A framework for the indexing, querying, clustering, and visualization of microbial genomes for surveillance and outbreak investigation

by

Aaron Petkau

A thesis submitted to
The Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements
of the degree of

Master of Science

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada
August 2022

Thesis advisor Author

Olivier Tremblay-Savard and Gary Van Domselaar **Aaron Petkau**

A framework for the indexing, querying, clustering, and visualization of microbial genomes for surveillance and outbreak investigation

Abstract

Whole-genome sequencing (WGS) has increasingly become a routine part of monitoring infectious diseases. The genomes of bacteria, viruses, or other infectious agents are sequenced and used to identify nucleotide variants or other genetic differences—providing a wealth of detailed information. This has particularly become relevant with the COVID-19 pandemic, where sequencing of millions of viral genomes over the course of the pandemic has been essential in early identification of new viral lineages. The continuous generation of WGS data at this scale has introduced a number of challenges for efficiently generating timely reports and searching for epidemiologically significant patterns.

I have designed and implemented a framework to address these problems—the Genomics Data Index (`https://github.com/apetkau/genomics-data-index`)—which uses ideas from the field of information retrieval to transform WGS data into a collection of genomics features (*nucleotide variants*, *kmers*, and *genes*) and index these features for rapid querying. I provide a command-line interface and Python API for incrementally adding new data and querying the index. The query API integrates with existing methods for working with tabular and phylogenetic data to provide a common interface for clustering, visualization, and statistical analysis of microbial genomes.

I evaluated this framework using three datasets containing assembled genomes and sequence reads. Indexing assemblies was more sensitive for nucleotide variant detection when there were fewer variants (sensitivity = 0.948 for 6.77% divergence compared to reads sensitivity = 0.663), but sensitivity when indexing with reads surpassed assemblies as variants increased. The software was able to scale to tens of thousands of SARS-CoV-2 genomes (2.17 hours for loading 20,000 genomes) and construct phylogenies consistent with the existing Pangolin lineage system. Constructing phylogenies using nucleotide variants derived from bacterial WGS reads was found to consistently group outbreak-related bacteria into monophyletic clades (4/4 correct clades), while kmer clustering was able to group bacteria only at the species level.

I have already applied this software to aid in investigating new lineages of SARS-CoV-2, and I believe this software will be of great benefit for future research and genomic surveillance of other infectious diseases.

Contents

List of Figures

List of Algorithms

List of Listings

List of Tables

Acknowledgments

I would like to begin by thanking my advisors, my committee, my parents, and all the people who have supported me along the way. I would like to acknowledge the support of the Visual and Automated Disease Analytics graduate training program (VADA), which provided specialized courses, workshops, and training during my studies. I would like to thank Dr. Andrew G. McArthur and the other members of the McArthur laboratory for providing me with the opportunity to join as a visiting student in the Department of Biochemistry and Biomedical Sciences at McMaster University from July to August 2020. I would like to thank the members of the Bioinformatics laboratory and other students within the Department of Computer Science with the University of Manitoba for the amazing support and discussions during my studies. I would like to thank the members of the Bioinformatics laboratory and other groups within the National Microbiology Laboratory under the Public Health Agency of Canada for all the amazing support and opportunities I have received. I would also like thank both Mohsen Yousefian and Kimia Shadkami—two amazing students at the University of Manitoba who worked with me to develop a benchmarking system and web interface for the software presented in this thesis respectively.

This work is dedicated to my family, without which I would not have been able to complete my Master's degree. It is also dedicated to all those who have been impacted by the COVID-19 pandemic.

Chapter 1

Introduction

Infectious disease investigations are increasingly relying on whole-genome sequencing (WGS)—where the genome of the infectious agent is read by a machine and stored within a file—to aid in tracking the spread of the disease and identifying the contaminated source [1]. WGS provides a significantly greater amount of information over former methods, down to capturing single nucleotide variants (SNVs) in the genetic code. These variants can be used to reconstruct the genealogy of microbes and classify them into different subtypes—helping to track the origin and spread of an infectious disease. WGS is routinely used by many countries around the world to help track foodborne [2; 1] or human-transmitted [3] diseases and has experienced a significant uptake during the course of the COVD-19 pandemic [4]. While WGS is extremely useful, the process of translating the WGS data from large numbers of bacteria or viruses into a human-interpretable form can be difficult and time-consuming [1]. This requires the identification of genetic features, whether SNVs or entire genes, and comparing large collections of these features to cluster microbial genomes and assign human-readable identifiers to these clusters—defining a subtype (or sequence type) [5; 6]. This has become all the more important during the progress of the COVID-19 pandemic, where worldwide genomic sequencing has significantly contributed to our understanding of the evolution of the SARS-CoV-2 virus. The pandemic has lead to the extensive investigation into variation of the viral genome, novel typing methods [6], and has brought terms like "variant of concern"—variants of SARS-CoV-2 which are shown to have increased transmissiblity, virulence, or decrease the effectiveness of public health measures—into mainstream usage [7].

Identifying features within microbial genomes and clustering or classifying these genomes is part of the much broader topic of information retrieval [8]. Information retrieval studies the different methods used for processing large collections of data—from files on a personal computer to web pages on the internet—and provides mechanisms to quickly search through this data to find relevant information. Information retrieval methods are used by people every day, such as searching for information using search engines like Google or categorizing books found within a library. This field provides a rich background of information to draw upon for solutions to the management of large collections of genomics data.

1.1 Motivation

The primary motivation for this work is to aid the process of applying microbial WGS to infectious disease investigations and surveillance and ultimately to improve public health. The COVID-19 pandemic has brought microbial (in particular viral) genomics into the mainstream, which has greatly contributed to our understanding of the SARS-CoV-2 virus throughout the pandemic. Genomics has provided real-time information on the global evolution of this virus, which has provided opportunities for early intervention and response to emerging variants or mutations of concern [9]. However, prior to the COVID-19 pandemic, microbial genomics was already routinely applied for monitoring a variety of other infectious or foodborne illnesses. Organizations such as PulseNet [2] have adopted WGS for routine monitoring of a variety of foodborne pathogens—from *Salmonella enterica* to *Listeria monocytogenes*. The diversity of pathogens being monitored, desire for near real-time results, and increasing amount of data being stored and analyzed worldwide has led to unique challenges, which must be faced as the world continues to adapt to the routine use of genomics for infectious disease monitoring.

1.2 Challenges

1.2.1 Large-scale comparisons

Comparison of genetic features and assigning subtypes using whole-genome sequence data introduces a number of challenges, with one major challenge being scalability. Public archives contain petabytes of genomic sequence data and are growing daily [10]. The COVID-19 pandemic has further expanded the amount of data generated as part of the worldwide effort to sequence and archive SARS-CoV-2 genomes. The largest archive—GISAID—holds on the order of ten million genomes and is continually growing [11; 4]. Processing data at this scale requires complicated systems [2] or innovative indexing and searching techniques [12].

Dynamic data analysis

A related challenge arises due to the dynamic nature of data for genomic surveillance or outbreak investigations. Rather than being static and analyzed and published as a single unit, microbial WGS in these situations is dynamic, with a continual stream of new data being delivered, requiring up-to-date reports. This leads to situations where data analysis needs to be performed once for an earlier collection of genomes, and then new data needs to be integrated into the existing results on an on-going basis. However, some software used for genomic epidemiology [13] cannot integrate new data into an existing set of results—leading to a situation where analysis needs to be repeated from the beginning whenever new data is added. This is inefficient as computational resources are wasted repeating the same work over and over again.

1.2.2 Variety of analysis methods

Additionally, different types of use cases may benefit from different analysis methods—using different genomic features such as genes or SNVs [2; 3]—to monitor genetic changes over time and classify microbial samples into subtypes. For example, monitoring SARS-CoV-2 samples makes use of nucleotide-level variation, which is used to define different viral lineages [6]. However, monitoring *Salmonella enterica* may benefit more from gene-level information [14] to classify genomes into different subtypes in addition to nucleotide-level variation.

Likewise, clustering genomes using a variety of analysis methods and feature resolutions is challenging due to the frequent use of different specialized software suited to each unique feature (*e.g.,* gene-level [15] vs. nucleotide-level [13] features). Some clustering methods may be most appropriate for large-scale data collections due to their lower computational resource requirements but may offer poor separation of very closely related genomes [5]. It can be difficult to switch between these different resolution levels in one single system and may require ad-hoc methods to load the data (*e.g.,* nucleotide-level features into a system designed for gene-level features).

1.2.3 Missing data

Missing or unknown data refers to portions of the genome where there is uncertainty as to the exact nucleotide sequence. This could either be due to poor-quality sequencing in a particular region, strict filtering thresholds for removing potential false positive features, or could be related to the inherent limitations of different sequencing platforms (*e.g.,* short read lengths leading to gaps in the sequenced genome at repetitive regions [16]). Missing regions can lead to ambiguities in phylogenetic analysis [17; 18] (*i.e.,* collections of identically scoring trees, called a terrace) or can easily be misinterpreted as a region with no variation compared to some other genome.

1.2.4 Data integration

Genomic feature data is not the only type of information that is useful for infectious-disease data analysis. **Epidemiological metadata** (or contextual data) is also extremely important for the proper interpretation of genomics data [19]. This can include information such as the collection date of a genomic sample, the geographic region, or a variety of additional information. Additionally, genomics-derived results such as **phylogenetic trees** are a useful aid in interpreting the metadata [20].

Integration of genomic-derived data (genetic features) and these other types of data can be a multi-step process. For metadata, one method is to export genetic information into a tabular data structure and then load it within existing statistical data analysis packages for further processing (such as Python or R). For phylogenetic trees, packages such as the ETEToolkit [21] can be used to load and manipulate the tree. Once additional data is loaded, both R and Python provide programmatic APIs [22] to manipulate data to produce summaries or visualizations. Likewise, software such as Jupyter (`https://jupyter.org/`) [23] provide interactive web-based environments for writing and executing code for data analysis. However, the initial step of loading all the information into a common platform can be challenging as it requires defining and structuring multiple sources of data.

1.3 Solution

In this work, I focus on addressing the challenges of using WGS for infectious disease surveillance and investigations. I have designed and developed software which breaks WGS data up into individual genomic features, indexes the microbial genomes using a variety of feature types, and provides mechanisms to cluster and query indexed genomes. I have implemented a method to handle missing data by extending the standard boolean operations (*true*, *false*) to include a third truth value (*unknown*), which can be used for querying. I have implemented the ability to index based not just on a single type of feature, but on all of the common feature types used for microbial genomics (*nucleotide variants*, *kmers*, and *genes*). I also provide both a command-line interface and a Python-based API, which allows the integration and analysis of genomic and external data in a single system. An overview of the components of the software is provided in Figure 1.1 and listed below.

1. **WGS data analysis**. This component translates raw WGS data into collections of individual genomic features to be used for indexing.

2. **Genomics index**. This component loads the genomic features into a data structure that enables easy comparison across many genomes.

3. **Querying**. This component provides a mechanism to query the index of genomic features to ask questions such as "which genomes are closest to genome X" or "which genomes contain a particular feature"?

4. **Clustering**. This component clusters genomes using a variety of methods from the identified genomic features.

5. **Visualization**. This component provides summary statistics and visualization capabilities of the data stored in the index.

The software is called the **Genomics Data Index (GDI)** and is available on GitHub (`https://github.com/apetkau/genomics-data-index`). I have also included tutorials on the usage of this software with example data (`https://github.com/apetkau/genomics-data-index-examples`). This system should help as the world continues to adopt WGS as a routine technique for infectious disease investigations.

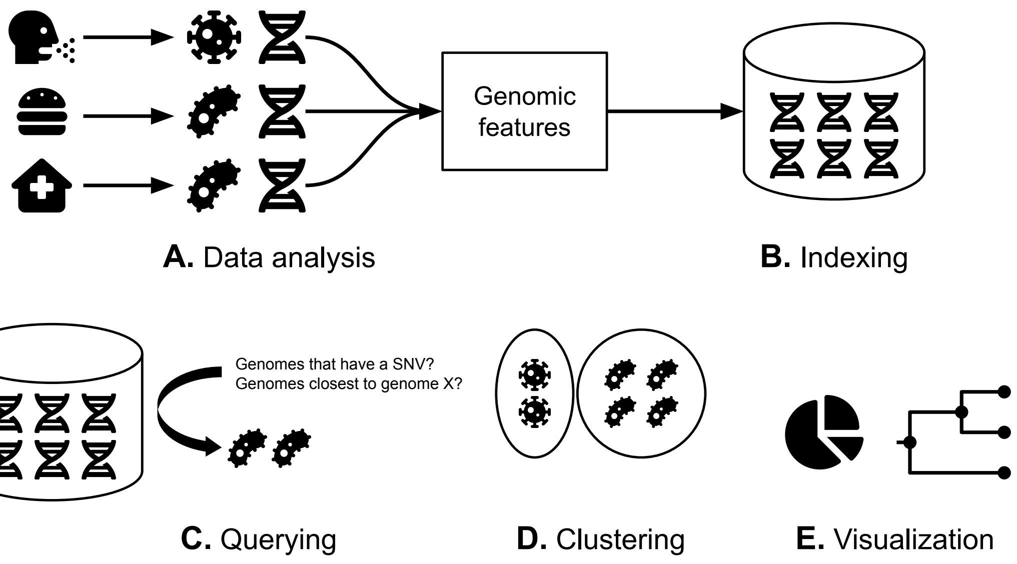


Figure 1.1: An overview of the Genomics Data Index software components. **(A) Data Analysis.** Whole-genome sequencing is performed on nucleic acids (DNA and RNA) derived from microbes and the sequence data is processed to produce sets of genomic features. **(B) Indexing.** The genomic features are used to construct an index for fast querying. **(C) Querying.** The index can be queried to find subsets of genomes which match specific criteria. **(D) Clustering.** Genomes loaded into the index can also be clustered to find closely related genomes. **(E) Visualization.** Data stored in the index can be visualized using the provided API.

Chapter 2

Background & Literature Review

In order to design the Genomics Data Index, I used a combination of existing work that intersects with both the **Information Retrieval** and **Microbial Genomics** fields of study. I introduce both of these topics and review existing methods. Finally, I end with an introduction to different **Workflow Management** software, which are useful when combining together many existing software components into a single data analysis pipeline.

2.1 Information retrieval

Information retrieval consists of the collection and searching through documents to find information of interest [8]. Information retrieval systems are found across a wide variety of use cases, from large-scale, like web search engines, to smaller and more specific cases, like searching through files on a personal computer. I provide a brief overview of topics and common terminology in this field.

2.1.1 Documents and terms

One key concept in information retrieval is a **document**, which is a unit of information that is of interest for a particular use case [8]. A document could refer to a book, a web page, a file on a desktop computer, a movie, or a song. Documents can be broken up into a smaller unit of information referred to as a **token** [8; 24]. For example, a particular book (document) consists of paragraphs, sentences, phrases, and words—each of which could be considered a token. Alternatively, a particular page within a book could be considered a document, while the words within a page could be considered tokens. The particular scale used to define a document (book, page, *etc.*) or a token (sentence, word, *etc.*) can vary depending on the use case. One of the goals of an information retrieval system is to retrieve documents relevant for a particular purpose. This can be accomplished through the use of a **query**, which can consist of one or more search terms. A **term** can be distinguished from a token in that a term refers to the common name we give to the same class of token across many documents (*e.g.,* the same word across many documents). A token is a particular instance of a term (*e.g.,* a word within a particular document) [8]. The set of all terms is referred to as a **vocabulary**.

2.1.2 Indexing

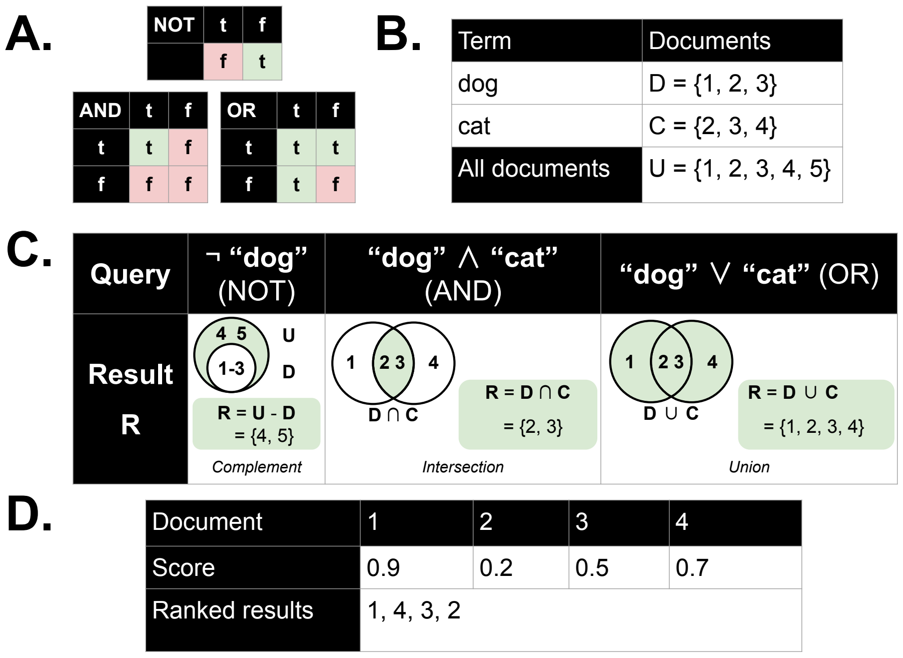
A common method of returning results related to a particular query is to first preprocess the documents to construct a data structure—an index—and use this index to aid in executing the query. A common index type used for information retrieval is referred to as an inverted index.

An **inverted index** is a data structure which matches terms found within collections of documents to information about the documents, such as a document identifier or the respective locations of the terms within each document [8]. Documents are first split into the individual tokens, a process called **tokenization**, and these tokens are organized into a **dictionary** of terms and information about the terms in the documents—the **postings list**.

A postings list can be **schema-dependent**, where the division of tokens into separate documents (the schema) is defined at the index time [8]. If the postings list matches terms to only the document identifiers containing these terms, then the index is called a **docid** index. If the frequency of the term is included alongside the document identifier, then the index is a **frequency index**. If the positions of the term within a document are included, then the index is a **positional index**. Finally, a **schema-independent** index is one where there is no division of terms into their respective documents defined at index time and thus instead must be defined at query time. This provides a mechanism to change the definition of a document for a particular query.

An index can also be **static**, that is the set of documents and terms does not change following the indexing process. By contrast, in a **dynamic** index, the documents and terms may change over time (*e.g.,* new documents may be added, or documents may be modified or removed).

In contrast to an inverted index, a **forward index** matches documents to the set of terms contained within that document. This can be used to produce summaries of terms within a particular document or set of documents.



A.

NOT	t	f
	f	t

AND	t	f
t	t	f
f	f	f

OR	t	f
t	t	t
f	t	f

B.

Term	Documents
dog	D = {1, 2, 3}
cat	C = {2, 3, 4}
All documents	U = {1, 2, 3, 4, 5}

C.

Query	¬ "dog" (NOT)	"dog" ∧ "cat" (AND)	"dog" ∨ "cat" (OR)
Result R	R = U - D = {4, 5}	R = D ∩ C = {2, 3}	R = D ∪ C = {1, 2, 3, 4}

D.

Document	1	2	3	4
Score	0.9	0.2	0.5	0.7
Ranked results	1, 4, 3, 2			

Figure 2.1: Basic information retrieval querying methods. **(A-C) Boolean retrieval**. **(D) Ranked retrieval**. **(A)** Boolean truth tables used for boolean retrieval. **(B)** An example set of terms which act as identifiers for sets of documents in boolean retrieval. Documents are numbered 1 to 5 and the letter U represents the *universe* of documents (*i.e.,* the collection of all documents in the index). **(C)** A set of example queries using boolean operators (NOT, AND, OR). These are implemented as operations on the sets of documents matching each term (*complement*, *intersection*, *union*), which are visually depicted using Venn diagrams with green representing the set of matching documents to a query. **(D)** In **Ranked retrieval**, scores are assigned to documents for a particular query which are used to order query results to return the most relevant results first.

2.1.3 Querying

Querying is the process where we search for the occurrence of particular terms within the inverted index and return a list of results. There are a number of different methods for query processing, with two basic categories being **boolean retrieval** and **ranked retrieval** (Figure 2.1).

Boolean retrieval

In boolean retrieval, terms are interpreted as referring to sets of matching documents and combinations of terms can be specified using boolean operations (AND, OR, NOT). These are converted to operations on the sets of associated documents (union, intersection, inversion/complement) to produce the final results [8]. As an example, let the term "dog" refer to the set of document identifiers $D = \{1, 2, 3\}$ and "cat" refer to the documents $C = \{2, 3, 4\}$ (Figure 2.1.B). In boolean retrieval, the query "dog AND cat" can then be converted to the set operation $D \cap C = \{1, 2, 3\} \cap \{2, 3, 4\} = \{2, 3\}$. The query "dog OR cat" can be converted to $D \cup C = \{1, 2, 3\} \cup \{2, 3, 4\} = \{1, 2, 3, 4\}$ (Figure 2.1.C).

Ranked retrieval

Ranked retrieval is used to rank the results of a query and return only the most-relevant (best-ranked) results. This can be applied all on its own to respond to a query or can be applied following boolean retrieval to further refine the set of results, which may be unreasonably large. Ranking can be performed using a variety of information, such as the proximity of query terms in a document. As an example, consider the query "dog OR cat" shown above, which results in the documents $D \cup C = \{1, 2, 3, 4\}$. These could be further refined to assign a score to each document based on some measure of how well it matches the query. For example $\{(1, 0.9), (2, 0.2), (3, 0.5), (4, 0.7)\}$ (given as (*docid*, *score*)). These ranked results can then be sorted to return only the most relevant results first $[(1, 0.9), (4, 0.7), (3, 0.5), (2, 0.2)]$ (Figure 2.1.D).

2.1.4 Categorization, classification, and clustering

Categorization is the process of assigning labels to documents to group them into particular categories [8]. The categories to use are provided ahead of time and depend on the particular use case. When categorization is performed automatically by first learning to distinguish between patterns in a set of example data, it can be termed **classification**, which is a machine-learning approach to automatically assign items (*e.g.,* documents) to a particular class [8; 25].

Classification can be both **supervised**, where the learning dataset includes the class labels, or **unsupervised**, where the learning dataset does not include the class labels. An alternative term for unsupervised classification is **clustering** [25]. As there is no prior knowledge of the categories to assign to an item, the categories could be termed **clusters**, and the goal is to learn from only the information contained in the dataset which clusters an item should be assigned to.

Classification and clustering may make use of **features**—properties about a document that could be useful for clustering—which can be organized into a vector. Features may consist of tokens in a document, frequencies of tokens, or other information. There are a number different types of clustering techniques [25], with two of these types being **hierarchical clustering** and **partitioning**.

Hierarchical clustering

In **hierarchical** clustering, clusters are built iteratively from the data, making use of some distance measure between pairs of data elements [25]. Building clusters from the bottom-up, where each data element is initially considered its own cluster and these clusters are joined based on some distance criteria, is called **agglomerative** clustering. Conversely, **divisive** clustering is a top-down approach where all the data is considered initially in the same cluster and then iteratively split into separate clusters based on some distance measure. The hierarchical structure of these clusters leads to a representation of these clusters within a tree, which can be drawn as a dendrogram. Leaf nodes in the tree represent individual data elements while internal nodes represent clusters of more than one element.

The iterative process of merging or splitting clusters requires some definition of distances between clusters in addition to distances between data elements—a **linkage distance**. These linkage distances can be one of three varieties: **single-linkage**, **complete-linkage**, and **average-linkage**. In **single-linkage** clustering, the distance between two clusters is defined as the minimum (*i.e.,* closest) distance between any pair of points, where one point is in each cluster. **Complete-linkage**, by contrast, defines the distance between clusters as the maximum (*i.e.,* furthest) distance between any pair of points, one point in each cluster. **Average-linkage** clustering defines the distance between clusters as the average distance between all pairs of points, one point per cluster. If similarity scores are used instead of distances, then these linkage categories still apply but would have to be slightly adjusted (*e.g.,* by using the *maximum similarity* for **single-linkage** as opposed to the *minimum distance*).

Due to the hierarchical nature of this approach, a single data item can be contained in multiple clusters of increasing size (from a cluster containing only the item itself to a cluster containing all items in the dataset). To convert this cluster hierarchy into a **flat partition**—a partition of the dataset where each item is contained only in a single cluster—we can choose different portions of the cluster hierarchy to divide our dataset [26]. This can be illustrated using a dendrogram where constructing a flat partition could be considered as equivalent to cutting the dendrogram at some particular distance threshold (or multiple distance thresholds).

Partition clustering

In contrast to the multiple levels of nested clusters generated from hierarchical clustering, in **partition clustering**, data elements are assigned to one of a number of clusters without any hierarchical structure [25]. One method is **k-means** clustering, which iteratively partitions the data items into k different clusters. This can be visualized by imagining that each data element exists as a point in some multi-dimensional space with distances between data elements corresponding to distances between points. The number of clusters, k, is chosen ahead of time and k random data points are chosen initially to represent the centers of each cluster (the *centroids*). The distance between every item in the dataset and the centroids is computed (using a distance metric, often Euclidean [27]) and each item is assigned to the cluster with the closest centroid. New centroids are computed for each of the k clusters based on the data points within each cluster. Every item in the dataset can then be re-assigned to clusters based on these new centroids. This process is iteratively repeated until the centroids converge to some stable position. The data elements assigned to each of the k centroids forms the k different clusters.

2.2 Microbial genomics

2.2.1 Microbial whole-genome sequencing

Whole-genome sequencing (WGS) is the process of converting the entire genome of an organism from its genetic material into a form that can be used for further analysis [28]. The genome of an organism consists of the nucleic acid, typically DNA but sometimes RNA, that encodes the instructions for the molecular machinery used to perform different functions within the organism. This code consists of long chains of linked units in a polymer—the nucleotides—consisting of four different types: Adenine (A), Thymine (T), Guanine (G), and Cytosine (C) [29]. In RNA, Thymine (T) is replaced with Uracil (U). The order of the nucleotides defines the genetic code and can be represented as a string (*e.g.,* ATCGG). Hence, the output of whole-genome sequencing consists of strings from the alphabet {A, T, C, G} (or {A, U, C, G} for RNA)—often broken into small fragments called reads that are hundreds to tens of thousands of characters long, depending on the underlying technology used to generate the sequence [30]. Microbial WGS focuses specifically on the genomes of microbes (bacteria, viruses, *etc.*).

There are a number of different instruments and methods for performing whole-genome sequencing. One common method is sequencing by synthesis using the Illumina-based sequencing instruments (`https://www.illumina.com`), which produce shorter reads (few hundred base pairs), but the reads often have fewer errors than other sequencing technologies [30]. Two common methods to sequence genomes using Illumina instruments are single-end sequencing or paired-end sequencing. **Single-end** sequencing starts from only one of the two ends of a DNA/RNA fragment to produce a single read. **Paired-end** sequencing will read a DNA/RNA fragment once from both ends, producing a pair of reads from opposite ends of the fragment. This pair of reads provides more information than a single read, and this additional information can be used downstream to improve data analysis [31]. Another sequencing technology is Oxford Nanopore [30], which produces much longer reads than Illumina (thousands of base pairs) but suffers from a larger number of errors.

Translating these genomic sequence reads into useful results requires careful data analysis, which can be broken up into a number of broad stages: **primary analysis**, **secondary analysis**, and **tertiary analysis** [16].

2.2.2 Primary analysis

This first stage involves assessing the quality of the sequence reads and possibly removing poor-quality reads from downstream analysis. During the sequencing process, it is possible that errors are introduced, which means that sequence reads will not necessarily match the nucleotides in the DNA/RNA molecule (*e.g.,* the reads could have an `A` in a particular location when the underlying DNA molecule actually has a `T` in this location).

To account for these errors, reads are often combined with an additional set of quality scores, which represents the probability of a particular nucleotide being incorrect. These scores are represented using a logarithmic scale called the **phred scale** [32], which relates the probability of an error in a nucleotide, P, to a quality score, Q, according to the formula $Q = -10\log_{10}(P)$. These scores are distributed together with the sequence reads using a file format—called FASTQ [32]—which uses an ASCII encoding to represent both the nucleotides (*e.g.,* A, C, T, G for DNA) and quality scores (which are converted to an integer and stored starting with ASCII 33 or `!` representing a score of 0).

A second method to account for errors in the nucleotides of individual reads is to increase the redundancy of the reads. That is, to generate an entire collection of overlapping reads with the idea that even if one nucleotide character in a single read is incorrect it can be accounted for by other overlapping reads to generate a consensus sequence that is, ideally, error-free. This redundancy is referred to as the **depth of coverage** (or sometimes just **coverage**), which refers to the amount of redundancy from overlapping reads for any given nucleotide in a genome [16]. For example, a coverage value of 10X for a particular nucleotide on a genome means that there are 10 reads overlapping this particular nucleotide (so if a single read has an error in this position it can be detected and accounted for by the other 9 reads). If there are N reads that are the same length, L, then the mean coverage, C, across an entire genome of length G can be estimated as $C = LN/G$ [33; 16]. If the reads do not all have identical lengths, then the sum of lengths of all reads can be used in place of LN [13] (*i.e.,* $C = \sum_{r \in R} \mathit{length(r)}/G$, for the set of reads R).

2.2.3 Secondary analysis

Once the primary analysis is complete and the sequence reads are validated to be of a high-enough quality, data processing can proceed to the secondary analysis. This includes two separate, but sometimes complementary, data analysis paths, referred to as **genome assembly** and **reference alignment**.

Genome assembly

Genome assembly is the processes where the small sequence fragments (reads) are fit together to attempt to re-construct a string of characters representing the full genome [16]. The output of the genome assembly process is long contiguous sequences of characters—called **contigs**—which represent, as closely as possible, the biochemical organization of the original DNA or RNA molecule. It is not always possible to reconstruct the order of nucleotides for the entire molecule in one single contiguous sequence due to repetitive regions, which create ambiguities on how the reads derived from the DNA/RNA molecule fit together [34].

Genome assembly can be further categorized as ***de novo* assembly** or as **reference guided** assembly. In *de novo* assembly there is no reference genome used as an aid during the assembly process [16] and instead the final set of contigs are constructed using only the sequence reads. Common *de novo* assembly software often makes use of a graph, with two broad categories being overlap-layout-consensus (OLC) and de Bruijn approaches [34]. In OLC, a graph is constructed which encodes overlaps between reads as edges between nodes (overlap). This graph is then used to determine the order of the reads within the original genome (layout) and finally produce a collection of contigs representing the genome (consensus). Alternatively, de Bruijn approaches first break reads up into smaller fragments of size k—called a kmer—and construct a special graph (a de Bruijn graph) which joins sequential kmers together with an edge. Paths through the de Bruijn graph can then be followed to reconstruct the original genome (as a set of contigs). Many modern *de novo* genome assembly software follow the de Bruijn graph approach, with some examples being SPAdes [35] and SKESA [31].

Reference alignment

As an alternative (or complementary) approach to genome assembly, reference alignment can be used when a reference genome is available that is closely related to the sequenced genome [16]. The reference genome provides a guide for the locations where individual sequence reads (or read pairs) should be best positioned. The choice of reference genome for this analysis can have a large impact on the results, with the ideal reference genome being as closely related to the sequenced genome in question. More distantly related reference genomes can lead to issues with aligning reads to the reference or can result in reads that do not align to any region on the reference genome as the particular region of the genome they are derived from does not exist in the reference genome.

There is a varied collection of software for read alignment to a reference genome, which often relies on breaking up reads into short fragments (*e.g.,* kmers) to aid in finding the best position to place a read. Examples include BWA [36] for aligning short reads, BWA-MEM [37] or Bowtie2 [38] for longer reads and Minimap2 [39]. One of the advantages of Minimap2 is the ability to also align full-length genomes against a reference genome. This can be useful if reads are not available and you wish to treat a genome assembly the same as other read datasets.

2.2.4 Tertiary analysis

Tertiary analysis refers to any analysis that completes the final step from processing WGS reads into some human-interpretable result [16]. This can include a variety of applications, such as identifying genes within an assembled genome or comparisons of large collections of genomes. When these genomes are derived from microbes related to an an infectious-disease outbreak, the genetic content can be used to infer epidemiological relatedness of the pathogens (*i.e.,* the transmission network of the disease) [30].

Worldwide organizations—such as PulseNet International [1; 5] or GenomeTrakr [40]—have been increasingly using WGS for infectious-disease investigations, and the ongoing COVID-19 pandemic has accelerated the application of WGS for infectious-disease tracking [6]. Two major methods are often applied for infectious disease investigations or monitoring [20; 5]: **microbial sequence typing** and **phylogenetics**. **Microbial sequence typing** is where genomes are categorized into separate groups based on shared genetic content [20]. This is a form of categorization and classification (Section 2.1.4) applied to microbial whole-genome sequencing data. Assigning a sequence type can be useful during an infectious-disease outbreak investigation, as it provides a simple identifier to aid in communication [5]. This method can be applied both to bacterial genomes (such as identifying subtypes of *Salmonella enterica* [14]) as well as to viral genomes (such as the Pangolin system [6] for typing SARS-CoV-2, the virus causing COVID-19). Alternatively, **phylogenetics** is the process of constructing a hypothetical model of the geneaology of a collection of genomes—represented as a phylogenetic tree. This could be considered a form of hierarchical clustering applied to whole-genome sequence data (Section 2.2.5). Phylogenetics—like other clustering methods—does not require specific labels (sequence types) to be defined ahead of time. This means phylogenetics can be used to aid in identifying and naming novel clusters [41], which can then be used to help build a classifier to automatically assign sequence types to genomes [6]. Both phylogenetic and microbial sequence typing methods are specific applications of classification and clustering that are part of machine learning. As such, they both require as input sets of features [25]—properties derived from the genomics data. A variety of features and algorithms that make use of these features have been developed. These can be divided into three different approaches: **nucleotide variant**, **kmer-based**, and **gene-based** methods [5]. Each feature type may be used for both microbial sequence typing or for phylogenetic analysis (Figure 2.2).

