Representation of Eukaryotic Pangenomes in a Graph Database

Mehmet Akdel
Wageningen University, Bioinformatics Department
Project Supervisor: Sandra Smit

ABSTRACT The introduction of New Generation Sequencing methods brought a recent leap in numbers of sequenced eukaryotic genomes. Currently, these sequences are stored as separate entities and sequence variation between closely related genomes are usually stored only relative to reference genomes. Due to the large size of eukaryotic genomes, inspection of variation between non-reference genomes is currently computationally demanding, making this kind of information hardly accessible to many biologists. In order to change the currently limiting sequence storage and representation methods, in this project, we proposed a reference free pangenome sequence representation of any given clade by compressing their genome sequences with their annotations into a single entity that inherently stores sequence variation. We achieved this by constructing a compressed de-Bruijn graph, relating this graph to existing annotation and orthology data and storing this highly connected data in a graph database. The results have shown that the database is scalable with apparent sequence data size compression and enables the user to retrieve biologically relevant information. These include investigation of orthology relationships between genes and deeper sequence level similarities between these genes.

KEYWORDS Pangenome; De-Bruijn; Graph; Genome; Database

1. Introduction

Today, biologists in the genomics field have access to a substantial number of eukaryotic genome sequences of closely related species and strains of the same species. Each of these groups of organisms have similar genome sequences but also have distinct phenotypes. A problem that is faced at this point is that it is becoming more and more exhaustive to compare and locate similar sequences, on these genomes, by straightforward approaches (Thorisson *et al.* 2009) (Marx 2013) . A pangenome representation of the data of this kind can be regarded as a possible approach to condense the sequences into a single entity, which would make the data more accessible and informative. A pangenome is previously defined as the similarity based representation of the total set of genes, which are present in a group of closely related species or strains of a single species (Rasko *et al.* 2008) . In a pangenome, the total gene set is divided into three groups: core genes, dispensable genes and unique genes. Core genes are shared by all genomes, dispensable genes are shared by a subset of the genomes and unique genes belong to a single genome . More recently, with the availability of a high number of genome sequences, pangenome was redefined by several research groups with a sequence based perspective where core, dispensable and unique gene sets were replaced by core, dispensable and unique sequences found in the multiple genome set (Marcus *et al.* 2014).

One of the applications of the use of a pangenome sequence representation approach could be in the applied plant sciences field. Genome sequence variation between domesticated plant species, including their wild counterparts, are under the focus of applied plant sciences projects (Baker 2012), (Zhang *et al.* 2015). Despite the highly redundant genome content in recently sequenced closely related plant variants, each one of them possess distinct set of desirable or undesirable phenotypes. This fact lends us the idea to compress down the sequence redundancy, which would bring forward the variable parts of multiple genomes that define the genotypic contribution towards distinct phenotypes. This would, in effect, enable us to represent the large amount of genome sequence data in the form of a sequence based pangenome which is more accessible for multiple genome sequence analysis .

Various tools are available for sequence-based inspection of multiple genomes. However, deep access and inspection of all this data for multiple purposes, such as finding genomic

variations and coding sequence homologies at sequence level, is computationally demanding. Graphs are used to represent multiple genomes as a single entity with the use of tools, such as SplitMEM and multiple genome alignment tools (Marcus *et al.* 2014) (Paten *et al.* 2011) . Other tools, which are based on Mummer, such as Panseq are used to obtain core, accessory and distinct segments of multiple genomes (Laing *et al.* 2010) . All these tools are directed towards prokaryotic genomes. Their use with eukaryotic sequence data pipes out overwhelming amount of data, which is usually structured as a graph. Currently, this output can only be accessed and investigated by loading and working with it in memory. With the consideration of eukaryotic genome size, it can be pointed out that the pangenome should be represented in a database to allow the users to access it without any memory constraints.

Several databases, which store sequence based variations in a single species exist (Lappalainen *et al.* 2013) . These platforms limit the perspective of the user with the point of view of a single reference genome, which is annotated in terms of sequence variations. Similar to structural variation annotations, functional annotations also exist for many sequenced genomes. However, these annotations are directed towards single genomes. Most of the sequenced genomes also contain functional annotations. Nearly all of the gene annotations have their homologous counterparts in the closely related genomes. This enables us to make connections between different genomes by using coding sequence homology relationships, which would make it more efficient to look for similar coding sequence segments between multiple genomes. Several pangenome inspection pipelines, including Panseq, functionality also extends to clustering orthologous gene sequences and represents these clusters as multiple genome sequence segments (Xiao *et al.* 2015) . However, this information is not represented in the form of a single interconnected entity but rather as a list of such segments and limits the biologist to directly ask questions such as "what are the nearest genes to an ortholog gene group in all genomes?". In addition to these pangenomics tools, Ensenble Genome Browser webtool allows the users to find out on orthologs accross a number of distantly eukaryotic species, however, the reference genomes of each species are used for each genomic representation whilst disregarding the functional annotation on other closely related genome sequences (Stalker *et al.* 2004).

Here, we present a scalable solution to compress multiple annotated genomes into a single pangenome representation. We construct a compressed de-Bruijn graph structure to store the genome sequences and their variation in a graph database. This storage allows us to handle large eukaryotic genomes. Functional information, such as gene annotations and orthology relationships, is integrated to the pangenome graph. We enable retrieval of biologically relevant information through a combination of de-Bruijn graph traversal, R-Tree index searches, and graph database relationship queries. In addition, sequence data can be extracted from the database in an efficient manner. The algorithms for construction, storage, and searching of the graph are described in section 2. M&M. An overview of the functionality, pan-genome applications, and performance are described in the Results.

2. Materials and Methods

The pangenome graph database construction pipeline was put together by using Python programming language. The pipeline involves construction of a python graph object (PGO) which contains the node and edge data, which is then imported into a graph database for storage and use (Figure 1). In this PGO, there are 5 main types of nodes:

1. Sequence nodes, which makes up the compressed de-Bruijn graph
2. Annotation nodes, which stores annotation data and relates to sequence nodes
3. Orthology nodes, which relates to sets of orthologous nodes
4. Segment nodes, which are used to locate nodes at given nucleotide positions
5. Index nodes, which makes up the tree-index graph and relates to annotation and sequence nodes

The input files fed into the pipeline are Fasta sequence files of all sequences that will be included in the pangenome graph, GFF files for genomes (optional) and OrthoMCL coding sequence group file (optional). The pipeline makes use of SplitMEM for compressed de-Bruijn graph construction.

