Journal Title Here Vol. , No. , pp. 1–9
doi: DOI HERE
Advance Access Publication Date: Day Month Year
Paper

A framework for large scale phylogenetic analysis

Bruno Lourenço[1, 2, *]

[1]CSE Dept, Instituto Superior Técnico, Av. Rovisco Pais 1, 1049-001, Lisbon, Portugal and [2]INESC-ID, R. Alves Redol 9, 1000-029, Lisbon, Portugal
[*]Corresponding author. bruno.leandro@tecnico.ulisboa.pt

ABSTRACT

With growing exchanges of people and merchandise between countries, epidemics have become an issue of increasing importance and huge amounts of data are being collected every day. Hence, analyses that were usually run in personal computers are no longer feasible. It is now common to run such tasks in High-performance computing (HPC) environments and/or dedicated systems. On the other hand, we are often dealing in these analyses with graphs and trees, and running algorithms to find patterns in such structures. Hence, although graph oriented databases and processing systems can be of much help in this setting, as far as we know there is no solution relying on these technologies to address large scale phylogenetic analysis challenges. This work aims to develop a modular framework that exploits such technologies, namely Neo4j. We address this challenge by proposing and developing a framework which allows representing large phylogenetic networks and trees, as well as ancillary data, that supports queries on such data, and allows the deployment of algorithms for inferring/detecting patterns and pre-computing visualizations, as a Neo4j plugin. This framework is innovative and brings several advantages to the phylogenetic analysis process, like the management of the phylogenetic trees, which will avoid having to compute them again, and the use of multilayer networks, that will make the comparison between them more efficient and scalable. The experimental evaluation results showcase that it can be very efficient in the mostly used operations and that the supported algorithms comply with their time complexity.

KEYWORDS: Phylogeny, Data processing, Data storage, Graphs, Database

INTRODUCTION

Phylogenetics is the study of the evolutionary history and relationships among individuals or groups of organisms, which aims to produce a diagrammatic hypothesis about the history of the evolutionary relationships of a group of organisms known as phylogenetic tree. The relationships are inferred through the analysis of the traits of the individual or group, that is, by applying computational algorithms, methods and programs to the phylogenetics data.

With the growing exchanges of people and merchandise between countries, epidemics have become an issue of increasing importance. The computational phylogenetics were mostly performed in personal computers and desktops. However, this kind of analysis is not feasible anymore, since huge amounts of data are being collected every day, and there are certain operations that require a considerable amount of memory or time. Instead, it is now common to run such tasks in high performance computing environments and/or dedicated systems. Therefore, there is a need to find a better way to store and maintain the data rather than in personal computers and desktops.

In large scale phylogenetic analysis of microbial population genetics, it is often needed to sequence and type the information of the organisms, and afterwards to apply a set of phylogenetic inference methods to produce a diagrammatic hypothesis about the history of the evolutionary relationships of a group of organisms. The computation and analysis of microbial population genetics often produce graphs

and trees, which have many relationships. As graph databases naturally apply to these data structures and are optimized to perform queries and operations over them, that is, they are designed specifically to deal with highly connected data, it should be possible to store them in a graph database.

A graph database management system is an online database management system with create, read, update, and delete (CRUD) methods that expose a graph data model. Graph databases are generally built for use with Online Transaction Processing (OLTP) systems. Accordingly, they are normally optimized for transactional performance, and engineered with transactional integrity and operational availability in mind (1). This type of database addresses the problem of leveraging complex and dynamic relationships in highly connected data. Graph databases offer appealing characteristics, such as performance and flexibility. Regarding the performance of graph databases, it tends to remain relatively constant, even as the dataset grows, because the queries only use the respective portion of the graph. In terms of flexibility, a graph database allows adding new nodes, labels, and relationships, to an existing structure, without disturbing the existing queries and application functionality.

Although graph oriented databases can be of much help in this setting, as far as it is known there is no solution relying on these technologies to address large scale phylogenetic analysis challenges. Thus, a study on which database engine better adresses the needs of this challenge was conducted to provide new insights and lead to innovative approaches, comparing graph databases such as Neo4j (2), Titan Aurelius (3), JanusGraph (4), Dgraph (5), Allegrograph (6), and Apache Rya (7). The comparisons made are presented in the fully extended version of this article (8) and they suggest that Neo4j offers the most interesting set of features and capabilities.

Neo4j is one of the most popular graph database management systems, and is currently active and open source. It is implemented in Java and accessible from software written in other languages using the Cypher query language (9). It is built over a native graph storage and processing engine system. This database has community-driven libraries available, that together with the official available algorithms, provide many algorithms to use over the graphs stored in the system, and allows to extend itself with plugins to support any other graph algorithm. It supports the storage of 34 billion nodes (10), allows to visualize data with tools that connect directly to the database such as Neovis (11) and Popoto (12), and supports the integration with other data processing tools (DPT) such as Apache Spark (13). Hence, Neo4j is the graph database system that is used to address the large scale phylogenetic analysis challenge.