Nucleotide variant approaches

A nucleotide variant is some variation at the nucleotide-level between genomes [42]. This could include single-nucleotide variants (SNVs)—where a single nucleotide in the genome is substituted for another nucleotide (*e.g.,* an A is substituted for a T)—or insertions/deletions of nucleotides (indels). These methods commonly focus on SNVs and so could also be called SNV-based approaches. SNVs can be referred to as a Single Nucleotide Polymorphisms (SNP) if the substitution occurs above some threshold frequency in a population [43], but sometimes SNV and SNP are used interchangeably [13]. Nucleotide variant approaches use WGS data to identify SNVs (or other variants) between organisms and use the SNVs to infer their evolu-

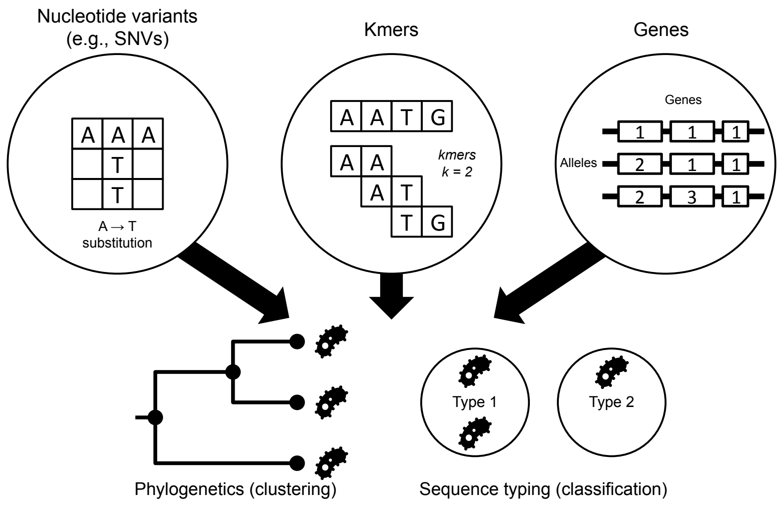

Nucleotide variants (e.g., SNVs)
Kmers
Genes
A→T substitution
kmers k = 2
Genes
Alleles
Type 1
Type 2
Phylogenetics (clustering)
Sequence typing (classification)

Figure 2.2: Microbial comparison and typing methods. Three different types of features can be used for comparing genomes at differing levels of resolution: **nucleotide variants**, **kmers**, and **genes**. Each feature can be used either for phylogenetic analysis (clustering of genomes), or for microbial sequence typing (classifying genomes into existing categories).

tionary relationship, represented as a phylogenetic tree [20].

Software implementing the nucleotide variant approach include the CFSAN SNP Pipeline [44], Lyve-SET [45], SNVPhyl [13], and snippy [46]. These software programs operate by comparing the WGS data to a reference genome to identify nucleotide variants (using the *reference alignment* approach). Often, one of the main outputs is a phylogenetic tree that, when combined with epidemiological metadata, can be used to help understand the transmission of a particular disease [20].

To facilitate the storage and querying of identified nucleotide variants, the SnapperDB system [47] provides a database for indexing and searching through collections of identified SNVs, along with a method to assign hierarchical codes to different SNV (SNP) combinations—called a SNP address. While SnapperDB was primarily designed for microbial genomes, it is capable of handling a wide variety of organisms through user-provided reference genomes used to identify SNVs. There also exists organism-specific databases as well, such as NextStrain [3], CovidCG [48] and outbreak.info [49] for the storage and investigation of SNVs derived from SARS-CoV-2 (or other viral) genomes.

Storing nucleotide variant-level data has an advantage beyond sequence typing, where individual variants can be monitored to see if they are increasing in frequency, which could potentially be an indication of a variant associated with a higher transmissiblity or immune escape [9].

Kmer-based approaches

Whereas nucleotide variant approaches operate on single nucleotide differences between genomes (or other small variants), kmer-based approaches break up genomes into fragments of length k, a $k$mer, and compare these kmers to help classify or cluster bacteria. While many traditional data analysis methods, such as *genome assembly* or *reference alignment*, may rely on breaking up sequences into kmers or small fragments, they often have additional steps which transform the kmers into some final result (*e.g.,* a *de novo* assembly). The advantage of kmer approaches for clustering or classification of microbial genomes is that they operate directly on kmers to produce their final result, with few intermediate steps. Thus, they can skip some of the secondary analysis methods to save on time or memory.

Mash [50] is one such kmer-based approach, based on the MinHash [51] algorithm originally developed to detect similarities in documents. In Mash, genomes are broken up into kmers and MinHashing is applied to select a subset of kmers and store the associated hash codes in a reduced dataset called a **sketch**. Sketches have the advantage of greatly reducing the number of kmers to record for each genome, while still retaining the ability to approximate the similarity between two genomes (defined as the Jaccard index [52] or the proportion of shared kmers between genomes). The similarity scores procured from the sketches can be used to hierarchically cluster genomes into related groups [53]. More recent software using the MinHashing approach includes sourmash [54], which includes the ability to construct and save a sketch using multiple kmer sizes, which can be useful to help balance speed vs. accuracy when performing MinHash comparisons.

A second method, which focuses on indexing large amounts of genomics data, is BIGSI [12]. BIGSI indexes kmers using a bit-sliced signature index—previously used by web search engines such as Bing [55]—to develop a system for rapidly searching for genes or other small sequences of interest within hundreds of thousands of bacterial genomes. COBS [56]—an extension of BIGSI—attempts to reduce the storage space required for generating these large indexes through variable-sized (as opposed to fixed-sized) Bloom filters used by the bit-sliced signature indexes.

Another interesting method—that intersects with kmer based and nucleotide variant-based approaches—is BioHansel [57]. BioHansel identifies SNVs which differentiate bacterial subtypes and organizes the SNVs into a SNV scheme, which associates sets of SNVs with particular subtypes. Next, kmers which contain the SNVs are used to rapidly screen and classify new WGS data, assigning each dataset a subtype derived from the pre-defined SNV scheme.

Gene-based methods

Gene-based methods operate at the level of genes—a fundamental unit of a genome consisting of a sequence of nucleotides that encodes some particular component of the machinery used by the organism [29]. Genes are of variable length and are much larger than the typical value of k chosen for kmers. Gene-based approaches include multi-locus sequence typing (MLST)—used to classify bacteria based on differences within particular genes [58]. In MLST, each gene represents a particular locus (location on the genome) to examine for variation, and the set of loci (genes) together define a scheme. Every unique variant of a gene (called an allele) is assigned a number (*e.g.,* allele 1). The set of allele numbers for all loci together are assigned a specific multi-locus sequence type. Different sequence types can be used to delineate different lineages of an organism.

While originally proposed to operate on small numbers of loci, MLST has since been extended to include many more loci—up to all the genes in a particular organism [59]. These extended MLST schemes are referred to as core-genome MLST or whole-genome MLST. In core-genome MLST only those genes shared among a group of organisms (the core genes) are used. Whole-genome MLST includes both the core genes of a group of organisms (such as a species) as well as genes that may only be found in a subset of this group of organisms (the accessory genes) [20].

There is a diverse set of software packages for identifying the MLST types of a collection of bacteria. One of the earlier programs is SRST2 [15], which identifies sequence types by comparing WGS reads to a curated set of reference loci (using *reference alignment*). More recent software—such as MentaLiST [60] and STing [61]—avoid the complexity of reference alignment used by SRST2 by directly working with kmers. Finally, there exists species-specific software, such as SISTR [14] for *Salmonella*, which uses core-genome MLST alongside additional classification methods. Comparison of MLST-based methods to SNV-based methods suggests that they tend to produce fairly concordant results when classifying closely related organisms [45]. For more distantly related organisms, SNV-based methods may detect a greater variation since they can account for multiple nucleotide differences within a single gene.

Central data repositories are used to to compare and track large numbers of MLST schemes and data. One popular repository is PubMLST.org, which hosts MLST schemes for a large collection of bacteria and other organisms [62]. Originally designed to work with basic 7-gene MLST, PubMLST—backed by the software BIGSdb [63]—has grown to encompass both core-genome and whole-genome MLST schemes. A similar repository is Enterobase [64], which focuses on maintaining whole-genome and core-genome MLST schemes and classifications of enteric bacteria—for example containing classifications for over 100,000 Salmonella genomes. One interesting feature of Enterobase is the hierarchical division of bacteria into clusters (called HierCC) based on similarities in their identified MLST alleles at differing thresholds [65].

2.2.5 Phylogenetics

Phylogenetic trees are a model of the genealogy of some particular biological entity as a tree [66; 67]. One example is the relationship of all living organisms on earth, which can be modeled with a tree where leaf nodes correspond to specific named species and internal nodes in the tree corresponding to known or hypothetical ancestors. Such a tree would be considered a *species tree*—as it models the genealogy of different species—with internal nodes corresponding to speciation events. An alternative type of phylogeny would be a *gene tree*, where the tree is a model of the genealogy of a collection of genes. Phylogenetic trees can either be **rooted**—where one node is a parent of all other nodes—or **unrooted**. A **clade** (or monophyletic group/clade) is a collection of organisms which all share the same ancestor, represented on a tree as a node and all of its descendants [68] (*i.e.,* a sub-tree rooted at a chosen node). There have been a number of different approaches for constructing a phylogeny, but two large categories are **distance-based** and **character-based**.

Distance-based methods

In **distance-based** methods, only the pairwise distances between different genomes are considered for constructing a phylogenetic tree. These distances can be calculated using a number of methods and evolutionary models. When pairwise distances are all defined, then constructing a tree can proceed through a variety of methods, which include the hierarchical clustering algorithms already discussed. In particular, the **UPGMA** method (unweighted pair group method with arithmetic averages) is a type of *agglomerative hierarchical clustering* for constructing a phylogenetic tree that uses average-linkage distances [67; 26] and produces a rooted tree. The cluster-tree can be interpreted as a phylogeny, where leaf-nodes correspond to individual sampled genomes, while internal nodes correspond to hypothetical ancestors. UPGMA, due to its use of average-linkage distances, also produces an ultrametric tree, which can be interpreted to represent a scenario of a constant molecular clock (*i.e.,* all genomes evolve at the same rate with respect to time) [67]. This assumption may or may not be realistic. An alternative algorithm, which does not require this molecular clock assumption, is **neighbor-joining** [67; 66]. Unlike UPGMA, **neighbor-joining** is a form of *divisive hierarchical clustering*, which does not produce a rooted tree, and so a root would have to be identified using some other method (such as including an outgroup in the tree). Both UPGMA and neighbor-joining still require the assumption of distances being *additive*, which means that the distance between genomes should be equal to the sum of distances along the path in the tree connecting them [67]. The additive property may or may not hold depending on the distance measure between genomes.

Character-based methods

In contrast to distance-based methods, **character-based** methods make use of the actual character data (either nucleotide or amino acid) [66]. This character data is provided in the form of a multiple sequence alignment, where the individual characters are organized into columns (a site in the alignment) containing homologous characters. Ideally these homologous characters should all have some shared ancestry (a common ancestor), which is what lets us use the sites of the alignment to infer a phylogeny reflecting the shared ancestry of the sequences as a whole. For microbial whole-genome sequence data, these multiple sequence alignments can be constructed either using the *assembled genome* or through a *reference-alignment* approach.

Three common character-based methods include: **maximum parsimony**, **maximum likelihood**, and **Bayesian**. Maximum parsimony takes the approach of assigning characters to internal nodes of a tree (representing hypothetical ancestors) to minimize the number of character changes required to explain the observed sequences in the leaves of the tree. The tree requiring the fewest character changes (and hence maximal parsimony) is chosen as the tree that best represents the genealogy of the sequences [66].

In contrast to maximum parsimony, both maximum-likelihood and Bayesian methods are probabilistic methods [67]. If we let T represent a tree (*i.e.,* a model of the shared ancestry of sequences) and let D represent our input data (the alignment), then $P(D|T)$ represents the probability of our data given a particular tree. In maximum-likelihood, we are treating $P(D|T)$ as a function of T (we are searching for a particular tree T where we already have the data D), hence $P(D|T)$ is referred to as a likelihood (instead of a probability), and our goal is to find the tree that maximizes this likelihood. Conversely, for Bayesian methods, we are interested in $P(T|D)$, which we can related to $P(D|T)$ through Bayes' theorem. Additional details on these methods can be found in [66; 67].

Evolutionary models

For the probabilistic character-based methods (as well as for distance methods) we can make use of different models of evolution to define distances between sequences (genomes) [66; 67] and make corrections to better reflect the true evolutionary distance. For nucleotide data, these models attempt to account for real-world differences in rates between nucleotide substitutions (*e.g.,* possible differences between an A $\rightarrow$ T vs. A $\rightarrow$ G substitution). The simplest model is JC69 [69], which assumes an equal rate for all possible substitutions from one nucleotide to another. More complicated models such as K80 [70] (different rates between transitions and transversions) or HKY85 [71] (extends K80 to include differences in equilibrium frequencies of nucleotides) introduce additional parameters to capture differences in substitution rates. The GTR [72] model (General Time Reversible) is the most complicated model that is also time-reversible (*i.e.,* the rate of any substitution like A $\rightarrow$ T is the same as its reverse T $\rightarrow$ A [66]). GTR includes the previous described models within it by assuming certain parameters are fixed (*e.g.,* equal rates for all substitutions and equilibrium frequencies for the JC69 model).

Phylogenetics and clustering

As described in Section 2.1.4, clustering is an unsupervised machine-learning approach to partition data into different categories (called clusters). Of the different methods for clustering, hierarchical clustering will group data into separated clusters by organizing these clusters into a larger tree, which can be drawn as a dendrogram. Both UPGMA and neighbor-joining are forms of hierarchical clustering where the constructed clustering tree can be interpreted as a phylogeny. Additionally, we can construct flat clusters/partitions (as opposed to hierarchical clusters) from either of these trees (or any hierarchically clustered tree) by cutting this tree at different distance thresholds [26]. This method works the best when the tree is ultrametric (all path lengths from the root to the leaves are the same), which is the situation for UPGMA-produced trees but not neccessarily for other types of phylogenies (like neighbor-joining or maximum-likelihood) [73].

There has been a number of other methods developed to attempt to group genomes into flat clusters by making use of a phylogenetic tree and which do not need the tree to satisfy the ultrametric property. One such method is *Cluster Picker* [74], which identifies flat clusters from a rooted phylogenetic tree by searching for monophyletic clades (sub-trees) that match user-defined thresholds. Cluster Picker starts from the root of a phylogeny and examines each node through a depth-first search, pruning the clades rooted at each node that fail either branch support or pairwise-distance thresholds. The nodes (and corresponding clades) that pass these thresholds are returned as separate clusters, with leaves of each clade corresponding to the members of each cluster. An alternative method is *TreeCluster* [73], which attempts to cut a phylogenetic tree (either rooted or unrooted) into the minimal number of sub-trees such that each sub-tree satisfies some defined constraint (such as a maximum pairwise-distance threshold). With a rooted tree, TreeCluster can use the additional constraint of restricting each cluster to a monophyletic clade.

2.3 Feature identification and storage

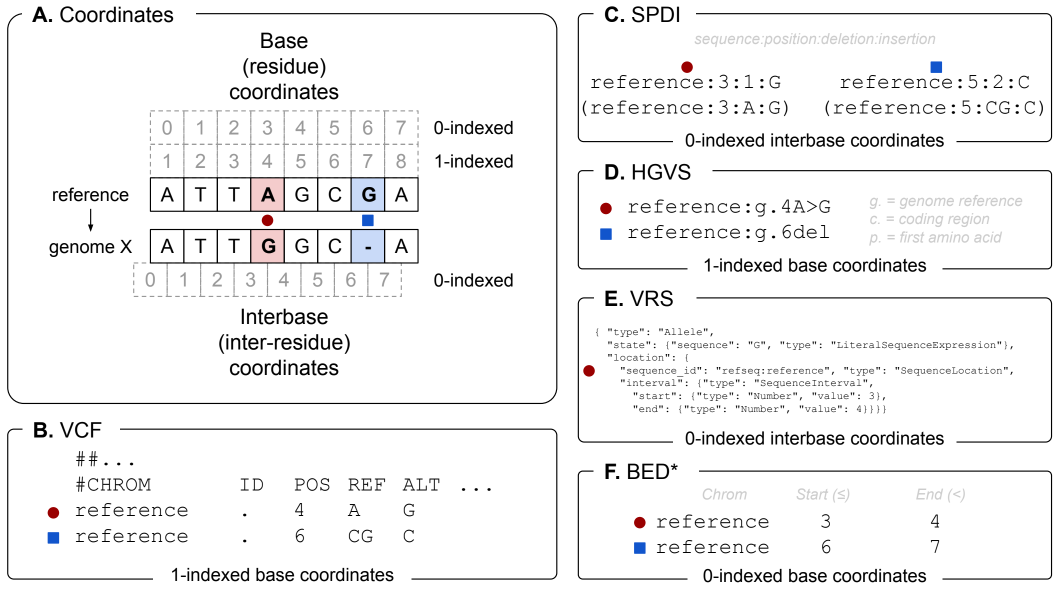
To identify, store, and operate on features from the three identified levels—**nucleotide variants**, **kmers**, and **genes**—there is a need for standard data models and file formats. A number of these have been previously identified and described, particularly for nucleotide variant data.

2.3.1 Nucleotide variant models

A nucleotide-level variant describes some variation in the underlying nucleotide sequence that make up either the DNA or RNA molecules found within an organism [75; 76]. Often, these are described as a change from a reference sequence to some sequence of interest (the alternative sequence) at some particular location. For example, `A → T` at position `10` describes a substitution where the reference nucleotide, `A`, changes to a nucleotide `T` at position `10` of the reference genome. For microbial WGS-based analysis, the reference sequence could be a previously sequenced genome that is stored on public archives (*e.g.,* RefSeq [77]), or could be some particular chosen organism within a collection that has undergone *de novo* assembly (Section 2.2.3) but is not otherwise available in public archives. A number of common data models and file formats for describing nucleotide-level variation are shown in Figure 2.3 and are described below.

VCF

One common storage format for nucleotide-level variants is the Variant Call Format (VCF) [75] (Figure 2.3.B), which is a text-based format for storing nucleotide



```
##...
#CHROM        ID   POS  REF  ALT  ...
● reference   .    4    A    G
■ reference   .    6    CG   C
```

```
●  reference:3:1:G          ■  reference:5:2:C
(reference:3:A:G)          (reference:5:CG:C)
```

```
● reference:g.4A>G
■ reference:g.6del
```

```
{ "type": "Allele",
  "state": {"sequence": "G", "type": "LiteralSequenceExpression"},
  "location": {
●   "sequence_id": "refseq:reference", "type": "SequenceLocation",
    "interval": {"type": "SequenceInterval",
      "start": {"type": "Number", "value": 3},
      "end": {"type": "Number", "value": 4}}}}
```

```
● reference     3     4
■ reference     6     7
```

Figure 2.3: An overview of different nucleotide-level variation representation models, with an emphasis on the coordinate systems used (labeled at the bottom of each box). **(A)** The different classes of coordinate systems used by each model. These are either *base* (or *residue*) coordinates, which count each nucleotide in a sequence, and *interbase* (or *inter-residue*) coordinates, which count the spaces in between each nucleotide in a sequence. These classes can then either be *0-indexed* (starts counting from 0) or *1-indexed* (starts counting from 1). Two example nucleotide-level variants are represented, labelled by a red circle (a substitution) and a blue square (a deletion). **(B)** The VCF [75] model and file format. Each nucleotide-level variant is represented as a separate row in a tab-delimited file, which records the sequence identifier (*CHROM*), position (*POS*), reference nucleotides (*REF*) and alternative nucleotides (*ALT*). **(C)** The SPDI [42] model. Each variant is represented by four attributes (sequence, position, deletion, insertion) concatenated by a `:` that describes how to transform a reference sequence into the sequence with the variant. The deletion string can have two different representations, either as the number of nucleotides to delete from the reference (the top identifier) or the nucleotide-string to delete (the bottom identifier). **(D)** The HGVS [78] model. This represents each variant by a sequence identifier (`reference`) followed by the type of sequence (`g.` for a genomic reference) and the position and type of variation. **(E)** The VRS [76] model. Each variant is represented by a hierarchical data structure written as JSON that stores the type of variation and the location. Only the red-circle substitution variant is shown. **(F)** The BED [79] format. This stores intervals on a genome given as a sequence identifier (*Chrom*) a *Start* and an *End* in a tab-delimited file format. The intervals are inclusive of the start, but exclude the end (i.e. the half-open interval: $start \leq p < end$ for all integers p). *BED differs from the other models in that it is primarily used to store intervals on a genome and not variants.

feature records—one record per line. Each record is divided up into a number of columns separated by tab characters. The columns include a name for the chromosome (`CHROM`), the position of the variant (`POS`), the reference genome sequence (`REF`), and a list of alternative alleles that replace the reference sequence (`ALT`). VCF files can also be used to store a lot of additional statistical or other information related to a variant. They can even be used to store a list of samples that have a particular variant (and so could be used as an inverted index that matches variants to sets of samples).

SPDI

A second, but related, model for storing nucleotide features is the Sequence Position Deletion Insertion (SPDI) model [42] (Figure 2.3.C). The SPDI model identifies each nucleotide variant with four different attributes that define how a reference sequence is transformed to contain the named nucleotide variant. Each attribute is concatenated by the `:` character to form a single string that identifies a variant. Specifically, a single identifier would look like: `[sequence]:[position]:[deletion]:[insertion]` (*e.g.,* `seq1:5:G:T` or `seq1:5:1:T`). The attributes consist of: **(1)** the name of the sequence (*e.g.,* `seq1`), **(2)** the position of the variant (*e.g.,* `5`), **(3)** the nucleotides deleted from the reference sequence or the length of deletion (*e.g.,* `G` or `1` to represented a deletion of 1 nucleotide), and **(4)** the inserted nucleotides that replace the deleted portion (e.g,. `G`).

HGVS

A third model is the Human Genome Variation Society (HGVS) model [78] (Figure 2.3.D), which describes a method of identifying variants in a human-readable string. These aren't necessarily restricted to nucleotide-level variants, but can be applied to other types of variation as well, such as amino-acid variants, or RNA variants. An example of an HGVS-identified variant is `seq1:g.5G>T`. This variant identifier consists of **(1)** a sequence identifier (`seq1`), **(2)** the type of sequence identified (`g.` for a genomic reference), **(3)** the position on the sequence (`5` for position 5), and **(4)** the variation at this position (`G>T` for a substitution of G $\rightarrow$ T). There are many different modifications to this basic HGVS identifier that provide more details on the variant. For example `p.` can be used in place of `g.` to describe an amino-acid level variant, or `5_10del` can be used to describe a deletion from positions 5 to 10.

VRS

A fourth model is the Variant Representation Specification (VRS) [76] (Figure 2.3.E), which is developed as part of the Global Alliance for Genomics and Health (GA4GH, `https://www.ga4gh.org`). This model is also more general than just a representation of nucleotide variation, such as substitutions or insertions/deletions (all part of what is termed "molecular variation"), but can also include copy-number or genotype variation (part of "systemic variation"). These different variation types are represented using a number of abstract components that are structured according to a JSON schema. The top-level component is

"Variation", which can be further divided into "MolecularVariation" and then "Allele". An Allele is used to represent a particular nucleotide-level variant, which is described by a "SequenceLocation" on a sequence and a "SequenceExpression" describing the substitution/insertion/deletion. One issue that can arise is that it is possible to represent the same variant using different JSON data structures (*e.g.,* if you use different sequence identifiers for the same genome). To combat this issue, the VRS system also includes a method to condense the variant identifier into URI (called a CURIE), which will be identical for identical variants regardless of their JSON representation. The CURIE for a variant is structured like `[namespace]:[type][base64 encoded hash]` (*e.g.,* `ga4gh:VA.2X6ThYd...`). This also has a benefit of reducing the verbosity of the full JSON definition of a variant, though at the expense of losing human-readability.

BED

Finally, a related format which is commonly used to describe blocks of contiguous regions on a genomic sequence is the Browser Extensible Data (BED) format (Figure 2.3.F). While originally developed to define regions to display in the Human Genome Browser at UCSC [79], the development of suites of software such as BEDTools [80] and pybedtools [81] for working with this format makes it useful for operations on intervals of genomic data (*e.g.,* unions, intersections, subtractions). The format stores a sequence identifier along with a start and end position for an interval (as the half-open interval $start \leq p < end$ for all integers p) in a tab-delimited file, one interval per line. This format is not used to store nucleotide-level variation like the other formats.

Coordinate systems

One complexity of all these different variation models and file formats is the different coordinate systems used to number nucleotides (Figure 2.3). These can be divided into two broad categories: *base* (or *residue*) coordinates, where it is the nucleotides that are numbered, and *interbase* (or *inter-residue*) coordinates, where it is the spaces in-between nucleotides that are numbered. The numbering can also further be sub-divided to be *0-indexed* (counting starting from 0) and *1-indexed* (counting starting from 1). Both VCF and HGVS used a 1-indexed base coordinate system (so they count the nucleotide bases and start counting with a 1). Both SPDI and VRS use a 0-indexed inter-base coordinate system (so they count the spaces in between nucleotides and start counting with 0 prior to the first nucleotide in the sequence). The BED format uses 0-indexed base coordinates (counts the nucleotides and starts with a 0). No variation system reviewed here uses a 1-indexed interbase coordinate system. Care should be taken to account for these coordinate system differences to avoid off-by-one errors when mixing and matching these variation models.

Storage models and other data

So far, I have only described how to name and store nucleotide variants—that is some transformation that changes a reference nucleotide sequence to an alternative nucleotide sequence (*e.g.,* an `A → T` substitution). However, there is other information that may be of importance that could be stored alongside nucleotide variants. In general, there are four different classes of information that could be described:

1. **Variants**. These are any variation in the nucleotide sequence of a reference genome that transforms it into an alternative genome (*e.g.,* an `A → T` substitution).

2. **Reference**. These are regions on the reference genome that are identical to regions on the alternative genome.

3. **Missing/unknown**. These are regions on the reference genome where there is missing data on the alternative genome. Thus, these regions cannot be described as either *variants* or *reference.*

4. **Novel**. These are regions on the alternative genome that are missing on the reference genome (*i.e.,* novel regions).

In order to account for these additional regions, a number of different variant storage models have been proposed [82]. The simplest model would be to store only variant nucleotides—the **Variant Only Storage Model**. This minimizes the amount of information stored, but may lead to misinterpretations of data in non-variant positions (where it cannot be determined if the nucleotide is the reference nucleotide, or is of poor quality/missing). Alternatively, every position on the reference genome could be assigned either as variant, reference, or missing and stored all together—the **Full Storage Model**. This provides a way to differentiate between reference and missing data, but at the expense of increased storage space (in the models described in [82] novel regions are not considered). A more compact form could be termed the **Block Storage Model**, where contiguous blocks of *reference* or *missing* regions are stored as a single entry with a range of positions (*e.g., 100-200*) instead of separate entries per each nucleotide. This model is very similar to the Genomic VCF (gVCF) format [83] (`https://github.com/sequencing/gvcftools`)—an extension of the VCF format to include blocks of contiguous *reference* or *missing* nucleotides in a single VCF record. Finally, the **Negative Storage Model** stores variant nucleotides and contiguous blocks of missing data, while leaving reference genome blocks unstored. This is because they can be inferred as those positions on the reference that are neither variant nor missing (assuming the full reference genome sequence is stored elsewhere).

2.4 Workflow management

Microbial genomics data analysis can often involve running a collection of different software—feeding the output of one component to the input of another—as data progresses through the different analysis stages (*i.e., primary* $\rightarrow$ *secondary* $\rightarrow$ *tertiary*). The variety of methods and software choices available for analysis can lead to complexity in defining and maintaining a full analysis pipeline. Further complications arise when attempting to install and execute all this software in a portable manner across different execution environments (*e.g.,* on a single machine or in a cloud computing environment).

Workflow management software is one solution to this problem. These software are designed to simplify the overhead of common tasks when designing a data analysis pipeline and provide a mechanism for executing the software through a common interface [84]. Additional benefits of workflow managers are the support for installation of individual software components and dependencies through the use of package managers, such as Conda [85] and Bioconda [86], or container technologies, such as Docker [87] or Singularity [88].

A comparison of some different workflow systems can be found in Table 2.1, which is derived from existing literature [84; 89] and the documentation provided by each workflow manager. The table is divided into two main categories of comparison criteria: **development** and **execution**.

2.4.1 Development

The design and development of a workflow is one major way to compare different workflow managers (Table 2.1). Each evaluated workflow manager is able to execute workflows defined in some text-based language (*e.g.,* YAML for *Snakemake* and *Pegasus* or Groovy for *Nextflow*). These languages provide the capability to join together the inputs and outputs of multiple existing software to define the flow of data across multiple stages from input to a final output. Most workflow managers provide the capability to easily write these stages as small collections of command-line tools (*e.g.,* `grep "string" input.txt | sort`), which provides an easy transition for developers familiar with working in a command-line environment to begin writing a workflow using these languages. One exception is *Galaxy* [92], which operates at the more abstract level of a *tool wrapper* that must be predefined in *Galaxy* prior to incorporating it into a workflow (though the *Galaxy*

Category	**Criteria**	**Snakemake [90]**	**Nextflow [91]**	**Galaxy [92]**	**Pegasus [93]**	**Cromwell [94]**
Development	Language	YAML/ Python	Groovy	JSON	YAML/ XML / API	WDL/ CWL
	Supports CWL [95] or WDL [96] [a]	CWL[f]	CWL[f]	CWL[f]	CWL[f]	CWL/ WDL
	Flow-control[b]	✓	✓		✓	✓
	Design[c]	T	T	TGC	T	T
Execution	Environment[d]	LHC	LHC	LHC	LHC	LHC
	Dependencies[e]	CDS	CDS	CDS	DS	DS
	Interface	CLI/ API	CLI	GUI/ API / CLI	CLI	CLI/ API
	Resuming	✓	✓	✓	✓	✓

Table 2.1: Comparison of different workflow languages and implementations.

[a] CWL = Common Workflow Language (`https://www.commonwl.org`),
WDL = Workflow Description Language (`https://openwdl.org`)

[b] Flow-control is the ability to enable or disable parts of the workflow to control data flow.

[c] T = text-based workflow design, G = graphical workflow design, C = command-line software for aiding design

[d] L = local, H = high performance computational cluster, C = cloud

[e] C = conda (`https://conda.io`), D = docker (`https://docker.io`), S = singularity (`https://sylabs.io`)

[f] Partial implementation or conversion scripts for CWL

community provides an enormous variety of tool wrappers contributed by bioinformaticans).

In addition to software-specific languages, two other languages for defining workflows include both the Common Workflow Language (CWL) [95] and the Workflow Description Language (WDL) [96]. These are intended to be software-independent standards which can be executed within a variety of workflow engines. Most workflow engines support CWL to some extent, though often only a partial implementation or by providing scripts to convert their custom language to a CWL workflow. WDL is supported by fewer workflow managers, with one notable exception being *Cromwell* [94], where it is the primary language used to define workflows.

Some workflow managers are able to directly incorporate code from existing programming languages. In particular, *Snakemake* [90] can incorporate Python code within a workflow while *Nextflow* [91] uses an existing programming language—Groovy—which is derived from Java. These provide greater flexibility to fine-tune the operation of the workflow when there does not exist command-line tools for the particular task at hand. This also ties into the capability to include advanced flow-control logic (such as if-else statements) for controlling the execution of different branches of a workflow. Most workflow managers support these flow-control structures except for *Galaxy*, which is limited in how different branches of a workflow can be executed.

One criteria of designing workflows where *Galaxy* excels is the large variety of tools and resources for designing workflows. Workflows are primarily designed using a graphical user interface (the *Galaxy* web application), but there is also experimental support for a text-based design of workflows (`https://github.com/galaxyproject/gxformat2`). In addition, there are a collection of command-line tools (the *Planemo* suite [97]) to assist in developing, testing, and sharing workflows or individual tools. Most other workflow managers primarily rely on writing workflow files in text editors which can then be executed by the workflow manager.

2.4.2 Execution

The second major category for comparing workflow managers is in how workflows are executed (Table 2.1). Every workflow manager is capable of executing the same workflow on either a local computer, a high-performance computational cluster, or in a cloud-based environment. This is one of the advantages of using workflow managers—the capability of executing the same code in a variety of environments. Additionally, nearly every workflow manager provides multiple ways to automatically install and manage software dependencies required by a workflow. Both Docker [87] and Singularity [88] are container solutions that are supported by every workflow manager. Containers provide the capability to encapsulate a piece of software and all its dependencies and isolate it from the main host machine, which makes it portable across a wide variety of computational environments. An additional method of software dependency management is via *conda* and the *bioconda* project [86], which is a package manager and repository of software packages, respectively, that are popular with bioinformaticians. All workflow managers except *Pegasus* and *Cromwell* support dependency management using *conda.*

All workflow managers provide a command-line interface for the execution of a workflow. A few (*Snakemake*, *Galaxy*, and *Cromwell*) also provide programmatic APIs for the execution of workflows. *Galaxy* alone provides a full web-based and graphical interface for the execution of workflows and management of data. In any method of executing workflows, all management software also provides the capability to resume workflows that have either failed or paused for some reason or another.

Chapter 3

Software design

I have chosen to implement an indexing, querying, clustering, and visualization framework for microbial genomics using Python, which includes both a Python-based API as well as a command-line interface. The software is named **Genomics Data Index (GDI)** (`https://github.com/apetkau/genomics-data-index`, version *0.6.0*, doi: 10.5281/zenodo.6485506). It is available for installation as a Python package using pip (`pip install genomics-data-index`). It requires some additional non-Python dependencies as described in the installation guide (it is recommended to use conda to install these). Example code and tutorials are available on GitHub (`https://github.com/apetkau/genomics-data-index-examples`) and provide a walkthrough of some of the features of the software using example datasets.

The overall design of this software falls into five separate components (Figure 3.1). These can be divided up into the **mandatory components** and **optional components**. The mandatory components consists of indexing genomic features that have been identified elsewhere and then querying this index for features or other information of interest. The optional components include a data analysis pipeline for identifying genomic features prior to indexing as well as clustering and visualization. These are described in more detail below:

1. **Data analysis pipelines**. This takes the microbial genomes (either as assembled genomes or as WGS reads) and processes them to identify the underlying genomic features to be indexed. These features consist of one of: **nucleotide variants**, **genes**, or **kmers**. Identifying these genomic features requires specialized software organized into a data analysis pipeline to move from genomics data to genomics features.

2. **Indexing genomics features**. This stage indexes the identified features to help speed up the querying stage. That is, the features are loaded, transformed, and stored within a special data structure (an inverted index stored in a relational database). This stage can either process files identified from the previous **Data analysis pipelines** stage or from **External data analysis** (where genomic features are generated and saved independently).