Neo4j graph database management system (DBMS) was chosen as the graph database tool. The entire python graph object data is initially written into disk as a comma separated value (csv) file, which is formatted to be accepted by Neo4j Batch Import tool (see section 2.D. on database construction). The end product is a functional graph database that stores pangenome graph representation.

A. Pangenome sequence graph construction

Initial part of the pipeline involves compressed de-Bruijn graph construction from the complete set of genome sequences. Compressed de-Bruijn graph was chosen as the basis of the sequence pagenome representation, as it is able to show the core, unique and dispensable sequences of the pangenome (Figure 2).

SplitMEM was the tool of choice in constructing the pan genome graph, as it was the only tool available that was specifically made to construct compressed de-Bruijn graph representations of the pangenome. SplitMEM accepts sequences in a single fasta file. The only parameter for the graph construction is the minimum length of Maximal Exact Match (MEM), which indicates the minimum length of a possible shared sequence fragment between more than one genomes. This value will be annotated as k in the rest of the paper.

First phase of the pipeline functions to reconstruct the compressed de Bruijn graph compartment of the PGO by running SplitMEM and using its output file. This compartment of the graph will be notated as sequence graph in the paper. The output file of SplitMEM is in dot file format. A dot file is a type of graph description language written in text. Each line of a dot file contains either edge or node information. The project pipeline parses the dot file, extracting information required to reconstruct the graph as an initial PGO. This initial PGO only contains data on relationships between compressed de Bruijn graph nodes and assigned sequence indexes to nodes, according to the provided fasta sequence input. The PGO data is then extended by the further addition of chromosome/scaffold/contig IDs and sequences to their corresponding nodes. This is done by extracting node sequences and chromosome IDs from the sequence file by using the sequence indexes stored in each node. Thus, the sequence graph eventually contains, in addition to node relationship data, information on the node sequence and its position in the genome sequences.

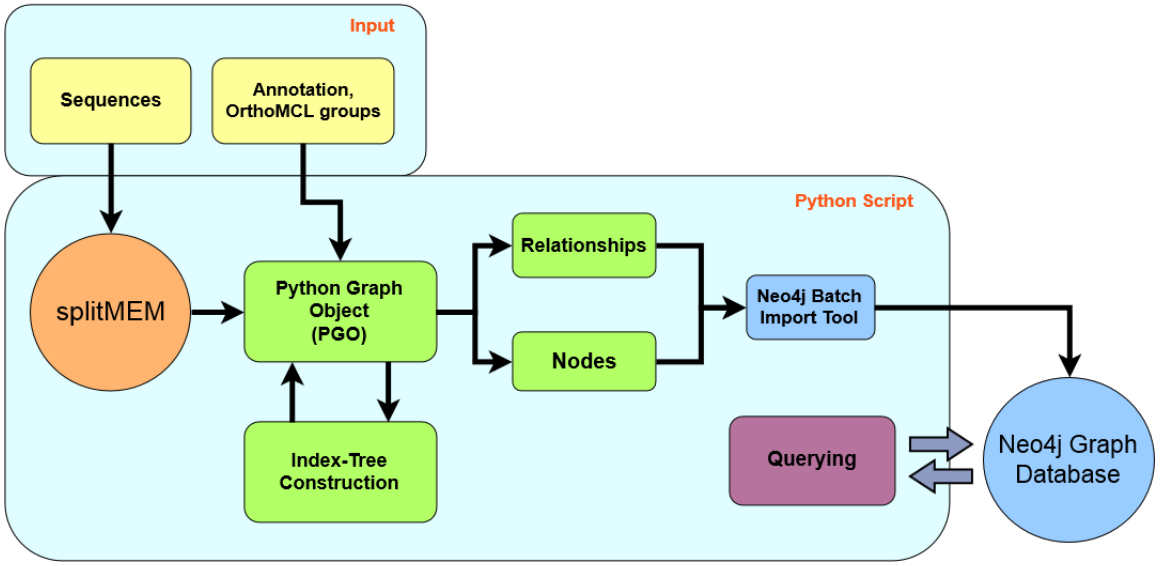


Figure 1 The diagram represents the work flow in the pipeline. In summary, the sequence input is taken to SplitMEM and processed into a compressed de-Bruijn graph output file (Dot file). The graph file, annotations and orthoMCL are then processed in python to build the python graph object and written into node and relationship csv files (green boxes). These files are then used as input to Neo4j Batch Import tool to form the complete Neo4j Graph database. Database can later on be queried by http requests through python.

B. Integration of Annotation

Next phase is to integrate annotation data into the PGO. At first, annotation nodes are formed with their feature IDs and nucleotide position ranges on the corresponding chromosomes' forward strand. This information is extracted in the script by parsing GFF file format which is provided beside the genome sequence file (fasta). Each node range is then used to relate to the correct sequence node. This is done by connecting the starting point of the range on the sequence graph by finding the sequence node that intersects with it. This edge is labeled as "start". Similarly, the end point of the range is connected to the sequence node that intersects this index. This edge is labeled as "end". A summary of the process of finding of "start" and "end" nodes is as follows: both the annotation and sequence nodes, corresponding to the same chromosome, are sorted in terms of their nucleotide position indexes. Then, starting from the first annotation, the sorted sequence nodes are iterated until the annotation position is found and marked. For the second annotation, the sequence node iteration starts from where it was left of and move forward to find the next annotation. As a result of "start" and "end" marks in the sequence graph, traversing from the start to the end node would lend us the complete set of nodes that lie between the annotation range (see 2.E.). As the annotation "start" and "end" positions usually lie inside a sequence node, the sequences in the beginning and ends is cut off, according to the stored annotation range, to extract the exact sequences.

