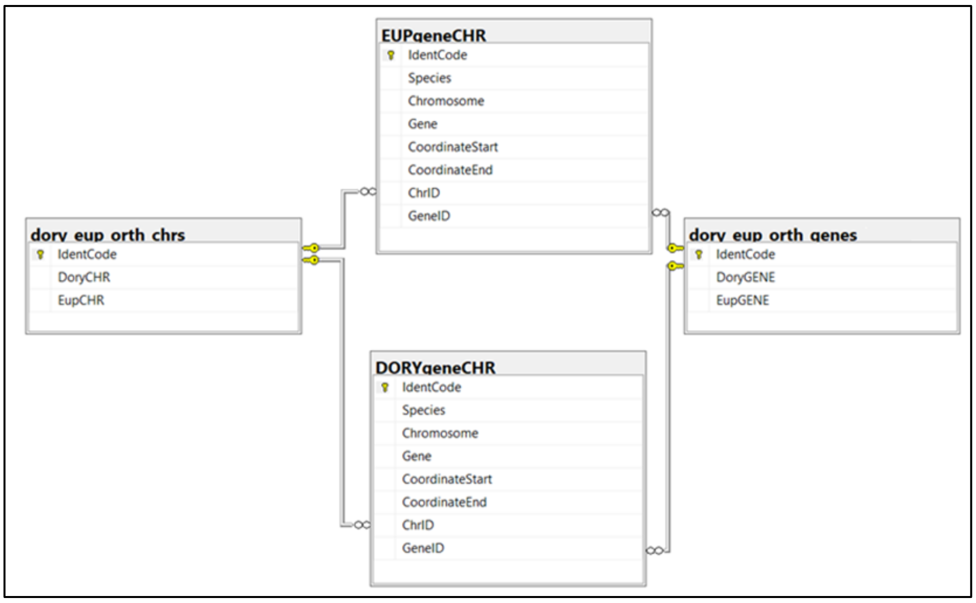

Figure 4: UML (unified modelling language) diagram describing the structure of the implemented MSSQL server and the relationships between the tables.

3.3. Methods

In order to compare the performance of the graph database Neo4j to the relational database MSSQL three different tests were performed: an evaluation of the storage space used by the database instances for the data described above and the runtime of two different queries when executed 1000 times.

3.3.1. Storage space used

The used storage space was determined in two different ways for the databases because of varying possibilities within the database systems.

For Neo4j the official online hardware sizing calculator was used to determine the estimated graph size. Therefore, the number of nodes, number of relationships, properties per node and properties by relationship must be entered into the calculator. The number of nodes, relationships and properties in the Neo4j database instance can be seen in the Neo4j user interface or determined by a query, asking for a count of all nodes or relationships. (https://neo4j.com/hardware-sizing-calculator/)

The used storage space of the MSSQL server was determined with the transact SQL command "EXEC sp_spaceused @updateusage = N'TRUE';". This command displays the space usage information of the current database after updating it to the latest database status in order to ensure the most recent information.

3.3.2. Query runtime

Two queries were performed in order to compare the efficiency of the databases under different circumstances, more accurately, a simple query which returns lots of data records and a more complex query which requires several relationships between the data. In Neo4j it is possible to return the results as a table the same as in MSSQL and other relational databases. Additionally, the results of a query can be returned and visualized as a network graph as well. The queries were tried both as a table and a graph in Neo4j; however, only returning the result as a table provides a fair comparison to the MSSQL server.

The first, simple query returns all genes and chromosomes of Doryteuthis. The second, more complex query returns all genes of Doryteuthis which are not on the eleventh Doryteuthis chromosome but are orthologous to an Euprymna gene which is on the fifth chromosome of Euprymna. For context, the fifth Euprymna chromosome and the eleventh Doryteuthis chromosome are homologous to each other.