3. **Querying**. This stage provides support for searching for genomes with particular features. Optionally, querying for closely related genomes can be performed using data from the **Clustering** stage.

4. **Clustering**. This stage groups together genomes stored in the index into clusters based on shared features.

5. **Visualization**. This stage implements methods to visualize genomes, features, or clusters of genomes.

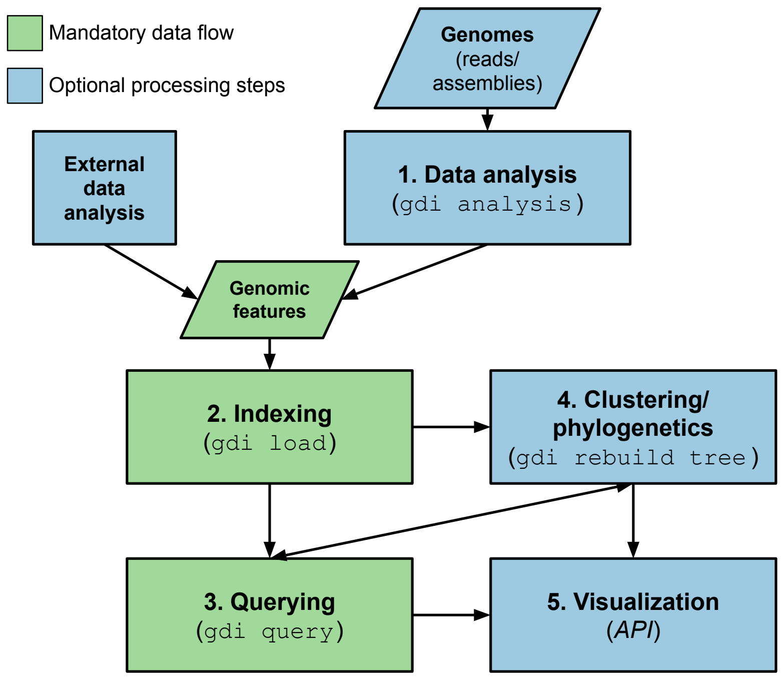

Mandatory data flow
Optional processing steps
Genomes (reads/ assemblies)
External data analysis
1. Data analysis (gdi analysis)
Genomic features
2. Indexing (gdi load)
4. Clustering/ phylogenetics (gdi rebuild tree)
3. Querying (gdi query)
5. Visualization (API)

Figure 3.1: An overview of the indexing software and data flow. There are two categories of analysis stages: the **mandatory stages** (in green), which go from genomic features to a query-able index, and **optional stages** (in blue), which either help to identify features in genomics data or cluster the data for additional querying options. Names of the commands in the command-line interface used to perform each stage are given in brackets below the stages. Each command can additionally be initiated from the Python API. The **visualization** stage can only be performed from the Python API. **External data analysis** refers to the ability to generate genomic features from external software before indexing.

3.1 Data analysis pipelines

The first step in the overall software design is transforming raw WGS data into collections of genomic features that can be indexed. The main input format for GDI is microbial genome sequence reads in the FASTQ [32] format (either compressed or uncompressed), or assembled genomes. Additionally, for performing sequence read alignment to a reference genome, a reference genome is needed as input (as a FASTA or Genbank file).

Once the input files are provided, the next step is running all the necessary underlying bioinformatics software needed to identify the genomic features. I chose to make use of Snakemake [90], which is a Python-based workflow language that provides a mechanism to write a workflow and schedule the individual tools for execution in a variety of environments (described in detail in Section 2.4).

3.1.1 Pipeline design

The overall data analysis pipeline is designed as a single workflow, written using *Snakemake*, which will either enable or disable certain stages depending on the type of input data and parameters (Figure 3.2). For example, if sequence reads or assembled genomes are given as input, then the respective portions of the analysis pipeline will be executed (Figure 3.2.A for reads and Figure 3.2.B for assemblies). Regardless of whether input data was reads or assemblies, kmer-based sketching (using sourmash [54]) can also be performed. These sourmash sketches are saved as gzipped JSON files for every input genome and combined with the additional output described in details below.

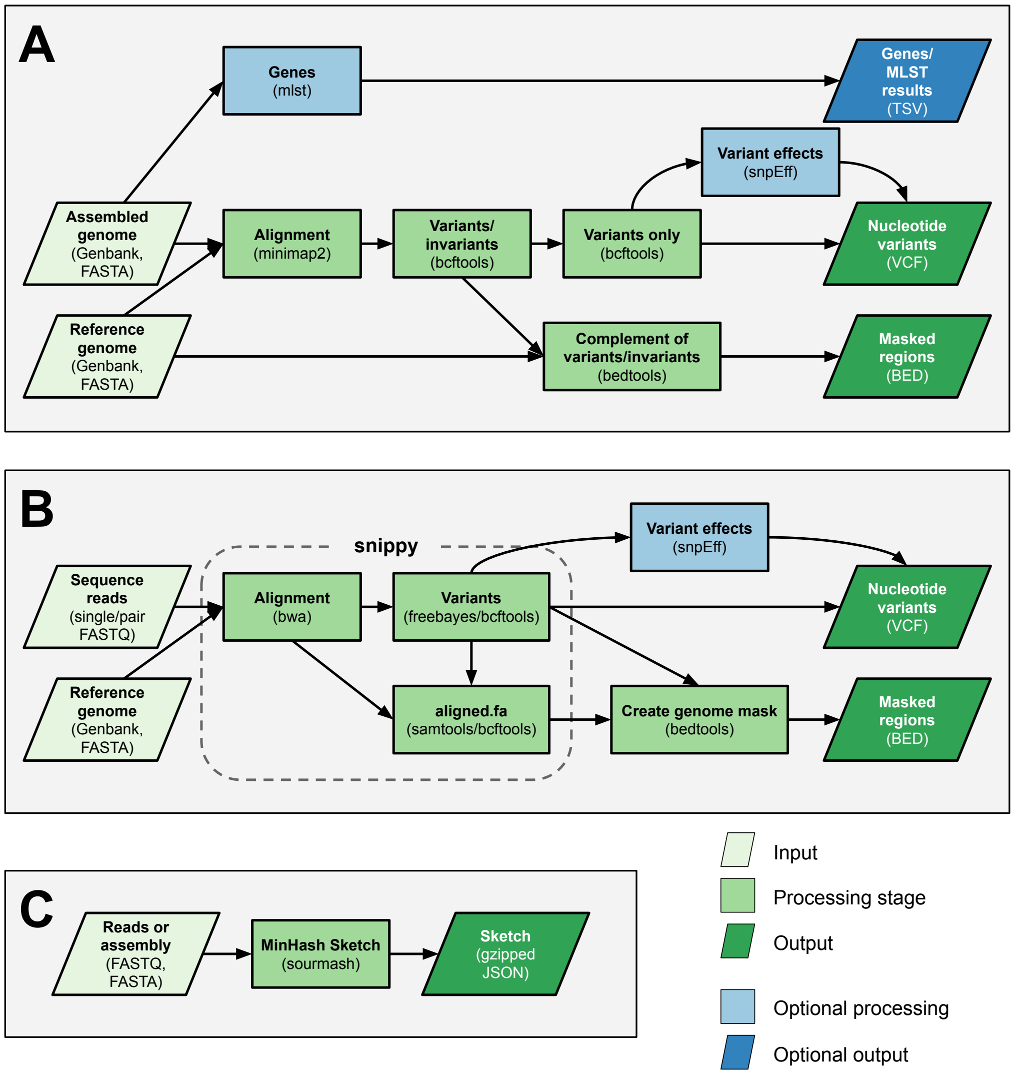

A
Genes (mlst)
Genes/ MLST results (TSV)
Variant effects (snpEff)
Assembled genome (Genbank, FASTA)
Alignment (minimap2)
Variants/ invariants (bcftools)
Variants only (bcftools)
Nucleotide variants (VCF)
Reference genome (Genbank, FASTA)
Complement of variants/invariants (bedtools)
Masked regions (BED)
B
snippy
Variant effects (snpEff)
Sequence reads (single/pair FASTQ)
Alignment (bwa)
Variants (freebayes/bcftools)
Nucleotide variants (VCF)
Reference genome (Genbank, FASTA)
aligned.fa (samtools/bcftools)
Create genome mask (bedtools)
Masked regions (BED)
C
Reads or assembly (FASTQ, FASTA)
MinHash Sketch (sourmash)
Sketch (gzipped JSON)
Input
Processing stage
Output
Optional processing
Optional output

Figure 3.2: An overview of the **data analysis pipelines** for transforming genomics data into genomics features. Each section is combined together within a single Snakemake [90] workflow, but are shown separately for clarity. Text shown in braces underneath each stage name shows the software or file format being used. **(A)** The genome assembly pipeline which processes assembled genomes to produce files containing nucleotide and gene (MLST) features. **(B)** The sequence reads analysis pipeline, which processes WGS reads to produce nucleotide features (this process excludes gene features). **(C)** The pipeline stages for constructing MinHash sketches (kmer features).

Genome assembly input

When genome assemblies are given as input (Figure 3.2.A), the pipeline uses minimap2 [39] to perform a pairwise alignment for every input genome assembly to a chosen reference genome. Minimap2 was chosen as it can align both reads as well as assembled genomes, and produce as output a BAM file [98], which is a file format used to store alignments. The aligned genomes are next passed to a combination of SAMtools and BCFtools [99] with specific filtering criteria to identify nucleotide variants and reference genome entries in the VCF format. The nucleotide variants are next passed—assuming the reference genome is in Genbank format and contains gene annotations—to SnpEff [100] to identify nucleotide variant effects (such as amino acid changes). This will produce a new VCF file with these effects encoded inside. If the reference genome does not include gene annotations, then the SnpEff stage is skipped (and so no variant effects, like amino acid changes, will be associated with the nucleotide variants).

To handle missing data, I produce as part of this pipeline a BED file which contains the coordinates of missing regions in the alignment or those regions with ambiguous bases on the input genome (*e.g.,* `N`). This is constructed using the output of BCFTools (a VCF file) that contains records for both nucleotide variants and positions identical to the reference genome, but excludes missing regions. Thus, coordinates in the the complement of this VCF file are the missing regions, which are processed using bedtools [80] to produce the BED file.

An optional step in this portion of the pipeline is to also include gene-level features. This is accomplished using the *mlst* software [101], which makes use of databases from PubMLST [63; 62]. This only produces traditional MLST results and core-genome (or whole-genome) MLST is not included.

The final output of this portion of the pipeline is a tab-delimited file which lists the VCF and BED files for each genome (the nucleotide-level variants) along with a tab-delimited MLST file.

Sequence reads input

When input is given as sequence reads (Figure 3.2.B), then the main analysis method for nucleotide variants is *snippy* [46]. Snippy internally makes use of the software *bwa* [98; 102; 37] for read alignment and *freebayes* [103], *bcftools*, and *samtools* [99] for variant identification. As output, snippy provides a VCF file for each genome, recording the high-quality nucleotide variants (*e.g.,* those that passed the quality thresholds defined by snippy). I use this VCF file as input to SnpEff [100] to annotate the VCF file with variant effects. Snippy also includes a stage where SnpEff is used, but I disable this and run SnpEff myself so that I have greater control over the parameters being used.

To handle missing data, I make use of the `snps.aligned.fa` file produced by snippy that contains gaps `-` and ambiguous bases `N` aligned to the reference genome for missing or low-quality data. I convert the positions of these gaps and ambiguous bases to a BED file containing coordinates of missing regions. I then subtract off any variants found in the VCF file produced by snippy (to include deletions as part of the nucleotide variants and not as part of missing data).

The final output of this portion of the pipeline is a tab-delimited file which lists the

VCF and BED files for each genome. There is no gene-level (MLST) analysis of any kind performed in this stage.

3.1.2 Parallelization of the Analysis stage

The use of the Snakemake [90] workflow language provides automatic support for parallel processing of genomes. Snakemake makes use of the provided list of input genomes alongside the defined workflow to construct a Directed Acyclic Graph (DAG) showing dependencies among all the individual stages of the workflow. Figure 3.2 shows a descriptive overview of a DAG for the GDI workflow and Figure A.1 shows an automatically-constructed DAG from Snakemake for three input genomes.

The constructed DAG is used to schedule and execute jobs for each stage. A single path through the DAG defines a set of jobs that must be executed sequentially, while parallel paths define jobs that can be executed in parallel. Snakemake executes individual jobs according to the DAG, which will involve a combination of sequentially-executed and parallel-executed jobs. The maximum number of jobs that can be executed in parallel is controlled by a parameter passed to Snakemake (the `--jobs` parameter), which is ultimately derived from the maximum number of cores provided to the GDI command-line interface.

Analysis sample batching

When running large numbers of samples (*e.g.,* tens of thousands) within a single Snakemake workflow, a significant amount of time may be spent inferring the very large DAG prior to executing jobs. To speed up processing, Snakemake provides a mechanism to process input files in batches (using the `--batch` argument) to split the analysis into smaller collections of files. I make use of Snakemake's mechanism of sample batching for processing large datasets, which is ultimately controlled by the `--batch-size` argument passed to the GDI command-line interface.

3.2 Indexing

Once the data analysis pipeline (or an external source) has broken up genomes into individual features, the next step is to index these features. Indexing is the process of constructing a data structure which can make it more efficient to search for particular items of interest (Section 2.1.2). Two large challenges to be addressed by GDI are to provide a mechanism to 1) search through and 2) compare large collections of genomes using a variety of different genomic features commonly used within microbial genomics (Section 1.2). To support these operations, I make use of a data structure often used within the field of information retrieval—an **inverted index** (Section 2.1.2).

An inverted index is a data structure which provides a way to match a particular term to a set of documents containing that term. For my use case, the documents correspond to genomic samples while terms correspond to genomic features. Hence, I need to implement a way to associate a feature identifier with a set of genomes. This requires a method to uniquely identify a specific genomic feature to be used as a key in the index. This is handled for each type of feature separately.

3.2.1 Feature identifiers

I define a *feature* as some particular region of a genome of interest. There are three broad classes of features that are to be stored within GDI: **nucleotide variants**, **kmers**, and **genes** (described in Section 2.2.4). In order to work with these types of features, I need a consistent way of naming and talking about them.

Nucleotide variants

To identify nucleotide variants, I chose a **concatenated version of the four attributes of a VCF** variant identifier (`CHROM:POS:REF:ALT`). This corresponds nearly identically to the the Sequence Position Deletion Insertion (*SPDI*) model [42], but instead uses a *1-indexed base coordinate* system instead of the *0-indexed interbase coordinate* system used by SPDI (Figure 2.3). However, I adopt the terminology of the SPDI model (sequence, position, deletion, insertion), since it uses more generic terminology (not every sequence in a reference genome is a chromosome). Additionally, I borrow the aspect of the SPDI model where a deletion string can be represented either as a string of nucleotides to delete or as the length of the deletion (which is useful for long deletion strings to make them more compact). Some examples are given below:

- A **substitution** `seq1:100:A:T`. This represents a transformation on the sequence with the name `seq1` by moving to nucleotide `100`, deleting the nucleotide `A` and inserting the nucleotide `T` in its place. That is, it identifies an $A \rightarrow T$ substitution at position *100* of *seq1*.
- An **insertion** `seq1:20:A:AT`. This represents a transformation on the se-

quence with the name `seq1` by moving to nucleotide `20`, deleting the nucleotide `A` and inserting the nucleotides `AT` in its place. Since *A* is both deleted and inserted, the net effect is no change in this nucleotide, which leaves only an insertion of *T* immediately following position *20*.

- A **deletion** `seq1:40:GC:G`. This represents a transformation on the sequence with the name `seq1` by moving to nucleotide `40`, deleting the nucleotides `GC` and inserting the nucleotides `G` in its place. Since *G* is both deleted and inserted, the net effect is no change in this nucleotide, which leaves only a deletion of *C* immediately following position *40*. An acceptable alternative representation of this nucleotide variant would be `seq1:40:2:G`, where the nucleotides to delete `GC` is replaced with the length to delete `2`. This representation is used internally by GDI as part of the primary key in the table of variants where it is more efficient to store the number of nucleotides to delete (*e.g.,* storing only the number *500* rather than a string of all 500 nucleotides). Using an identifier with the string of characters to delete is used elsewhere where human-readability is of greater importance.

I purposely chose to differ from the SPDI standard due to the expectation that these variant identification strings will be used by humans (*e.g.,* during querying). If I was to strictly use the SPDI model, I would have to remind users (and myself), who copy/paste variant identification strings into a query in the Python API (Section 3.3.1), to always convert any VCF coordinates (or HGVS coordinates) to SPDI coordinates. This would likely lead to many off-by-one issues. Hence, to avoid this issue, I chose to modify the SPDI standard slightly.

Genes and alleles

In order to name features consisting of alleles of genes (as defined by the gene/MLST method) I developed a model, inspired by SPDI [42], that I call the Scheme Locus Allele (SLA) model. The SLA model requires three different values to uniquely define a particular gene-allele (the Scheme, Locus, and Allele), and these are joined together into a single string separated by the character `:`. For example, `ecoli:adk:100` would be a name for allele `100` of gene `adk` on the `ecoli` scheme.

Kmers

With the current version of GDI, identifying and searching for specific kmers is not supported, hence I have no specific identifier for a kmer. However, I store sourmash [54] sketches within my index. These are stored as gzipped JSON files which contain a list of the hash values derived from a subset of all kmers within a particular genome for some given value k. These are stored as a list of integers (*e.g.,* `[10, 45, 200, ...]` in JSON) and would be the closest values that correspond to kmer identifiers stored in my index. However, these are not loaded into the inverted index like the other features as I store only a subset of all hashes from all kmers (a selection of the minimum hash values as specified in the MinHash/Mash [51; 50] algorithm).

Unknown/missing data

In some situations, it's possible to encounter genomic features which are unknown or missing. For example, with nucleotide variants it is possible for certain regions of a genome to be missing or ambiguous (represented by an `N`, meaning it could be any one of $\{A, T, C, G\}$). Or, for gene/MLST results, missing data may be represented by the number `0` as an allele identifier (or possibly an empty string or an NA value).

In order to account for this, I define an explicit mechanism to name these unknown/missing data. I choose the character `?` to represent unknown/missing data for both nucleotide variants and gene-allele variants. I decided against the character `N` as this represents any one of $\{A, T, C, G\}$ (for nucleotide variants), but I also want to represent a case of a gap `-`. The missing data character `?` would be used by my defined feature identifiers as follows:

1. **Nucleotide variants**: Here, a nucleotide feature which is unknown/missing is named like `CHROM:POS:REF:?`, where `?` is used for the `ALT` value of the identifier to indicate that this position on a reference sequence is missing/unknown. As an example, `seq1:10:A:?`, which means that position `10` on sequence `seq1` consists of an `A` to `?` change (that is, it is unknown what is on position 10).

2. **Gene alleles**: Here, the identifier would look like `Scheme:Locus:?`, which indicates that there is missing data for the locus in a particular MLST scheme. For example, `ecoli:adk:?`, which can be read as an unknown allele `?` for gene `adk` on scheme `ecoli`.

This method of storing both variant features and missing features within an inverted index is similar to a **Negative Storage Model** [82] (Section 2.3.1). Since I store the reference genome sequence elsewhere (Section 3.2.3), the unstored sequence for a particular variant/sample can be inferred from the reference as those which are not missing and non-variant. One difference between this model and the original negative storage model [82] is that the authors are not using an inverted index for this model.

3.2.2 Storing sets of genomic samples

In an inverted index, each key/identifier (a term) is associated with a set of genomic samples (the documents). Genomic samples often have a name associated with them (*e.g., SampleA*). However, it's often more efficient to work with integers rather than arbitrary strings, so each of these sample names are first associated with a numerical identifier. Once I have numerical identifiers, I can reduce the problem down to storing sets of integers rather than sets of strings.

In order to efficiently store sets of integers I made use of Roaring bitmaps [104], which are designed as an efficient data structure and collection of operations to work with sets of (unsigned) integers. While Roaring bitmaps are called bitmaps, they are specifically a mixed data structure for storing sets of 32-bit integers to address some of the issues with bitmaps (primarily the storage space). Roaring bitmaps divide the storage of integers into three separate classes:

1. **Lists** of integers (*e.g.,* `[1, 5, 9, ...]`)

2. **Lists of Runs** of integers. A run consists of a starting integer and the length

of the run (*e.g.,* `8,3` is used to store the integers `8, 9, 10`)

3. **Bitmaps** of integers (*e.g.,* `0110`, which has the second and third bit set to 1 and so stores the integers `1, 2`)

To make use of these different classes, Roaring divides up the space of all 32-bit integers into blocks of 16-bits each (capable of storing 65,536 integers) and chooses one of the above three classes (lists, runs, or bitmaps) to store each 16-bit block. Heuristics are used to choose which type of method to use for storing integers based on the distribution of integers in each block. Separate implementations are used for set operations on integers stored in each block type (*e.g.,* intersections of a **run** of integers with a **list** of integers). Roaring also provides a mechanism to serialize/deserialize a set of integers into a string of bytes.

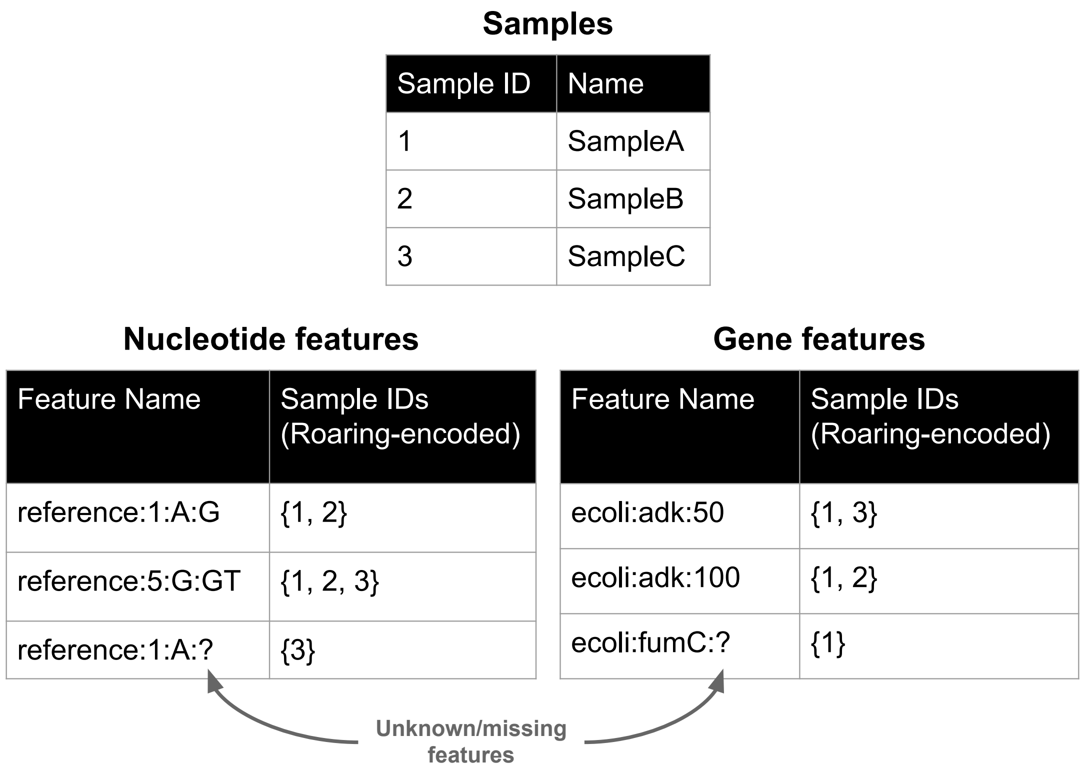
In order to use Roaring bitmaps, I first map a sample name (as a string, like `SampleA`) to a sample identifier (as a number, like `1`) through a table in relational database software (*e.g.,* SQLite or MariaDB). In other words, I define a table with a sample identifier as the primary key. These identifiers are then added to a Roaring bitmap and serialized into a string of bytes. I store this string of bytes in a relational table with the primary key of the table consisting of the feature identifiers described above (Figure 3.3, Figure A.2). To lookup the set of samples that contain a particular genomic feature, I use this feature as a key and lookup the Roaring bitmap corresponding to this key.

Alternative method (foreign keys)

An alternative method for associating sets of sample identifiers to a genomic feature would have been to explicitly include these identifiers as a foreign key in a table that maps feature identifiers to samples containing those features (in other words a table with two columns: *feature identifier*, the primary key, and *sample identifier*, the foreign key referencing the samples table). However, I chose to not use this method as it would have greatly increased the number of rows in my inverted index table in a relational database. In particular, this method of storage means that the rows grow as $rows = samples * features$ (the two columns in the table).

A rough estimate of the number of rows required if I wished to store the global SARS-CoV-2 genome dataset would be 10 million genomes (from GISAID [4] as of April 2022) multiplied by 30,000 different features (30,000 is the approximate length of the SARS-CoV-2 genome and I assume I store one feature per nucleotide to account for unknown/missing data). This gives me $\approx 3 * 10^{10}$ or on the order of tens of billions of rows.

However, if I instead used Roaring bitmaps as described above, then the number of rows grows with respect to only the number of features ($rows = features$). A rough estimate of the number of rows for storing the same set of SARS-CoV-2 genomes in this case would be $\approx 30,000$ rows (one row per feature identifier). This is on the order of tens of thousands of rows (as opposed to tens of billions). Additionally, the number of rows no longer grows with respect to the number of samples in my database (as each sample identifier is encapsulated within a Roaring bitmap instead of explicitly represented as a row in the database).



Samples

Sample ID	Name
1	SampleA
2	SampleB
3	SampleC

Nucleotide features

Feature Name	Sample IDs (Roaring-encoded)
reference:1:A:G	{1, 2}
reference:5:G:GT	{1, 2, 3}
reference:1:A:?	{3}

Gene features

Feature Name	Sample IDs (Roaring-encoded)
ecoli:adk:50	{1, 3}
ecoli:adk:100	{1, 2}
ecoli:fumC:?	{1}

Unknown/missing features

Figure 3.3: An overview of how feature names are associated with sample identifiers in the database. The **Sample** table stores sample names associated with a unique, incremental identifier in a relational database. Additional feature tables (for **Nucleotide** and **Gene features**) store a **Feature Name** per row, with each name associated with a set of **Sample identifiers**, encoded as a Roaring bitmap. Unknown/missing features are stored in their respective tables, but using the symbol "?" to identify them.

Given these rough estimates for the number of rows, I ended up choosing to use Roaring bitmaps to store sets of sample identifiers. However, this has the disadvantage of shifting the logic for associating an integer in the Roaring bitmap to a sample from the database/SQL to my Python code (*i.e.,* I must implement joins between sample identifiers and sample names in Python rather than using SQL). Some of these disadvantages are discussed in the limitations section (Section 5.3.4).

3.2.3 Index storage

The inverted index and associated information is stored within a mixture of a relational database as well as files on a file system (Figure 3.4). This information is all grouped together within a common directory, which is referred to as a *project* in GDI. An empty *project* directory can be created using the `gdi init` command and stores a configuration file containing connection information for a relational database, as well as the location of directories containing additional files on the file system. Within a *project* directory there are three main categories of data stored: **Relational database**, **Reference genome storage**, and files on a **File system**. One advantage of storing all data within a single *project* directory is that the entire index can be copied and distributed to separate machines.

Relational database

The main database is constructed and queried using the Python package SQLAlchemy [106], which provides a common API for interacting with a variety of relational database management software. The default relational database software used is SQLite (`https://www.sqlite.org/`), which stores data in a self-contained file that does not need external database management software to be installed. SQLite was chosen to minimize dependencies for using GDI and provide the capability to copy an index to other machines with minimal overhead. However, support for MariaDB (`https://mariadb.org/`) or other relational database software is also provided.

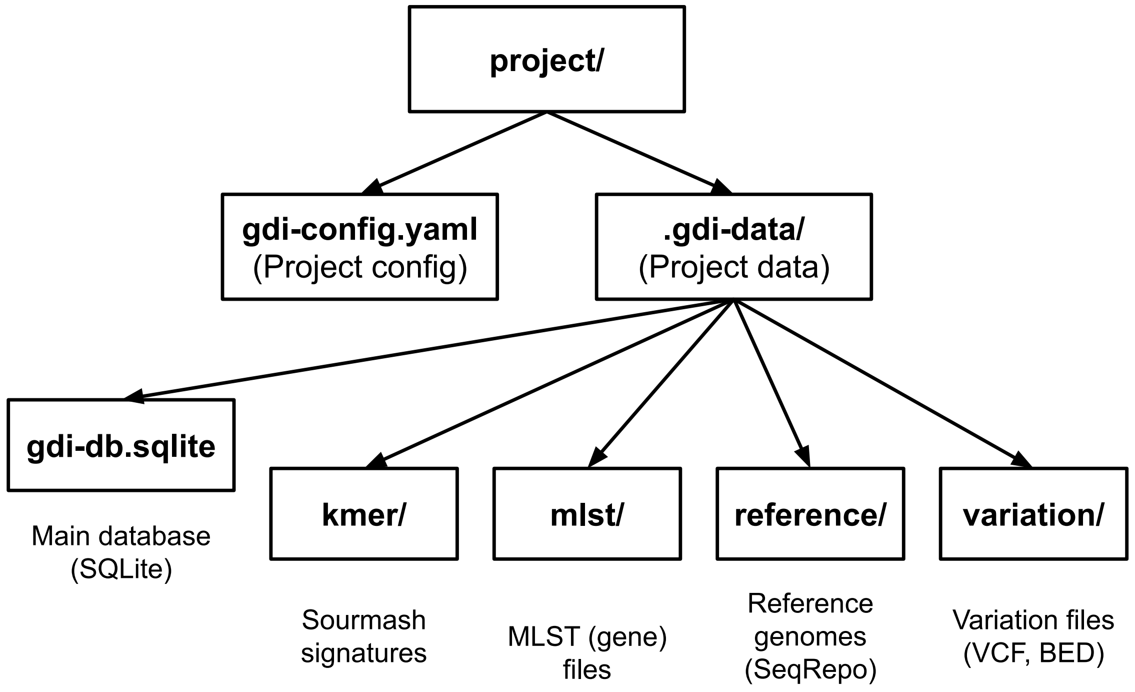


Figure 3.4: The directory structure and files used by GDI to store data. The main directory is a `project`, which contains a configuration file (`gdi-config.yaml`) and a sub-directory of additional files (`.gdi-data`). Underneath this directory is the main SQLite database file (`gdi-db.sqlite`), a directory containing the reference genome storage (`reference` containing SeqRepo [105] data). Additionally, there are sub-directories (`kmer`, `mlst`, and `variation`) containing additional genomic feature files stored by GDI.

Reference genome storage

Indexing genomes using nucleotide variants requires the use of a reference genome to provide a common genomic sequence to compare to other genomes. In order to store and access reference genome sequence data, I make use of SeqRepo [105], which is a Python library for storage and access of large collections of genomic sequence data. SeqRepo uses a combination of a SQLite database as well as files (sequence files in FASTA format) stored on the file system. SeqRepo is itself accessed with the help of the GA4GH Variation Representation System (VRS) [76], which provides Python libraries for working with data from SeqRepo in addition to nucleotide variants. The entire SeqRepo database files are stored within a `reference` directory (Figure 3.4).

File system

The final category of data stored is additional files on the file system. These are divided into separate directories: `kmer`, `mlst`, and `variation`. The `kmer` directory contains sourmash [54] signature files, one file per genome, stored as gzipped JSON. The `mlst` directory is available for additional MLST (gene-level) data. The `variation` directory stores nucleotide features—stored as a pair of gzipped VCF (nucleotide variants) and BED (missing/unknown regions) files for each genome.

3.2.4 Parallelization of the indexing stage

The indexing stage supports parallel processing through the Python `multiprocessing` package. This is used during two stages of indexing: **saving features** and **constructing a feature DataFrame**.

Saving features

Saving features involves pre-processing feature data (*e.g.,* nucleotide variant files) to convert to a standardized form and saving these files into the GDI-managed directory. This involves executing external software, such as BCFTools [99] and

BEDTools [80], for every sample to be loaded into the index. I use Python's `multiprocessing` package to spawn a number of sub-processes to handle a collection of genomes in parallel during this stage.

Constructing a feature DataFrame

Constructing a feature DataFrame involves loading features for each sample (*e.g.,* a VCF file) into a pandas [22] DataFrame prior to constructing the inverted index. I make use of the Python `multiprocessing` package to split this task up among a collection of sub-processes for every sample to be loaded (*e.g.,* every VCF file to convert to a DataFrame). Once a new DataFrame is constructed for every sample, these DataFrames are concatenated together to construct a single DataFrame containing every sample to index. The combined DataFrame is used to construct the inverted index that matches a particular feature to a set of samples containing that feature.

Index sample batching

The two multiprocessing stages of GDI are further sub-divided to process batches of samples (*e.g.,* 2000 samples in a batch). This is to avoid large memory usage that would occur if a very large number of samples were loaded at once (*e.g.,* loading tens of thousands of samples into a nucleotide variants DataFrame). This is controlled by the `--sample-batch-size` argument to the GDI command-line interface.

3.3 Querying

Two major challenges for applying microbial WGS to infectious disease investigations are to perform *large-scale comparisons* and to *integrate multiple types of data* (*e.g.,* epidemiological metadata or phylogenetic trees) to aid in the analysis and visualization of this data (Section 1.2). To address these challenges, I have devised a query language which provides the capability to search through the constructed index to select subsets of genomic samples that match particular criteria. Querying is exposed through two separate user interfaces: the **Python API** and the **command-line interface (CLI)**.

3.3.1 Query API

The GDI Python API provides the capability to select subsets of genomic samples by using Python code. This API was inspired by both pandas [22] and SQLAlchemy [106], which provide their own API to query for data. In particular, the GDI Python API is intended to be used alongside pandas and Python-based visualization packages such as matplotlib [107] and the ETEToolkit [21] for easy analysis and visualization of genomics data and phylogenetic trees. This addresses one of the challenges of working with large amounts of genomics data and metadata—mainly integration into a common system (Section 1.2).