The objective of this work is to develop a modular framework for large scale phylogenetic analysis that exploits the Neo4j graph oriented database technology to allow the management of the phylogenetics data, without needing to load it into the clients computers. This framework should have a data model that allows the representation of large phylogenetic networks and trees, as well as the ancillary data. It should support queries on such data and allow the deployment of algorithms for inferring/detecting patterns and for pre-computing visualizations.

APPROACH

The solution consists in a framework that complies with the phylogenetic analysis process and uses a graph database. This framework consists of several components, namely a database containing the plugin which holds the algorithms, an application server, and an authorization server. These components are represented in Figure 1.

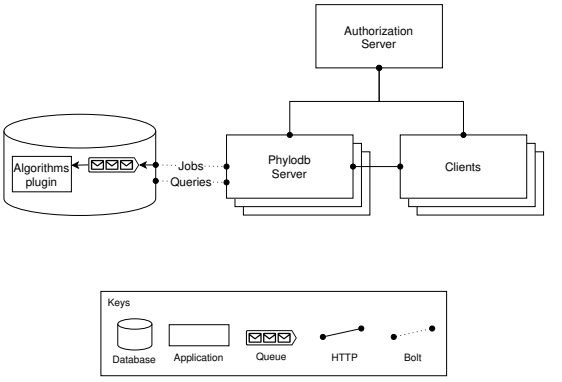

Fig. 1. Client-Server architectural view.

The `Phylodb Server` component provides a Spring (14) web application programming interface (API), to perform several operations over the data stored in the database, namely access data, load datasets, execute algorithms and obtain results. It should be possible to scale horizontally to handle more operations by adding more instances of the `Phylodb Server` component. The `Phylodb Server` interacts with the Neo4j database component in two different ways. That is, it can normally query data, but it can also queue executions of the algorithms that are deployed in the database and reside in the `Algorithms Plugin` component. These algorithms also read the needed inputs and write the computed results. The `Authorization Server` component, which relies on the Google Identity Provider (15), manages the user information and provides operations to perform the authentication of a user.

The solution was implemented considering an agile methodology and is publicly available at