The created annotation nodes were then further grouped into orthologs by parsing the data on previously formed gene groups from OrthoMCL output. OrthoMCL is a tool that takes translated coding sequences from a clade of interest as input and clusters them into highly homologous gene groups by using all *versus* all BlastP alignment method (Li *et al.* 2003) . Thus, by parsing the OrthoMCL output, orthology nodes are formed in the PGO and connected to their corresponding genes.

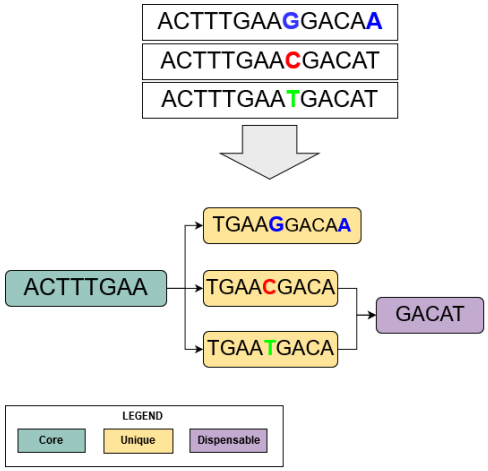


Figure 2 The diagram above shows the representation of 3 input sequences in a compressed de-Bruijn graph (k=5). The colored nucleotides represent single nucleotide differences.

C. Integration of Tree-Index

Tree index graph was built and integrated into the main graph object for two purposes: 1) accessing an annotation node and its related sequence nodes at a given nucleotide position and 2) accessing a nucleotide position in sequence graph. Tree indexing methods enable finding of position intervals that intersect with a coordinate input in an efficient manner. Initially, a currently available Neo4j add-on, called Neo4j-Spatial, was used in neo4j database itself to create and integrate a modifiable R-Tree index graph. However, due to the limitations on the particular add-on, a second approach, which involved construction and integration of the Tree-Index into the PGO, was used.

For the purpose indicated in first point, the annotations are grouped and their position intervals are used as input to con-

struct a one dimensional R-tree index (Martin 2007) . In summary, the one dimensional R-tree construction algorithm involves initially sorting nucleotide position intervals of the annotations, with respect to the chromosome, into an interval array. Then, the mid-points of these intervals are paired from either side of the interval array. Each of these pairs, which can be regarded as child nodes, are then packed into a single interval which will be represented as an internode. Thus, in the tree structure, the internode connects two of the child nodes. In the next step, the internodes are taken into account as the child nodes and packed again into the next level. This is repeated until only a single node called root node is left (Figure 3). In the resulting tree, each tree leaf corresponds to an annotation interval which (annotation node) would also point the start and end sequence nodes in the compressed de-Bruijn graph.

The chromosome sequences are not represented as single sequence entity in the sequence graph as each node contains only a part of this sequence. Reaching a nucleotide position on a given chromosome requires reconstruction of the continuous sequences while traversing. Therefore, the straightforward way to reach to a given nucleotide location involves traversing the chromosome graph from the beginning until reaching the particular position. Yet another way of achieving this is to iterate through all nodes that corresponds to the chromosome until the given position is found. Both of these ways require starting and working with the complete chromosome sequence graph, which renders them inefficient. To avoid this issue, a new set of nodes named "segment" nodes are introduced. Each "segment" node corresponds to a sequence interval, such as 50000 to 100000. Similar to annotation nodes, each segment node points to the beginning and end of the interval they correspond to on the sequence graph. Finally, segment node intervals are used as leaves to construct the index tree. The target sequence node, which intersects with the position of interest, could be reached by traversing from the start labeled sequence node onward and checking for the particular input coordinate (section 2.E.). In summary, the use of index tree focuses the position search on a short sequence segment instead of a complete chromosome.

In the end product, each one of the annotated distinct continuous sequences (chromosomes/scaffolds/contigs) would contain an index tree for its corresponding annotation nodes and an index tree for its segmentation nodes.

D. Graph Database Construction and Data Structure

Due to the highly interconnected nature of a pangenome sequence graph, the use of a native graph database for its storage is a logical choice. Native graph databases are known to perform faster in retrieving highly interconnected patterns of data and traversing graph data. Graph databases are index and schema free in which the data is stored in node and edge entities, which holds information where the edge or the node is linked to and with what type of relationship. Neo4j graph database management system (DBMS) was chosen to base the project upon, because of the following reasons:

1. Most widely used graph DBMS which have resulted in plenty of community based guidelines and extensive official documentation
2. Known and tested to be highly scalable
3. Database can be accessed by using a http port.
4. Has a user friendly query language, Cypher

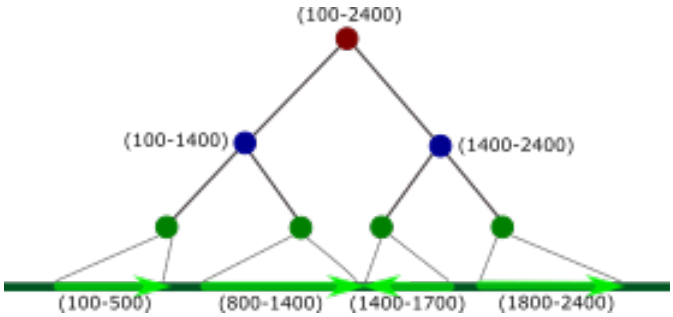
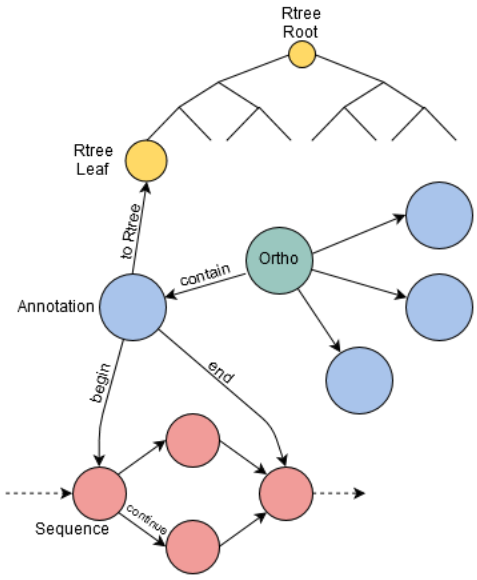


Figure 3 The diagram shows a simple one dimensional Rtree. The dark green line on the bottom represents a single chromosome and the light green arrows shows the annotation intervals on the current chromosome. On the index tree, the green nodes correspond to leaf nodes, which store the nucleotide intervals of annotations for the chromosome. The blue nodes are intermediate nodes which are made by packing closest neighbours. Lastly, the red node packs the complete range of annotation intervals.