Query objects

A query in the GDI Python API is initiated from a connection to the genomics index database (Section 3.2.3). The query is encapsulated in an instance of a Python

`SamplesQuery` class I have defined. An instance of this class is referred to as a query object, and stores sets of matching genomic samples alongside additional information used to further refine the query (Figure 3.5.A). There are three types of additional information that can be attached (or joined) to a query:

1. **Genomics index**: This is the default object attached to a query and provides a connection to the inverted index used to further refine a query.

2. **Tree**: This consists of a phylogenetic (or hierarchical clustering) tree loaded using the ETEToolkit [21]. The tree can be used to further refine a query using the distances or topology of the tree between genomic samples.

3. **Metadata**: This consists of additional metadata associated with genomic samples stored as a pandas DataFrame [22]. Adding metadata to a query requires a join-column in the metadata table used to match up samples in the index with rows in the metadata table (*e.g.,* a column of sample names in the metadata table).

Each query can be refined using one of three different operations (Figure 3.5.B). Each operation uses the attached additional data from a query to select a subset of the samples in the query and returns a new `SamplesQuery` instance encapsulating this subset. The operations consist of:

1. `q.hasa("[FEATURE]")`: This can be read as "select all samples in query *q* that **have a (has a)** feature named *[FEATURE]*" (*e.g.,* a nucleotide variant or gene-allele). This makes use of the attached *inverted index* (Section 3.2). As an example, `q.hasa("seq1:10:A:T")` would return a query object that

has selected all genomes that have an `A→T` substitution at position `10` on sequence `seq1` (uses the feature identifiers from Section 3.2.1).

2. `q.isa([EXPRESSION])`: This can be read as "select all samples in query *q* that **are a (is a)** result of *[EXPRESSION]*". The *[EXPRESSION]* can either be a sample name (*e.g.,* `SampleA`) or, if metadata is attached to the query, a particular value in a column of the metadata (*e.g.,* if a column is called `lineage` then one could use `q.isa("X", isa_column="lineage")` to select all samples belonging to lineage `X`).

3. `q.isin([EXPRESSION])`: This can be read as "select all samples that **are in (is in)** the result of *[EXPRESSION]*". The *[EXPRESSION]* could either be a list of sample names or a query involving a phylogenetic tree. The expressions involving a phylogenetic tree can either be based on distance (in units of *substitutions/site* or *substitutions*) or selections based on the most recent common ancestor of a group of samples. For example `q.isin(["Sample A", "Sample B"], kind="mrca")` would select all samples in the query `q` that are found in the clade rooted at the most recent common ancestor to samples `Sample A` and `Sample B`.

Missing/unknown features

One challenge that arises, in particular with querying for genomic features (using the `hasa()` operation), is how to handle missing or unknown features (Section 3.2.1). This can result in cases where it is unknown if a particular genomic

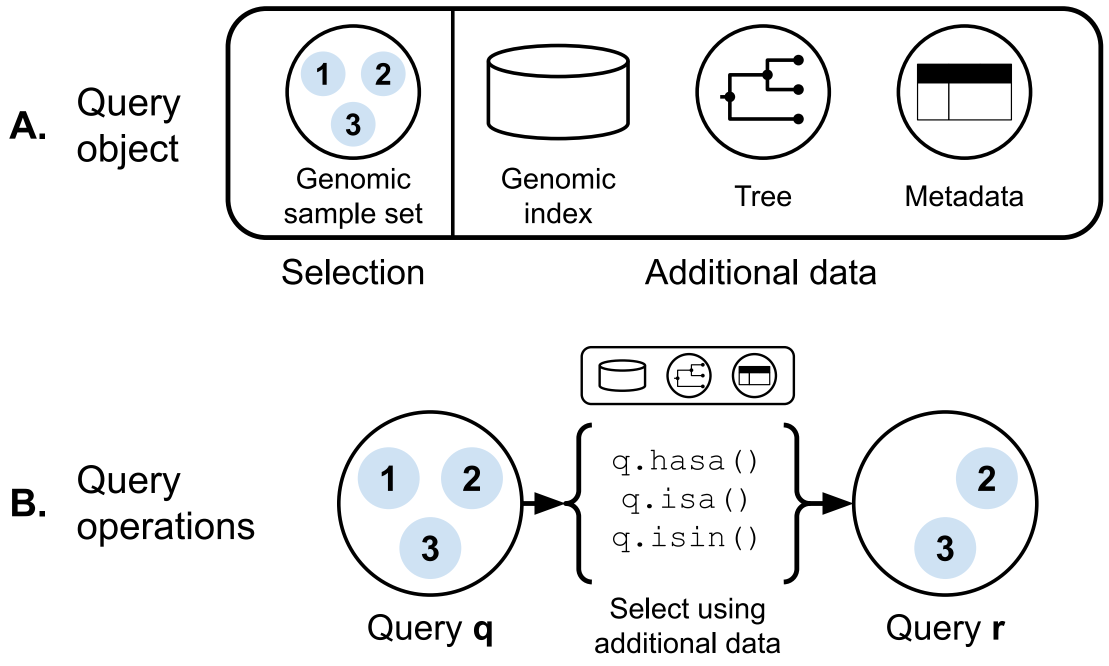


Figure 3.5: An overview of the Query API. **(A)** A query starts with a query object in Python (an instance of the `SamplesQuery` class). The object consists of sets of selected genomic sample identifiers (as integers) along with connections to additional data structures to assist in querying. These additional data structures include: the *Genomics index* itself, an optional *Phylogenetic tree*, and an optional *metadata* table storing additional data associated with the genomic samples. **(B)** Different operations (`isa()` , `hasa()` , and `isin()`) can be performed on a query object, `q` , to select subsets of samples using the additional data joined to query `q` . The operations return a new query object, `r` , which consists of a subset of the samples from the original query `q` .

sample has a particular feature (*e.g.,* if a genome has an "N" on position 10 then it cannot be determined if the A→T variant exists at position 10).

As an example, consider a genomic sample *SampleA* which has missing data at position *10* on the sequence *seq*. Also consider a query `r = q.hasa("seq:10:A:T")` that is used to select samples that have an *A*→*T* variant at position *10* of sequence *seq*. A problem arises if we try to determine whether or not *SampleA* should be selected by this query. This is equivalent to asking the question "is it **True** or **False** that *SampleA* has variant `seq:10:A:T`" (by the definition of `q.hasa()` above). It would not make sense to say that this statement is **True**, nor would it make sense to say that this is **False** since both of these are saying more than what is known. All that is known is that it is unknown if *SampleA* has mutation `seq:10:A:T`. It would be useful to have a third truth-value, **Unknown**, in addition to **True** and **False** to better capture these situations. This is described more formally below alongside the introduction of logical operations between queries.

Logical operations

Querying an index can be implemented using a variety of methods—two categories of which are **Boolean retrieval** and **Ranked retrieval** (Section 2.1.3). In **Boolean retrieval**, a query can be broken down into logical operations on different terms (*e.g.,* "dog OR cat"), which can be implemented by searching for matching sets of documents to each term in the inverted index and combining these terms using set operations (union, intersection, complement) to return the final set of documents (Figure 2.1).

To provide a similar method of searching for genomic samples using queries, I have implemented an extension of Boolean retrieval which operates on the features stored in GDI. A **feature** is analogous to search term and a **genomic sample** is analogous to a document. Hence, Boolean retrieval is implemented by converting logical operations (*i.e.,* NOT, AND, OR) on a query combining genomic features to sets of matching genomes. However, as introduced in Section 3.3.1, I am not using two-valued logic, $\{True, False\}$, but I instead add a third truth value, $\{True, False, Unknown\}$, to account for missing or unknown genomic information. I extend the ordinary two-valued Boolean logic truth tables to incorporate a third truth value by using Kleene's strong logic of indeterminacy [108; 109]. This provides a way to extend the methods used for regular Boolean retrieval (Figure 2.1) to incorporate a third truth value. This requires the addition of a third set of genomic samples (to represent the set of *Unknown* samples in addition to *True* and *False*). Using a third truth value (*Unknown*) to account for missing data has also been used by the pandas [22] library by defining a *Nullable Boolean* data type to handle missing values (indicated with NA) in a table (`https://pandas.pydata.org/docs/user_guide/boolean.html`). Below, I formally define the structure of a query to satisfy three-valued logical operations.

I define a query, $Q = (T, N, U)$, as a tuple of three different sets of integers (Figure 3.6.A). Each integer corresponds to a unique identifier for a sample in the index (in my implementation, these identifiers are assigned by a relational database and stored as Roaring bitmaps [104]). These three sets can be used to derive a fourth set, F, all of which are defined as follows:

1. T: The set of samples for which the query Q is **True**.
2. N: The set of samples for which the query Q is **Unknown**.
3. U: The **Universe** set of all samples for a query Q.
4. F: The set of samples for which the query Q is **False**. This is not needed as part of the definition of a query but can be derived from the other sets from

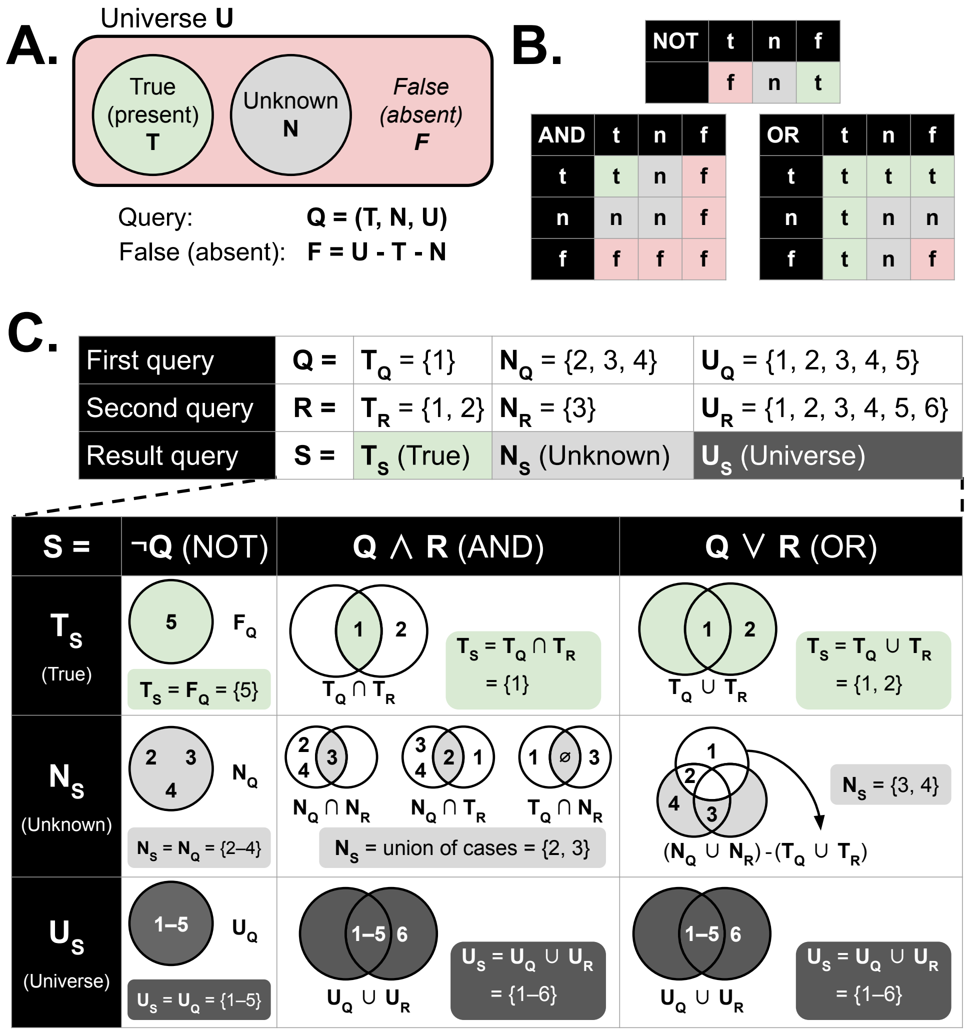
the relations below.

Given the above definitions, the following relations hold on a query Q:

1. $T \subseteq U$: That is, the set of samples for which a query is **True**, T, is part of the universe of samples U.
2. $N \subseteq U$: That is, the set of samples for which a query is **Unknown**, N, is part of the universe of samples U.
3. $T \cap N = \emptyset$: That is, the sets of **True** and **Unknown** samples are mutually exclusive (a sample cannot be both True and Unknown).
4. $F = U - T - N$: That is, the set of samples in universe U for query Q that are not **True** and not **Unknown** must be **False**.

From these definitions of a query, I can define logical operations—NOT, AND, and OR—on queries. Let $Q = (T_Q, N_Q, U_Q)$ and $R = (T_R, N_R, U_R)$ be two separate queries. Let $S = Q|R$ be a new query which is the result of some logical operation $|$ on Q and R. The defined logical operations are $| \in \{\neg, \wedge, \vee\}$ (NOT, OR, AND), which correspond to Kleene's truth tables [108] (shown in Figure 3.6.B). Table 3.1 contains the formulas for logical combinations of the queries Q and R using set operations on the underlying sample identifiers. Figure 3.6 shows a detailed view of these formulas with example queries.

From the formulas in Table 3.1 it can be seen that the universe of a combined query $S = Q|R$ is given by $U_S = U_Q \cup U_R$. That is the universe of the combined query is the union of the universes of the two original queries Q and R. One assumption required for this to be true is that if a sample x is in the universe of Q



NOT	t	n	f
	f	n	t

AND	t	n	f
t	t	n	f
n	n	n	f
f	f	f	f

OR	t	n	f
t	t	t	t
n	t	n	n
f	t	n	f

First query	Q =	$T_Q = \{1\}$	$N_Q = \{2, 3, 4\}$	$U_Q = \{1, 2, 3, 4, 5\}$
Second query	R =	$T_R = \{1, 2\}$	$N_R = \{3\}$	$U_R = \{1, 2, 3, 4, 5, 6\}$
Result query	S =	T_S (True)	N_S (Unknown)	U_S (Universe)

Figure 3.6: An overview of the definition of a query and logical operations on queries (green is **True**, red is **False**, light gray is **Unknown**, dark gray is **Universe**). **(A)** The samples operated on by a query, Q, can be defined as an ordered tuple of three sets: $Q = (T, N, U)$. These correspond to sets of sample identifiers for those samples where the query holds true, T, where the query has an unknown status, N, and the universe U of all samples that are a part of the query. The samples where a query is false, F, can be derived from $F = U - T - N$. **(B)** Some of the extended truth tables from Kleene's strong logic of indeterminacy [108; 109]. Here, t is **True**, n is **Unknown**, and f is **False**. **(C)** Two example queries, Q and R, along with the logical operations used to create a new query $S = \text{operation}(Q, R)$. The queries, Q and R, consist of sets of sample identifiers (integers). The table showing the resulting query, S, consists of columns correspond to different logical operators—*NOT*, *AND*, *OR*—and rows correspond to the underlying sets of sample identifiers for $S = (T_S, N_S, U_S)$. Each cell in the table shows the set operations used to define the sample identifier sets for query S.

Result $S =$	$\neg Q$ (NOT)	$Q \wedge R$ (AND)	$Q \vee R$ (OR)
$T_S =$	$F_Q = U_Q - T_Q - N_Q$	$T_Q \cap T_R$	$T_Q \cup T_R$
$N_S =$	N_Q	$(N_Q \cap N_R) \cup (N_Q \cap T_R) \cup (T_Q \cap N_R)$	$(N_Q \cup N_R) - (T_Q \cup T_R)$
$U_S =$	U_Q	$U_Q \cup U_R$	$U_Q \cup U_R$

Table 3.1: Logical operations between two queries Q and R.

but not R ($x \in U_Q$ and $x \notin U_R$) then it is assumed that sample x has a status of **False** for query R (since $x \notin U_R$ so $x \notin T_R$ and $x \notin N_R$).

The above definitions of a query Q and logical operations on combinations of queries are implemented as three different sets of genomic sample identifiers and logical operations on instances of a `SamplesQuery` class. The logical operations overload the behaviour of the `&` (AND), `|` (OR), and `~` (NOT) operators in Python. This provides a concise syntax to write queries in the Python API (Listing 3.1).

Listing 3.1 Example code for logical operations on queries q and r

```python
import genomics_data_index.api as gdi

# Load database/index from directory "db1"
db = gdi.GenomicsDataIndex.connect("db1")

# q is an instance of SamplesQuery
# consisting of genomes with a (hasa) particular nucleotide feature
q = db.samples_query().hasa("ref:20:A:T")

# r is an instance of SamplesQuery
# consisting of genomes with a (hasa) particular gene (MLST) feature
r = db.samples_query().hasa("mlst:ecoli:adk:100")

# s is a query selecting samples in q AND r
s = q & r

# s is a query selecting samples in q OR r
s = q | r

# s is a query selecting samples in the complement of q
s = ~q

# Returns the selected (True) samples from query s
# stored in-memory as a Roaring bitmap
s.sample_set

# Returns the unknown/missing samples from query s
# stored in-memory as a Roaring bitmap
s.unknown_set

# Returns the universe set from query s
# stored in-memory as a Roaring bitmap
s.universe_set

# Returns the unselected (False) samples from query s
# derived from the other sets and returned as a Roaring bitmap
s.absent_set
```

3.3.2 Query CLI

In addition to the Python API, I also provide a command-line interface (CLI) for managing genomics indexes. The command-line interface is initiated by the main command `gdi` and consists of a number of sub-commands. Four large categories of sub-commands include the following:

1. **Managing genomic indexes**: This consists of `gdi init` which can be used to create a new index (SQLite database and directories on a file system) within a directory (*e.g.,* `gdi init index` to create an index in directory *index/*). This directory is also referred to as a *project* in the GDI interface. Subsequent commands can refer to this index/project directory (*e.g.,* `gdi --project-dir index ...`). Additionally, `gdi db` can be used to report information about the data stored in an index (*e.g.,* size on disk).

2. **Analysis**: This consists of `gdi analysis`, which will run the data analysis pipeline on a set of input data to identify genomic features. Additionally, `gdi input` or `gdi input-split-file` can be used for preparing the input data files for analysis.

3. **Loading data**: This includes `gdi load` for loading genomic features from a variety of files and `gdi rebuild` for constructing and saving a new phylogenetic tree from indexed data.

4. **Querying:** These commands include `gdi list` for listing sample names or reference file names, and `gdi query` for querying the index.

The `gdi query` command is implemented using the GDI Python API (Section 3.3.1) and provides a mechanism for using a simplified set of operations to query for particular genomes or summarize results. Table 3.2 shows a set of API and CLI equivalent commands.

3.4 Phylogenetics and clustering

Once genomes are indexed, the data stored within GDI can be used to cluster genomes or to construct a phylogenetic tree (Figure 3.1). As described in Section 2.2.5, both phylogenetic analysis and hierarchical clustering methods share a lot of similarities. Primarily they will group genomes into a hierarchical structure (a tree) which can be depicted as a dendrogram. One major difference is that phylogenetic analysis is explicitly attempting to infer the ancestry of genomes, where internal nodes of a tree represent hypothetical ancestors, and will often correct distances on branches of the tree to better reflect realistic models of evolution. Clustering algorithms, in general, do not have this requirement, though some phylogenetic analysis methods are also hierarchical clustering methods (*e.g.,* UPGMA).

Given this overlap between these two topics, I will at times refer to a phylogenetic tree as a clustering of genomes, with the intended meaning being that a phylogenetic tree (when rooted) could be viewed as a hierarchical clustering of genomes, which could be converted to a flat clustering through methods described in Section 2.2.5. This allows me to use a common codebase for both phylogenetic trees and hierarchically clustered genomes.

When the features indexed are nucleotide-level features (*i.e.,* SNVs or inser-

API command	CLI command	Description
`q.hasa(feature)`	`gdi query "hasa:feature"`	Finds all genomic samples in the index that have a particular feature.
`q.isa(sample)`	`gdi query "isa:sample"`	Finds the specific named sample.
`q.isin("sample", kind="distance", ↪ unit="substitutions", distance=5)`	`gdi query ↪ "isin_5_substitutions: ↪ sample"`	Searches for all genomes that are within 5 substitutions of the sample "sample".
`q.hasa(feature1) & q.hasa(feature2)`	`gdi query "hasa:feature1" ↪ "hasa:feature2"`	Searches for genomes that have both "feature1" **and** "feature2".
`q.hasa(feature1) | q.hasa(feature2)`	*Not defined*	Searches for genomes that have either "feature1" **or** "feature2".
`~q.hasa(feature1)`	*Not defined*	Searches for genomes that do **not** have "feature1".
`q.summary()`	`gdi query --summary`	Summarizes the number of *Present*, *Unknown*, and *Absent* samples in query.
`q.features_summary(kind="mutations")`	`gdi query ↪ --features-summary ↪ mutations`	Summarizes counts of all features found among all samples in the query (replace "mutations" with "mlst" for gene/MLST features).
`q.features_comparison( ↪ sample_categories="variable", ↪ categories_kind="dataframe", ↪ kind="mutations", unit="percent")`	*Not defined*	Groups samples into categories as defined in a column named "variable" in a dataframe and summarizes the percent of samples that have a particular feature (*kind="mutations"* in this case).

Table 3.2: Equivalent Python API and CLI commands for GDI.

In this table, q is a query initiated in the Python API.

tions/deletions) I make use of the maximum-likelihood phylogenetic method to construct a phylogenetic tree to cluster genomes. In other scenarios (in particular, for kmer-based features), I make use of hierarchical clustering methods to cluster genomes and return the cluster-tree, which can be visualized as a dendrogram similar to a phylogenetic tree.

In both cases, clustering can be initiated by the Python API, which makes use of a query object (described in Section 3.3) to select the set of genomes for clustering. Clustering geomes through phylogenetic analysis can also be initiated via the CLI. In the Python API, running `query.build_tree()` will return a new query that has been associated with the cluster tree. This new query can now be used to access the cluster-tree or for additional querying commands that make use of this cluster-tree (such as querying by distance). Listing 3.2 shows a snippet of code for clustering using the Python API.

Listing 3.2 Code for constructing a phylogenetic tree or hierarchical cluster tree

```python
import genomics_data_index.api as gdi

# Load database/index from directory "db1"
db = gdi.GenomicsDataIndex.connect("db1")

# Select a set of samples to cluster from the database
query = db.samples_query().isa(["SampleA", "SampleB", "SampleC"])

# Build a kmer tree from the selected samples
cluster_query = query.build_tree(kind="kmer", kmer_size=31)

# Build a maximum-likelihood tree from the selected samples
# Using the selected reference genome named "reference"
cluster_query = query.build_tree(kind="mutation", scope="reference")

# The tree object is associated with the new query object
cluster_query.tree

# The "cluster_query" object can now be used to query
# based on distances in the tree
cluster_query.isin("genome", kind="distance",
                   distance=1e-7, units="substitutions/site")
```

3.4.1 Building phylogenies from nucleotides

The maximum likelihood approach to phylogenetic analysis provides a way to construct a hypothetical evolutionary tree using the genomic sequencing data. This approach is used when there are nucleotide features available for all genomes in question and where each genome is already aligned to the same reference genome. These conditions provide a mechanism to construct a multiple sequence alignment, one of the key inputs to the maximum-likelihood approach for constructing a phylogeny.

Constructing a multiple sequence alignment

In order to construct a multiple sequence alignment, I first make use of the VCF [75] and BED [79] file formats as well as the *SAMtools* and *BCFtools* suite of software [99]. I use the `bcftools consensus` command (`https://samtools.github.io/bcftools/bcftools.html#consensus`) to create a consensus sequence for each genome by applying substitutions and deletions (I exclude insertions) stored in a VCF file to the corresponding reference genome. BCFtools includes an additional parameter to mask out regions stored in a BED file with an `N` character, which is also applied to the consensus sequence to indicate regions on a genome where data is missing/unknown.
To construct a multiple sequence alignment from each individual consensus sequence, I could use alignment software such as MAFFT [110]. However, as a shortcut to avoid running this software (which would require additional time and resources), I instead concatenate each consensus sequence together into a single file, which should be equivalent to a multiple sequence alignment. This works under the condition that each nucleotide in every consensus sequence is homologous (and so concatenating each consensus sequence constructs an alignment where every site in the alignment is homologous). I am (mostly) able to satisfy this condition due to the following constraints imposed when constructing each consensus sequence with `bcftools consensus`:

1. Every consensus sequence is constructed from the same reference genome.

This means that any substitutions introduced are in the same position on the reference genome (and so are homologous).

2. I do not include insertions. Insertions would change the coordinates of downstream substitutions (and so substitutions would no longer remain homologous).

3. I replace deletions with a `-` (gap) character. If I had instead removed the actual sequence in the place of a deletion, I would change downstream coordinates of substitutions and so consensus sequences would no longer remain homologous.

Given that these conditions hold due to my parameter choices for `bcftools consensus`, the concatenated alignment should align different homologous sites with each other. One possible exception is in regions with many substitutions or other criteria which could lead to a sub-optimal alignment. This is discussed in more detail in the limitations section (Section 5.3.3).

Constructing the phylogenetic tree

Once a multiple sequence alignment is constructed, I use the software iqtree [17] to generate a phylogenetic tree using the maximum likelihood method. This phylogenetic tree can either be saved to a separate file on the file system (as a Newick formatted file [111]), or stored in the relational database. In my CLI, the command to build a maximum-likelihood tree is `gdi build tree [OPTIONS]`. Alternatively, the command `gdi rebuild tree [OPTIONS] [REFERENCE]...` can be used to build a maximum-likelihood tree and save the corresponding tree to the relational database associated with the selected reference genome. Alternatively, in the Python-based API the command to build a tree is `query.build_tree()`, where *query* is a query that selects a set of samples to be used for building a tree.

3.4.2 Hierarchical clustering

When nucleotide sequence data is not available (*e.g.,* when examining genomes using kmers or genes), character-based methods for constructing phylogenetic trees to cluster genomes will not work. In these cases, I instead rely on algorithms derived from distance-based methods of phylogenetic analysis (hierarchical clustering algorithms in particular). I have implemented this for clustering genomes using kmers as identified using sourmash [54].

Clustering with kmer distances

Clustering by kmers can be initiated using the `query.build_tree()` command in the Python API (where *query* is a query object which has selected the set of genomes to cluster). Unlike the maximum-likelihood method, there is currently no facility in GDI to save the kmer-based hierarchical cluster. However, generating these clusters is much quicker when compared to constructing a maximum-likelihood phylogenetic tree.

The first step to clustering genomes by their kmer content is to calculate and record all pairwise distances between different genomes into a distance matrix. I use the `sourmash compare` command to construct a pairwise distance matrix of Mash distances (which are approximations of the Jaccard distances between genomes using kmer content). This pairwise distance matrix is then clustered using the `scipy.cluster.hierarchy` module from *scipy* [112] to construct a single-linkage hierarchical cluster of the kmer distances. This is further refined by the *scikit-bio* [113] Python package to convert the cluster to a tree in Newick format. The tree can then be passed to the *ETEToolkit* [21] to visualize as a dendrogram or saved as a newick file.

3.4.3 Parallelization of the clustering stage

To support parallel processing of the clustering stage, I make use of the built-in mechanisms for parallel processing provided by some of the software used for clustering. For constructing a **maximum-likelihood phylogeny** using iqtree [17], I make use of the `--threads-max` and `-T AUTO` command-line arguments, which will specify the maximum number of threads to use when constructing a tree and auto-detect an appropriate number of threads respectively. The auto-detected number of threads may be less than the maximum and is measured internally by iqtree. When constructing a **distance-based kmer tree**, I make use of the `--processes` argument, which is passed to `sourmash compare` during construction of the pairwise similarity matrix (which is then converted to a distance matrix).

The number of processes (or threads) to use when clustering genomes is provided by the number of processes passed to the GDI command-line interface. Or, alternatively, when using the GDI Python API, each clustering method provides an `ncores` parameter which can be used to set the number of processing cores.

3.4.4 Flat clusters to tree comparison

To (partly) facilitate the evaluation of how well a hierarchical cluster or phylogenetic tree corresponds to a flat cluster of genomes, I have developed and implemented a mechanism to compare a flat cluster to a rooted tree. There exists options to evaluate how well two different flat clusters correspond to each other (such as Silhouette scores [114]), but a hierarchical cluster could be converted to many different flat clusters depending on how the tree is cut ([73] and Section 2.2.5). In order to avoid the issue of how to best cut a tree into clusters prior to comparing different flat partitions to each other, I developed a scoring system to directly compare a flat partition to a phylogenetic (or hierarchical clustering) tree.
In order to construct a scoring system, I first define a way to map a single set of leaves from a tree T to a score represented by a number from 0 to 1. This score reflects how well the set of leaves match the structure of the tree (*i.e.,* are the chosen leaves scattered across the entire tree or do they group together under a single branch). Next, I define a way to examine collections of sets of leaves from a tree (representing different clusters of genomes) by either examining the distribution of the corresponding scores or defining a single numerical score (*e.g.,* mean or median) from a list of scores. An example of this scoring system is shown in Figure 3.7.

Defining the score for a single set of genomes to a tree

Let T be some arbitrary rooted tree and let L be a chosen set of leaves from this tree. To define a mechanism to assign a score to the leaves L for tree T, I first define the following requirements:

1. Given a rooted cluster/phylogenetic tree, T, and a set of genomes (leaves) on the tree L, I wish to define a function, S, which maps this set of genomes to some number between 0 and 1. That is $S(T, L)$ is a function which returns some number in the interval $[0, 1]$.

2. The highest possible score, $S(T, L) = 1$, is achieved if and only if all genomes L are the leaves of a monophyletic clade on the tree T.

3. The value $S(T, L) = 0$ represents the lowest possible score for genomes L.

From these requirements, I define this cluster scoring according to Algorithm 3.1. This algorithm will satisfy these three requirements (proof in Appendix E), with the caveat that the minimum score $S(T, L) = 0$ is only achieved in the limit as the size of the tree T increases (specifically the minimum score is $2/N$ where N is the number of leaves in the tree T and is achieved when a pair of leaves has a most recent common ancestor of the root of the tree, see Appendix E).

Defining the score of a collection of sets of genomes to a tree

To extend Algorithm 3.1 to a scoring system for a collection of genomes compared to a tree let us first make a few definitions. Let T be some tree containing a collection of genomes (represented as leaves L of the tree). Let $G = (g_1, g_2, ..., g_n)$ be some tuple of n subsets of genomes such that $g_m \in G$ is some subset of leaves of the tree T (*i.e.*, $g_m = \{l_1, l_2, ...\}$ where l_x is a leaf on tree T). I define a cluster scoring function, R, to be a function that maps this n-tuple of genomes to an n-tuple of scores. In other words, $R(T, G)$ returns some n-tuple of scores for the G clusters on tree T.

Algorithm 3.1 Cluster score for a set of leaves from a tree

```
Require: A non-empty set of leaves $L$ from a rooted tree $T$.
1: if $|L| = 1$ then
2:     $L_m \leftarrow L$
3: else
4:     $m \leftarrow$ the most recent common ancestor of $L$ in $T$.
5:     $L_m \leftarrow$ the set of descendant leaves of node $m$.
6: end if
7: $S \leftarrow |L \cap L_m| / |L \cup L_m|$        ▷ $S$ is the Jaccard index of $L_m$ and $L$
8: return The score $S$.
```

Algorithm 3.2 Cluster scores for a tuple of sets of leaves from a tree

```
Require: An $n$-tuple of sets of non-empty leaves $G = (g_1, g_2, ..., g_n)$ from a rooted
    tree $T$.
1: $A \leftarrow$ empty list
2: for each $g \in G$ do
3:     Append $S(g, T)$ to $A$        ▷ $S$ is from Algorithm 3.1
4: end for
5: return $A$
```

The scoring function in Algorithm 3.2 implements this function R and will return an n-tuple of numbers from 0 to 1. To assign an overall score to a tree T, I can either calculate statistical summaries of this list of numbers (*e.g.,* mean or median) or I can examine the distribution of these numbers as a whole (Figure 3.7.D).

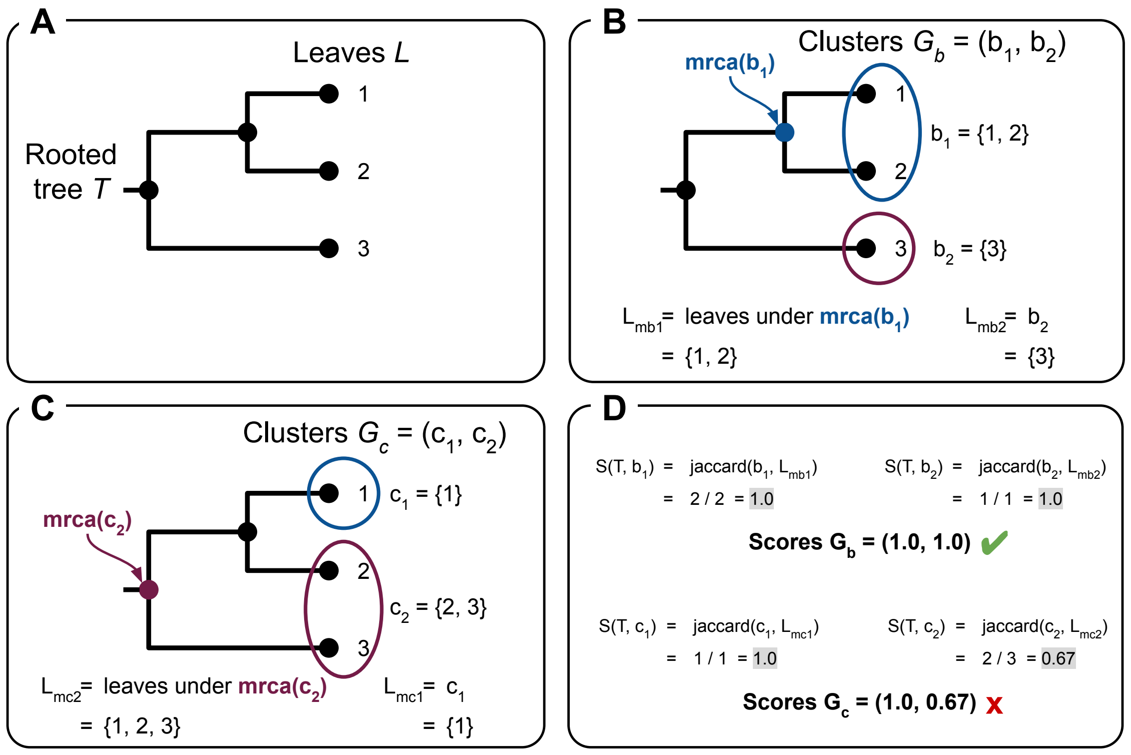

A
Leaves L
Rooted tree T
1
2
3
B
Clusters G_b = (b_1, b_2)
mrca(b_1)
b_1 = {1, 2}
b_2 = {3}
L_mb1 = leaves under mrca(b_1) = {1, 2}
L_mb2 = b_2 = {3}
C
Clusters G_c = (c_1, c_2)
mrca(c_2)
c_1 = {1}
c_2 = {2, 3}
L_mc2 = leaves under mrca(c_2) = {1, 2, 3}
L_mc1 = c_1 = {1}
D
S(T, b_1) = jaccard(b_1, L_mb1) = 2 / 2 = 1.0
S(T, b_2) = jaccard(b_2, L_mb2) = 1 / 1 = 1.0
Scores G_b = (1.0, 1.0) ✓
S(T, c_1) = jaccard(c_1, L_mc1) = 1 / 1 = 1.0
S(T, c_2) = jaccard(c_2, L_mc2) = 2 / 3 = 0.67
Scores G_c = (1.0, 0.67) x

Figure 3.7: An example of calculating different cluster scores for the same tree according to Algorithms 3.1 and 3.2. **(A)** A rooted tree T which has three leaves $L = \{1, 2, 3\}$. **(B)** Two clusters $G_b = (b_1, b_2)$ consisting of sets of leaves are shown on the tree T. The most recent common ancestor to b_1 (*mrca(b_1)*) is labeled on the tree. The leaves that are descendants of *mrca(b_1)* are used to define the set L_{mb1}, which is used in Algorithm 3.1 for calculating a score for this cluster. The value L_{mb2} is equal to the set b_2 since it consists of a single leaf. **(C)** Two clusters $G_c = (c_1, c_2)$ consisting of sets of leaves are shown on the tree T. The most recent common ancestor to c_2 (*mrca(c_2)*) is labeled on the tree. The leaves that are descendants of *mrca(c_2)* are used to define the set L_{mc2}, which is used in Algorithm 3.1 for calculating a score for this cluster. The value L_{mc1} is equal to the set c_1 since it consists of a single leaf. **(D)** The rest of the calculations for the scores for clusters G_b and G_c. The label *jaccard(b_1, L_{mb1})* is shorthand for the Jaccard index of sets b_1 and L_{mb1}. These result in two ordered pairs of numbers (labeled **Scores** G_b and **Scores** G_c), which can be compared against each other to show that the clusters G_b match the tree better than G_c (the minimum score of 1.0 in the ordered pairs is greater than 0.67). This is indicated with a green check-mark next to **Scores** G_b and a red x next to **Scores** G_c.