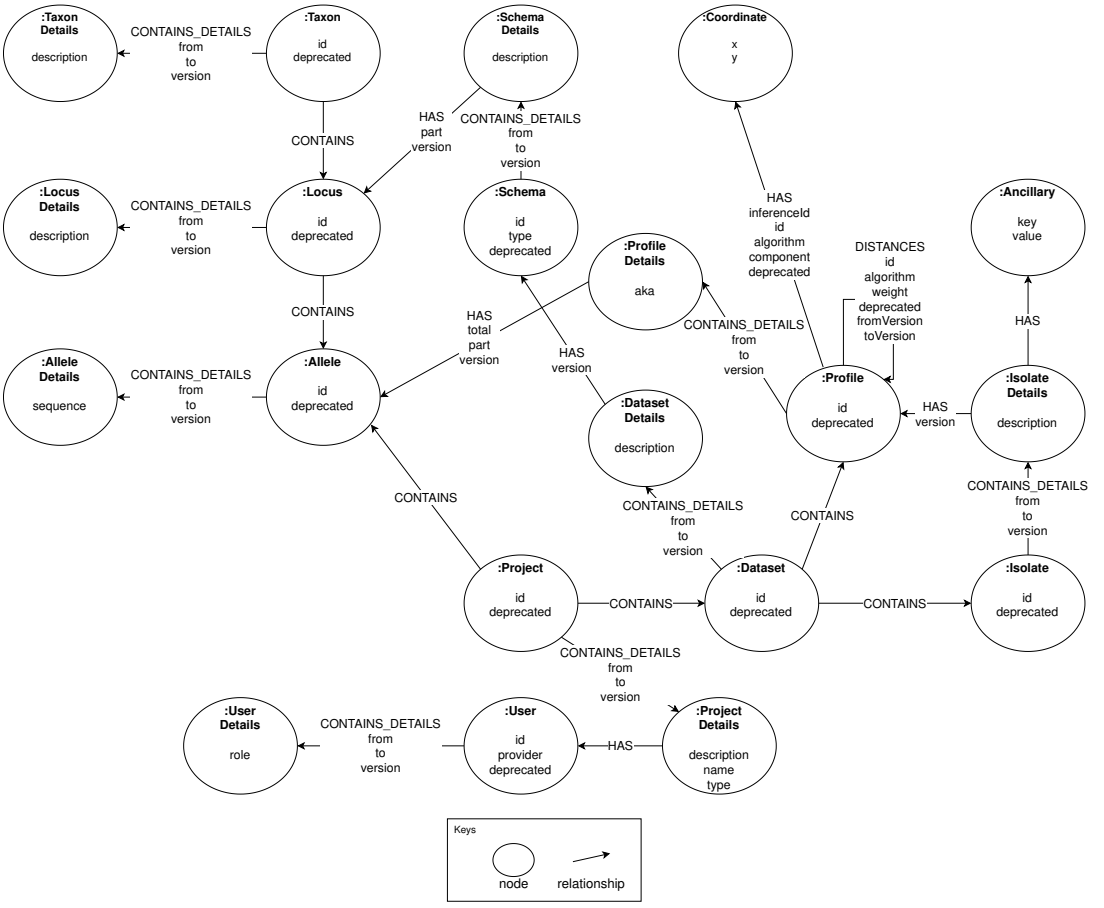

Fig. 2. Data model represented by a set of nodes and relationships to compose a graph data model, which is the format used by Neo4j.

`https://github.com/Brunovski/phyloDB` along with its issues, milestones, and documentation.

Database

The database follows a data model, which is represented in Figure 2, that allows to represent all the entities considered in the phylogenetic analysis and their respective relationships (16; 17).

The analysis of the phylogenetics data is based on the need to pass files through a series of transformations, called a pipeline or a scientific workflow. The input of this scientific workflow is the organism information that comes from the laboratories. Hence, given such information that comes as biological samples, a Next-Generation Sequencing (NGS) process is applied to obtain the genetic sequences. Then, alignment tools or assembly tools are executed to assembly the genomes (18; 19).

The sequences assembled in the alignment process may occupy a given position of a locus and define distinct alleles of that locus. A locus is a specific location in the chromosome, and every unique sequence, either Deoxyribonucleic acid (DNA) or peptide depending on the locus, is defined as a new allele. An allele can also be defined as a viable DNA coding sequence for the transmission of traits, and it is represented with a number identifying the allele and string containing the sequence.

That is, the phylogenetic data are composed of **taxonomic units**, ***loci***, and **alleles**. The taxonomic units consist of several *loci*, thus this is represented in the data model by a relationship named `CONTAINS` between taxonomic units and *loci* nodes. Moreover, the *loci* may hold specific locations for a set of alleles, which is represented by a relationship named `CONTAINS` between the nodes of each locus and the associated nodes of the alleles.

After the alignment phase, a typing methodology is applied to identify or fingerprint each organism based on the genes that are presented in almost

all organisms, which are named as conserved genes. Bacterial identification and characterization at subspecies level is commonly known as microbial typing. This process provides the means to execute phylogenetic inference methods, which then produces a hypothesis about the history of the evolutionary relationships about a group of organisms. There are several typing methodologies, such as the Multilocus Sequence Typing (MLST) (20; 21), Multiple-Locus Variable Number Tandem Repeat Analysis (MLVA) (22), and Single Nucleotide Polymorphism (SNP) (23).

One of the most popular methodologies is the MLST, which is an unambiguous procedure for characterizing isolates of bacterial species using the sequences of internal fragments. This methodology types several species of microorganisms, and when applied, the set of alleles identified at the *loci* are considered to define a Sequence Type (ST), a key identifier for this methodology, that can also be defined as an allelic profile. The chosen *loci* are usually different for each species, although some species may share some or even all *loci* in their MLST schemas. The number of chosen *loci* can vary and be greater or smaller than the seven *loci* more commonly adopted. The generated sequences are compared to an allele database and for each gene, the different sequences are assigned as distinct alleles and, for each isolate, the alleles at each of the *loci* define the allelic profile.

That is, the **typing schemas** can use several *loci* to characterize different **allelic profiles**, and this is expressed in the data model by the relationships `HAS` between the details of a schema node and the respective *loci* nodes. The allelic profiles belong to a specific **dataset** as they are a result of applying a typing methodology. Thus, this is described by the relationship `CONTAINS` between dataset and profile nodes. These profiles follow the same schema, hence they should be related to the typing method used. To impose this concern, the `HAS` relationship is used between the dataset and schema nodes, which means that all profiles from that dataset follow the related schema. However, having the dataset connected to the schema, only allows to perceive what *loci* were used in the typing operation. Therefore, to know what is the allele that characterizes a profile for each locus used in the schema, the details node of a profile must be connected to the respective allele nodes. Hence, this is represented by using a relationship called `HAS` between the profile details and the alleles nodes.

The main goal of the typing methods is the characterization of organisms existing in a given sample. However, some microorganisms from the sample collected need to be isolated to be characterized. Thus, each organism isolated from the microbial population becomes an **isolate**. An isolate can be associated with typing information and ancillary details. **Ancillary** details include information about the place where the microorganism was isolated, the environment, the host, and other possible contextual details.