The runtime of the queries was then determined using two python3 scripts, one for Neo4j and one for the MSSQL server. These scripts create a connection to the database instances using the specific database drivers and perform the query 1000 times. For Neo4j the GraphDatabase driver from the python module neo4j was used while the pymssql driver was used for the MSSQL server. The runtime of the queries was then determined by measuring the current time immediately before and after the execution of the query. The results were not printed to the command line in order to limit the measured time to the execution of the queries unaffected by the time it takes to print thousands of results to the command line.

3.4 Results

The database tests were performed as described above in order to compare the performance of the graph database Neo4j to the relational database MSSQL. The results evaluating the used storage space and runtime of queries with varying complexities are documented below.

3.3.3. Storage space used

Implementing the dataset into Neo4j results in a network graph containing 61538 nodes, 72081 relationships and 5 properties by nodes, which are 17.3 MB according to the calculator. However, 218202 bytes can be subtracted, 41 bytes per property according to the Neo4j developer guide, because the chromosome-nodes only have two properties. Consequently, the Neo4j database instance requires about 17.1MB to store the dataset.

In the MSSQL server the data is stored in four tables, which are connected to each other using keys. The transact SQL command "EXEC sp_spaceused @updateusage = N'TRUE';" returns the storage space of the database in several categories: reserved (allocated) space, data, index_size and unused space. The total amount of space used for the data and indices is 10480 KB, which is nearly seven megabytes less space than Neo4j used; however, MSSQL automatically allocates more space to the database files and the total allocated space for the database is 14256 KB.

3.3.4. Query runtime

Performing the simple query, which asks for all genes of Doryteuthis, returns 35416 data rows. It required 52.95 seconds for 1000 executions in Neo4j when the results are returned as a table and 76.71 seconds when returned as a network graph. In contrast, 1000 executions of the simple query in MSSQL required only 0.69 seconds. The visualization of this query as a graph in Neo4j can be seen in the figure 5 below.

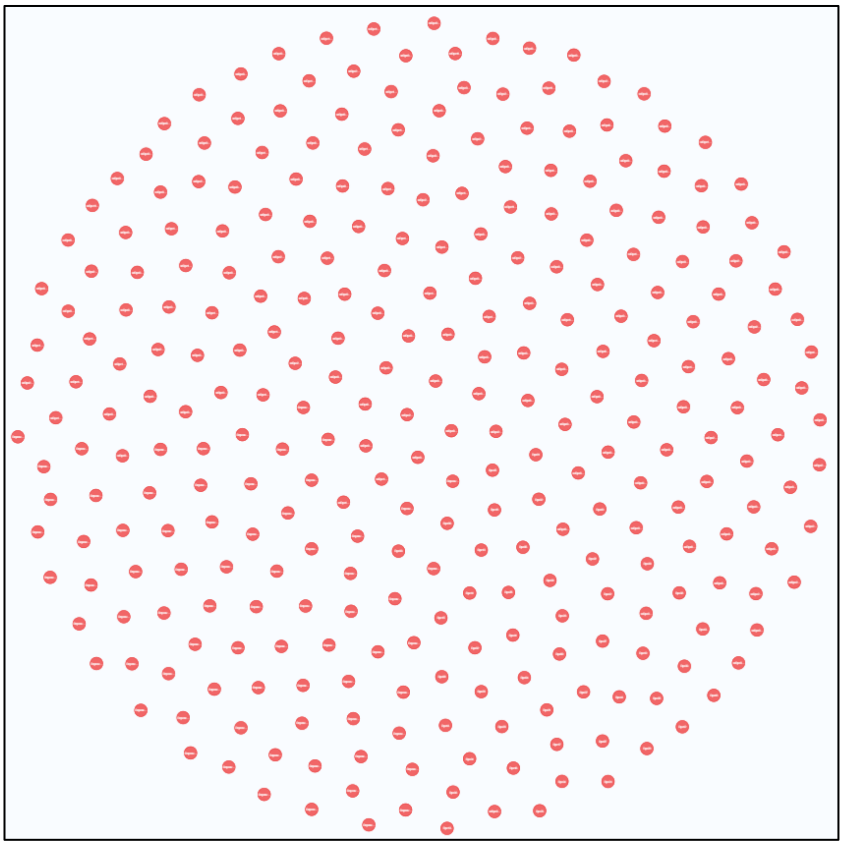

Figure 5: Neo4j visualization of the simple query described above. Each orange dot represents a gene of Doryteuthis stored as a node with its five properties: gene name, species name, chromosome name, start- and end-position of the gene on its chromosome. The data in the graph is not connected, because there are no relationships defined between the individual genes of Doryteuthis.