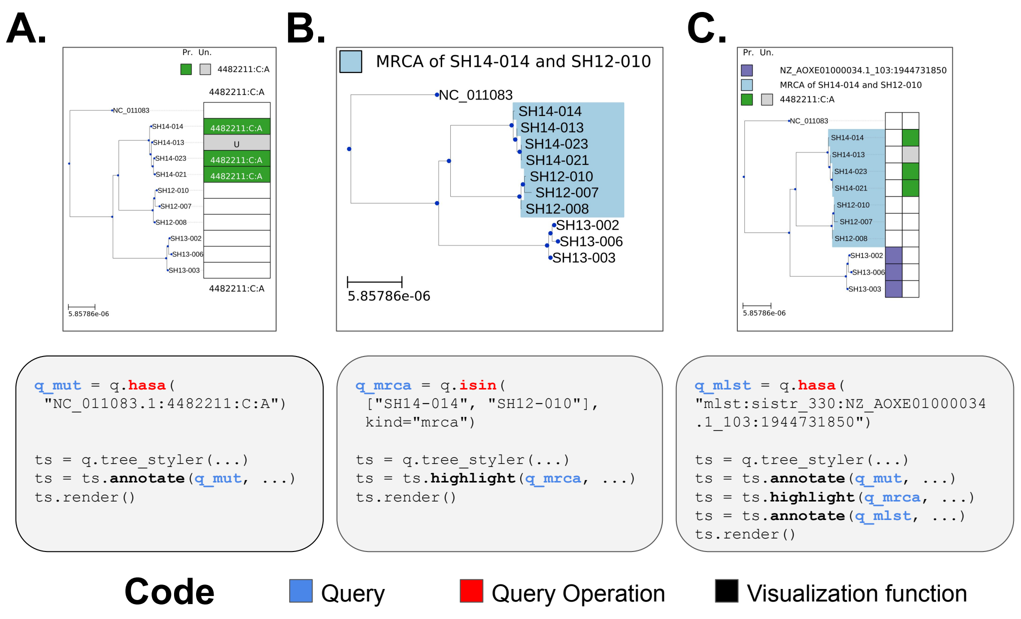
Implementation of the cluster scoring system

These two algorithms (Algorithms 3.1 and 3.2) are implemented in my indexing software through a class `ClusterScorer`. This takes as input a query that must select some set of genomes and must have an attached tree (whether a phylogenetic or hierarchical cluster tree). This version of `ClusterScorer` will return a single score for the selected genomes from the query for the tree using Algorithm 3.1. Alternatively, `ClusterScorer` can also take as input a collection of queries which select sets of genomes. In this case, `ClusterScorer` will return a list of scores for each query using Algorithm 3.2. Example code is given in Listing 3.3.

3.5 Visualization

A visualization component is included in the Python API to visualize phylogenetic trees and highlight arbitrary queries on these trees. This is implemented by using the ETEToolkit [21], but includes functionality to more easily construct a visualization using queries. The phylogenetic tree to visualize can be loaded directly from a tree saved in the index, or a tree can be attached to a query using the `join_tree()` method—providing support for visualizing samples on trees generated from external methods.

Figure 3.8 shows the two main methods used to display the output of a query on a tree: `highlight` and `annotate`. The `highlight` method will colour the background of leaves of a tree (corresponding to samples from a query). The `annotate` method will add a track next to the leaves of a tree and colour this track based on



```
q_mut = q.hasa(
 "NC_011083.1:4482211:C:A")


ts = q.tree_styler(...)
ts = ts.annotate(q_mut, ...)
ts.render()
```

```
q_mrca = q.isin(
 ["SH14-014", "SH12-010"],
 kind="mrca")

ts = q.tree_styler(...)
ts = ts.highlight(q_mrca, ...)
ts.render()
```

```
q_mlst = q.hasa(
"mlst:sistr_330:NZ_AOXE01000034
.1_103:1944731850")

ts = q.tree_styler(...)
ts = ts.annotate(q_mut, ...)
ts = ts.highlight(q_mrca, ...)
ts = ts.annotate(q_mlst, ...)
ts.render()
```

Figure 3.8: An overview of the Python API used to visualize queries on a phylogenetic tree. The lower portion shows the code while the upper portion shows the visual produced by the code. A visualization is initiated by running the `q.join_tree(tree).tree_styler()` method on a query q. **(A)** The `annotate()` method is used to construct a heatmap showing the status of samples in the query aligned with the tree (status is one of *presence*, *absence*, or *unknown*). **(B)** The `highlight()` method colours the background of the leaves of a tree based on the status of samples in the query. **(C)** The `highlight()` and `annotate()` methods can be combined together to visualize arbitrary sets of samples.

the status of a query (effectively letting you display a heatmap next to the dendrogram). The visualization component will colour samples in a query using the three different states: **True** (present), **Unknown**, and **False** (absent). Colours and other display options can be modified using the Python API.

Listing 3.3 Example code for scoring clusters in the Python API

```python
import genomics_data_index.api as gdi
import ete3

# Load database/index from directory "db1"
db = gdi.GenomicsDataIndex.connect("db1")

# Load a phylogenetic tree from a newick file using the ete3 package
# I assume this tree is rooted
tree = ete3.Tree("tree.nwk")

# Initiate a query containing all samples in the index
query = db.samples_query()

# The query must have a joined phylogenetic/cluster tree
# Which can be achieved using "join_tree" on a query
query = query.join_tree(tree)

# Create a ClusterScorer from the query
scorer = ClusterScorer(query)

# Select some subset of genomes on the tree
# That have a (hasa) particular feature
q1 = query.hasa(...)

# Score this subset of genomes based on
# how well it corresponds to a monophyletic group in the tree
scorer.score(q1)
# Returns a single number (e.g., 0.5)

# Define a list of subsets of genomes on the tree
# as a list of queries
q1 = query.hasa(...)
q2 = query.isa(...)
queries_list = [q1, q2]

# Score the list of queries (by passing a list to score())
scorer.score([q1, q2])
# Returns a table of scores, one row for each query.
```

Chapter 4

Evaluation

In order to evaluate GDI, I used three separate scenarios consisting of different data. I included an additional fourth evaluation consisting of a qualitative comparison of GDI to other, similar-functioning, software.

1. **Data simulation**. Here, I evaluated the software using simulated nucleotide variants. I processed this simulated data using the *Data analysis pipeline*, *Indexing*, *Clustering*, and *Querying* stages, and examined the sensitivity and precision of GDI at detecting nucleotide variants and constructing phylogenetic trees.

2. **Real-world genome assemblies**. I evaluated GDI by measuring computational resource usage for processing real-world SARS-CoV-2 genome assembly data.

3. **Real-world genome sequence reads**. I evaluated GDI by measuring computational resource usage for processing real-world WGS reads from a variety

of different microbial organisms.

4. **Comparison to existing software**. I qualitatively compare GDI to existing software.

The code for the evaluations is available as a set of Bash, Python, and Jupyter notebooks (`https://github.com/apetkau/genomics-data-index-evaluation`, version *1.0.1*, doi: 10.5281/zenodo.7021387). All of these evaluations were run using GDI version *0.6.0* (doi: 10.5281/zenodo.6485506). Recording resource usage (run time and peak memory) was performed using the CMDBench [115] software package. The evaluations were run using a computer with 62 GB of memory, a 1 TB solid state drive (*Samsung SSD 860*), and 2 physical processors (2 x *Intel(R) Xeon(R) Gold 5120 CPU @ 2.20 GHz*) with 14 cores each (hyper-threaded to 28) for a total of $2 * 28 = 56$ processing units.

Unless otherwise stated, I ran all of the data analysis and indexing using GDI with 32 processing cores. These processing cores are used for parallel processing of each of the different stages of GDI (Figure 3.1). The **Data analysis** stage performs parallel processing through Snakemake's construction of a DAG and scheduling of jobs (Section 3.1.2). The **Indexing** stage performs parallel processing by spawning sub-processes to handle saving data for individual samples and constructing a large DataFrame prior to creating an inverted index (Section 3.2.4). The **Clustering/phylogenetics** stage performs parallel processing by using the built-in arguments to the underlying software for tree construction (Section 3.4.3). The **Querying** and **Visualization** stages do not have any parallel processing implemented, but they also tend to execute much quicker than any of the other stages.

4.1 Data simulation

In order to test out how well the software is able to read nucleotide variants and reconstruct a phylogenetic tree, I evaluated the software using a simulated dataset. The simulations were performed using Jackalope [116], which provides a mechanism to simulate genomes and sequence reads corresponding to a provided phylogenetic tree. The leaves of the provided tree correspond to genomes to simulate, while the branch lengths and topology of the tree are used to control the generation of substitutions, insertions, and deletions. As output, Jackalope produces a set of WGS reads corresponding to simulated genomes represented by leaves of the provided tree.

4.1.1 Methods

I constructed the simulated dataset by first using an input phylogenetic tree consisting of 59 genomes plus a reference genome which was constructed from already published WGS data derived from multiple outbreaks of *Salmonella enterica* [117]. This phylogenetic tree was used as a template to simulate substitutions and insertions/deletions that scale with the branches of the tree. The particular tree was chosen as it contained multiple clades derived from different outbreaks of the same organism and is representative of a tree that would be constructed during a real-world outbreak investigation (the particular organism here does not matter as I am simulating genomes from this tree). I constructed a random reference genome using Jackalope [116] with length 19,699 bp (composed of two sequences of lengths 10,834 bp and 8,865 bp, respectively) to be used as the starting point for simulating nucleotide variants. I chose a random reference genome so I could adjust the length of the genome to speed up the data simulations and I chose to simulate two sequences to represent a scenario a bit more complicated than just a single sequence.

To simulate data I used the input phylogenetic tree, reference genome, and the HKY85 [118; 71] evolutionary model to simulate 59 different sets of substitutions (and insertions/deletions) using the reference genome as a template and following the topology and branch lengths of the input phylogenetic tree. I saved each simulated genome to a consensus sequence file (in FASTA format). As these consensus sequence files are derived directly from the Jackalope simulations (as opposed to being assembled from reads), they represent an idealized assembly that is unaffected by any artifacts introduced by *de novo* assembly of reads. However, for additional validation, I also simulated Illumina paired-end reads from these consensus sequences (2 X 250 bp read length, mean fragment length 550 bp, standard deviation fragment length 100 bp). Nucleotide variant and sequence read simulation was repeated for two separate sets of scenarios: 1) I simulated a single collection of 59 genomes with a fixed transition rate of 0.2 in the HKY85 model and adjusted the coverage of the simulated sequence reads to be: 5X, 10X, 20X, 30X, 40X, 50X; 2) I simulated multiple collections of 59 genomes with a fixed coverage of 30X and adjusted the transition rate in the HKY85 model to be: 0.05, 0.1, 0.2, 0.5, 1.0, 2.0, 5.0. In each case the transversion rate was set to be half the transition rate.

I next indexed each set of simulated data using my software and the default pipeline (`gdi analysis ...`). This was repeated in each of the above scenarios for both the saved consensus sequences (the **assemblies** scenario, which represents a case of an idealized, perfect assembly) as well as the sequence reads (the **reads** scenario). As part of the indexing process, I also constructed a maximum-likelihood phylogenetic tree of the 59 genomes plus the reference genome by exporting a whole-genome alignment of nucleotide data from the index and using this alignment as input to the software iqtree [17] with the `--fast` option and the *GTR+F+R4* evolutionary model (see Section 3.4.1 for details).

To compare the nucleotide features detected by my analysis pipeline and indexing process, I used the index to construct a table of all detected nucleotide features following indexing. I then defined a $(sample, feature)$ pair by concatenating the sample identifier with the feature name (*e.g.,* `SampleA:Feature1`, `SampleA:Feature2`, ..., `SampleB:Feature1`, ...). Each of these $(sample, feature)$ pairs was used to determine True Positives (TP), False Positives (FP), and False Negatives (FN) (where the Truth set is defined by the $(sample, feature)$ pairs simulated by Jackalope). As most positions on the reference genome are not associated with features, there is a large imbalance between the relatively few true positives and the much larger number of true negatives. In such a scenario, it is recommended to make use of sensitivity (*i.e.,* recall) and precision instead of specificity to evaluate performance [119]. Sensitivity (SN) is defined as $SN = TP/(TP + FN)$, while precision (PR) is defined as $PR = TP/(TP + FP)$. A third measure is the F1 score, which combines sensitivity and precision into a single value and is defined as $F1 = 2 * PR * SN/(PR + SN)$. As none of these measures require True Negatives (TN), I do not record TN.

In order to compare the constructed phylogenetic trees, I used the APE [120] package in the R programming language to load up and manipulate the phylogenetic trees. I computed a pairwise distance between each tree and the initial reference tree (taken to represent the True tree). I computed two different distance measures: the normalized Robinson-Foulds distance [121] as implemented in phangorn [122] and the Kendall-Colijn metric [123] as implemented in treespace [124].

4.1.2 Comparison across coverages

Figure 4.1 shows the counts of TP, FP, and FN $(sample, feature)$ pairs across a range of depth of coverage values, and Figure 4.2 shows the sensitivity, precision, and F1 scores. From these figures it can be seen that for lower depth of coverage values, the number of TP and FP is also low for **reads** [415 TP and 19 FP $(sample, feature)$ pairs for a coverage of 5X, a sensitivity of 0.025]. At this same coverage value, the number of FN is very high (16,517 FN for a coverage of 5X). However, as the coverage increases above the minimum depth of coverage value of 10X, the TP, FP, and FN's quickly plateau at a depth of coverage of 20X (twice that of the minimum coverage of 10X). These values reach 14,401 TP, 666 FP, and 2,531 FN for a depth of coverage of 20X—resulting in a sensitivity of 0.851 and precision of 0.956. Further increase in coverage does not appear to impact the sensitivity or precision as dramatically, with a sensitivity of 0.886 and precision of 0.957 for 50X. The **assemblies** are unaffected by the depth of coverage and hence appear as a horizontal line across all possible coverage values. These consistently perform better than the read dataset—16,762 TP, 0 FP, and 170 FN—which shows that detecting nucleotide variants using reads is less effective than using the ide-

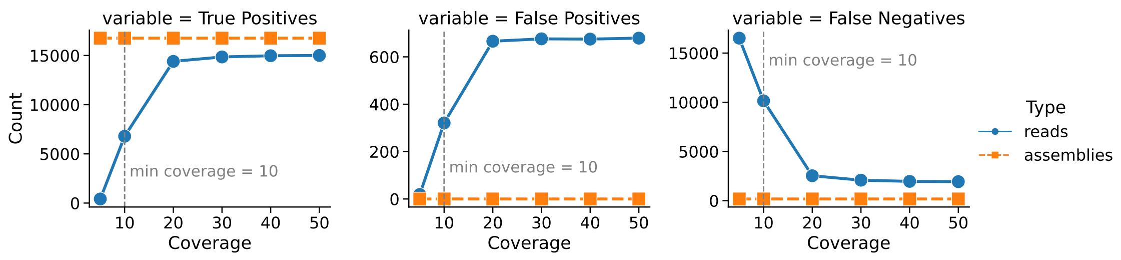


Figure 4.1: A comparison of the counts of true positive/false positive/false negative $(sample, feature)$ pairs across different simulated read coverages. These are divided up into two different data types: **reads** and (idealized or perfect) **assemblies**. As read coverage does not apply to assemblies, the values appear as a horizontal line (with each data point a replication of constructing the same index). The gray dotted line corresponds to a coverage value of 10X, which is the default minimum depth of coverage used to detect a nucleotide variant in GDI. True Negatives are not shown due to the reason described in the Methods (section 4.1.1).

alized assembled genomes simulated by Jackalope. These results are mirrored in the scores from Figure 4.2, which also plateau at a depth of coverage of 20X, but where the reads are consistently worse than the assemblies (sensitivity of 0.851 and precision of 0.956 for reads at 20X compared to a sensitivity of 0.990 and precision of 1.00 for assemblies).

Figure 4.3 shows the impact that adjusting the read coverage depth has on a phylogenetic tree constructed from the detected nucleotide variants. The phylogenetic tree was compared to the initial phylogenetic tree used to simulated data to determine a distance using two different measures: Normalized Robinson-Foulds (NRF) [121] and the Kendall-Colijn (KC) metric [123] (with $\lambda = 0.5$, meaning that both branch lengths and topology of the tree contribute equally to the computed distance). Similar to the case of detected variants, the tree distances are higher for a lower coverage value for the **reads** scenario (NRF = 0.948 and KC = 135 for a read coverage of 5X). However, these distances drop as the coverage increases (NRF

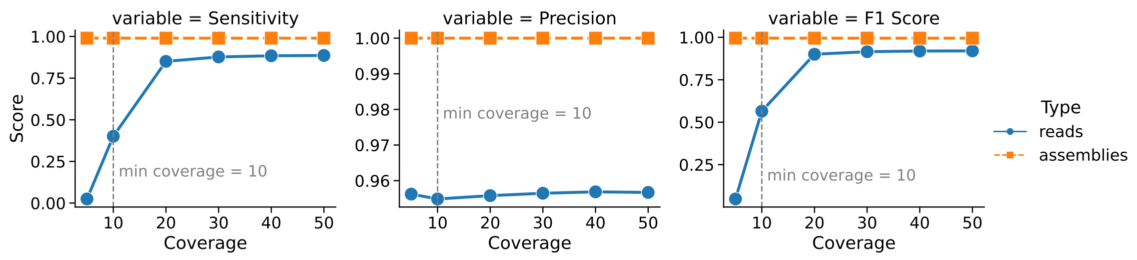


Figure 4.2: A comparison of the sensitivity/precision/F1 scores for detected $(sample, feature)$ pairs across different simulated read coverages. These are divided into two different data types: **reads** and (idealized or perfect) **assemblies**. As read coverage does not apply to assemblies, the values appear as a horizontal line (with each data point a replication of constructing the same index). The gray dotted line corresponds to a coverage value of 10X, which is the default minimum depth of coverage used to detect a nucleotide variant in GDI.

= 0.724 and KC = 79.3 for a read coverage depth of 20X), indicating that as the depth of coverage increases, the constructed phylogenetic tree more closely resembles the original simulated phylogenetic tree. For the (idealized or perfect) **assemblies** scenario, the constructed phylogenetic tree is unaffected by the read depth of coverage, similar to Figure 4.1. However, construction of a phylogenetic tree may vary depending on initial conditions for the maximum-likelihood approach, leading to variability in the distance scores. Hence, for the **assemblies** scenario, I chose to repeat the construction of the tree 6 times and selected the tree with the maximum NRF score (and thus the worst-performing tree, shown on Figure 4.3 as an orange dotted line). The scores for the worst-performing **assemblies** tree is NRF = 0.707 and KC = 70.2, which is lower than all the trees constructed from the **reads** scenario (Figure 4.3). Since this is the worst-performing (most distant) tree of 6 trials for the (idealized or perfect) assemblies scenario, this suggests that building a tree from nucleotide variants identified from an assembled genome (even the worst-

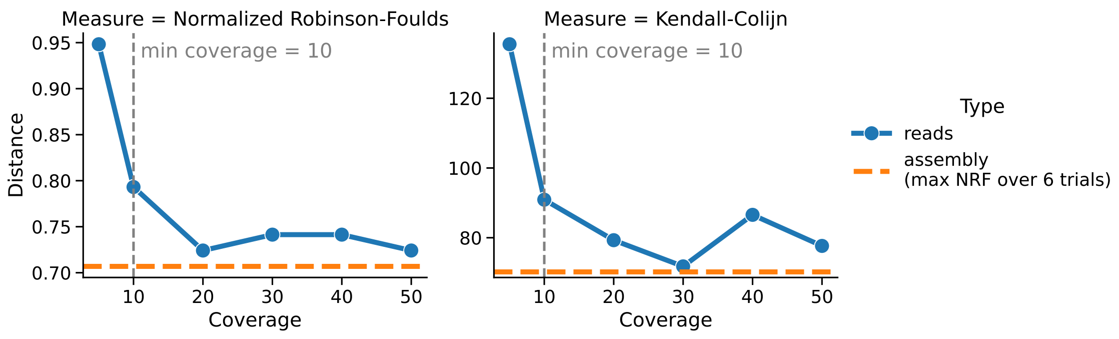


Figure 4.3: A comparison of the distances between the initial input tree and the constructed phylogenetic trees over different read depth of coverages. The results are divided up into two different data types: **reads**, and (idealized or perfect) **assemblies**. Two different distance measures are employed: **Normalized Robinson-Foulds** (NRF) and **Kendall-Colijn** (KC). As read coverage does not apply to assemblies, the values appear as an orange horizontal line corresponding to a tree with a maximum NRF score out of 6 replicates. The gray dotted line corresponds to a coverage value of 10X, which is the minimum depth of coverage used to detect a nucleotide variant in GDI for the **reads** scenario.

performing tree) outperforms directly using reads to identify nucleotide variants regardless of the read coverage. Supplemental figures Figure B.3 and Figure B.4 show a comparison of the initial phylogenetic tree and the best constructed phylogenetic tree for reads (at coverage 20X) and assemblies (max NRF score out of 6 replicates, hence worst tree) respectively. Figure B.5 shows a comparison of the worst phylogenetic tree for reads (a coverage of 5X) to the initial reference genome.

4.1.3 Comparison across substitution divergences

To examine how well GDI performs across a range of substitution divergences among genomes, I also simulated data where I kept the read coverage depth constant at 30X and varied the rate of substitutions. To vary the rate of substitutions, I adjusted the *transition rate* parameter as part of the HKY85 model using the values $\{0.05, 0.1, 0.2, 0.5, 1.0, 2.0, 5.0\}$. The *transversion rate* was set to be half the *transition rate* in all cases. I converted the *transition rate* values to an *observed substitution divergence* value in order to make the data interpretation easier. To do this, I pre-selected the genome in the initial input phylogenetic tree which had the longest path to the reference genome (*SH14-012*, see Figure B.1). I then counted the number of substitutions inserted into *SH14-012* across different transition rates. I normalized the number of substitutions to be a percentage of the reference genome length, which I use as the *substitution divergence* values (*i.e.,* 0% represents 0 observed substitutions between the *reference* and *SH14-012*, while 100% represents 19,699 observed substitutions, the same length as the reference genome). Table 4.1 shows the transition rate and corresponding substitution divergence values. Figure B.2 shows a plot of these values. Due to time constrains I only simulated one collection of genomes for every transition rate.

Figure 4.4 shows the counts of TP, FP, and FN $(sample, feature)$ pairs across a range of substitution divergence values. From this figure it can be seen that the number of TP increases with larger substitution divergences (which is expected as a larger divergence means a larger number of TP). However, this hits a peak and then drops off again. This is especially apparent for the (idealized or perfect) **assemblies**, where a peak TP value (49,552) is found for a divergence of 6.77% but then drops off to 1,395 for a divergence of 10.4%. This is reflected in the sensitivity value of 0.948 for 6.77% divergence, but then dropping to 0.0164 for 10.4% divergence (Figure 4.5). An explanation for this is provided at the end of this section (Section 4.1.3) and involves parameters passed to the Minimap2 [39] software

Transition rate	Number of substitutions	Substitution divergence
0.05	310	1.57%
0.1	371	1.88%
0.2	473	2.40%
0.5	827	4.20%
1.0	1,333	6.77%
2.0	2,049	10.4%
5.0	3,481	17.7%

Table 4.1: Transition rates, number of substitutions, and divergence between *reference* and *SH14-012*. Reference genome length is 19,699 bp. I only performed a single simulation for every transition rate.

used for aligning assemblies. This same trend is not observed for reads, where a divergence of 6.77% has a sensitivity of 0.663 and falls to 0.519 for a divergence of 10.4%. However, looking at the FP counts, for assemblies, these remain constant at 0 FP for all divergence values (hence precision is constant at 1.0). For reads, the FP increase for larger divergence values and the precision decreases for higher divergence values (with the lowest precision of 0.590 for a divergence of 17.7%). Examining the F1 score (Figure 4.5) we can observe that the (idealized or perfect) assemblies outperform reads for lower divergence values (at 6.77% there is an F1 score of 0.973 for assemblies to 0.736 for reads). However the F1 score for assemblies drops to a score of near 0.0 beyond a substitution divergence of 6.77% (at 10.4% an F1 score of 0.0323 for assemblies compared to 0.608 for reads).

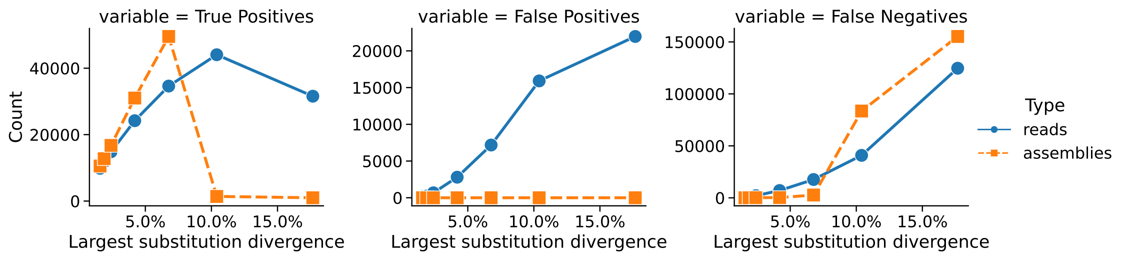


Figure 4.4: A comparison of the counts of true positive/false positive/false negative $(sample, feature)$ pairs across different simulated substitution divergences between *SH14-012* and the *reference* genome. Substitution divergences were controlled by adjusting the transition rate from 0.05 to 5.0 (see Table 4.1). The results are divided up into two different data types: **reads**, and (idealized or perfect) **assemblies**. True negatives are not shown due to the reason described in the methods (section 4.1.1).

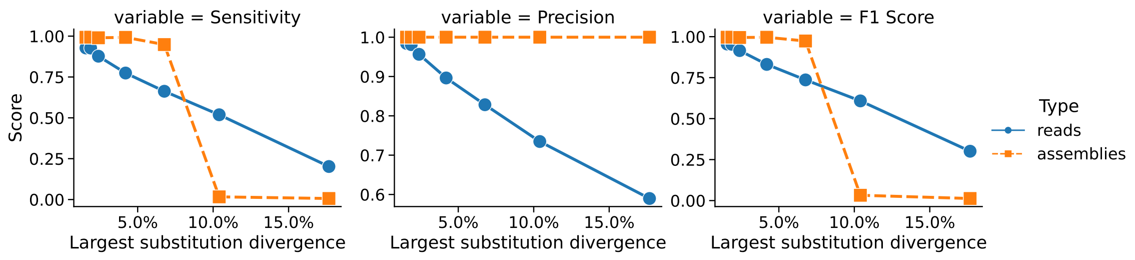


Figure 4.5: A comparison of the scores of $(sample, feature)$ pairs across different simulated substitution divergences between *SH14-012* and the *reference* genome. Substitution divergences were controlled by adjusting the transition rate from 0.05 to 5.0 (see Figure B.2). The results are divided up into two different data types: **reads**, and (idealized or perfect) **assemblies**.

Figure 4.6 shows the impact that adjusting the substitution divergence has on a phylogenetic tree constructed from the detected nucleotide variants. This figure shows the distance between the initial phylogenetic tree and the tree constructed by GDI after detecting and loading all the nucleotide variants. For lower substitution divergence values, the (idealized or perfect) **assembly** trees appear to have a lower distance (and so are a closer match to the original tree) than the **reads** trees.

The best performing assembly tree (measured by the NRF score) occurs with a divergence value of 1.88% (NRF = 0.672, KC = 78.4). This can be contrasted with the reads tree, which is the worst performing for this divergence value (divergence = 1.88%, NRF = 0.759, KC = 87.6). Supplemental figures Figure B.6 (assembly) and Figure B.7 (reads) show a comparison of the initial phylogenetic tree and the constructed phylogenetic trees for each of these cases respectively.

For a divergence of 6.77% and beyond the (idealized or perfect) assemblies trees failed to be constructed by GDI and hence are missing from Figure 4.6. The assembly tree that gets successfully built with the highest divergence occurs with divergence 4.20% and has NRF=0.741 and KC=70.3. By contrast, the reads tree at this divergence actually has a lower NRF distance but a higher KC distance (divergence = 4.20%, NRF = 0.724, KC = 93.0). Supplemental figures Figure B.8 (assembly) and Figure B.9 (reads) show a comparison of the initial phylogenetic tree and the constructed phylogenetic trees for each of these cases respectively.

Investigation of poor performance for assemblies at high divergences

To examine a bit more closely what is going on with (idealized or perfect) **assemblies** at a divergence greater than 4.20%, I inspected the nucleotide variants listed in the VCF file for *SH14-012* (the file stored in my index which lists all the identified nucleotide variants) and the proportion of unknown/missing sites listed in the corresponding BED file for this sample (the file in my index that lists ranges of positions that are unknown/missing for the nucleotide-level features). For a divergence of 4.20% I find that there are 712 records in the VCF file, and the BED file

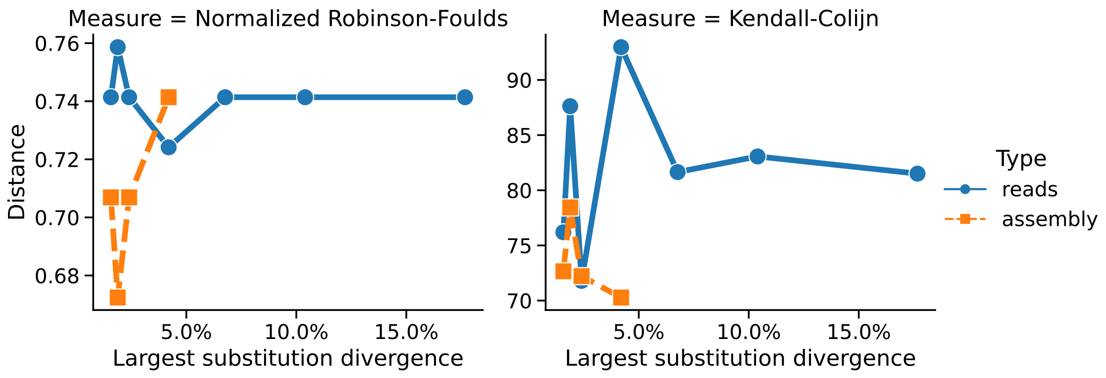


Figure 4.6: A comparison of the distance to constructed phylogenetic trees over different substitution divergences between *SH14-012* and the *reference* genome. Substitution divergences were controlled by adjusting the transition rate from 0.05 to 5.0 (see Figure B.2). The results are divided up into two different data types: **reads**, and (idealized or perfect) **assemblies**. Two different distance measures are employed: **Normalized Robinson-Foulds** and **Kendall-Colijn**. Data points for divergence values $\geq$ 4.20% for the **assembly** case are missing as iqtree [17] was unable to construct phylogenetic trees for these divergence values.

lists 0.02% of the reference genome as being missing/unknown (4/19,699 bp). However, for a divergence of 6.77% I find 0 records in the VCF file and 100% of the reference genome is missing/unknown from the BED file (19,699/19,699 bp). Hence, when exporting a multiple sequence alignment used for constructing a phylogenetic tree, genome *SH14-012* is marked as consisting entirely of missing/unknown nucleotides (indicated with an `N` character in the alignment). When I run iqtree [17] on this multiple sequence alignment to construct a maximum-likelihood tree I see the message "Sequence SH14-012 contains only gaps or missing data". Hence, I conclude that this is the reason why I cannot construct phylogenetic trees for **assemblies** for divergences greater than 4.20%. This is also reflected by the large drop in F1 score for higher divergences for assemblies (Figure 4.5).