That is, isolates may have related ancillary data. Thus, in the data model this is expressed as a relationship named `HAS` between the detail of an isolate and ancillary data nodes. Since an isolate may be associated to a profile, there is also a relation between the two, which is called `HAS`. The detail of this isolate also has several relationships `HAS` to each ancillary data associated to it.

Succeeding the typing process follows the execution of a phylogenetic inference method to the results. A phylogenetic inference method is the application of computational algorithms, methods, and programs to phylogenetic data that allows to produce a diagrammatic hypothesis about the history of the evolutionary relationships of a group of organisms. There are several types of phylogenetic inference methods, such as distance matrix methods, maximum parsimony, maximum likelihood, and Bayesian inference. In this work, we have focused in distance matrix methods (24; 25; 26). Distance matrix methods rely on the genetic distance between the sequences being classified. The distances are often defined as the fraction of mismatches at aligned positions, with gaps either ignored or counted as mismatches. The Globally Optimized eBURST (goeBURST) algorithm (27) is an example of an implementation of these algorithms.

Therefore, the inference algorithms rely on the genetic **distances** between profiles. These distances are calculated by computing a distance matrix. Based on these distances, the algorithm is then executed and relationships `DISTANCES` are created between the different profile nodes to compose the resulting graph. This strategy allows to consider multilayer networks since the same nodes shall be used to represent different graphs.

After executing an inference algorithm, a visualization algorithm, such as Radial Static Layout (28), or GrapeTree (29), is executed to compute the optimal coordinates for each node of the received graph or tree. Afterwards, the coordinates are provided to a render framework which then presents each profile and relationship to a user interface.

Thus, the visualization algorithms execute over the graphs resulting from the inference algorithms, and create visualization **coordinates** for each node of the graphs. Hence, a relationship `HAS` between a profile and coordinate nodes exists to represent the coordinate of some profile, for a given inference and visualization algorithm.

This data model also incorporates versioning and soft deletes concerns to mitigate some problems that occur nowadays, such as the impossibility to delete a wrongly inserted profile after executing an algorithm that generates a graph containing it. Such profiles can not be removed, because the generated graphs would then become invalid. In this case, by

considering a versioning and soft delete strategy, these removals should be possible, since the graphs would be linked to the statuses of the profiles and not to the profiles themselves. The versioning strategy to achieve this behaviour is to separate each object from its state, link them through a relationship with the respective version number, and capture changes by having different state nodes (30). In the data model, the name of the status nodes end with `Details`, and the version relationships are named as `CONTAINS_DETAILS`.

API

The API provided by the framework was implemented based on three layers, namely `Controllers`, `Services` and `Repositories`, as demonstrated in Figure 3.

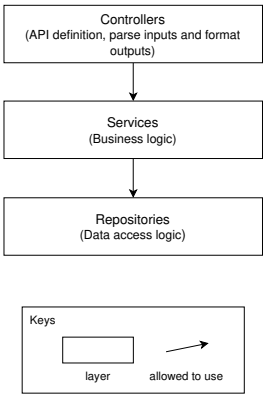

Fig. 3. Layered architectural view of the PhyloDB Server component.

When a request is received by the API, it is passed through the `Controllers` layer. This layer contains the controllers that parse the received input, execute the respective service, and retrieve the response containing the respective status code and the formatted content. The `Services` layer contains the services that perform the business logic and use the needed repositories. The `Repositories` layer holds the repositories that shall provide operations to interact with the database. Apart from these layers, there is a validation logic that verifies the request authenticity and the user permissions before the request is processed by them. Thus, several factors are considered in the implementation of this API such as, the representational state transfer (REST) architecture, the Hypertext Transfer Protocol (HTTP) semantics, the imports and exports of datasets, the error handling, the security concerns, and more.

The interaction between the API and users is accomplished through HTTP requests. Hence, the HTTP protocol allows the API to retrieve different types of responses depending on several factors, such as the parameters used in the request, the type and result of the operation that is being executed.

The security concerns are considered through the definition of a security pipeline, which is based on the Spring interceptor components (31). These components allow to implement the `preHandle` method, that is executed before a request is passed to the controllers. Thus, the security pipeline is composed by the `AuthenticationInterceptor` and `AuthorizationInterceptor` interceptors. These components implement the `SecurityInterceptor` interface. This interface defines the method `handle`, which shall contain the authenticity and user permissions validations. However, to ensure that these validations are executed before a request is passed to the controllers, the `SecurityInterceptor` must implement the Spring `HandlerInterceptor` interface and define that the `handle` method is executed within the `preHandle` method.

The authentication is based on the bearer token authentication (32). This type of authentication is based on tokens that are acquired after an authentication process with an identity provider.