The more complex query, which asks for Doryteuthis genes on the eleventh chromosome which are orthologous to Euprymna genes on the fifth chromosome, returns 32 rows. It took Neo4j 11.61 seconds to return the query 1000 times as a table and 20.50 seconds as a graph. The MSSQL server took 20.47 seconds for 1000 executions of the more complex query. The visualization of the second query as a graph in Neo4j can be seen in the figure 6 below.

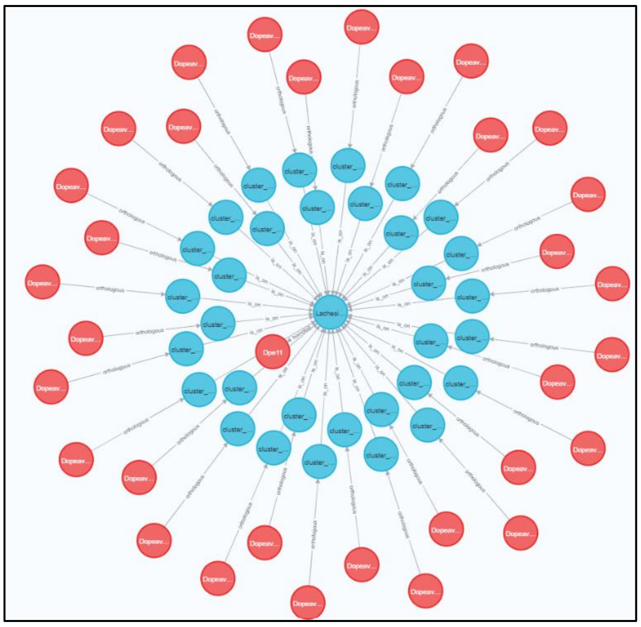

Figure 6: The nodes on the outside arranged in two circles are the returned genes, Doryteuthis in orange Euprymna in blue. The two dots in the middle are two chromosomes, more accurately, the eleventh Doryteuthis chromosome (orange) and the fifth Euprymna chromosome (blue). Additionally, there are three different types of relationships connecting the nodes: the two chromosomes are homologous to each other, the Euprymna genes are positioned on the fifth Euprymna chromosome and the Doryteuthis genes are orthologous to the respective Euprymna gene.

All the results of the database performance tests are summarized in table 5 below.

Table 5: Database performance test results of Neo4j and MSSQL. The first storage space used value for MSSQLis the allocated space, while the value in brackets is the space used by the data including indexing. The values in brackets for the Neo4j query runtimes are from returning the results as a network graph instead of a table.

	Neo4j	**MSSQL**
Storage space used (MB)	17.1	14.256 (10.480)
Simple Query runtime (s)	52.95 (76.71)	0.69
Simple Query runtime (s)	11.61 (20.50)	20.47

4. Storing Sequence Data in Neo4j

The chapter below describes the entire workflow of how sequence data was implemented in a Neo4j database instance. Therefore, FASTA files are processed by a python3 script, and the resulting CSV files are loaded into theNeo4j database, stored as a network graph and then queried.

This workflow supports both nucleotide and amino acid sequences, as long as they are in the FASTA format. The number of sequences specified in the FASTA files are also not limited. For the implementation, querying and analysis sequence data of SARS-CoV-2 was implemented, more accurately, four different complete genome FASTA files were used. The GenBank accession numbers and the colloquial names of these SARS-CoV-2 genomes are summarized in the table below.

Table 6: SARS-CoV-2 variants used with their GenBank accession number.