To investigate why I got 0 records in the VCF file for a divergence of 6.77%, I examined the output of minimap2 [39], which is used for alignment prior to identifying nucleotide variants. I ran minimap2 for genomic sample *SH14-012* (using the command `minimap2 -t 1 -a -x asm5 reference.fasta.mmi SH14-012.fasta.gz`), which produces a SAM file [98] storing the genome alignment. Part of the SAM file format is an encoding of matches/mismatches/insertions/deletions between genomes encoded in a string using the CIGAR format [98]. For *SH14-012* and the above parameters for minimap2, the output SAM file has no data in the CIGAR string field. However, if I adjust the minimap2 parameter `-x asm5` to `-x asm10`, I do find a valid CIGAR string entry. The parameter `asm10` impacts the sequence divergence for mapping (from the minimap2 usage statement: *"asm5/asm10/asm20 - asm-to-ref mapping, for 0.1/1/5% sequence divergence"*). Hence, I conclude that the parameter `-x asm5` in minimap2, used by default in my analysis pipeline, is the cause of failure for higher-divergence datasets.

4.2 SARS-CoV-2 data analysis

4.2.1 Methods

Data preparation

In order to test out the applicability of this software to real-world data, I evaluated indexing and querying a collection of SARS-CoV-2 genomes. SARS-CoV-2 is an RNA virus with a genome length of $\sim$ 30 kbp. I downloaded a set of 1,095,217 publicly available genomes from GenBank (and processed by Nextstrain [3]) as of August 12, 2021 (`https://docs.nextstrain.org/projects/ncov/en/latest/`

`reference/remote_inputs.html#summary-of-available-genbank-open-files`). I first removed any genomes where the number of ambiguous or undetermined bases was $\geq$ 50% of the length of the reference genome (NC_045512 with length 29,903 bp). I then selected a subset of 100,000 random genomes from this collection to act as a pool of genomes from which I draw subsets for testing.

Data analysis and indexing

In order to evaluate the run time and performance of indexing and querying on increasing numbers of genomes, I divided the 100,000 test genomes into cases of 10, 20, 50, 100, 200, 500, 1,000, 2,000, 5,000, 10,000, and 20,000 genomes. These were divided up such that each larger collection of genomes contains all the previous smaller collections (*e.g.,* the case of 20 genomes contains the same genomes from the case of 10 plus 10 additional genomes). I ran GDI to process and load these sets of genomes into an index. The processing by GDI can be divided up into three different stages: **Data analysis**, **Indexing**, and **Clustering** (Figure 1.1 and Figure 3.1). These are referred to in the evaluation results as **Analysis**, **Index**, and **Tree**, respectively.

The **Analysis** stage consists of alignment and identification of small variants (single-nucleotide variants and indels) of the SARS-CoV-2 genomes with respect to the reference genome (NC_045512). This also includes steps to construct sourmash sketches using kmer sizes of $k \in \{31, 51, 71\}$. This stage uses parallel processing provided by Snakemake during the execution of the DAG (defining portions of the data analysis workflow that can be executed in parallel, see Section 3.1.2). The

Index stage consists of constructing an inverted index of the individual nucleotide features. This stage uses parallel processing while pre-processing VCF and BED files and loading the data into a DataFrame prior to constructing an inverted index (Section 3.2.4). The **Tree** stage consists of constructing a multiple sequence alignment of the identified variants followed by building a phylogenetic tree using iqtree [17] using the `--fast` option and the *GTR+F+R4* evolutionary model. This stage uses parallel processing by passing appropriate arguments to iqtree (`--threads-max` and `-T AUTO`, see Section 3.4.3). Due to the time it takes to build a phylogenetic tree, I only ran the Tree stage for up to 500 genomes and skipped this stage for the cases with more than 500 genomes. I used 32 processing cores to run each of the Analysis, Index, and Tree stages and I recorded the run time, peak memory, and disk usage (repeated 3 times). I also calculated the **Total** time, which is the sum of the previous three stages.

Querying

To evaluate querying performance, for each of these indexes I recorded the run time (using the Python `timeit` function) for a number of different query operations in the **Python API** (repeated 10 times). These query operations include: `q.isa()`, `q AND r`, `q.summary()`, `q.join()`, `q.features_summary()`, `q.hasa()`, `q.toframe()`, `q.features_comparison()` (see Table 3.2 for a description of these operations). I did not record memory usage in this case as it was difficult to determine how to separate out the memory usage of my query operation from the memory usage of the rest of the application (*i.e.,* it is difficult to measure the memory usage of calling a single Python function). To evaluate querying performance using the **command-line interface**, I recorded the run time and memory usage of a number of operations. These operations include `query hasa`, `query isa`, `query --summary`, `query --features-summary`, and `list samples` (see Table 3.2 for a description of these operations). Here, I could record the memory usage, which is defined as the peak memory used by the entire Python application. In all cases, query operations use only a single processing core (they are not multi-process or multi-threaded).

Clustering and phylogenetic analysis

To evaluate how well a phylogenetic tree (or hierarchical cluster tree) constructed using the nucleotide variants indexed by my software groups related genomes into clades, I compared these clades to independently defined Pangolin lineages [41], which themselves are defined using phylogenetics and machine learning models [6]. The phylogenetic tree was constructed using 500 samples and a maximum likelihood approach where the whole-genome alignment of the samples is exported and used to construct a phylogenetic tree with iqtree [17] (details in Section 3.4.1). The hierarchical cluster was defined using a kmer-based MinHash method for $k \in \{31, 51, 71\}$ [50; 54] (details in Section 3.4.2). Pangolin lineages were identified using the pangolin software [41] (version *3.1.20*, pangoLEARN *2022-02-28*, pangodesignation *v1.2.127*), which was run on the SARS-CoV-2 genome assemblies. I also included a comparison to a phylogenetic tree constructed from the Augur pipeline—a phylogenetic pipeline that is part of Nextstrain [3] and which operates independently of GDI by constructing a multiple sequence alignment (using MAFFT [110]) and a maximum-likelihood tree (using iqtree [17]). The tree constructed using Augur is used to provide an independently constructed phylogenetic tree for comparison to GDI.

The hierarchical structure of Pangolin lineages are recorded in the lineage names using a dotted-decimal notation [41] (e.g,. lineage *B.1.1* is part of *B.1*, which is part of lineage *B*). I split these lineages based on their hierarchical levels (the alphanumeric portion in between the period `.`) and assign genomes to every level. That is, if genome X is part of lineage *B.1* I split this into two clusters—*B.1* and *B*—and I assign genome X to be a part of both clusters. Consequently, the clusters I define using these lineages are not mutually exclusive (a genome can and likely does belong to more than one cluster). An additional complexity in this process is that some lineages are aliases of others (*e.g., AY.1* is an alias of *B.1.617.2.1*). I solve this by first replacing each alias with the expanded version of a lineage code prior to splitting lineages (*e.g.,* I substitute *AY.1* with *B.1.617.2.1*). The pangolin lineage aliases are derived from the table located at `https://cov-lineages.org/lineage_list.html`.

In order to quantify how well a lineage-level (cluster) is accurately grouped together by a phylogenetic or hierarchical clustering tree, I assign scores to every lineage based on how well it forms a monophyletic group in the tree. The scores range from 0 to 1, with a score of 1 meaning that all samples form a monophyletic group. Scores of <1 mean that samples in a cluster do not form a monophyletic group, with scores closer to 0 being interpreted as mixing more samples from other clusters within the cluster-in-question. This scoring system is described in more detail in Section 3.4.4.

I evaluated each tree (whether a phylogenetic tree or hierarchical cluster tree) through this scoring system by scoring each lineage-level to produce a distribution of scores for each tree, one per lineage-level. I removed scores derived from lineage-levels consisting of only a single genome, as these will always be assigned the maximum score of 1 for every tree, thus I did not consider them informative for comparing different tree-building methods. Considering only lineage-levels with 2 or more genomes results in 41 lineage-levels and thus 41 different scores (numbers from 0 to 1) for each tree. I compared the distribution of these scores for each tree (Figure 4.17).

4.2.2 Data analysis and indexing

To evaluate the performance for data analysis and indexing, I constructed genomics indexes for batches of genomes up to 20,000 and measured the run time, peak memory, and disk usage when using 32 processing cores. The run time of indexing genomes is dominated by the run time used to build a phylogenetic tree (Figure 4.7, Type = *Include Tree*). With 500 genomes, this took 5.26 ± 0.41 hours out of the total run time of 5.35 ± 0.41 hours or 97% of the time (mean $\pm$ standard deviation over 3 trials). When ignoring building a phylogenetic tree (Figure 4.7, Type = *Exclude Tree*), the major contributor to the total run time was the *Analysis* stage, which for 20,000 genomes took 2.17 ± 0.00 hours out of a total of 2.55 ± 0.01 hours or 85% of the total (mean $\pm$ standard deviation over 3 trials).

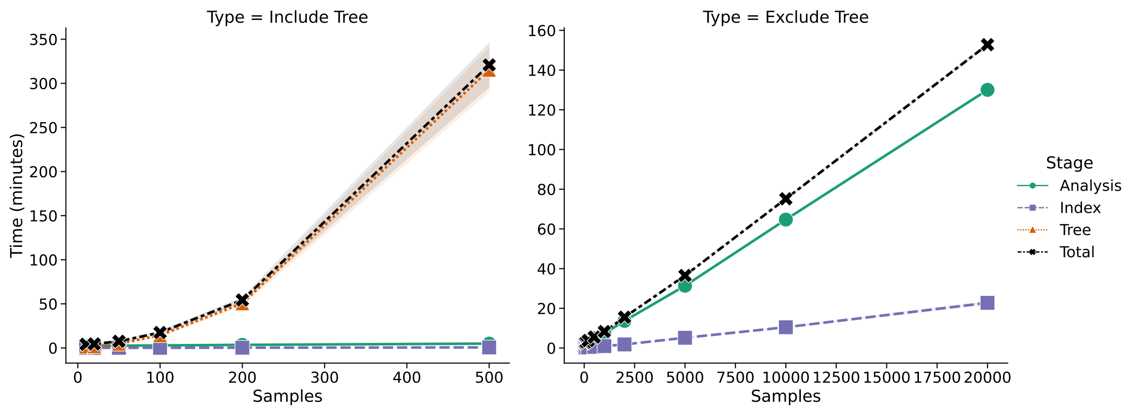


Figure 4.7: Time to index different numbers of SARS-CoV-2 genomes (using 32 processing cores). The time is divided into three different stages: **Analysis**, **Index**, and **Tree**. The left plot shows the time spent including building a phylogenetic tree. The right plot shows the time spent when building of a tree is skipped. The mean over 3 different trials is shown, with confidence intervals corresponding to ± one standard deviation from the mean (most are too small to see).

Unlike run time, the peak memory usage (Figure 4.8) is primarily driven by the peak memory of the *Index* stage. For 500 genomes, the peak memory usage occurred in the *Index* stage with 5.63 ± 0.01 GB and the next highest peak was in the *Analysis* stage with 1.90 ± 0.46 GB.

Interestingly, between 2,000 genomes and 5,000 genomes there was a large jump in the peak memory usage in the *Index* stage from 6.15 ± 0.01 GB to 23.6 ± 0.23 GB (Figure 4.8). One hypothesis for this jump is that I process genomes in batches during the *Index* stage to avoid constructing very large data structures (Section 3.2.4). The default batch size is 2,000 genomes, so moving from 2,000 genomes to 5,000 genomes would mean the *Index* stage goes from processing all genomes in one batch to requiring three batches. In order to confirm this hypothesis I re-ran the entire indexing process (starting from 2,000 genomes) with a batch size of

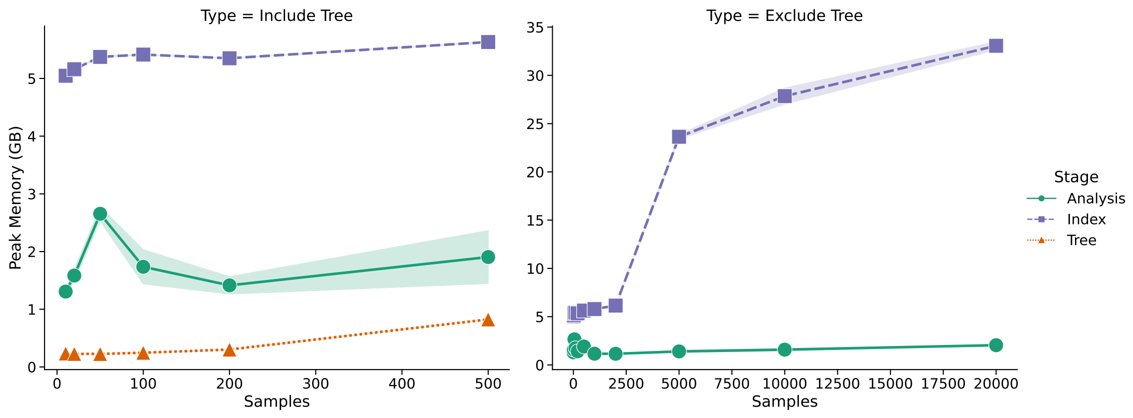


Figure 4.8: The peak memory used to index different numbers of SARS-CoV-2 genomes. The memory is divided into three different stages: **Analysis** (identifying nucleotide variants), **Index** (loading nucleotide variants into a database), and **Tree** (building a maximum-likelihood phylogenetic tree). The left plot shows the peak memory when including building a phylogenetic tree. The right plot shows the peak memory when building of a tree is skipped. The mean over 3 different trials is shown, with confidence intervals corresponding to ± one standard deviation from the mean.

10,000 (instead of 2,000) and compared the memory usage between both scenarios (Figure 4.9). As can be seen, moving from a batch size of 2,000 to 10,000 shifts the large jump in memory to between 10,000 and 20,000 genomes. Hence, I conclude that the batch size is the likely cause of this jump in memory.

The disk usage of indexing is primarily driven by the *Analysis* stage (Figure 4.10). For 500 genomes, the *Analysis* disk usage is 3.52 ± 0.00 GB out of a total of 3.53 ± 0.00 GB or 99.7% of the total. For 20,000 genomes, the *Analysis* disk usage is 7.10 ± 0.00 GB out of a total of 7.45 ± 0.00 GB, or 95% of the total. The *Index* disk usage for 20,000 genomes is only 0.34 ± 0.00 GB, or only 4.6% of the total. Hence, the vast majority of the disk space used for indexing is only temporary, as the files used by the *Analysis* stage are cleaned up after the full index has been

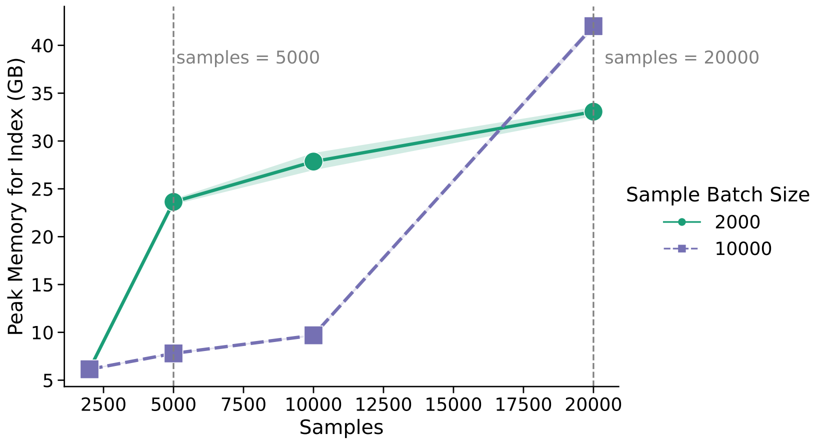


Figure 4.9: The peak memory used for the **Indexing** stage across sample batch sizes of 2,000 and 10,000. For the case of a batch size of 2,000, the large increase in memory occurs when indexing 5,000 samples (more than 2,000). For the case of a batch size of 10,000, the large increase in memory occurs when indexing 20,000 samples (more than 10,000 samples). The mean over 3 different trials is shown, with confidence intervals corresponding to ± one standard deviation from the mean.

constructed. Additionally, unlike for the peak memory, there are no obvious jumps or irregularities as the number of genomes increase.

4.2.3 Querying

In order to evaluate the performance when querying an index or summarizing information from the index, I chose a set of 7 different operations in the GDI Python API and used the Python `timeit` module to evaluate the run time over 10 different iterations. I also evaluate a subset of these 7 operations using the GDI command-line interface. All query operations use only a single processing core for execution.

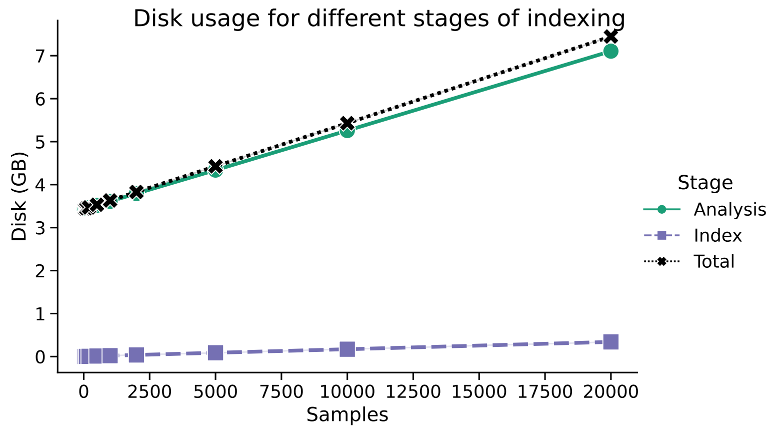


Figure 4.10: The disk space used to index different numbers of SARS-CoV-2 genomes. The disk space is divided into two different stages: **Analysis** (identifying nucleotide variants) and **Index** (loading nucleotide variants into the database). The mean over 3 different trials is shown, with confidence intervals corresponding to ± one standard deviation from the mean (these are too small to see on this figure).

Python API

Figure 4.11 shows the relationship between the number of genomic samples in the index and the run time. This is divided up into three different time scales: **Short**, **Medium**, and **Long**. In the **Short** time-scale the `q AND r` and `q.summary()` operations appear fairly constant across a large range of samples—taking only 0.024 ± 0.000 ms and 0.35 ± 0.01 ms respectively (mean ± standard deviation). Both of these operations depend primarily on set operations using Roaring bitmaps [104] and thus should be expected to be fairly fast. In particular, the sample identifiers are stored in a Roaring bitmap, which stores sets of 32-bit integers organized into blocks of 16-bits (the first 16-bits of an integer is used as a key to identify the block) [104]. Given that I am storing up to 20,000 samples where the identifiers are defined sequentially (from 1 to 20,000) then I am only ever manipulating a single roaring-bitmap block with all sample identifiers in it (16-bit blocks can store a

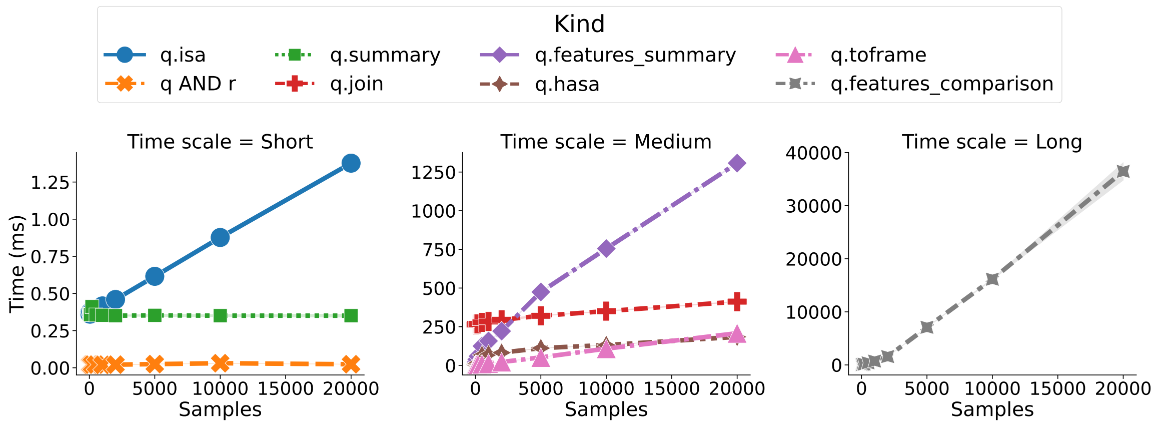


Figure 4.11: The time spent during different operations of the query API compared to the number of samples. The three plots compare query API calls, which use different lengths of time: **Short**, **Medium**, and **Long**. The mean over 10 different trials is shown, with confidence intervals corresponding to ± one standard deviation from the mean (confidence intervals are too small to see in these figures).

maximum of $2^{16} = 65,536$ unsigned integers). Hence, even with 20,000 samples I would not expect the time taken to perform set operations for Roaring bitmaps to change much, and it should only change as I get closer to 65,536 samples stored in an index. By contrast, the `q.isa()` operation requires connecting to the SQLite database and this is reflected in the increased time as the number of samples increase, reaching 1.38 ± 0.00 milliseconds for 20,000 samples (mean ± standard deviation).

For the **Medium** time-scale query operations, the largest-increasing operation is `q.features_summary()`, which takes $1,310 \pm 3.4$ ms for 20,000 samples. For the **Long** time-scale operations there is only one operation `q.features_comparison()`, which takes $36,500 \pm 1,700$ ms for 20,000 samples. Both `q.features_summary()` and `q.features_comparison()` involve summarizing information for all features found in all samples, and hence take much longer to execute.

In order to get a better idea on the query run time, I also compared the time to the number of genomic features. Genomic nucleotide features can be divided up into two classes: nucleotide variants (referred to as **known** features and shown on Figure 4.12) and missing/unknown features (referred to as **unknown** features and shown on Figure 4.13). Nucleotide variants consist of features corresponding to single or multiple nucleotide variants (*e.g.,* an A to G mutation at position 100) or insertions/deletions. The unknown/missing features consist of entries in my index representing samples where it is unknown what the nucleotide is on a particular region. The figures are divided into the same three time-scale of query operations: **Short**, **Medium**, and **Long**.

For the nucleotide variants (**known** features shown on Figure 4.12), we can see that most operations increase with increasing number of features except for `q AND r` and `q.summary()` . The operation `q AND r` consists of a logical operation on sets of samples (samples in query `q` and samples in query `r`), and so are not directly related to the number of features. The operation `q.summary()` prints summary statistics for a particular query and so is also unrelated to the number of features. Every other query increases with the number of features, with the greatest increases for each of the three time scales consisting of: `q.isa()` , `q.features_comparison()` , and `q.features_summary()` .

For the unknown/missing features (**unknown** features shown on Figure 4.13), we see that most operations appear fairly constant except for `q.features_summary()` and `q.hasa()` , which increase slightly with respect to the number of unknown/missing features. Neither of these operations depend on the number

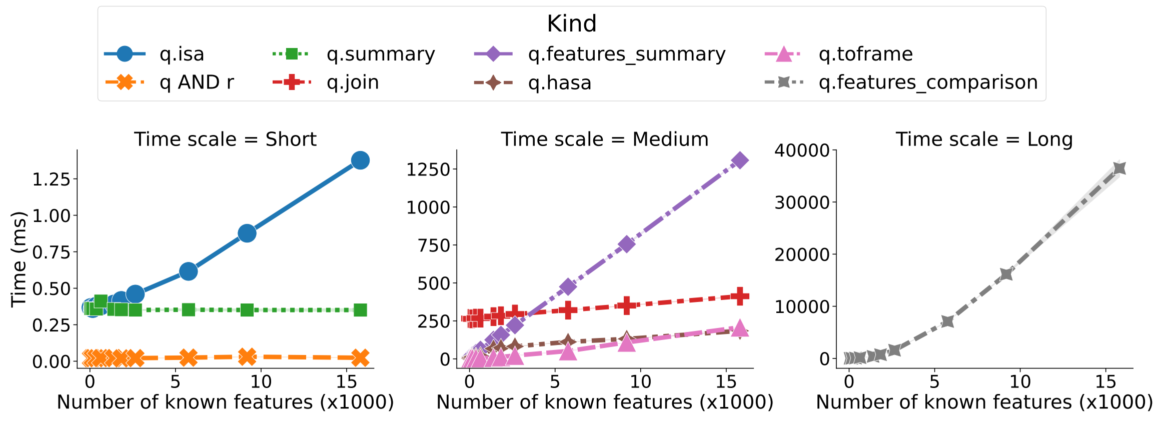


Figure 4.12: The time spent during different operations of the query API compared to the number of nucleotide variant features (excluding unknown features). The three plots compare query API calls using different lengths of time: **Short**, **Medium**, and **Long**. The mean over 10 different trials is shown, with confidence intervals corresponding to ± one standard deviation from the mean (confidence intervals are too small to see in these figures).

of unknown/missing features by default, but the increase could potentially be explained by a correlation between the number of nucleotide variants and unknown/missing features (that is increasing the number of samples increases both types of features, see Figure 4.14). Another interesting aspect is the large spike in run time at around 30,000 unknown/missing features (Figure 4.13). The number of unknown/missing features stored in my index is limited by the length of the reference genome (one unknown/missing feature entry per position on the reference genome), which for the NC_045512 reference genome is 29,903 bp. This means that the number of unknown/missing features becomes saturated at the reference genome length, which occurs at around 500 samples for this dataset (Figure 4.14). Any further increase in the number of genomic samples leads to an increase in the number of indexed nucleotide variants but no increase in the number of unknown/missing features. Since increases in the number of nucleotide variants lead

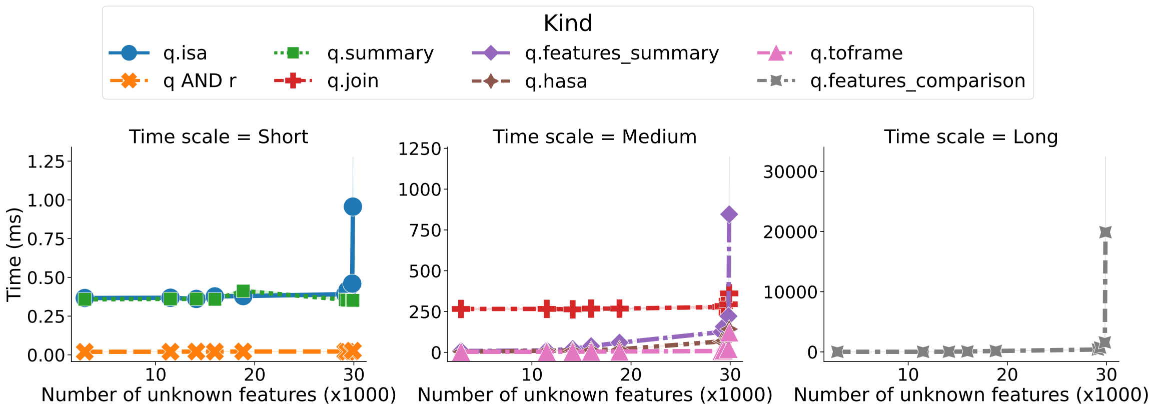


Figure 4.13: The time spent during different operations of the query API compared to the number of unknown/missing nucleotide features. The three plots compare query API calls using different lengths of time: **Short**, **Medium**, and **Long**. The mean over 10 different trials is shown, with confidence intervals corresponding to ± one standard deviation from the mean (confidence intervals are too small to see in these figures).

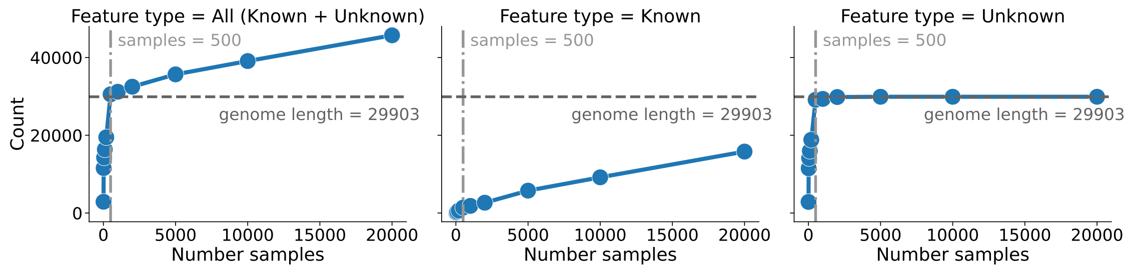


Figure 4.14: The number of features stored in the index compared to the number of samples for the SARS-CoV-2 dataset. Features are divided into two classes: the **Known** type, which includes single/multiple nucleotide variants and indels, as well as the **Unknown** features type, which corresponds to missing or unknown positions on the reference genome. The **All** features type includes the count of both unknown and known features. The number of unknown features is limited by the length of the reference genome (29,903 bp) since there can only exist one unknown/missing feature per position on the genome. For this dataset, the number of unknown features reaches nearly the genome length when indexing 500 samples.

to an increase in run time, this provides an explanation for why there is a large spike near 30,000 features for the unknown/missing features figure (Figure 4.13).

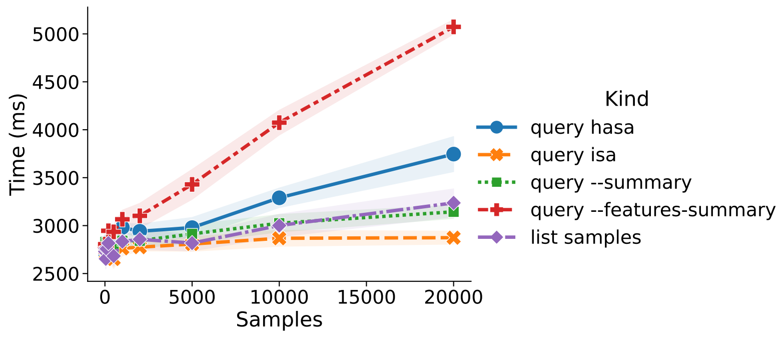


Figure 4.15: The run time of selected operations in the command-line interface (CLI) as the number of samples in the index increases. The mean over 10 different trials is shown, with confidence intervals corresponding to ± one standard deviation from the mean.

Command-line interface

In addition to examining the run time of the API, I also evaluated the run time of the command-line interface (CLI) (Figure 4.15). Not all operations available in the API are also available in the CLI (Table 3.2), so only a subset of equivalent operations are shown. The overall trend in increasing run time as the number of samples increases also holds for the CLI, but the plot starts with 2,500 milliseconds instead of 0 milliseconds. This reflects the overhead introduced by using the CLI to start up the application. For example, for two equivalent operations (`q.hasa()`) with 20,000 samples the API takes 184 ± 0.69 milliseconds, while the CLI takes $3,750 \pm 180$ milliseconds (mean ± standard deviation over 10 trials). This means the CLI is $\sim$ 20X slower than the API in this particular case due to the overhead of starting up the application.

In addition to the run time, I also evaluated the memory usage for the CLI (Figure 4.16). In general, there is an increasing memory usage for querying using the

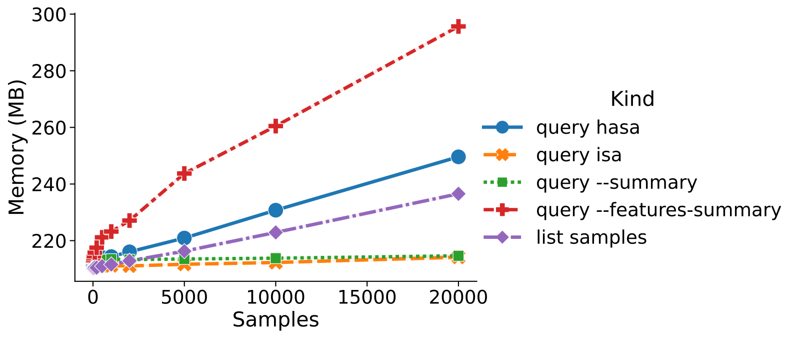


Figure 4.16: The peak memory usage of selected operations in the command-line interface (CLI) as the number of samples in the index increases. The mean over 10 different trials is shown, with confidence intervals corresponding to ± one standard deviation from the mean (confidence intervals are too small to be seen in this figure).

CLI as the number of samples increases. For a case of 20,000 samples, the highest memory used is by the `query --features-summary` command with 296 ± 0.33 MB (mean ± standard deviation over 10 trials). Measuring the memory of the individual GDI Python API commands is more difficult and thus was not performed for these evaluations.

4.2.4 Phylogenetics and clustering

As a final set of evaluations of the SARS-CoV-2 dataset, I examined how well different tree-building methods group genomes into clades corresponding to independently defined clusters. These clusters are defined as the different Pangolin lineages [41]. For 500 samples, I had a total of 41 different lineages (or lineage-levels), each of which was assigned a numerical score from 0 to 1. Figure 4.17 shows the distribution of these scores. Here, it can be seen that the maximum-likelihood trees (constructed both from Augur as well as GDI) both have the majority of the lineages lying within monophyletic or near-monophyletic clades (scores all are equal to or close to 1.0). The median score for both my internal maximum-likelihood approach as well as the Augur pipeline is 1.0. This is in contrast to the kmer trees, which all tend to have scores closer to 0 (the highest median is for $k = 51$ with a median score of 0.058). Hence, the topology of the tree for both maximum-likelihood approaches are a much better match to the externally defined (Pangolin) clusters than for any kmer-based clustering (with $k \in \{31, 51, 71\}$). Figures C.1, C.2, C.3, C.4, and C.5 show the dendrograms alongside a heatmap of genomes and lineages for the Augur pipeline, GDI-constructed maximum-likelihood tree, and the three kmer trees respectively.