In the current project, Neo4j Community Edition with version 2.2.5 was used. The default configuration files were used for the memory configuration and server configuration. It is necessary to have a basic understanding of the data units and how they are connected to each other in a Neo4j database to further elaborate on the data structure used in the current project. All Neo4j node entities hold information on which ingoing/outgoing relationship (edge) entities they extend to, and all relationships hold information on to which and from which node they point to. These nodes and relationships can also contain label and property entities. Properties hold data as keys and their corresponding values. Labels can hold string data which can be used to specify the type of node/relationship. Similarly, node and relationship entities also hold information on which properties and labels they are linked to and *vice versa* (Tobias 2015).

Here, in essence, the data stored in PGO was transformed into an essential set of properties which would allow the user to match useful information or traverse the sequence graph correctly. Similar to the PGO, six node labels, "annotation", "Sequence", "ortho", "indexTree", "flank" and "segment", were used to categorize the nodes according to the type of information they hold. This enables the user to send queries directly to the set of nodes which are under interest. For instance, if the user is looking for annotation with a specific gene/protein id, letting the database know that this id is found in one of the nodes that is labeled with 'annotation' limits the database to check only this group for the gene ID, thereby increasing the search speed. The relationships between the nodes were left identical to the PGO explained in previous sections. A visual summary of relationships between different node categories are given in Figure 4. Sequence nodes and relationships can be regarded as the most complex entity in the database since they hold the de-Bruijn graph structure. Traversing and extracting this sequence graph details was optimized by reducing the properties into a minimized set of properties that hold essential data required for these purposes. The end result of this data structure makes it possible to reconstruct the complete set of sequences with a single traversal job. These essential properties used in sequence nodes are sequence (stores sequence string), length (stores nucleotide length of the sequence) and chromosome/scaffold/contig names



(Figure 3)

Figure 4 The diagram shows relationships between different kinds of nodes.

which share the sequence (stores the position of sequence in corresponding chromosome/scaffold/contig). The complete set of properties and labels that are in the database are given in supplementary table 1 with the reasons of their inclusion.

The initial graph database was created by using an official command-line tool named Neo4j-import. At first, the PGO constructed in the python pipeline was written into csv text file. The format of the csv file was adjusted with regards to the Neo4j-import tool guidelines. Then, the csv files were used as input to create the database.

Neo4j database has a dedicated http port for retrieving data and traversing the graph. The database has a Representational State Transfer (REST) system through the http port. This means of access was essential for the project as this was the only currently viable way to access the database from any scripting language that allows http requesting. This have allowed us to set specific conditions for traversing the sequence graph and the tree index graphs. Cypher query language was also used in obtaining various answers from the database. Cypher is a powerful and user-friendly query language, which is specific to Neo4j DBMS. It is a powerful language in terms of matching specified patterns in the graph. In this project, in addition to pattern matching (section 2.E.), Cypher was mainly used as a way to access and assess the integrity of the database by checking if any flaws exist. An example of this sort is to check how many annotations exist and if they are correctly pointing the start and end nodes on the sequence graph.

E. Traversal and Querying

In the pangenome database, pattern matching queries are useful for purposes such as looking for specific genes and its orthologs. In this case the bridge structure, where two gene annotations are bridged by an ortholog node, is matched. This pattern is also filtered to only include the given gene ID. These queries were accomplished by using Cypher. Graph traversal methods are necessary in order to be able to make use of the de-Bruijn graph and the index tree components of the database. Therefore, correct traversal and pruning rules were set and queried in Neo4j transactions. These transactions were based on http requests to Neo4j REST interface. Thus, both cypher and REST methods were used in combination to obtain answers for biologically relevant questions to the database. This was implemented with a set of python functions, written in a modular fashion, that can be put together to build specific query methods. The functions implied, can be segmented to three parts. First part functions as the index tree search module where a given contig/scaffold ID with a position under investigation is given as an input and the annotation/segment node ID is given as an output. The second part of the functions starts with an annotation node ID and lends us the IDs of sequence nodes to which annotation node points to as 'start' and 'end', together with start and end positions in the corresponding sequence stored in the sequence node. The third set of functions are used to traverse sequence nodes from the 'start' to 'end' node (Figure 5).

To give a complete example to a type of request implied above, this query can be given: 'Return the sequence of an annotation at position x on genome y'. This query is answered by first requesting the R-tree root node ID dedicated to the annotation nodes of the given 'genome y'. After the root is grabbed, the tree index is traversed from the root to the next levels by selecting the branches, at every step, that leads to an intermediate node that intersects with the given position. This traversal ends when the final leaf node is reached. Each leaf node points to the annotation node of interest. Thus, the next step is to move to the annotation node. Annotation node contains 2 outgoing edges. First one points to the beginning and the second one points to the end sequence node. Thus, the beginning and end sequence node IDs are grabbed. Lastly, starting from the beginning node, the sequence nodes are traversed until the end sequence node, while collecting the node sequences at each step. It should be noted that there is only one correct step to take while traversing the sequence nodes. The selection of the correct path is achieved by a breadth first approach where the correct step is taken by checking the sequence positions of nodes which are continued from the current node until the node with the correct position is found. The next node should contain the position which corresponds to the current node length minus the *k-1* of length added to the current node position. The end result is a list of sequences in a serial order which are to be overlapped by the *k-1* of length to obtain the linear sequence (see Figure 5).