Variant name	**GenBank accession number**
Alpha	BS001490
Gamma	OM095212
Delta	OM348664
Omicron	OM324054

4.1. Python3 programs generating CSV files

Two programs with different approaches were developed, one implements the sequences as a De Bruijn graph and the other as a pangenome graph. Both programs use FASTA sequence files as input and generate CSV files which are structured and formatted in such a way that they can be imported into a Neo4j database instance without any further adaptations by the user.

4.1.1. De Bruijn graph

The De Bruijn graph implementation requires the input of one or more FASTA files. Additionally, the user can specify the desired k-mer length, alternatively the default length of k=3 is selected.

The program splits the sequences in the FASTA files into overlapping k-mers. Based on these k-mers, the script determines all distinctive k-mers in the sequences and stores them in a CSV “node” file, which will specify all the nodes in the Neo4j network graph. Moreover, one “relationship” CSV file is created for each entered FASTA file. These relationship files contain the information needed to define the connections between the nodes in the network graph in Neo4j. This is achieved by specifying the start- and end-node of each relationship, as well as the relationship types and the position in the sequence. The latter is necessary in order to allow the correct traversing of the network graph in Neo4j according to the initial sequences.

While the algorithm used to determine the nodes and relationships for the De Bruijn graph implementation is simple, the size of the CSV output files provided a serious problem. The file containing the nodes is very small, since it only contains all the unique k-mers, 64 at mot when using the k-mer length of three; however, the relationship files are a lot larger than the initial FASTA sequence files. For instance, the relationship CSV file of the SARS-CoV-2 alpha variant with k=3 has a size of 631 KB, about 30 times more than the initial FASTA file. While this does not provide a problem for shorter sequences such as viral genome, it becomes a serious issue, particularly with eukaryotic genomes.

4.1.2. Directed pangenome graph with grouped nucleotides

The implementation as a pangenome graph takes two FASTA files as input and creates two CSV files, one containing the node information and one containing the relationship information for Neo4j.

It does so by first aligning the sequences, which is done by comparing the two sequences one nucleotide at a time. The alignment algorithm determines for each nucleotide position whether the two sequences are matching or not and correspondingly stores an alignment value. In case of a mismatch, the number of matches in the next 1000 nucleotides is counted for frameshifts from zero to one hundred in both sequences in order to determine the shift with the maximal number of matches. Depending on this calculation the mismatch is considered as a mutation if the optimal shift is zero or as a gap in the alignment for one of the species.

After the alignment of the two sequences the individual nucleotides are grouped together depending on whether they belong to both initial sequences or only to one of them. This second algorithm also simultaneously adds these grouped sequences as nodes in the output CSV file as well as the relationships between these nodes in the second CSV output. As a result, these CSV files can be loaded into Neo4j and stored as a network graph in the graph. Consequently, regions where both sequences are equal do not require to be stored twice.

4.2. Importing data into Neo4j

There are several different ways of implementing data into a Neo4j database instance, among them two methods for CSV files, which were both tried an analysed for the import of whole genome sequence data into Neo4j.

The “LOAD CSV” command is used directly in the cypher shell in Neo4j. It allows the user to add data to any Neo4j database instance at any time, even when the database is currently active. The nodes and relationships must be manually defined when using this command, which enables the user to specify the structure of the implemented data depending on their needs; however, the “LOAD CSV” command is limited by its performance. CSV files larger than ten megabytes are not suitable for this command and therefore, the “neo4j-admin import” tool has to be used when working with larger files instead.

The “neo4j-admin import” tool has the capabilities to load the data in CSV files far larger than 10 megabytes into a Neo4j database instance. This tool requires the role of each datapoint to be specified beforehand in the header row of the CSV file and separate input files for nodes and relationships. More accurately, the nodes need to have an ID and a label, while the relationships require a relationship type and a start and endpoint as specified by the node IDs. Consequently, the CSV files must be organized properly, and the structure of the network graph has to be precisely planned beforehand. Unfortunately, the “neo4j-admin import” tool does not work on remote Neo4j database instance as well as already filled databases. Therefore, the upload of large amounts of data, such as whole genome sequences, is limited to local Neo4j database instances. Moreover, the database cannot be expanded by adding more whole genome sequences.