After the request passes by the security pipeline it is handled by the controllers, that use services to perform the respective operation. These services rely on the repositories which define the queries to interact with the database, either to manage data or to schedule algorithms executions. Several concerns were considered in the implementation of the repositories, such as the use an object graph mapper (OGM), pagination, parameterized queries and more.

The use of the functionalities related to an OGM functionalities may add some overhead (33). Hence, it was decided that each query should be implemented from scratch to increase the performance of the data access operations. That is, the implementation of each repository method contains the respective Cypher query that it should perform.

Moreover, the use of pagination in these queries was adopted because this framework is intended to handle great quantities of data. This approach allows to control the quantity of data that is dealt with in the methods that retrieve many domain entities. Thus, memory issues are less likely to happen since a maximum number of records that can be retrieved by the queries is defined. To achieve this behaviour, the respective methods must receive the page and the limit of records to retrieve.

Also, each of those queries is parameterized. By using parameterized queries the performance can be increased because Neo4j can cache the query plans and reuse them in the following executions, which increases the following query speed. And, it also allows to protect from injection attacks, since parameters are never allowed to be interpreted as part of the query and have no means of escaping out of being anything other than a value of some sort

(33; 34). This can be achieved by executing a query that contains specific placeholders for each parameter and pass the values of the parameters, in the correct order, as arguments.

Plugin

The plugin of the database is based on the feature of user-defined procedures from Neo4j. A user-defined procedure is a mechanism that allows to extend Neo4j by writing custom code, which can be invoked directly from Cypher. These procedures can take arguments, perform operations on the database, and return results. Moreover, some resources can be injected into them from the database, which is similar to the dependency injection mechanism from Spring.

This plugin intends to extend Neo4j to support inference and visualization algorithms, which shall be available as procedures. The inference algorithms procedures are executed over the profiles of a dataset, while visualization algorithms procedures are executed over the results of the inference algorithms. These procedures can be invoked directly from Cypher like any other standard procedure.

The structure of the plugin is also based on three layers, namely `Procedures`, `Services` and `Repositories`, as shown in Figure 4.

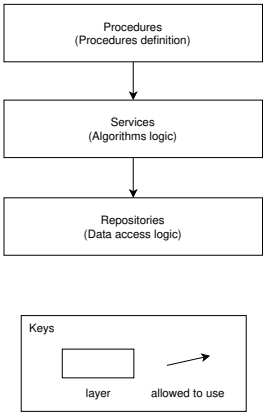

Fig. 4. Layered architectural view of the Algorithms Plugin component.

The `Procedures` layer holds the definitions of the operations that allow to execute the supported algorithms, hence a call to an algorithm is directed to them. They parse the received input and execute the respective service provided by the `Services` layer. The `Services` layer reads the input data for the algorithm from the database, compute the respective algorithm and store the obtained result back to the database. The reading and writing of data are accomplished by using the methods provided by the `Repositories` layer.

The algorithms receive inputs, perform the respective computation, and produce results. Hence, inference algorithms receive distance matrices and produce graphs, while visualization algorithms receive the graphs produced by inference algorithms and generate coordinates for each of the nodes of the graph. Thus, the interface `Algorithm` defines a method `compute` that is extended by the `InferenceAlgorithm` and `VisualizationAlgorithm` interfaces to specify the respective arguments and results. For example, `InferenceAlgorithm` defines that the `compute` method must receive a `Matrix` and retrieve an `Inference`.

The goeBURST algorithm is the only inference algorithm supported, and the Radial Static Layout algorithm is the only visualization algorithm supported. Therefore, a class that implements the `compute` method exists for each of these algorithms. For instance, the class `GoeBurst` extends from `InferenceAlgorithm` to implement the `compute` method, which receives a `Matrix` and retrieves an `Inference`, with the logic of the goeBURST algorithm. Implementations of the goeBURST and Radial Static Layout algorithms already exist, thus our implementations are based on them (35; 36).

EVALUATION

The tests were executed over a set of read and write operations, which were picked to be analysed in terms of time and memory. These operations are composed by the `Save Profiles`, `Run Inference`, `Run Visualization`, `Save Profile`, `Get Profiles`, `Get Resumed Profiles`, `Get Profile`, `Get Inference`, and, `Get Visualization`. In these tests, the `Save Profiles` saves a file containing an increasing number of profiles. The `Run Inference` executes the goeBURST algorithm over an increasing number of profiles that are stored in the database. The `Run Visualization` executes the Radial Static Layout algorithm over the result of the goeBURST algorithm execution with an increasing number of profiles and edges. It must be taken into account that these two operations not only compute the algorithm, but also include the work of gathering the data and storing the results. Note that, in the case of the inference algorithms, the calculation of the distance matrix is also included in this process. Then, the `Save Profile` saves a single profile in the database that holds an increasing number of profiles. The `Get Profiles` and `Get Resumed Profiles` retrieve the respective information about an increasing number of profiles. The `Get Profile` retrieves only a single profile independently of the quantity of profiles stored in the database. Finally, the `Get Inference` and `Get Visualization` retrieves the results of the algorithms executed over an increasing number of profiles and edges.