4.3 Read data

4.3.1 Methods

To evaluate the analysis and indexing of features derived directly from sequence reads, I made use of a standard benchmarking dataset [125] consisting of four collections of WGS reads derived from different bacterial organisms involved in real-world infectious disease outbreaks. These consist of 22 *Campylobacter jejuni*, 9 *Escherichia coli*, 31 *Listeria monocytogenes*, and 23 *Salmonella enterica* genomes. Each of the four different datasets from the different organisms can be further subdivided into a collection of genomes derived from bacteria that were confirmed to be epidemiologically related to a single outbreak (and so should be expected to be similar to each other) as well as genomes derived from bacteria that are more dis-

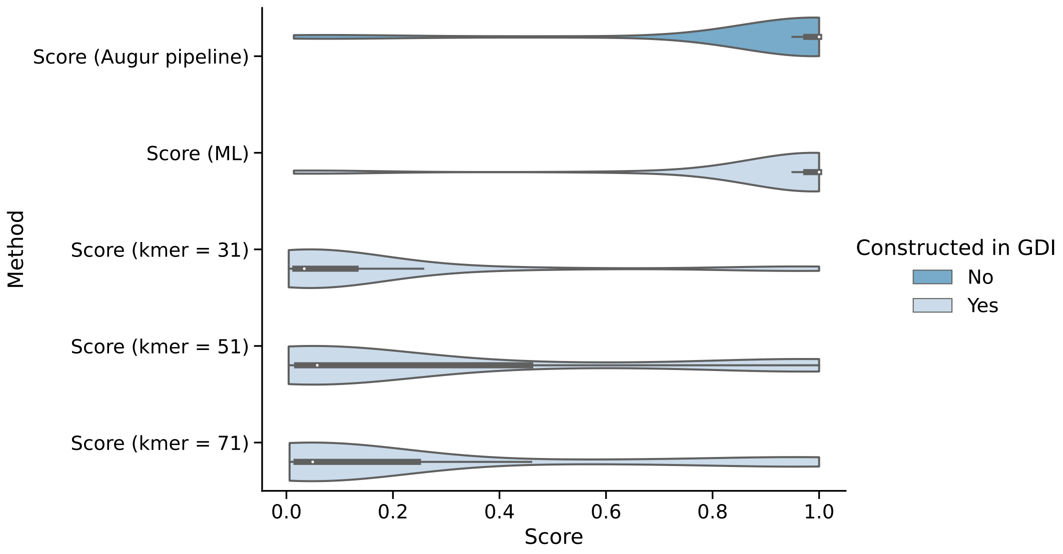


Figure 4.17: A comparison of the distribution of cluster scores from different methods of clustering genomes. Each of the 5 categories shows the distribution of all 41 lineages with assigned scores as a violin plot scaled to have identical areas. Each violin plot has an internal box-and-whisker plot. Scores range from 0 to 1, with 1 representing a situation where all genomes form a monophyletic clade. The **Augur pipeline** category shows the distribution of scores generated from constructing a multiple sequence alignment and a maximum-likelihood tree using the Augur pipeline [3] instead of GDI. The other 4 methods show the scores for a phylogenetic tree or hierarchical clusters constructed using GDI.

tantly related (the outgroup). For each organism, an additional reference genome is provided or recommended. I chose this dataset as one of the goals of this evaluation is to examine the differences in clustering genomes between a maximum-likelihood approach and a kmer approach as implemented in GDI. Specifically, I will show that clustering by both kmers and nucleotide variants are necessary, depending on the resolution desired for clustering data. Table 4.2 summarizes the dataset information.

In order to compare clustering of the genomes using kmers vs. a maximum-likelihood phylogenetic approach, I also processed and inserted the data for all four datasets into the same GDI index. I then constructed maximum-likelihood phylogenetic trees for each of the four datasets separately using iqtree [17] with the `--fast` option and the *GTR+F+R4* evolutionary model. I also constructed a single kmer-based tree (a hierarchical cluster tree using single-linkage clustering from pairwise distances calculated by sourmash [54] using a kmer size of $k = 71$ and a kmer scale factor of 1000).

In order to evaluate the resource usage of indexing this data, each of the four datasets was divided into a number of cases by selecting a random subset of genomic samples from each dataset corresponding to 4, 8, 16, and the full number of samples. For the *1405WAEXK-1 (Escherichia coli)* dataset, the case of 16 samples was ignored as there are only 9 samples in this dataset. In each case, the genomes were processed and inserted into a separate index for each individual dataset (so there were four GDI indexes in total). The run time, memory, and disk usage was recorded for each of these test cases over 3 separate iterations. The disk usage for the **Index** stage corresponds to the disk usage of also storing a phylogenetic tree (which is part of the GDI index and stored in the relational database). I do not divide up the disk usage of the GDI index into both an **Index** (without tree) and a **Tree** stage since the size of the phylogenetic tree (Newick file) is negligible when compared to the size of the rest of the index.

For each dataset and testing scenario, I ran the analysis pipeline to identify nucleotide variants from reads along with a sourmash sketch consisting of $k \in \{31, 51, 71\}$ and inserted the data into a GDI index. I used 4 cores for the data analysis, which was chosen to be equal to the smallest sample size (4 samples)

Dataset	Organism	Reference genome	Reference length (bp)	Number genomes (Outbreak + Outgroup)	Cases (number samples)
0810PADBR-1	*Campylobacter jejuni*	ASM187918v2	1,634,890	22 (14 + 8)	$\{4, 8, 16, 22\}$
1405WAEXK-1	*Escherichia coli*	Ec2011C-3609	5,412,686	9 (3 + 6)	$\{4, 8, 9\}$
1408MLGX6-3WGS	*Listeria monocytogenes*	ASM104771v2	2,939,733	31 (28 + 3)	$\{4, 8, 16, 31\}$
1203NYJAP-1 - Tuna Scrape Outbreak	*Salmonella enterica*	ASM43941v1	4,808,805	23 (18 + 5)	$\{4, 8, 16, 23\}$

Table 4.2: Dataset and organisms used for benchmarking with sequence reads

to make sure I'm always using the maximum number of processing cores assigned to the application no matter how many samples I am processing. All three of the **Analysis**, **Index**, and **Tree** stages perform parallel processing of the data (Section 3.1.2, Section 3.2.4, and Section 3.4.3 respectively). If I were to increase the maximum allowable cores from 4 to 8, for example, then when running GDI on only 4 samples, there would still be 4 unused processing cores that could have been occupied by work on processing additional samples (as the maximum allowable cores to use is 8). Hence, I set the maximum allowable cores to 4 so that there are no unused processing cores for any case I evaluate. This is to avoid any biases in my run time (or memory usage) that could come about due to unoccupied cores.

4.3.2 Running time

Figure 4.18 shows the mean time across each of the different sample scenarios for each of the four separate datasets, divided up into the separate data processing stages of my software. For 8 samples (the largest number of samples with timing information for all 4 datasets), the dataset that takes the longest total time is *Campylobacter jejuni* (22.5 ± 0.9 minutes, mean $\pm$ standard deviation). This is primarily driven by the time taken for the **Analysis** stage, which is 97% of the total time (21.9 ± 0.09 minutes). Surprisingly, while *Campylobacter jejuni* takes the most time overall, for the **Index** and **Tree** stage it takes the least amount of time with 0.346 ± 0.008 minutes and 0.320 ± 0.024 minutes respectively. Compare this to the dataset that takes the most amount of time for the **Index** and **Tree** stages—*Escherichia coli*—which takes 3.51 ± 0.21 minutes for the **Index** stage and 0.493 ± 0.018 minutes for the **Tree** stage. This is $\sim$ 10X longer for the **Index** stage when compared to *Campylobacter jejuni* for 8 samples.

One hypothesis for why the **Index** stage has such a dramatically different order of running time when compared to the **Analysis** stage is related to the number of genomic features identified during the **Analysis** stage. The **Index** stage consists of constructing an inverted index, which requires constructing a large table consisting of samples and collections of features contained in each sample and "inverting" this data structure (to construct a table of features and collections of samples containing those features). This means that we should expect run time (and memory usage) to depend both on the number of samples as well as number of features contained in each dataset. Figure 4.19 shows the relationship between samples and fea-

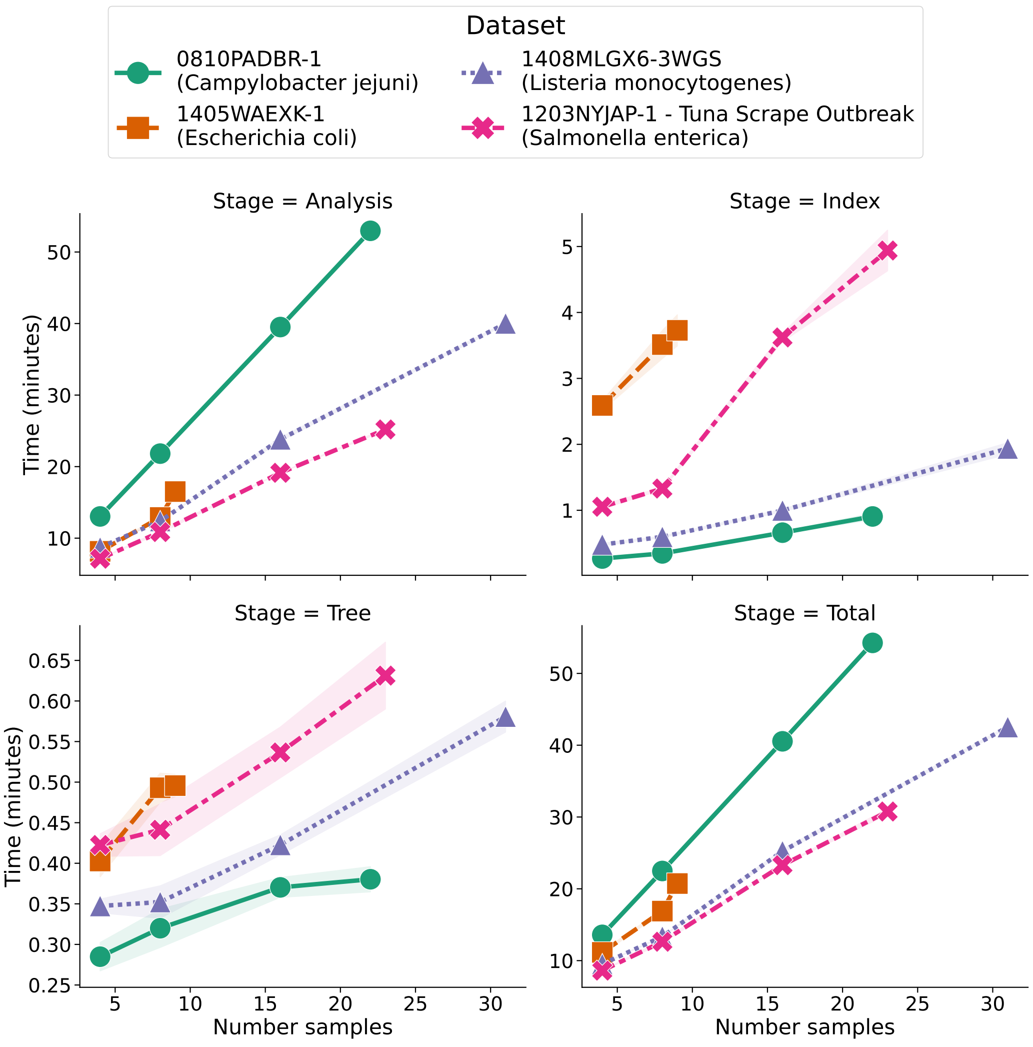


Figure 4.18: The running time for processing the read data from a set of four different organisms over a number of sample sizes and when processed using a maximum of 4 processing cores. Each of the four different figures shows the running time of one of the three different stages of GDI (**Analysis**, **Indexing**, **Tree**) plus the sum of the times over all stages (**Total**). The x-axis shows the number of genomic samples being indexed, which are divided up among the four different datasets (one dataset per organism). The time for indexing data from each dataset is divided into four different sample sizes: 4, 8, 16, and the full dataset size. The case of *Escherichia coli* lacks a data point for 16 samples as there are only 9 samples in this dataset. Each plot shows the mean over 3 trials with the confidence interval corresponding to ± one standard deviation from the mean.

tures for each dataset. From this figure we can see that *Escherichia coli* has by far the largest number of features in the case of 8 samples ($549,134$ features), while *Campylobacter jejuni* has the fewest ($44,494$). Hence, this is one possible explanation for why there is this inversion of running time for each dataset between the **Analysis** and **Index** stages. More detailed analysis (with a greater number of test cases) would have to be performed to confirm this hypothesis.

The relationship between the number of features and run time could also explain why the order of running time for the **Tree** stage matches the order for the **Index** stage (with *Escherichia coli* and *Salmonella enterica* taking the most amount of time, Figure 4.18). I would expect the time spent building a phylogenetic tree to depend on the number of features (amount of variation) between genomes, which is highest for *Escherichia coli* and *Salmonella enterica*. However, I would also expect the run time of tree-building to depend on the length of the reference genome used in the multiple sequence alignment (which is also highest for *Escherichia coli* and *Salmonella enterica*, Table 4.2). An additional factor that could play a role here is the number of unknown/missing features, which is highest for *Escherichia coli* (Figure 4.19). Missing data in a multiple sequence alignment leads to collections of trees with identical likelihood scores (called terraces, [18; 17]) which could have an impact on the time needed by iqtree. However, determining which of these factors plays a role in the run time of building a tree would require additional testing.

As a general trend, the running time increases with the number of samples (Figure 4.18) and appears to be ordered from the longest stage to the shortest as **Analysis**, **Index**, and **Tree**. One example is the running time for *Listeria monocyto-*

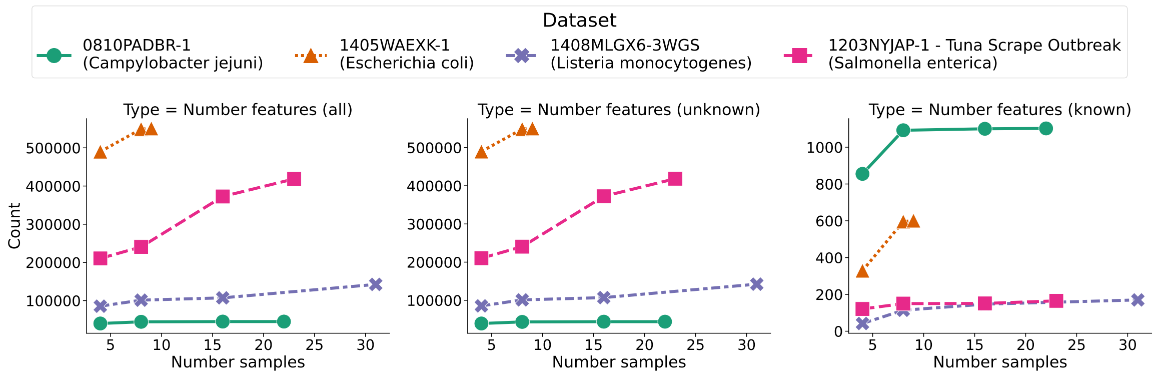


Figure 4.19: The relationship between the number of samples in a dataset and the number of features stored in the inverted index for this dataset. The three figures correspond to the case of counting all features (**all**), counting only unknown/missing features (**unknown**), and counting known features (*i.e.*, SNVs or indels, **known**). Each of the three datasets are shown as separate colours/data point styles on the figures. The features from each dataset are divided into four different sample sizes: 4, 8, 16, and the full dataset size. The case of *Escherichia coli* lacks a data point for 16 samples as there are only 9 samples in this dataset.

genes (the dataset with the highest number of genomes at 31), which is 40.0 ± 0.1, minutes, 1.94 ± 0.09 minutes, and 0.581 ± 0.019 minutes for the **Analysis**, **Index**, and **Tree** stages respectively. However, as observed in the *SARS CoV-2 data analysis* section, constructing a phylogenetic tree dominates the overall run time for a large number of samples. I would expect the read datasets to follow a similar trend, which is likely not observed due to the lower number of samples being indexed (31 samples for the reads compared to 500 samples for SARS-CoV-2 and building a phylogenetic tree).

4.3.3 Memory

Figure 4.20 shows the peak memory usage for each of the four separate datasets over the differing number of samples and divided up into the separate data processing stages. For 8 samples during the **Analysis** stage, the peak memory is the largest for the *Campylobacter jejuni* dataset (10.0 ± 0.3 GB) and smallest for the *Escherichia coli* dataset (4.87 ± 0.01 GB). The peak memory for the **Analysis** stage for 8 samples here is also the peak memory across all stages. The stage with the second largest peak memory for 8 samples is the **Index** stage, with 3.33 ± 0.00 GB for *Escherichia coli* and 1.09 ± 0.01 GB for *Campylobacter jejuni.* Similar to the *running time* section, *Campylobacter jejuni* goes from the largest memory usage in the **Analysis** stage to the smallest memory usage for the **Index** stage while *Escherichia coli* moves to the highest spot for the **Index** stage. This is likely due to a similar mechanism as for running time, where the greater number of features for *Escherichia coli* (Figure 4.19) leads to a greater amount of memory used when constructing the inverted index.

Similar to the running time scenario, the general order of memory usage among the three stages appears to be (from highest to lowest): **Analysis**, **Index**, and **Tree**. For the dataset with the most samples (*Listeria monocytogenes* with 31 samples) this would be 9.27 ± 0.36 GB, 2.54 ± 0.28 GB, and 0.927 ± 0.000 GB respectively.

4.3.4 Disk usage

Figure 4.21 shows the disk usage for each of the four separate datasets, divided up into the **Analysis** and **Index** stages. I did not include a separate **Tree** stage as the size of each phylogenetic tree (a Newick file) is negligible when compared to the size of the GDI index (0.004 MB compared to 38.1 MB or 0.01% of the index size for the case of 31 samples for the *Listeria monocytogenes* dataset, which would have

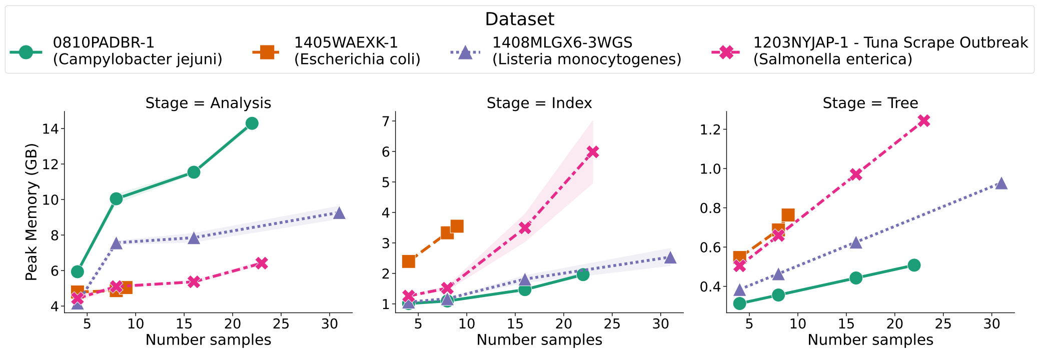


Figure 4.20: The peak memory usage for processing the read data from a set of four different organisms over a number of sample sizes and when processed using a maximum of 4 processing cores. Each of the three different figures shows the peak memory of one of the three different stages of indexing (**Analysis**, **Indexing**, **Tree**). The x-axis shows the number of genomic samples being indexed, which are divided up among the four different datasets (one dataset per organism). The peak memory for indexing data from each dataset is divided into four different sample sizes: 4, 8, 16, and the full dataset size. The case of *Escherichia coli* lacks a data point for 16 samples as there are only 9 samples in this dataset. Each plot shows the mean peak memory over 3 trials with the confidence interval corresponding to $\pm$ one standard deviation from the mean.

the largest phylogenetic tree).

Of the two stages, the **Analysis** stage takes up by far the largest amount of disk space (an **Analysis** size of $6,970 \pm 0.00$ MB compared to the **Index** size of 18.5 ± 0.01 MB for 8 samples of the *Campylobacter jejuni* dataset). However, the **Analysis** stage is temporary, needed only for converting the genomic sequence reads to genomic features prior to loading into the index. Out of the four datasets, *Campylobacter jejuni* requires the most amount of disk space for the **Analysis** stage but is closer to requiring the least amount of disk space for the **Index** stage (it requires the least amount of space for 4 samples, but for ≥ 8 samples the *Listeria monocytogenes* dataset requires the least disk space for the **Index** stage). This difference in disk space between the **Analysis** and **Index** stages is likely due to the same cause

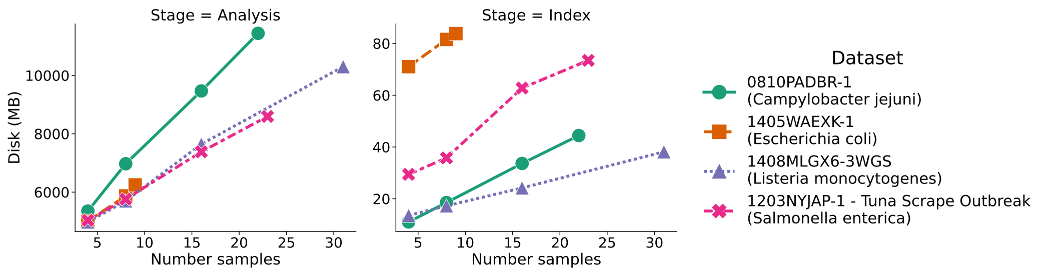


Figure 4.21: The total disk usage for processing the read data from a set of four different organisms over a number of sample sizes and when processed using a maximum of 4 processing cores. The two different figures show the disk usage over two of the indexing stages: **Analysis** and **Indexing**. The x-axis shows the number of genomic samples being indexed, which are divided up among the four different datasets (one dataset per organism). The disk usage for indexing data from each dataset is divided into four different sample sizes: 4, 8, 16, and the full dataset size. The case of *Escherichia coli* lacks a data point for 16 samples as there are only 9 samples in this dataset. Each plot shows the mean disk usage over 3 trials with the confidence interval corresponding to ± one standard deviation from the mean (intervals are too small to see).

as described in the *running time* section—mainly that *Campylobacter jejuni* consists of the fewest number of features overall across all sample sizes and so the GDI index is the smallest (Figure 4.19). The dataset with the largest disk usage for the **Index** stage is *Escherichia coli* with 81.5 ± 0.04 MB for 8 samples. This also mirrors what is observed in the *running time* section.

4.3.5 Clustering

Each organism's dataset is divided into a group of genomes from bacteria confirmed to be epidemiologically related alongside a number of outgroup genomes (Table 4.2). The genomes that are epidemiologically related should be expected to all cluster together based on shared genomic content as there has not been enough time for any individual bacterial genome to evolve significantly from the others to contribute to a larger genetic distance. This is confirmed by the existing publication on this data [125], which presents phylogenetic analysis to confirm that the outbreak-derived genomes all cluster together on a phylogenetic tree (and form a monophyletic group). I use this existing information to evaluate how well either the phylogenetic tree-building or kmer clustering mechanisms I included in GDI perform at clustering related genomes. To measure this, I construct a tree which clusters the genomes using either a maximum-likelhood (ML) method (from nucleotide features) or single-linkage method (from kmers). I evaluated each method based on whether or not it is able to group the outbreak-related isolates into a monophyletic clade. Results are shown in Table 4.3 with the trees shown in Figures D.1,D.2,D.3,D.4.

The maximum-likelihood method outperforms the kmer-based method at grouping outbreak-related genomes into monophyletic clades, with 4/4 of the organism datasets grouping outbreak-related genomes into a monophyletic clade for the ML approach compared to 1/4 for the kmer approach (Table 4.3). This suggests that a maximum-likelihood approach using nucleotide variants should be preferred when clustering or performing phylogenetic analysis of closely related genomes. However, one downside to this approach is that it relies on the choice of a reference genome with which to align sequence reads (Figure 3.2), and cannot be used to construct phylogenetic trees for more distantly related genomes (*e.g.,* across different organisms)—at least not without significant modifications, such as aligning key genes in common among all organisms rather than whole genomes. As these modifications for constructing maximum-likelihood phylogenies of distantly related

	Number in		Monophyletic	
Organism	Outbreak	Outgroup	ML	Kmer
Campylobacter jejuni	14	8	✓	
Escherichia coli	3	6	✓	✓
Listeria monocytogenes	28	3	✓	
Salmonella enterica	18	5	✓	

Table 4.3: Ability of a particular clustering method to group outbreak-related isolates into a monophyletic clade.

organisms are not implemented in my software, I do not evaluate them.

To evaluate clustering a more diverse set of organisms, I also tested clustering genomes using a kmer approach. In this approach, all organisms were grouped into their own separate clades in the overall tree (Figure 4.22). Hence, while a kmer approach is not as accurate for closely related genomes (Table 4.3), it can be used to cluster genomic samples which are more distantly related. This demonstrates that both a maximum-likelihood and kmer based approach are useful depending on the level of resolution at which you wish to examine the data.

4.4 Existing software

GDI fits within a collection of other existing bioinformatics software for either indexing genomes by features or storing genomic features or metadata about genomes in a database. I compare and contrast a collection of this software to GDI (Table 4.4). I divide this comparison up along four broad dimensions: **Usage**, **Index-**

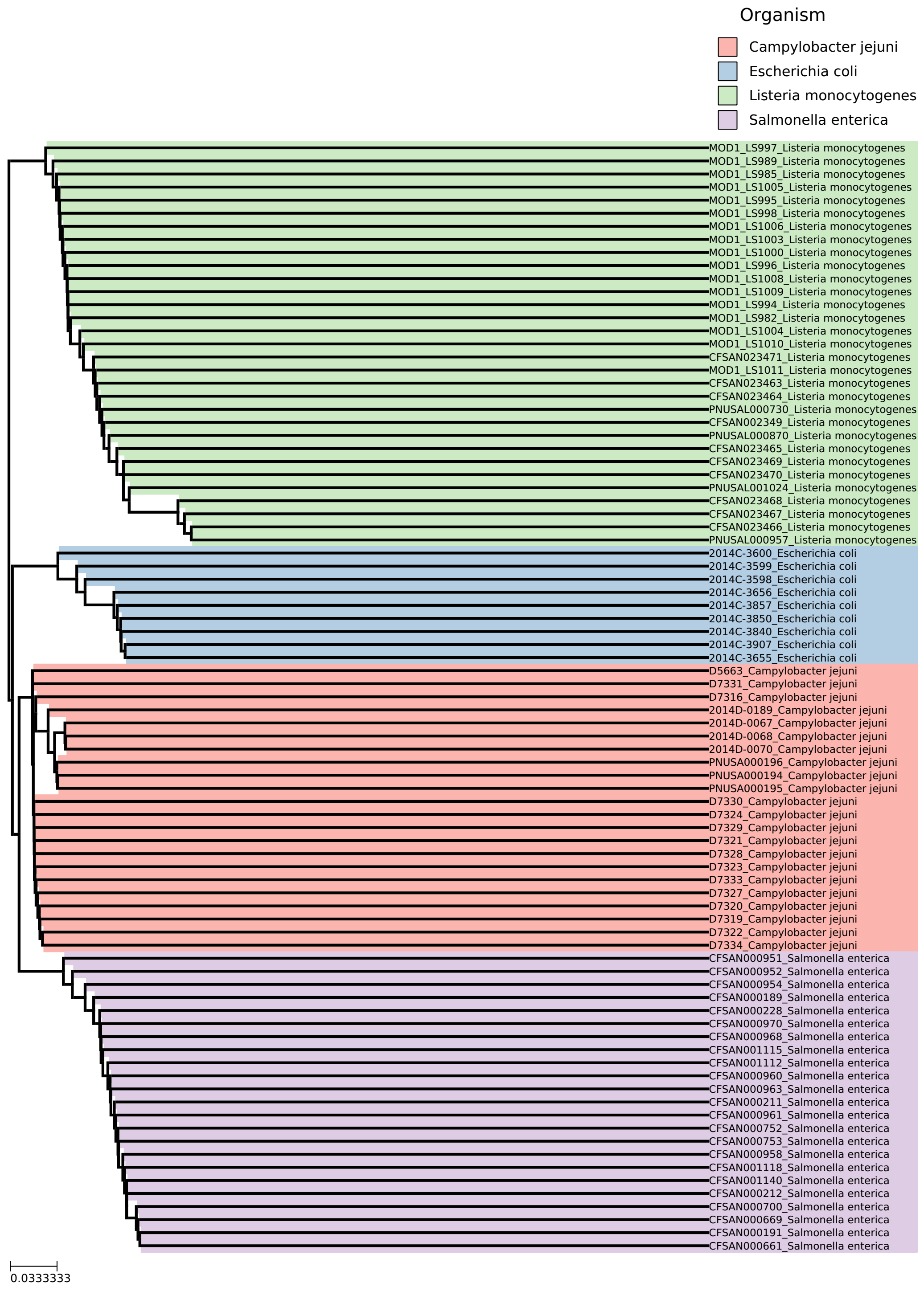

Organism
Campylobacter jejuni
Escherichia coli
Listeria monocytogenes
Salmonella enterica
MOD1_LS997_Listeria monocytogenes
MOD1_LS989_Listeria monocytogenes
MOD1_LS985_Listeria monocytogenes
MOD1_LS1005_Listeria monocytogenes
MOD1_LS995_Listeria monocytogenes
MOD1_LS998_Listeria monocytogenes
MOD1_LS1006_Listeria monocytogenes
MOD1_LS1003_Listeria monocytogenes
MOD1_LS1000_Listeria monocytogenes
MOD1_LS996_Listeria monocytogenes
MOD1_LS1008_Listeria monocytogenes
MOD1_LS1009_Listeria monocytogenes
MOD1_LS994_Listeria monocytogenes
MOD1_LS982_Listeria monocytogenes
MOD1_LS1004_Listeria monocytogenes
MOD1_LS1010_Listeria monocytogenes
CFSAN023471_Listeria monocytogenes
MOD1_LS1011_Listeria monocytogenes
CFSAN023463_Listeria monocytogenes
CFSAN023464_Listeria monocytogenes
PNUSAL000730_Listeria monocytogenes
CFSAN002349_Listeria monocytogenes
PNUSAL000870_Listeria monocytogenes
CFSAN023465_Listeria monocytogenes
CFSAN023469_Listeria monocytogenes
CFSAN023470_Listeria monocytogenes
PNUSAL001024_Listeria monocytogenes
CFSAN023468_Listeria monocytogenes
CFSAN023467_Listeria monocytogenes
CFSAN023466_Listeria monocytogenes
PNUSAL000957_Listeria monocytogenes
2014C-3600_Escherichia coli
2014C-3599_Escherichia coli
2014C-3598_Escherichia coli
2014C-3656_Escherichia coli
2014C-3857_Escherichia coli
2014C-3850_Escherichia coli
2014C-3840_Escherichia coli
2014C-3907_Escherichia coli
2014C-3655_Escherichia coli
D5663_Campylobacter jejuni
D7331_Campylobacter jejuni
D7316_Campylobacter jejuni
2014D-0189_Campylobacter jejuni
2014D-0067_Campylobacter jejuni
2014D-0068_Campylobacter jejuni
2014D-0070_Campylobacter jejuni
PNUSA000196_Campylobacter jejuni
PNUSA000194_Campylobacter jejuni
PNUSA000195_Campylobacter jejuni
D7330_Campylobacter jejuni
D7324_Campylobacter jejuni
D7329_Campylobacter jejuni
D7321_Campylobacter jejuni
D7328_Campylobacter jejuni
D7323_Campylobacter jejuni
D7333_Campylobacter jejuni
D7327_Campylobacter jejuni
D7320_Campylobacter jejuni
D7319_Campylobacter jejuni
D7322_Campylobacter jejuni
D7334_Campylobacter jejuni
CFSAN000951_Salmonella enterica
CFSAN000952_Salmonella enterica
CFSAN000954_Salmonella enterica
CFSAN000189_Salmonella enterica
CFSAN000228_Salmonella enterica
CFSAN000970_Salmonella enterica
CFSAN000968_Salmonella enterica
CFSAN001115_Salmonella enterica
CFSAN001112_Salmonella enterica
CFSAN000960_Salmonella enterica
CFSAN000963_Salmonella enterica
CFSAN000211_Salmonella enterica
CFSAN000961_Salmonella enterica
CFSAN000752_Salmonella enterica
CFSAN000753_Salmonella enterica
CFSAN000958_Salmonella enterica
CFSAN001118_Salmonella enterica
CFSAN001140_Salmonella enterica
CFSAN000212_Salmonella enterica
CFSAN000700_Salmonella enterica
CFSAN000669_Salmonella enterica
CFSAN000191_Salmonella enterica
CFSAN000661_Salmonella enterica
0.0333333

Figure 4.22: The kmer-based tree clustering genomes across four different organisms (shown in the legend at the top). Each branch leading to a leaf in the tree is coloured by the particular organism the leaf represents. The name of the leaf corresponds to the specific name of the genomic sample followed by the organism.

ing, **Querying**, and **Clustering and Visualization**.

4.4.1 Usage

Usage refers to any aspects related to either a regular user or system administrator to use or install the software. GDI provides both a *command-line interface (CLI)* as well as a *Python-based API* for using the software (the **Interface** column in Table 4.4). A large collection of other software packages also provide either a CLI or an API, however some software may also provide a *graphical-user interface (GUI)*—in particular: *BIGSdb*, *Enterobase*, *outbreak.info*, *covidcg*, and *NCBI Pathogens*. The GUI in these cases are provided as a website and in each case these software are available as a central web application for people to submit data to or to query existing data (the **Availability** column). All software compared is available for local installation except for the *NCBI Pathogens* project [126], which is only available as a web application hosted by NCBI (`https://www.ncbi.nlm.nih.gov/pathogens`, though certain components of their data analysis pipelines are available to install locally).

Input refers to the type of genomics data used as input for the software, either *reads (R)* or *assemblies (A)*. About half the software can support both reads and assemblies (*GDI*, *BIGSI*, *COBS*, *BCFTools*). *BCFTools* differs slightly from the others as it does not directly take as input reads or assemblies, but it can work with data (a VCF file) generated from either reads or assemblies. *BIGSI* accepts reads or assemblies as input, although it requires these data files to be pre-processed to identify kmers prior to indexing. *Enterobase*, while it accepts only

Software	Usage[1]					Indexing[2]		Querying[3]					Clustering and Vis.[4]		
	Inter.[a]	Avail.[b]	Input[c]	Multi-species	Diff.[o]	Feat.[d]	Inc. add.[e]	Samp.	Feat.	Dist.	Meta.	Unk.	Clust.	Lin.	Vis.
GDI	CLI/ API	L	RA	✓	●	NGK[g]	✓	✓	✓	✓	✓	✓	✓		✓
BIGSI [12]	CLI/ API	LW	RA[j]	✓	●●	K			✓						
COBS [56]	CLI/ API	L	RA	✓	●	K			✓						
BCFTools [99]	CLI/ API	L	RA[j]	✓	●●	N	✓	✓	✓			✓[i]			
BIGSdb [63]	GUI/ API	LW	A	✓	●●●	G	✓	✓	✓	✓	✓	✓[k]	✓	✓	✓
SnapperDB [47]	CLI	L	R	✓	●●	N	✓	✓		✓			✓	✓	
Enterobase [3]	GUI/ API	LW	R	✓	●●●	NG[l]	✓	✓	✓[l]	✓	✓	✓[k]	✓[l]	✓	✓
outbreak.info [49]	GUI/ API	LW	A		●●●	N	✓		✓[h]		✓				✓
covidcg [48]	GUI	LW	A		●●	N	✓		✓		✓		✓[m]		✓
NCBI Pathogens [126]	GUI	W	R	✓	N/A	NGK[g]	✓	✓		✓	✓	✓[n]	✓	✓	✓

Table 4.4: Comparison of different software to GDI.