4.3. Performing queries

Once the data is implemented in Neo4j it can be analysed by performing queries and visualized when using the Neo4j browser.

In cypher queries on the data implemented as a De Bruijn graph it is possible to specify the species, chromosome and nucleotide position on the chromosome. Consequently, certain any regions on the chromosome can be specified and compared; however, the shift of two genome sequences, which is a result of not using an alignment between the different sequences, must be considered when searching for the start- and end-position of corresponding genome regions.

In figure 7 below, the visualisation of a network graph in Neo4j, which contains a short section of the surface glycoprotein of four different variants of SARS-CoV-2 can be seen. In this region a single mutation, the substitution of an adenine instead of a cytosine differentiates the alpha and gamma form the delta and omicron variants. Recreating the initial sequences for this section, which was also done by querying the network graph, reveals this difference as well: “CGGTAGCACACCTTGTAA” for alpha and gamma, but “CGGTAGCAAACCTTGTAA” for delta and omicron.

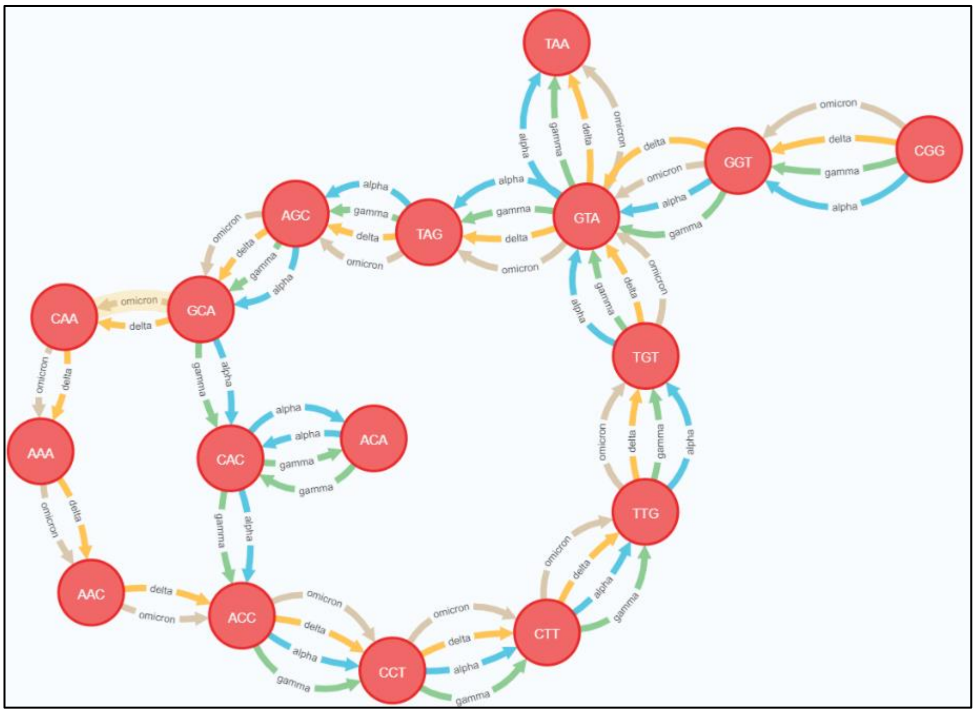

Figure 7: Neo4j browser De Bruijn graph visualization of a short sequence of the four SARS-CoV-2 variants: alpha (blue), gamma (green), delta (yellow) and omicron (brown).

Storing the sequences as a De Bruijn graph allows the nucleotide distributions to be determined by querying. For instance, when using a k-mer length of three, the trinucleotide composition can be returned in descending quantity. For the surface glycoprotein of the delta variant of SARS-CoV-2, which spans from position 21513 to 25328 of the genome, the most common trinucleotide is a triple-thymine with 161 occurrences; however, it has to be considered that the trinucleotides from all three frameshifts are counted.