F. Database Performance and Scalability Tests

Testing scalability involved inspection of databases built by using increasing amounts of genome sequences of different yeast strains. Number of genomes used were: 5, 8, 11, 20, 25 and 30. In addition, 3 different *k* values (20, 45 and 60) were used. Performance test involved recording data retrieval speed using a complex query which combines the query methods, which include both tree-index search and sequence graph traversals in constructed yeast pangenome databases. The query used in this case was asked to return the sequence of a gene found in a chromosome at a given nucleotide position. The memory configuration of Neo4j was left in default parameter, "weak". This setting only allows a limited part of the database to be uploaded to the memory to increase query speed.

Input Data

Saccharomyces cerevicae: A collection of 31 completed yeast genome sequences were taken from internet based Saccharomyces Genome Database (Stanford University 2015) . This

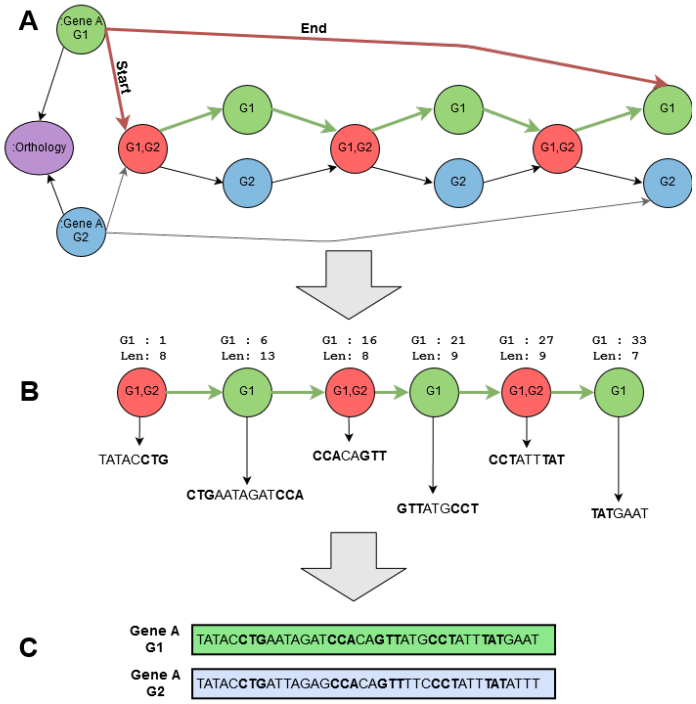


Figure 5 A) The diagram shows the GeneA genes for genome G1 and G2 which are grouped into an orthology node. If the sequence of GeneA in genome G1 was queried, the red lines, labeled "Start" and "End", are initially followed to mark the beginning and end nodes in the sequence graph. Then, traversal is started from the beginning node onward until finding the end sequence node. **B)** Here, the correct traversal path is shown. The "G1" keys, indicated above each node, has a position value of the particular node in genome G1. The "Len" keys store the length of the current sequence stored in the node. While traversing the path, the sequences are extracted from each node. **C)** The sequence of Gene A of genome G1 can be finally put together by overlapping each sequence fragment by *k-1*. Here the green box gives the sequence of GeneA of genome G1.

set included the reference strain, *S288C*. All sequences, except reference strain, consisted of varying numbers of contigs. The reference strain was fully assembled into chromosomes. All of the sequences were also annotated and paired with their corresponding annotation files. The gene naming in yeast was based on similarity between different strains. In other words, the strains that share a gene uses the same gene name followed by strain name. This enabled extracting orthology group information directly from parsing the GFF files.

Beauveria bassiana: 6 available genome sequences were used in *B. bassiana* pangenome construction. Two of the genomes were of strains B-15 and ASEF 2860. The rest of the 4 genomes were sequenced and assembled into scaffolds in Wageningen University. All genome sequences were annotated and OrthoMCL ortholog gene clusters were formed by using all of the annotations except ASEF 2860 strain.

3. Results and Discussion

The central achievement of this project was the construction of a Neo4j graph database that stores a Pan-genome sequence graph in compressed de-Bruijn graph structure, interlinked to a set of annotation and orthology data and an index tree built for accessing annotation nodes in given positions. This database is produced by supplying the pipeline with genome sequences of the clade of interest, genome annotation data in GFF file format and finally an ortho-mcl gene cluster (group) file. The database was shown to be scalable up to lower eukaryotic species pangenomes, such as *S. cerevisiae* (30 genomes) and *B. bassiana* (6 genomes). The database produced is also functional in terms of obtaining biologically relevant data, such as extracting sequences for genes at certain positions. As the annotation level graph also includes orthology relationships between gene annotations, access to the orthologues of a certain gene and additionally extract their sequences from the graph is enabled.

In this project, the most upscaled databases for *B. bassiana* and *S. cereviasae* pangenomes were created by using sequence, annotation and orthology data. The ratios of 3 different compartments: sequence, tree-index and annotation were inspected in databases to get an indication of the complete graph overview. For yeast pangenome, which contained 30 strains and a *k* set to 45, the sequence nodes take up about 71% of total nodes, whereas the annotations and tree-index nodes take up 8% and 16%, respectively. The nodes which are not included (5%) correspond to "flank" nodes that mark the beginnings and ends of chromosomes in the sequence graph. The total number of nodes in this database was about 2.1 million nodes. *B. bassiana* pangenome, which contained 6 strains and a *k* set to 45, contained node ratios of 82.3% for sequence, 3.1% for annotation and 6.4% for index-tree compartments. The *bassiana* database contained 2.2 million nodes in total. Here, the segment nodes and trees were not taken into account as they cover no more than 1% of the nodes.