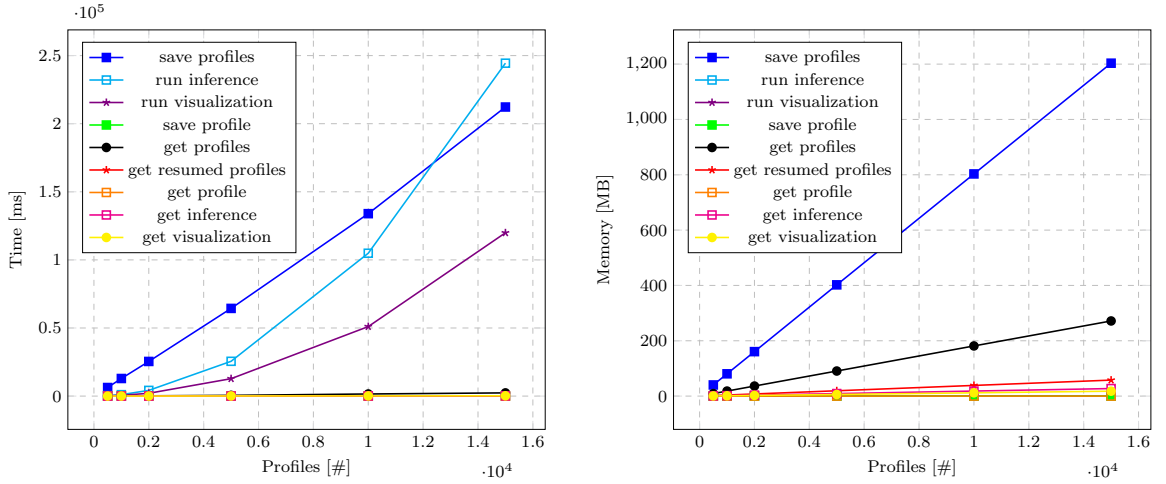

Fig. 5. Plots containing the time and the memory results, in milliseconds and megabytes respectively, as new profiles are incrementally integrated for the *Streptococcus pneumoniae* MLST dataset.

The tests relied on the *Streptococcus pneumoniae* (37) MLST dataset, which was specifically chosen because it is part of several published studies and also because it is publicly available, which will facilitate the interpretation of the results. This dataset contains a profile length of 7 and a total of around 16000 profiles currently.

This experimental evaluation was performed on a machine with an Intel Core I7 2.40 GHz quad core processor and 8 GB of memory, where 2 GB were allocated for the database and another 4 GB were allocated for the API.

The average running time that each operation took to complete, over an increasing number of profiles is presented in the plot represented by Figure 5, in milliseconds. The obtained results confirm that the graph database operations that operate over a fixed amount of data are not affected by the increasing volume of the database data. This is confirmed by analysing the results for the `Get Profile` operation, that shows that the respective line is flat, which means that the time of execution is constant and independent of the increasing number of profiles. Additionally, it can be concluded that the presented execution times for the algorithms comply with their time complexity, which is quadratic for the goeBURST algorithm (27), and linearithmic for the Radial Static Layout algorithm (28) since the children nodes are sorted. The presented results also reveal that relying in a graph database to handle this type of data allows to have a good performance in read and single write operations.

The average memory allocated that each operation consumed until completion, over an increasing number of profiles, is presented in the plot represented by Figure 5, in megabytes. The presented results reveal that the framework allocates the memory linearly proportional to the amount of data that it is handling, and that the read and single write operations allocate much less memory than batch writes operations. However, it must be noticed that this analysis only considers the amount of memory allocated in the API. This is relevant because the algorithms are executed within the database, which causes them to use the database memory instead of the API memory. Hence, the allocated memory for the algorithms executions are minimal.

CONCLUSION

Epidemics have become an issue of increasing importance due to the growing exchanges of people and merchandise between countries. Hence, phylogenetic analyses are continuously generating huge volumes of typing and ancillary data. And there is no doubt about the importance of such data, and phylogenetic studies, for the surveillance of infectious diseases and the understanding of pathogen population genetics and evolution. The traditional way of performing phylogenetic analysis is not feasible anymore as a result of the amount of data generated.

The goal of this work was to develop a framework that should comply with the phylogenetic analysis process and that exploits a Neo4j database to allow the management of the phylogenetics data. It should support queries on such data and allow the deployment of algorithms for inferring/detecting patterns and for pre-computing visualizations.