In addition to the De Bruijn graph, the SARS-CoV-2 data, more precisely the alpha and gamma variants, were implemented in Neo4j as a directed pangenome graph with grouped sequences. The whole genome sequences are stored as a network graph with the following structure in Neo4j: each node represents one or more nucleotides, and these nodes are connected by two different types of relationships, one for each genome sequence. A visualization of the starting region of the gamma version of SARS-CoV-2 can be seen in figure 8 below. This query was performed by filtering for the first few nodes and only considering the relationships, which correspond to the alpha variant. The full sequences that are stored in each node cannot be visualized if they are too long; however, the data is still stored in that node. It has to be noted, that the first 202 nucleotides of the genome are stored in only four nodes and three relationships.

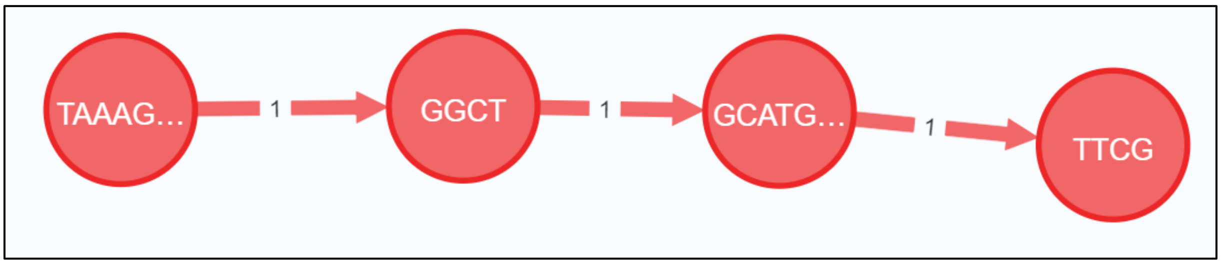

Figure 8: Directed pan genome graph containing the first 202 nucleotides of the SARS-CoV-2 alpha variant visualized in the Neo4j browser.

The information, which node belongs to which sequence is not stored in the nodes themselves, but in the relationship type. If both relationship types, the alpha (1) and gamma (2) variant are allowed, the resulting pangenome graph contains three additional nodes corresponding exclusively to the gamma sequence. A small sample graph with the same sequence region as in the example above is visualized in figure 9 below. It describes the genome sequences by adding the nucleotides defined in the nodes, which are passed through when walking through the network graph using the proper edges.