A. Application of the Pangenome Database

Effectiveness of the pangenome database stands out by being able to answer as many biologically relevant questions as possible with the currently integrated types of data. The highly connected structure of the graph makes it possible to relate different kinds of data in the database with ease. The database enables a user to freely inspect the pangenome graph by directly using cypher query language. As a simple example, the user can search for an annotation ID by querying "Match (n:annotation) where n.gene = 'geneA' return n". This matches and returns the node labeled "annotation" that holds data on geneA. Similarly, one can also return all the annotation nodes that are positioned on a specific chromosome. By further including relationship information in the query, another line of cypher query can be used to return annotation nodes which is orthologous to geneA. Cypher can also be used on sequence nodes to look for patterns such as "bubbles" which usually represent simple structural variations, such as SNPs and Indels, between sequences contained in the pan-genome. An extensive list of cypher queries, for retrieving relevant data is given in supplementary table 2. In many cases, a biologist would want to ask more complex questions which require graph traversals and additional graph algorithms to extract the desired data. For this reason, python scripting was used for retrieval of complex queries which were accomplished by using a combination of cypher query language, index tree traversal and sequence graph traversal algorithms (see section

2.E.). Answers to the following queries are obtained from these methods:

1. Obtaining a sequence between a given range
2. Retrieving a sequence of a known annotation
3. Retrieving annotations in a given position (point or position)
4. Retrieving a sequence/sequences of orthologous gene/genes of a given gene

The queries listed above are also answered by genomic sequence and annotation explorer tools, such as Ensemble Genome Browser web tool (Stalker *et al.* 2004) . However, these browsers are based on the perspective of reference genomes of various species. Therefore, this project shows that it is possible to have a tool which resembles Ensemble Genome Explorer in terms of functionality but extends this into a pangenomic format where the sequence structure also contains the information on variation between the genomes.

A database that contains annotation and OrthoMCL data together contains sufficient information to find out the core, dispensable and unique gene sets. The finding of these gene sets involved inspection of each orthologous gene group in the database. These findings can be also set to modify the database by storing the type of genes according to their gene set. This allows the user to, later on, access to these gene sets or ask which gene set a gene belongs to by simple cypher queries. *B. bassiana* was used as a test case to show this approach. The results based on this approach showed that 45,586 (88.80%) number of genes belong to the core, 5,497 (10.71%) genes belong to dispensable genome and 248 (0.48%) genes belong to unique genome. The genome that contained most unique genes (34%) was genome with ID g5078. As the annotation data is also connected to the sequence graph, the user can further extract sequences of each one of these genes (Figure 5).

In addition to sequence extraction, graph traversal while collecting relevant information in the sequence graph can be potentially used to ask questions to the database such as "What percentage of a given sequence range in a selected genome is shared between each of the other genomes?". This question can be answered by collecting data on which genomes appear on every single traversed nodes, in addition to the input genome. The answer to this question would indicate how conserved the given sequence interval is among the different genomes. This approach was tested on several genes of *B. bassiana* database. For instance, gene with an ID of Bb2597v1.0t05994.1, was found to share more than 65% of its sequence in 4 of the other genomes. These were compared to orthology data in the database and confirmed that indeed these sequences were indicated already as orthologous except one which did not contain any similarity in the graph. This incidence can be explained by the single directional nature of the graph, since the gene may be transcribed using the complementary strand. With a similar approach, traversed node information can also be used to search for annotations in the index tree that correspond to other genomes. This would limit the similarity search for only genes. Thus, this approach can be useful to apply if there is no previous orthology data available for gene annotations and also to see how conserved the sequences are in an orthology group. The same example orthology group given above also explains this, as one of the 4 genomes, which was highly similar to a gene from an orthology group, corresponded to an annotation on the genome that was not included in the orthoMCL run. Therefore, this annotation could be added to the orthology group due to its highly similar sequence composition.

Another potential question, also starting with an annotation, is traversing a gene input while checking sequence nodes if they are repeated in other genes of the same genome. The outcome of this process would point out for protein domains by finding out conserved coding sequence segments between different genes. This type of sequence information can be used to draw domain relationships between genes and also to group genes into gene families.

B. Performance and Scalibility

The project have shown that the explained approach is scalable by increasing number of genomes being represented in the form of a pangenome database. The notion of scalability here not only involves storing increased amount of sequence data but also keeping the query transactions efficient. The data size scalability can be attributed to Neo4j DBMS as it can potentially contain a database of several billions of nodes/edges and function (Tobias 2015). Therefore, it can be pointed out that the Neo4j platform should not limit the graph and database size. In terms of database functionality, the upscaling factor leads to increased number of nodes which should be taken into account while trying to navigate to given nucleotide locations through the sequence and annotation nodes. The use of index trees here prevented this issue by speeding up accession of nodes in given nucleotide positions.

Despite the scalable data storage and database functionality, SplitMEM is a potential limiting phase in the construction pipeline due to its use of considerably high memory for building the suffix tree. For instance, for building a suffix tree for 30 yeast strain genomes used up about 172Gb of memory. There are also several flaws of this software. These include incorrect positions on SplitMEM graph, concatenation of discontinuous sequences and existence of minor uncompressed parts on the graph. The first two problems were solved by modifying the output from SplitMEM, however, the last was not solved as the uncompressed parts does not make the graph wrong, in terms of data it represents, but less efficient in terms of data space. In addition to these flaws, the produced output only corresponds to the forward strand. In other words, the graph does not represent the bi-directional nature of the DNA. Thus, it is clear that replacing SplitMEM with a better functioning compressed de-Bruijn graph construction tool is ideal. A candidate would be a memory efficient tool which is being developed by Siavash Sheikhizadeh Anari in Wageningen University. This tool is directed for eukaryotic genomes and includes bi-directional representation.

The use of compressed de-Bruijn graph proved to be efficient in terms of its size when stored in the database, indicating data compression. For instance, storage of 25 yeast pangenome (with a k of 45) took up 146 Mb of space whereas the fasta input files took about 191 Mb of disk space. The database size is increased with the addition of more informative node properties, index-tree and annotation compartments. These add up to a size of 284 Mb with the same sequence graph, corresponding to 25 yeast genomes. It should be noted here that each set of genomes used were based upon the lower set except the 30 yeast genome sequence dataset which was based on 24 of the sequences used in 25 genomes set and 6 additional genomes. This was done to prevent an unresolved error from neo4j-import tool related tho

	Database Size (Mb)				
Number of Genomes	*k*=20	*k*=45	*k*=60	Max. Memory (Gb)*	Query transaction time (s) **
5	56	53	53	25	0.63
11	133	120	119	62	0.62
20	255	228	224	109	0.66
25	328	284	278	138	0.78
30	348	300	293	176	0.75