The implementation of this framework provides a data model that is designed to represent the

relationships between the several types of data and to consider multilayer networks. This data model also contemplates versioning and soft deletes concerns to mitigate currently known problems. Furthermore, the API implementation considers several concerns, such as importing and exporting of datasets, logging, error handling, and security concerns. Finally, the implementations of the algorithms are based on the user-defined procedures feature of Neo4j, which allows to extend its semantic with our algorithms.

Several tests were performed over the framework, which allowed us to conclude that the most important facts found are related with the read operations computational cost, when comparing to write operations, since the former represents a notable difference over the latter. We can also conclude that by using a graph database the operations executed over a fixed amount of data are not affected by an increasing volume of data. Furthermore, we observed that the execution times for the algorithms complied with their time complexity. Overall, we consider our implementation efficient in terms of read and single write operations. However, with the presented results and their analysis, we understand that the batch operations can still be improved.

There are several possible continuations of this work. One could be the extension of our solution to provide more algorithms, and make use of parallelization to improve their performance. The already provided algorithms by Neo4j make use of parallelization, hence we could improve our algorithms execution time by parallelizing their computations. Other potential development could be on how to use the background triggers functionality to achieve the dynamic computation of inference algorithms. Another possibility could be a study of how to perform the batch writes processing based on a queue mechanism.

FUNDING

This work was partly supported by national funds through FCT– Fundação para a Ciência e Tecnologia, under projects PTDC/CCI-BIO/29676/2017 and UIDB/50021/2020.

REFERENCES

1. Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases*. O'Reilly Media, Inc., 2013.
2. Florian Holzschuher and René Peinl. Performance of graph query languages: Comparison of cypher, gremlin and native access in neo4j. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, EDBT '13, pages 195–204, New York, NY, USA, 2013. ACM. URL: http://doi.acm.org/10.1145/2457317.2457351, doi:10.1145/2457317.2457351.
3. Aurelius. Titan: Distributed graph database., 2015. Last accessed 28 December 2020. URL: http://titandb.io.
4. The Linux Foundation. Janusgraph: Distributed graph database., 2017. Last accessed 28 December 2020. URL: http://janusgraph.org/.
5. Dgraph. Dgraph: A distributed, fast graph database., 2016. Last accessed 28 December 2020. URL: https://dgraph.io/.
6. F. Inc. Allegrograph., 2004. Last accessed 28 December 2020. URL: https://franz.com/agraph/allegrograph/.
7. Roshan Punnoose, Adina Crainiceanu, and David Rapp. Rya: A scalable rdf triple store for the clouds. In *Proceedings of the 1st International Workshop on Cloud Intelligence*, Cloud-I '12, pages 4:1–4:8, New York, NY, USA, 2012. ACM. URL: http://doi.acm.org/10.1145/2347673.2347677, doi:10.1145/2347673.2347677.
8. Bruno Lourenço, Cátia Vaz, and Alexandre Francisco. A framework for large scale phylogenetic analysis. *arXiv preprint arXiv:2012.13363*, 2020.
9. Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1433–1445. ACM, 2018.
10. Philip Rathle. Official release: 3 essentials of neo4j 3.0, from scale to productivity and deployment, 2016. Last accessed 28 December 2020. URL: https://neo4j.com/blog/neo4j-3-0-massive-scale-developer-productivity/.
11. Neo4j Contrib. Neovis.js. Last accessed 28 December 2020. URL: https://github.com/neo4j-contrib/neovis.js.
12. NHOGS Interactive. Popoto.js. Last accessed 28 December 2020. URL: http://www.popotojs.com/.
13. Andreia Sofia Teixeira, Pedro T Monteiro, Joao A Carriço, Francisco C Santos, and Alexandre P Francisco. Using spark and graphx to parallelize large-scale simulations of bacterial populations over host contact networks. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 591–600. Springer, 2017.
14. Spring Framework. Spring framework. *Available on:¡ https://spring. io/¿. Access in*, 3, 2018.
15. Google. Openidconnect. Last accessed 28 December 2020. URL: https://developers.google.com/identity/protocols/OpenIDConnect.
16. João Almeida, João Tiple, Mário Ramirez, José Melo-Cristino, Cátia Vaz, Alexandre P. Francisco, and João A. Carriço. An ontology and a rest api for sequence based microbial typing data. In Ana T. Freitas and Arcadi Navarro, editors, *Bioinformatics for Personalized*