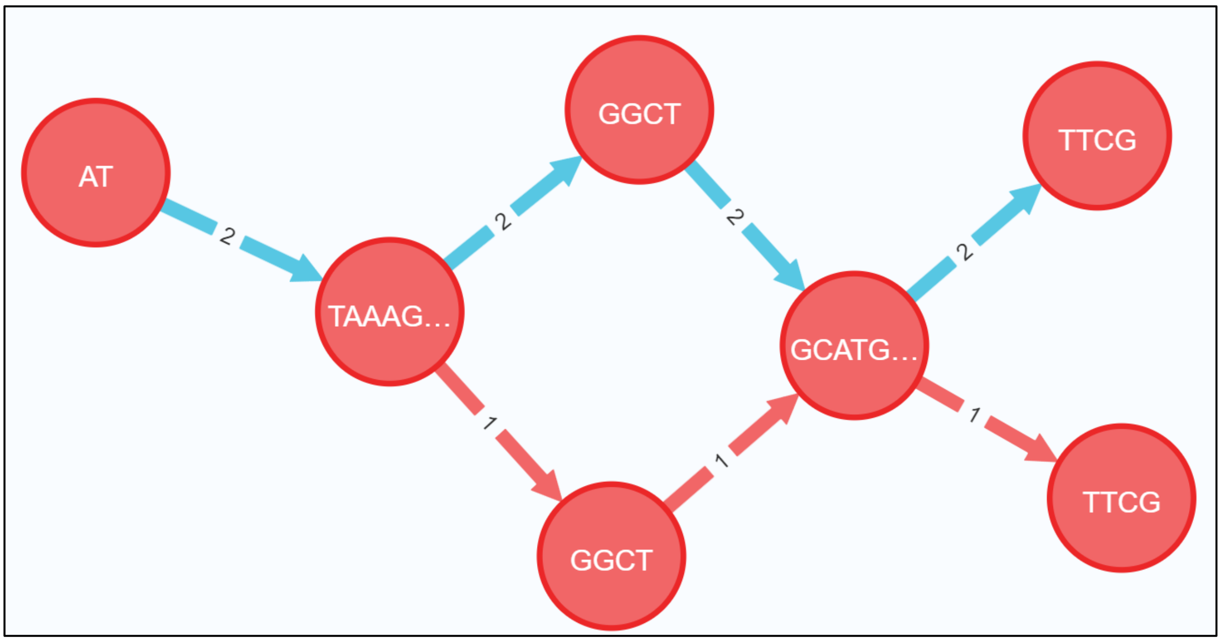

Figure 9: Directed pan genome graph containing the first 202 nucleotides of the SARS-CoV-2 alpha variant (blue edges, 2) and the first 204 nucleotides of the SARS-CoV-2 gamma variant (red edges, 1) visualized in the Neo4j browser. The three nodes at the top are part of the gamma variants sequence, the nodes at the bottom correspond to the alpha variant and the two nodes in the middle are part of both sequences.

5. Discussion

The database performance tests clearly showed the strengths and weaknesses of different types of databases for storing and querying connected biological data. The amount of storage space used in Neo4j was 17.1 megabytes, which is a bit higher than MSSQL (14.256 MB); however, because the MSSQL server automatically allocates space to its tables the difference was less than 3 megabytes, which is significant but not detrimental. The Neo4j database instance requires more storage space because the relationships between the data are already stored in the network graph itself, instead of being defined between tables using keys.

However, this "disadvantage" is also the biggest strength of graph databases like Neo4j, particularly when working with highly connected data. "Unlike other databases, Neo4j graph database doesn't need to compute the relationships between your data at query time. The connections are already there, stored right in the database." (https://neo4j.com/developer/graph-database/) Although the data model was rather small, containing only four tables, Neo4j already outperformed the MSSQL server at the more complex query, which contained all three different types of relationships defined in the data: homologous chromosomes, orthologous genes and which chromosome the genes are on.

On the other hand, for the simple query, which only asks for all gene data records without any relationships between the data, the performance of the MSSQL server really shines. While Neo4j required 52.95 seconds, MSSQL only needed 0.69s, which is a drastic difference. These results clearly show that relational databases were optimized in the past decades for working with high numbers of data records; however, the performance drastically decreases with each additional join operation. In comparison, the runtime of queries in Neo4j does depend on the connectedness of the data as much; however, Neo4j is not optimized for returning lots of data records as modern relational databases are.

While both developed python3 scripts for importing whole genome data in Neo4j are functioning implementations, it is clear that the grouped, directed pangenome graph approach is superior to the De Bruijn graph version. While the number of nodes is limited to 64 (all possible trinucleotides) in the De Bruijn graph when using a k-mer length of three, the number of relationships, 29831 for the alpha variant of SARS-CoV-2 alone, is far greater, which results in an increased amount of required storage space despite only using 64 nodes, which is contrary to the goal of implementing whole genome sequence data into a graph database. In comparison, the pangenome graph for the alpha and gamma variant contains only 2084 nodes and 2824 relationships.

Moreover, the De Bruijn python3 script itself has more stringent performance limitations. Even though the underlying algorithm, which creates overlapping k-mers and indexes them, is simple and fast, even for very large genomes, the output CSV files provide a big problem because auf their large size. For instance, the relationship file of the SARS-CoV-2 alpha variant in the De Bruijn approach is 631KB large compared to a 54 KB node file and a 33 KB relationship file for both the alpha and gamma versions processed as a directed pangenome graph. On the other hand, the algorithm used in the directed pangenome approach, which aligns the genome sequences, defines the relationships and groups the nucleotides accordingly, is more complex and slower, but the resulting CSV file sizes are manageable.