Table 1 The table shows details on disk and memory usage for storage and construction, respectively, by using different *k* parameters and different numbers of yeast genomes. Query transaction speed and number of Nodes/Edges are also given for databases with different numbers of genomes, which were constructed with *k*=45. *Corresponds to peak memory usage during construction. **Query involved finding an annotation at a given position and returning its sequence

the csv file input. As it can be noticed from Table 1, the increase in database size is linear with the addition of more genomes, until 25 yeast genomes. This was expected for the annotation and index tree compartments as the number of annotations increase linearly with addition of more sequence length and the tree index, having a binary structure, always corresponding to *2n-1* nodes, where n is the number of annotations. This points out that the size of sequence graph was also linear up to 25 genomes. This notion was confirmed by looking at sequence graph only databases built by different number of genomes. Regarding the 30 yeast genome dataset, the database size it produced was lower than expected. This may be attributed to higher similarity of the 6 genomes, which were not present in the rest. Databases were also built using different minimum MEM sizes (20, 45 and 60). It was evident that lowering of *k* lead to construction of larger graphs, with more nodes and edges, however, with higher resolution in terms of detailing shared parts between genomes. Thus, the choice of *k* depends on the clade being investigated. For instance, a if the genomes in the clade of interest are distantly related, then choosing a lower MEM would give more information on the shared parts and interruptions between similar sequence intervals.

A database shown highly scalable in size would have no use if it does not provide an efficient and logical way to access this data. It was implied earlier that, with the current data structure, the user is able to access informative data. The next step taken in this study was to also show that this data can also be accessed with an acceptable speed with a growing database size. Table 1 shows the results for returning sequence of a gene found in a scaffold in a given position. This is a complex form of query retrieval method that uses both graph traversals and index-tree tree search. The query was repeated with 5 genes of at least 1000 bp length and their average was taken. It should be noted that the retrieval time ranged from 0.3 to 2.2 seconds. This variation mostly depends on the number of nodes traversed for sequence extraction. As a result, the information retrieval time here is an indication of its usability.

Taken as a whole, no limitations were found up to the 30 *cerevisae* or 6 *bassiana* pangenome databases in terms of data retrieval and functionality. The databases constructed contained at most about 2.5 million nodes. This indicates that the database can be possibly further scaled up to higher eukaryotic organisms without any constraints.

4. Conclusion and Future Work

The pangenome database construction pipeline tool explained in the paper was undertaken as a response to an increased demand on a method to store multiple similar eukaryotic genome sequences in a highly informative structure which, in addition to obtaining sequence data, can be accessed to obtain information on sequence similarity and sequence annotation. The tool have proven that it is efficient to store and represent eukaryotic pangenome sequence graph in a compressed de-Bruijn graph structure. The de-Bruijn graph both compressed the sequence while storing information on sequence variation. The graph DBMS used in in the project, Neo4j, proved to be a scalable choice. However, sequence graph construction tool, SplitMEM, would be a possible limiting factor with larger genomes due to its high memory usage. Accession of corresponding annotations of sequence intervals was implemented in an efficient manner by the use of binary tree indexing. This allowed a fast way to grab an annotation which intersects with a given position. Taken as a whole, it is possible to further scale up the database since there was no apparent reduction in data retrieval speed. In terms of scalability and performance, an important improvement on the current methods would be to use an alternative tool for compressed de-Bruijn Graph construction. Another improvement would be to implement the database construction natively in Neo4j by using the provided java libraries. This would prevent any issues in importing large graph data. In addition, implementing the traversal query methods would also improve traversal speed as the http requests at each traversal step spends extra time (Holzschuher and Peinl 2013).

An apparent feature of the explained data structure was the inclusion of subgraphs of orthologous genes, which were defined by integrating orthology relationships between annotated genes. These subgraphs hold information on how similar the genes are and what are their differences. Some work on traversing the genes in these orthology subgraphs enabled inspection of sequence similarities between them.

The database explained in the paper is not limited in terms of data types being stored. Biological data is usually highly interrelated. Thus, other types of relevant data can also be integrated to the current database. For instance, different steps in metabolic pathways can be pointed by sequence annotations. Also, in this project, only the annotation data on coding sequences was used. Thus, as a next step, the complete information included in the GFF file can be included in the database. Another area which is open to improvements is deep sequence variation de-

tection in orthologous gene subgraphs. The current traversal methods used in the project are still open to improvements as not all the sequence variation information stored in the graph was exploited. Some of the information that can be extracted from these subgraphs in future can include type and location of sequence variations, such as, indels, single nucleotide variation and translocations between orthologous genes. Thus, it can be concluded that while the current project pipeline and implemented methods enables construction of a functional and informative pangenome graph database, its functionality can be extended further by including more types of data and/or implementing more graph algorithms for sequence variation detection.

Literature Cited

Baker, M., 2012 Structural variation: the genome's hidden architecture. Nature methods **9**: 133–137.

Holzschuher, F. and R. Peinl, 2013 Performance of graph query languages: comparison of cypher, gremlin and native access in neo4j. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, pp. 195–204, ACM.

Laing, C., C. Buchanan, E. N. Taboada, Y. Zhang, A. Kropinski, A. Villegas, J. E. Thomas, and V. P. Gannon, 2010 Pan-genome sequence analysis using Panseq: an online tool for the rapid analysis of core and accessory genomic regions. BMC bioinformatics **11**: 461.

Lappalainen, I., J. Lopez, L. Skipper, T. Hefferon, J. D. Spalding, J. Garner, C. Chen, M. Maguire, M. Corbett, and G. Zhou, 2013 DbVar and DGVa: public archives for genomic structural variation. Nucleic acids research **41**: D936–D941.

Li, L., C. J. Stoeckert, and D. S. Roos, 2003 OrthoMCL: identification of ortholog groups for eukaryotic genomes. Genome research **13**: 2178–2189.

Marcus, S., H. Lee, and M. C. Schatz, 2014 SplitMEM: A graphical algorithm for pan-genome analysis with suffix skips. Bioinformatics **30**: 3476–3483.