Medicine, pages 21–28, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

17. Cátia Vaz, Alexandre P. Francisco, Mickael Silva, Keith A. Jolley, James E. Bray, Hannes Pouseele, Joerg Rothganger, Mário Ramirez, and João A. Carriço. Typon: the microbial typing ontology. *Journal of Biomedical Semantics*, 5(1):43, 2014. doi:10.1186/2041-1480-5-43.
18. J A Carriço, A J Sabat, A W Friedrich, M Ramirez, and Collective on behalf of the ESCMID Study Group for Epidemiological Markers (ESGEM). Bioinformatics in bacterial molecular epidemiology and public health: databases, tools and the next-generation sequencing revolution. *Eurosurveillance*, 18(4), 2013. URL: https://www.eurosurveillance.org/content/10.2807/ese.18.04.20382-en, doi:https://doi.org/10.2807/ese.18.04.20382-en.
19. Keith A. Jolley and Martin CJ Maiden. Bigsdb: Scalable analysis of bacterial genome variation at the population level. *BMC Bioinformatics*, 11(1):595, 2010. doi:10.1186/1471-2105-11-595.
20. BG Spratt. Multilocus sequence typing: molecular typing of bacterial pathogens in an era of rapid dna sequencing and the internet. *Current opinion in microbiology*, 2(3):312—316, June 1999. URL: https://doi.org/10.1016/S1369-5274(99)80054-X, doi:10.1016/s1369-5274(99)80054-x.
21. Martin CJ Maiden, Jane A Bygraves, Edward Feil, Giovanna Morelli, Joanne E Russell, Rachel Urwin, Qing Zhang, Jiaji Zhou, Kerstin Zurth, Dominique A Caugant, et al. Multilocus sequence typing: a portable approach to the identification of clones within populations of pathogenic microorganisms. *Proceedings of the National Academy of Sciences*, 95(6):3140–3145, 1998.
22. Bjørn-Arne Lindstedt. Multiple-locus variable number tandem repeats analysis for genetic fingerprinting of pathogenic bacteria. *Electrophoresis*, 26(13):2567–2582, 2005.
23. Nicholas J Croucher, Simon R Harris, Christophe Fraser, Michael A Quail, John Burton, Mark van der Linden, Lesley McGee, Anne von Gottberg, Jae Hoon Song, Kwan Soo Ko, et al. Rapid pneumococcal evolution in response to clinical interventions. *science*, 331(6016):430–434, 2011.
24. Alexandre P Francisco, Cátia Vaz, Pedro T Monteiro, José Melo-Cristino, Mário Ramirez, and Joao A Carriço. Phyloviz: phylogenetic inference and data visualization for sequence based typing methods. *BMC bioinformatics*, 13(1):87, 2012.
25. Marta Nascimento, Adriano Sousa, Mário Ramirez, Alexandre P Francisco, João A Carriço, and Cátia Vaz. Phyloviz 2.0: providing scalable data integration and visualization for multiple phylogenetic inference methods. *Bioinformatics*, 33(1):128–129, 2016.
26. Cátia Vaz, Marta Nascimento, João A Carriço, Tatiana Rocher, and Alexandre P Francisco. Distance-based phylogenetic inference from typing data: a unifying view. *Briefings in Bioinformatics*, 07 2020.
27. Alexandre P Francisco, Miguel Bugalho, Mário Ramirez, and João A Carriço. Global optimal eburst analysis of multilocus typing data using a graphic matroid approach. *BMC bioinformatics*, 10(1):152, 2009.
28. Christian Bachmaier, Ulrik Brandes, and Falk Schreiber. *Biological networks*. 2014.
29. Zhemin Zhou, Nabil-Fareed Alikhan, Martin J Sergeant, Nina Luhmann, Cátia Vaz, Alexandre P Francisco, João André Carriço, and Mark Achtman. Grapetree: visualization of core genomic relationships among 100,000 bacterial pathogens. *Genome research*, 28(9):1395–1404, 2018.
30. Ljubica Lazarevic. Keeping track of graph changes using temporal versioning. Last accessed 28 December 2020. URL: https://medium.com/neo4j/keeping-track-of-graph-changes-using-temporal-versioning-3b0f854536fa.
31. Spring. Spring framework documentation. Last accessed 28 December 2020. URL: https://docs.spring.io/spring-framework/docs/current/reference/html/web.html.
32. Microsoft. The oauth 2.0 authorization framework: Bearer token usage. Last accessed 28 December 2020. URL: https://tools.ietf.org/html/rfc6750.
33. Christophe Willemsen. Cypher: Write fast and furious. Last accessed 28 December 2020. URL: https://neo4j.com/blog/cypher-write-fast-furious/.
34. Andrew Bowman Neo4j Staff. Neo4j security. Last accessed 28 December 2020. URL: https://community.neo4j.com/t/neo4j-security/16044.
35. Luana Silva. phylolib. Last accessed 28 December 2020. URL: https://github.com/Luanab/phylolib.
36. Leonardo Alexandre Luana Silva and Diogo Loureiro. Phyloviz-electron. Last accessed 28 December 2020. URL: https://github.com/DrLDiogo/PHYLOViZ-Electron.
37. PubMLST. Pubmlst. Last accessed 28 December 2020. URL: Available:http://pubmlst.org/.