The developed workflow for storing sequences as grouped, directed pangenome graph is a useful starting point for storing whole genome sequences in the graph database Neo4j; however, several improvements and additional functions could be added. For instance, the python3 script, which generates the CSV import files based on FASTA files, is currently limited to using only two different sequences, which could be extended to allow multiple different genomes as input. Additionally, there is no function for FASTA files which contain several different sequences corresponding to multiple chromosomes or contigs yet, which aligns the chromosomes to each other, assuming that this information is known. Moreover, while the neo4j admin-import tool works well for local, unused Neo4j database instances the usefulness of the database is limited by the fact that sequence data cannot currently be added to a used database instance.

In conclusion, Neo4j is a free graph database which can be used for storing, querying and visualizing biological data. It performs well compared to other databases, particularly when working with highly connected data. Some of the most suitable biological use cases for graph databases like Neo4j are biochemical pathways, complex metabolic interactions, such as gene-protein-metabolite-phenotype relationships, or genome structure and gene orthology data.

6. Supplementary material

The scripts developed for this study, the queries performed in the databases performance tests and the Euprymna scolopes and Doryteuthis paleii CSV files are documented on the GitHub repository on "https://github.com/MarkusBeicht/software-developement-project".

7. Acknowledgements

I would like to thank Julius Krämer, Univ.-Prof. Mag. Dr. Thomas Rattei, Assoz. Prof. Dr. Oleg Simakov and Fatih Sarigol for their support and ideas during this project.

I want to especially thank my supervisor, Thomas Rattei for guiding me through this project and providing feedback on the methods used in this project as well as the presentations.

Moreover, I want to thank Julius Krämer for helping me with the database performance tests, including the setup of the databases and the database drivers.

8. References

S. Yin and I. Ray, "Relational database operations modeling with UML," 19th International Conference on Advanced Information Networking and Applications (AINA'05) Volume 1 (AINA papers), 2005, pp. 927-932 vol.1, doi: 10.1109/AINA.2005.294.

Eric J. Naiburg and Robert A. Maksimchuk, UML for database design, Addison-Wesley, Boston, MA, USA, 2001

Compeau PE, Pevzner PA, Tesler G. How to apply de Bruijn graphs to genome assembly. Nat Biotechnol. 2011 Nov 8;29(11):987-91. doi: 10.1038/nbt.2023. PMID: 22068540; PMCID: PMC5531759.

Chikhi R., Limasset A., Jackman S., Simpson J.T., Medvedev P. (2014) On the Representation of de Bruijn Graphs. In: Sharan R. (eds) Research in Computational Molecular Biology. RECOMB 2014. Lecture Notes in Computer Science, vol 8394. Springer, Cham. https://doi.org/10.1007/978-3-319-05269-4_4

Siavash Sheikhizadeh, M. Eric Schranz, Mehmet Akdel, Dick de Ridder, Sandra Smit, PanTools: representation, storage and exploration of pan-genomic data, Bioinformatics, Volume 32, Issue 17, 1 September 2016, Pages i487–i493, https://doi.org/10.1093/bioinformatics/btw455

Eizenga JM, Novak AM, Sibbesen JA, Heumos S, Ghaffaari A, Hickey G, Chang X, Seaman JD, Rounthwaite R, Ebler J, Rautiainen M, Garg S, Paten B, Marschall T, Sirén J, Garrison E. Pangenome Graphs. Annu Rev Genomics Hum Genet. 2020 Aug 31;21:139-162. doi: 10.1146/annurev-genom-120219-080406. Epub 2020 May 26. PMID: 32453966; PMCID: PMC8006571.

Bhanu Gandham. An interactive, Javascript-based browser for microbial pangenome graphs Bhanu version, https://observablehq.com, July 2020

8.1. Webservers

Neo4j resources, accessed 26th December 2021, <https://neo4j.com>

SARS-CoV-2 lineages, accessed 24th January 2022, <https://cov-lineages.org>

NCBI National Center for Biotechnology Information, accessed 24th January 2022, <https://www.ncbi.nlm.nih.gov>