Martin, D., 2007 Packed 1-dimensional R-Tree for Geometric Algorithms. http://lin-ear-th-inking.blogspot.nl/2007/06/packed-1-dimensional-r-tree.html.

Marx, V., 2013 Biology: The big challenges of big data. Nature **498**: 255–260.

Paten, B., D. Earl, N. Nguyen, M. Diekhans, D. Zerbino, and D. Haussler, 2011 Cactus: Algorithms for genome multiple sequence alignment. Genome research **21**: 1512–1528.

Rasko, D. A., M. J. Rosovitz, G. S. Myers, E. F. Mongodin, W. F. Fricke, P. Gajer, J. Crabtree, M. Sebaihia, N. R. Thomson, and R. Chaudhuri, 2008 The pangenome structure of Escherichia coli: comparative genomic analysis of E. coli commensal and pathogenic isolates. Journal of bacteriology **190**: 6881–6893.

Stalker, J., B. Gibbins, P. Meidl, J. Smith, W. Spooner, H.-R. Hotz, and A. V. Cox, 2004 The Ensembl Web site: mechanics of a genome browser. Genome research **14**: 951–955.

Stanford University, D. o. G. a. t. S. o. M., 2015 Saccharomyces Genome Database. http://www.yeastgenome.org.

Thorisson, G. A., J. Muilu, and A. J. Brookes, 2009 Genotype–phenotype databases: challenges and solutions for the post-genomic era. Nature Reviews Genetics **10**: 9–18.

Tobias, I., 2015 The Neo4j Manual v2.2.5. http://neo4j.com/docs/stable/.

Xiao, J., Z. Zhang, J. Wu, and J. Yu, 2015 A brief review of software tools for pangenomics. Genomics Proteomics Bioinformatics **13**: 73–6, 2210-3244 Xiao, Jingfa Zhang, Zhewen Wu, Jiayan Yu, Jun Journal Article Research Support, Non-U.S. Gov't Review China Genomics Proteomics Bioinformatics. 2015 Feb;13(1):73-6. doi: 10.1016/j.gpb.2015.01.007. Epub 2015 Feb 23.

Zhang, Z., L. Mao, H. Chen, F. Bu, G. Li, J. Sun, S. Li, H. Sun, C. Jiao, and R. Blakely, 2015 Genome-Wide Mapping of Structural Variations Reveals a Copy Number Variant That Determines Reproductive Morphology in Cucumber. The Plant Cell p. tpc. 114.135848.

Supplementary Materials:

Supplementary Table 1:

Sequence Nodes:

Propery	Value	Reason
len	length of the node	Correct Traversal
sequence	Sequence of the node	To obtain sequence
scaffold_name	array of nucleotide positions which appear on the scaffold	Correct traversal

Annotation Nodes:

Propery	Value	Reason
gene	gene ID	Required for identification
Range	Sequence of the node	Required for obtaining its corresponding range on scaffold
genome	scaffold name	Required for identification
feature	genome name	Required for identification and forming pangenome gene sets (core, dispensible or unique)

Segment nodes are identical to annotation nodes, however, instead of gene names they store segment ids

Ortho Nodes:

Propery	Value	Reason
orthoID	unique orthology group ID	Required for identification
feature (optional)	gene set information (core, unique or dispensible)	Required for ortholog gene set type identification

Rtree Nodes:

Propery	Value	Reason
type	leaf, root or internal_node	Required for identification and Rtree traversal
bbox	Range	Required for Rtree traversal

Flank Nodes:

Propery	Value	Reason
feature	start/end	Required for finding start/end node to a corresponding scaffold on the seq. graph
genome	scaffold name	Required for scaffold identification

Relationships:

From (node)	To (node)	Property/Label	Function
Sequence	Sequence	cont	Not necessary but required by Neo4j
annotation	Sequence	genome:scaffoldname start:int or end:int	marks start and end of annotation on graph
ortho	annotation	ortho	Not necessary but required by Neo4j
flank	Sequence	flanking	Not necessary but required by Neo4j
segment	Sequence	genome:scaffold name start:int or stop:int	Marks the start and end of annotation on graph.

Supplementary information 2:

1)Find annotation from name:

```
match (n:annotation) where n.gene = ‘name’ return n;
```

2)Find orthologs of an annotation:

```
match (n:annotation)--(o:ortho)--(x:annotation) where n.gene = ‘name’ return x;
```

3)Return gene names which are unique to a genome, with the genome name:

```
match (o:ortho)-[r]-(a:annotation)
with n, count(r) as rr where rr = 1
match n-->x return x.gene, x.feature;
```

4)Return genes in a scaffold:

```
match (n:annotation) where n.genome = ‘scaffoldx’ return n;
```

5)Return genes in a genome:

```
match (n:annotation) where n.feature = ‘genomex’ return n;
```

6)Return sequence nodes that appear in a given scaffold:

```
match (n:Sequence) where ‘scaffoldx’ in keys(n) return n;
```

7)Return start node of a scaffold:

```
match (n:flank)-->(y:Sequence)
where n.feature = ‘start’ and n.genome = ‘scaffoldx’ return y;
```

8)Return end node of a scaffold:

```
match (n:flank)-->(y:Sequence)<--(x:Sequence)
where n.feature = ‘end’ and n.genome = ‘scaffoldx’ return x;
```

9)Return the root of an R-tree that contains a genome:

```
match (n:scaffoldx_Rtree) where n.type = ‘root’ return n;
```

Note: To return leaves or internal nodes change the type to “leaf” or “internal_node”, respectively.

10)Finding simple snp bubbles (with a kmer length of 80):

```
MATCH p = (n) --> (k1) --> (x) <-- (k2) <-- (n)
WHERE k1.len = k2.len and k2.len = 41 and k1 <> k2
RETURN SUBSTRING(k1.seq,20,1), SUBSTRING(k2.seq,20,1);
```

11)Finding simple indels (with a kmer length of 80):

```
match (n:hiv) match p = (n)-->(k1)-->(x)<--(k2)<--(n)
where k1.length = 41 and k2.length>k1.length
return substring(k2.kmer, 20,length(k2.kmer)-40) as insertion;
```