[1] Inter. = Interface, Avail. = Availability, Diff. = Difficulty to install and use local version

[2] Feat. = Features, Inc. add. = Incremental addition

[3] Samp. = Samples, Feat. = Features, Dist. = Distance, Meta. = Metadata, Unk. = Unknown/missing features

[4] Clust. = Clustering of indexed data, Lin. = Assignment of lineages/sequence types, Vis. = Visualization

[a] CLI = Command-line interface, API = Application programming interface, GUI = Graphical user interface

[b] W = web application, L = local installation

[c] R = whole-genome sequence reads, A = genome assembly

[d] N = nucleotide, G = gene (multi-locus sequence typing), K = Kmer

[e] Incremental addition refers to incrementally adding new genomes to the same index/database.

[f] Multiple species refers to indexing multiple different species within the same index/database.

[g] Here, K means support for clustering using kmers and not querying for individual kmers.

[h] Here, querying by feature produces only summary information.

[i] VCF files can be filtered to mark low-quality variant calls, which could indicate missing regions.

[j] Reads and assemblies require pre-processing with external software.

[k] Unknown/missing data is handled via methods such as partial matches or excluding missing data for pairwise distances.

[l] N is for nucleotide-level features (SNPs), which are used primarily for clustering and only limited support is provided for querying.

[m] Clustering here is only for building a tree of lineages, not individual genomes.

[n] Some support for representing partial matches of genes (for antimicrobial resistance).

[o] Difficulty ranking: ●= low, ●●= medium, ●●●= high

reads as input, will perform a *de novo* assembly on these reads using a standardized pipeline prior to processing the data. The rest of the software operates on either reads or assemblies, but not both.

Multi-species refers to the ability for the indexing and querying components to operate on multiple different organism species. Some software (*outbreak.info* and *covidcg*) only work for a particular type of organism (SARS-CoV-2). The rest of the software, including GDI, is not specific to any particular organism and can store and process a wide variety of WGS data.

Finally, another way to compare each software is how difficult or easy it is to install and use a local version (**Difficulty to install/use**). The provided results in Table 4.4 are based on how many steps are required for installing the software, how detailed the documentation is, or how much additional external software is required (such as relational database management software). The easiest to install and use includes *GDI* and *COBS*, which have minimal external dependencies and can directly take as input sequence reads or assemblies from a command-line interface. The medium level of difficulty includes: *BIGSI*, *BCFTools*, *SnapperDB*, and *covidcg*. These each require some additional dependencies to be installed (web servers or external database management software) or require pre-processing of input data prior to loading the data. The most complex software is *BIGSdb*, *Enterobase*, and *outbreak.info*, all of which have a more complicated installation process such as creating and modifying configuration files and setting up database management software. Finally, *NCBI Pathogens* is not available to install locally and so it cannot be ranked according to this category.

Notably, there is no simple way to rank difficulty-to-use as it depends on the use cases for the software. I chose to rank according to the difficulty to install locally, but if instead I chose to rank according to the difficulty to use the publicly available web applications, then any software which provides a web application (*e.g., NCBI Pathogens*, *BIGSdb*, *outbreak.info*) would outrank any other software package (including GDI, which has no public website with data available).

4.4.2 Indexing

Indexing refers to any aspect related to breaking up genomes into features and storing these features in some data structure to provide either querying functionality or to cluster genomes based on shared features. The **Features** column (Table 4.4) contains three different categories of features being compared: *nucleotide (N)*, *gene (G)*, or *kmers (K)*. Both *GDI* and the *NCBI Pathogens* pipeline are able to support indexing by all three features, though with the caveat that kmer-based indexing does not record all kmers in a genome and is primarily used for clustering genomes. Both *BIGSI* and *COBS* index based on kmers and—unlike *GDI* or *NCBI Pathogens*—they record information about all kmers and provide support for querying using all these kmers. *Enterobase* supports indexing by two features (nucleotide and genes/MLST), but with the caveat that nucleotide-level features are primarily used to construct a phylogenetic tree rather than being recorded in an index for later querying. All other software supports indexing by only one type of feature. In particular, *BCFTools* is a suite of command-line software which is focused on nucleotide variants only and can construct indexes of these variants for faster querying as well as set operations on nucleotide variants across samples. *BCFTools* itself is used as part of the *GDI* software to initially construct sets of nucleotide features from genomes.

Incremental addition refers to the ability to add new genomes into an existing index without having to re-construct the entire index. *GDI* supports this feature along with most other software. The exception is *BIGSI* and *COBS*, which both operate on kmers and require reconstructing an entire index if you want to add new genomes into it.

4.4.3 Querying

Querying refers to any operation where you input a request for data and a collection of results are returned. There are a number of different data types by which you can query within these software. Querying by **Samples** will provide a way to search for information related to a particular sample or set of samples. Most software supports this query method except for *BIGSI* and *COBS* (which are primarily intended to query by a kmer or collection of kmers in a sequence). The other type of software which does not support this feature is *outbreak.info* and *covidcg*, which are intended more to provide summary statistics of the worldwide collection of SARS-CoV-2 genomes rather than searching for individual genomes.

Alternatively, you can query by **Features**, which will return results of all genomic samples which contain a particular feature. Most software supports this method of querying except for *SnapperDB* (which is used more for clustering and assigning SNP types to genomes) and *NCBI Pathogens* (used more for clustering collections of genomes and assigning SNP or kmer types).

Querying by **Distance** refers to the ability to search for genomes within some particular distance of each other. Only about half the software compared here provides support for this method of querying. Software that does not support this feature includes *BIGSI* and *COBS* (used mainly for querying by genes or small sequences), *BCFTools* (a tool suite more for manipulating collections of nucleotide variants), *outbreak.info*, and *covidcg* (used more for showing summary information of collections of SARS-CoV-2 genomes).

Querying by **Metadata** refers to the ability to select genomes based on associated metadata (such as collection dates, externally assigned lineages/sequence types, or other information). Only about half the software provides support for this method of querying.

Support for **Unknown** (or missing) features while querying refers to the ability to differentiate genomes which lack a particular feature compared to those where data is missing (so it is uncertain if they have the feature not). *GDI* supports this, alongside about half the other software. However, all the other software that supports this come with caveats or have differences compared to how *GDI* handles this situation. *GDI* uses a three-valued logic system (Section 3.3.1) to represent cases where it is True that a genome has a feature, False that it has the feature, or Unknown if it has a feature or not. No other software supports such a three-valued logic system, and instead uses other methods to handle missing or unknown features. With *BCFTools*, there is a way to represent missing data by marking particular entries in a VCF file as failing to pass particular filters (whether they are nucleotide variants or reference nucleotides), which would indicate low-quality (missing) regions on a genome. With *BIGSdb* and *Enterobase*—both of which are primarily used for gene-level data—there is support for representing missing data or partial matches of genes in an MLST scheme. Finally, for *NCBI Pathogens*, there is also support for partial matches of genes, primarily when recording antimicrobial resistance.

4.4.4 Clustering and Visualization

Clustering refers to either clustering genomes by shared features or the assignment of new **Lineages** of genomes (*i.e.,* assignment of labels to clusters). *GDI* supports clustering of genomes—using both kmers and nucleotide features—however it does not support assigning lineages/labels to individual clusters. This is the only feature missing from *GDI* that other software packages support shown in Table 4.4. Only about a half of the other software supports clustering: *BIGSdb*, *SnapperDB*, *Enterobase*, *covidcg*, and *NCBI Pathogens*. The software *covidcg* only partially supports clustering, which is applied to the SARS-CoV-2 lineages (using CoVizu [127]) instead of individual genomes. Slightly less than half of the surveyed software supports the assignment of new lineages, which includes all software that supports clustering except for *covidcg* (and *GDI*).

Visualization refers to built-in support for visualizing collections of genomes, either through dendrograms or other visualization techniques. *GDI* supports visualization (both as dendrograms as well as support to easily work with data through existing packages like matplotlib). Both *covidcg* and *outbreak.info* provide a lot of different visualization methods to summarize SARS-CoV-2 genomic data. *BIGSdb* and *Enterobase* provide visualization of bacterial gene-level (MLST) data. Finally, *NCBI Pathogens* provides visualization of a number of bacterial pathogen genomic data, including an integrated web-based phylogenetic tree viewer.

Chapter 5

Discussion

With the increasing relevance on whole-genome sequencing of microbial genomes for infectious disease surveillance there is a need for innovative methods for the management and analysis of large collections of these genomes. The most relevant example is the COVID-19 pandemic, which has lead to the generation of millions of SARS-CoV-2 viral genomes and development of highly specialized resources to investigate this data [48; 49; 3]. While a large collection of software exists to handle this data (Table 4.4), there are still a number of challenges for the large-scale application of microbial genomics to infectious disease surveillance.

I developed the **Genomics Data Index (GDI)** to help address some of these challenges by providing components for pre-processing, indexing, querying, clustering, and visualization of genomes. Chapter 1 described in detail some of these challenges and Chapter 2 described existing solutions and background concepts. Next I described how I address these challenges with the design of my software (Chapter 3). I then evaluated the performance of GDI (Chapter 4) using three separate datasets: a simulated dataset, a collection of SARS-CoV-2 genomes, and a collection of WGS reads derived from four different bacterial organisms. I included a fourth method of evaluation where I compared GDI to existing software for infectious disease data analysis. I now turn to a discussion of the implications of my evaluation and I present some cases where I have applied GDI to the task of studying SARS-CoV-2.

5.1 Evaluation

5.1.1 Data simulation

I have evaluated GDI using simulated data to assess the capability of identifying nucleotide-level variants and using these variants to construct phylogenetic trees. I used a simulated reference genome (consisting of two sequences 10,843 bp and 8,865 bp long, respectively) and simulated a set of 59 genomes such that substitutions and indels match the topology of a reference phylogenetic tree. This data was then indexed by my software under a number of different scenarios: 1) fixing the transition and transversion rate and adjusting the sequence coverage for the reads and 2) fixing the coverage and adjusting the transition rate. In each of these scenarios, I ran my data analysis pipeline and indexing stage (stage 1 and 2 in Figure 3.1) on both the simulated FASTA files with nucleotide variants (the idealized or perfect *assemblies* scenario) and the simulated reads from these FASTA files (the *reads* scenario).

I found that for *sequence reads*, the sequencing depth of coverage can have a large impact on the amount of nucleotide variants detected, with a sensitivity of 0.025 for 5X coverage compared to 0.851 for 20X coverage (Figure 4.2). However, this quickly plateaus with less gain in sensitivity as coverage increases (0.886 for 50X compared to 0.851 for 20X). As the value of 20X is twice that of the minimum read coverage of the data analysis pipeline for identifying nucleotide variants (minimum coverage of 10X), it is recommended to verify that WGS reads are at least twice this minimum coverage value to avoid large numbers of false negatives.

Indexing genome *assemblies* is not impacted by read coverage since there are no reads being used as input. However, when varying the amount of sequence divergence, I found that my indexing system is capable of identifying more true positive and less false positive nucleotide variants for assemblies when the substitution divergence is low to moderate (up to 6.77%), which is reflected in the greater F1 score values (F1 score of 0.973 for assemblies compared to 0.736 for reads at 6.77%, see Figure 4.5). However, I did find that for high sequence divergences (greater than 6.77%), GDI was incapable of identifying most true positives using genome assemblies, which is reflected in the large drop in F1 scores (0.0323 for assemblies compared to 0.608 for reads at 10.4%, see Figure 4.5). For reads, the drop in F1 scores is much more gradual as the substitution divergence increases (Figure 4.5). This was also reflected in the increase in tree distances for assemblies (or failure to build beyond a divergence of 4.20%, see Figure 4.6). Some additional investigation revealed that the large drop in assembly sensitivity is possibly caused by parameters to the minimap2 software (the `-x asm5` parameter), which may need to be adjusted as the diversity of the genomic data increases.

Overall, this suggests that for very closely related organisms or where a high precision and sensitivity is important, it would be recommended to process genome *assemblies* for best results. In particular, I note that the *assemblies* scenario resulted in 0 false positives no matter how high the divergence was between simulated genomes (Figure 4.4). However, this should be interpreted with caution as these assembly test cases were an idealized or perfect scenario, where there is no missing data or errors introduced due to the *de novo* assembly process. Alternatively, indexing using *reads* would be recommended as a general-purpose type of input data as the sensitivity and precision are less susceptible to specific parameter settings in the underlying software. However, I give this recommendation with the caveat that this is based on a test of a very small genome ($\sim$ 20 kb) and bacterial-sized genomes (millions of bp) may show different results.

5.1.2 SARS-CoV-2 data analysis

I evaluated GDI using a set of real-world SARS-CoV-2 genomes available in public archives. I divided up a set of SARS-CoV-2 genomes into subsets of 10, 20, 50, 100, 200, 500, 1,000, 2,000, 5,000, 10,000, and 20,000. Each of these subsets were processed through the GDI data analysis pipeline and indexing system (**Data analysis** and **Indexing** in Figure 3.1, referred to as **Analysis** and **Index** in the evaluation). For subsets of 500 or fewer genomes I also built a maximum-likelihood phylogenetic tree (**Clustering/phylogenetics** in Figure 3.1, referred to as **Tree** in the evaluation). I measured the run time, peak memory, and disk usage of all three of these stages. I next measured the run time and peak memory for querying the index using the query API and CLI. Finally, I examined how well the constructed phylogenetic tree for 500 genomes matched pre-existing classes of genomes (Pangolin lineages of SARS-CoV-2) by using the scoring system implemented in GDI (described in Section 3.4.4).

Analysis, indexing, and building a tree

I found that when it comes pre-processing and indexing genomes, building a phylogenetic tree of all genomes takes up an increasing proportion of the time of the full data analysis process (97% of the time for 500 genomes, see Figure 4.7). This suggests that building a tree should be performed with a bit more thought as to when it is really necessary, perhaps only for smaller datasets. However, this was performed with only one software package (iqtree [17]) and a fixed evolutionary model. There is potential that some fine-tuning of the evolutionary model and other parameters of the phylogenetic software could significantly speed up tree-building. Additionally, constructing a phylogenetic tree is not necessary to query the tree with GDI, as externally generated trees can be combined with an existing index instead of an internally constructed tree (using the `query.join_tree()` method in the GDI Python API).

The stage that took the largest amount of memory was the **Index** stage (Figure 4.8). There was an interesting trend that was detected, where a large spike in memory was found at a particular number of samples (from 2,000 to 5,000 samples). This was found to correspond to the batch size for loading genomes into the index (Figure 4.9). Increasing the batch size from 2,000 to 10,000 genomes delayed this large spike in memory usage. I included this batch size parameter since, while testing, I found that indexing larger datasets (tens of thousands of genomes) would overwhelm the amount of memory in the machine I was using. The capability to limit the number of genomes to index at any given time fixes this issue, but introduces some different issues, such as how to best set the batch size to optimize performance. Optimizing the batch-processing to use less memory would likely require some additional work.

Querying

I investigated the performance of querying using both the GDI Python API and the CLI. For the API, the query operations have a large range in their evaluated running time—from fractions of a millisecond to half a minute (Figure 4.11). Two of the operations that increase at a large rate are `q.features_summary()` (reaching 1.3 seconds for 20,000 samples) and `q.features_comparison()` (reaching 36 seconds for 20,000 samples). Both of these operations involve summarizing all genomic features in the index, which fits with the trend of increasing running time as more samples (and features) are added to the index. Other operations (*e.g.,* `q.hasa()`) increase at a much more moderate rate (reaching 0.18 seconds for 20,000 samples, Figure 4.11). This has implications when working with large-scale data in an interactive environment (*e.g.,* Jupyter), where individual query operations (like `q.hasa()`) may remain very quick while data summary operations (like `q.features_comparison()`) may start to take a significant enough time that their results may have to be pre-computed and cached.

One noticeable difference between the API and the CLI is that querying in the CLI has an overhead of a few seconds, likely due to starting up Python and loading the GDI application (Figure 4.15). This renders some query operations significantly slower with the CLI (∼20X slower for the CLI for `q.hasa()`). It could be beneficial to further investigate how to speed up the CLI. However, the Python API is designed to integrate with other Python-backed statistical analysis and visualization packages (pandas [22] or the ETEToolkit [21]) to aid in moving from simple queries to a final figure or table in the same environment. Hence, for optimal performance when performing repeated queries, it would be recommended to work directly with the Python API as opposed to something like the CLI within a Bash script.

Clustering

I also examined the capability of a maximum-likelihood tree to group genomes into monophyletic groups that corresponded to pre-existing lineages. This was measured using a tree-to-cluster scoring system I implemented as part of GDI (described in Section 3.4.4). I found that the maximum-likelihood trees constructed using GDI grouped genomes into clades that correspond to SARS-CoV-2 lineages just as well as the existing Augur pipeline [3] (Figure 4.17). However, the kmer-based single-linkage clustering methods were not able to cluster genomes into clades (when interpreting the hierarchical clustering tree as a phylogenetic tree) nearly as well, regardless of the kmer-size ($k \in \{31, 51, 71\}$). This suggests that kmer-based clustering is not appropriate for very closely related genomes (as is the case with SARS-

CoV-2 genomes). However, the maximum-likelihood approach implemented in GDI is just as capable as existing software to construct a phylogenetic tree consistent with the Pangolin lineage classification scheme.

5.1.3 Read data

I evaluated the data analysis (**Analysis**), indexing (**Index**), and phylogeny-building (**Tree or clustering**) components of GDI using sequence read data derived from a previously published dataset intended for benchmarking [125]. This consisted of 22 *Campylobacter jejuni*, 9 *Escherichia coli*, 31 *Listeria monocytogenes*, and 23 *Salmonella enterica* genomes. I subsampled these datasets and ran these genomes through the data analysis, indexing, and clustering stages. I measured the running time, memory, and disk usage of processing the data with GDI. The kmer and maximum-likelihood trees were compared with respect to how well they were able to group genomes at the organism level as well as at the pre-established division of genomes into outbreak and outgroup categories for each organism.

Analysis and indexing

Similar to the SARS-CoV-2 assemblies dataset, I found the data analysis stage, where I identify nucleotide variants, to occupy the majority of the overall processing time (the **Analysis** takes up 97% of the **Total** time for 8 samples of *Campylobacter jejuni*, Figure 4.18). However, there was a large amount of variability with regards to the running time among the different organisms and across different processing stages. For the **Analysis** stage, I found *Campylobacter jejuni* to take the longest time, but for the **Index** stage I found *Escherichia coli* to require the most amount of time. This trend was repeated for the peak memory and the total disk usage. By examining the relationship between the number of samples and number of features (Figure 4.19) I was able to determine that the large running time for *Campylobacter jejuni* for the **Index** stage is likely due to the larger number of nucleotide variants (features) per sample. Hence, the relationship between running time and the particular data being indexed is a bit complicated. Isolating the specific relationships would likely require testing different datasets from a variety of organisms.

Tree or clustering

I examined how well GDI is able to cluster genomes using both the kmer-based and the maximum-likelihood approaches. I found that kmer-based approaches are able to group genomes at a higher taxonomic rank (such as the species level, Figure 4.22). However, for use cases where there are more closely related genomes, the kmer-based approach is unable to classify genomes appropriately (Table 4.3). This means that while kmer-based approaches have their uses for a higher-level overview of a large number of genomes, they are not an appropriate method for identifying which genomes are part of, or outside of, an infectious disease outbreak based on only a small number of nucleotide variants. In this case, a maximum-likelihood approach would be recommended.

5.2 Applications

The ongoing COVID-19 pandemic has provided a unique set of problems to attempt to address, as well as a large collection of real-world data to help test out GDI. During the pandemic, I used the challenges faced by others at processing this data as guidance on how to structure GDI and the sort of commands I should provide. Some examples include providing analysis pipelines which can support SARS-CoV-2 genomes, providing support for querying by the amino-acid change resulting from a nucleotide variant, supporting missing data, as well as supporting a phylogenetic-based visualization system for viewing nucleotide variants alongside a tree. Each of these features was inspired by challenges faced by myself or others when trying to make sense of the growing collection of SARS-CoV-2 genomes. Solving these challenges has provided opportunities to apply GDI to help make sense of the data being gathered throughout this pandemic. Here, I present two cases where I have applied GDI to COVID-19 data.

5.2.1 The SARS-CoV-2 Delta variant in Canada

The Delta variant of SARS-CoV-2 was first listed as a variant of concern by the World Health Organization on May 11, 2021 [7]. This variant quickly took hold worldwide and become the dominant variant. This includes within Canada, where the Delta variant was the dominant variant during the Summer and Fall of 2021, prior to Omicron. The term Delta is assigned by the World Health Organization, but an alternative name is B.1.617.2, which is assigned in the Pangolin [6] naming system. B.1.617.2 is also alternatively named AY in this system, with sub-lineages being recorded in a dotted-decimal notation (*e.g.,* AY.1, AY.2, ...).

In Canada, two sub-lineages of Delta that were found to be represented in greater proportions than the rest were AY.25 and AY.27. To better understand the evolution and spread of these Delta sub-lineages I collaborated with a team of people to use GDI to investigate the nucleotide variation within AY.25 and AY.27 and visualize the data on a phylogenetic tree [128].

Figure 5.1 and Figure 5.2 show two phylogenetic trees of the AY.25 and AY.27 sub-lineages of the Delta variant respectively, as found in [128]. The phylogenetic analysis itself was performed using the Augur pipeline (part of Nextstrain [3]) along with data of the SARS-CoV-2 virus derived from GISAID [4]. I ran GDI on the downloaded assemblies of the genomes (using the `gdi analysis` command) and joined the constructed index with the phylogenetic tree produced from Augur alongside metadata provided by GISAID listing the dates and locations of each genome. These were combined together into the final figures shown using the GDI Python-based API. These figures provide a visual representation (as a heatmap) of which nucleotide variants are associated with which SARS-CoV-2 genomes. The figures, alongside associated data analysis, were then used to identify a new sub-lineage of the Delta variant of SARS-CoV-2 (AY.25.1, `https://github.com/cov-lineages/pango-designation/issues/313`). Documentation and Jupyter notebooks showing all the code used to construct these figures is available on GitHub (`https://github.com/phac-nml/ay25ay27`).

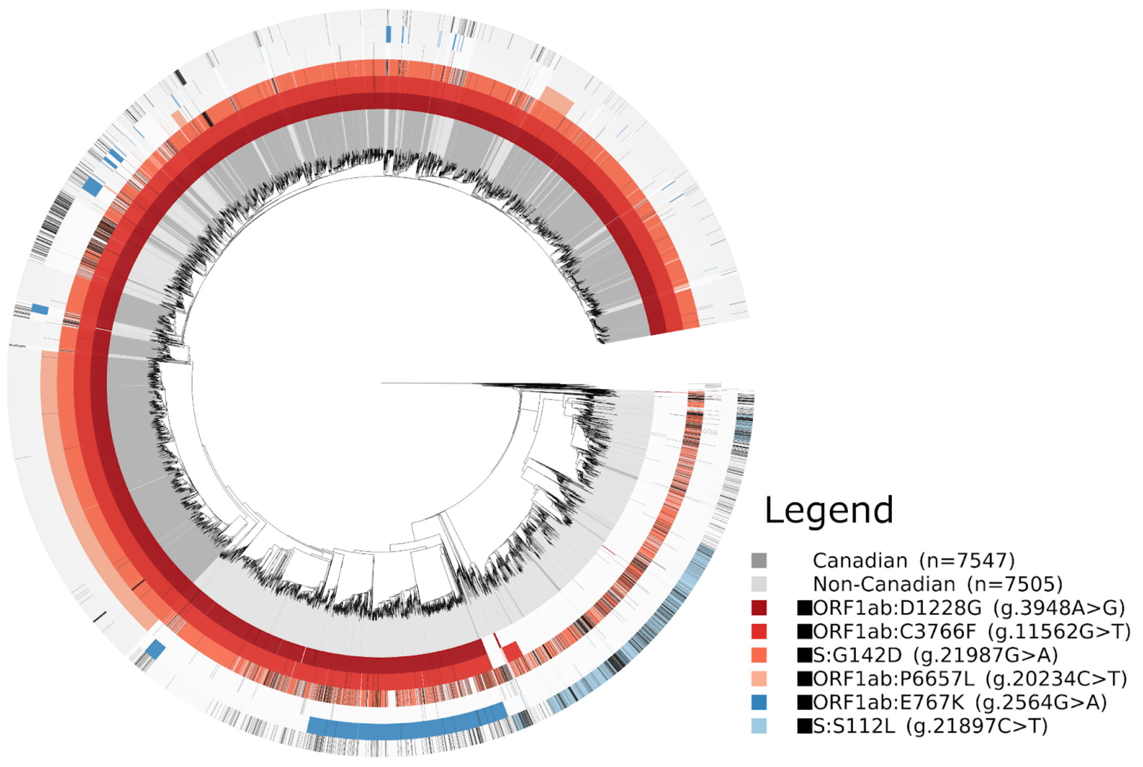

Legend
Canadian (n=7547)
Non-Canadian (n=7505)
■ORF1ab:D1228G (g.3948A>G)
■ORF1ab:C3766F (g.11562G>T)
■S:G142D (g.21987G>A)
■ORF1ab:P6657L (g.20234C>T)
■ORF1ab:E767K (g.2564G>A)
■S:S112L (g.21897C>T)

Figure 5.1: Phylogenetic tree of the AY.25 lineage of the Delta variant of SARS-CoV-2 from [128] and visualized using GDI. The inner track shows a time-scaled dendrogram representing the inferred ancestry of a collection of SARS-CoV-2 viruses. The next track indicates whether a particular virus was sampled in Canada (dark gray) or outside of Canada (light-gray) or is not AY.25 (white). The next tracks indicate whether a nucleotide variant is present (coloured), absent (transparent), or missing/unknown (black), with the particular variants listed in the **Legend**.

5.2.2 The SARS-CoV-2 Omicron variant

The Omicron variant of SARS-CoV-2 was named and defined as a variant of concern by the World Health Organization on November 26, 2021, after having been first detected only a few days prior [9]. One of the unique aspects of Omicron was the large number differences in the genome when compared to any other SARS-CoV-2 variant. Over the months of November and December 2021, there

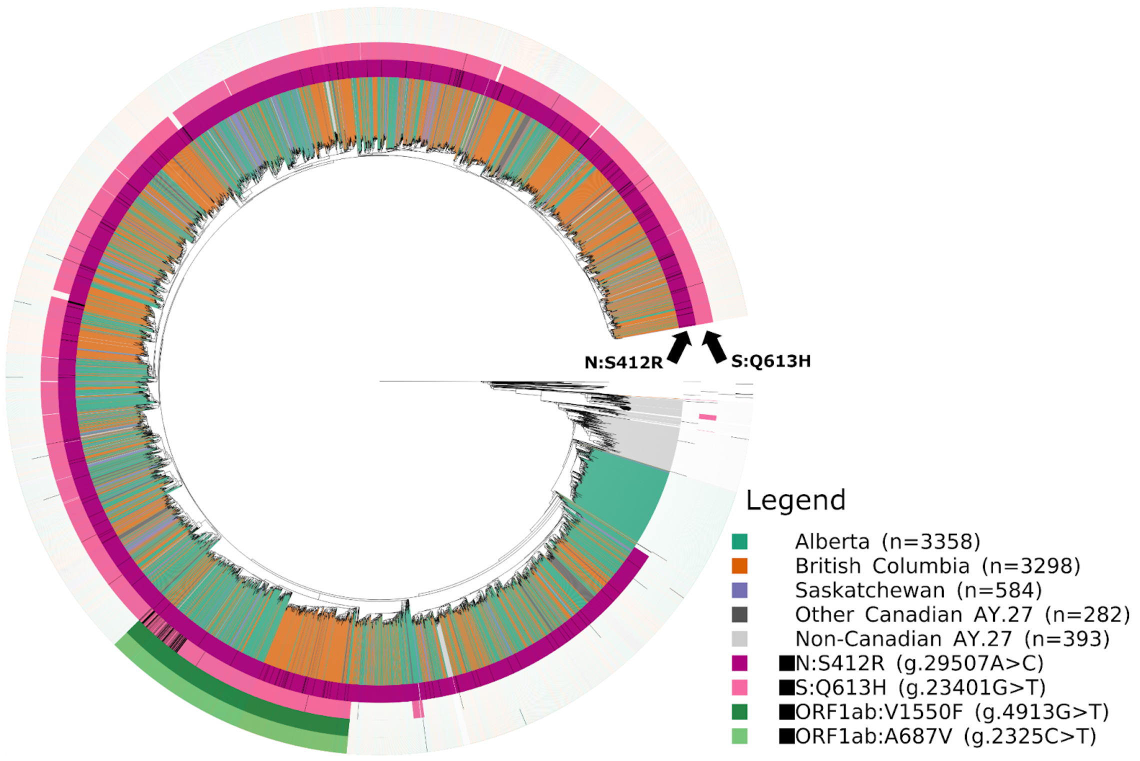

N:S412R
S:Q613H
Legend
Alberta (n=3358)
British Columbia (n=3298)
Saskatchewan (n=584)
Other Canadian AY.27 (n=282)
Non-Canadian AY.27 (n=393)
■N:S412R (g.29507A>C)
■S:Q613H (g.23401G>T)
■ORF1ab:V1550F (g.4913G>T)
■ORF1ab:A687V (g.2325C>T)

Figure 5.2: Phylogenetic tree of the AY.27 lineage of the Delta variant of SARS-CoV-2 from [128] and visualized using GDI. The inner track shows a time-scaled dendrogram representing the inferred ancestry of a collection of SARS-CoV-2 viruses. The next track indicates the location where the particular virus was sampled, either in Canada (coloured by province), outside of Canada (light gray), or non-AY.27 (white). The next tracks indicate whether a nucleotide variant is present (coloured), absent (transparent), or missing/unknown (black), with the particular variants listed in the **Legend**.

was a large shift worldwide to focusing investigation efforts to Omicron and the nucleotide variation it contained. One issue with Omicron when compared to other SARS-CoV-2 variants was the difficulty in sequencing particular regions of the viral genome, which lead to large amounts of missing data in viral genomes being uploaded to GISAID. I had already implemented methods to visualize nucleotide variants on a phylogenetic tree as well as account for missing data on a genome. I used GDI to process Omicron genomes of SARS-CoV-2 and visualize these genomes as a phylogenetic tree alongside nucleotide variants (mutations) unique to Omicron (Figure 5.3). This helped gain a better understanding of the diversity of nucleotide variants in Omicron when compared to other SARS-CoV-2 variants-of-concern, as well as providing insight into the particular regions that were more difficult to sequence within Omicron (appearing as "missing" data coloured black in Figure 5.3). Of particular note is the S:K417N mutation (column 26), for which Canada had a pre-existing RT-PCR-based diagnostic screen, but was originally thought to be polymorphic in Omicron and thus not suitable for its detection; however, my analysis using GDI demonstrated that the apparent polymorphism was due to missing sequence data, and was indeed suitable for Omicron detection. The screen was thus deployed in Canada for the detection of Omicron.

5.3 Limitations

5.3.1 Contiguous nucleotide-level variants

One limitation of GDI is related to cases with complicated mixtures of nucleotide-level variation, such as one variant contained within another variant. As an example, consider the multiple nucleotide variant `AT→GC`, which contains both an `A→G` and a `T→C`. In such a situation, you could use GDI to search for genomes with `AT→GC`, but not for `A→G` or `T→C` individually. This is due to the way nucleotide-level features are identified and stored in the VCF file format, which may group contiguous nucleotide variants together into a single entry: `AT→GC`.

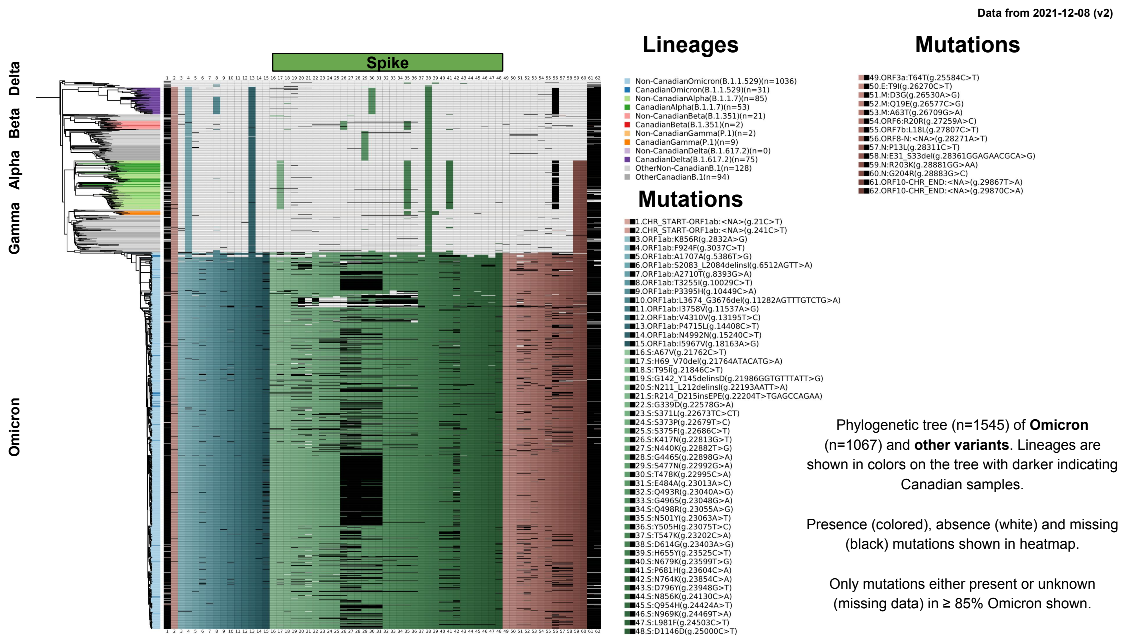


Figure 5.3: A phylogenetic tree and heatmap of the Omicron variant of SARS-CoV-2 alongside related genomes. The figure was generated from data on December 8, 2021 and shows a time-scaled dendrogram representing the inferred ancestry of a collection of SARS-CoV-2 viruses. The left-most portion shows the dendrogron with a heatmap of nucleotide variants (mutations) shown alongside the dendrogram. The presence (coloured), absence (white), and unknown/missing status (black) for each nucleotide variant is shown in the heatmap. The list of nucleotide variants is defined as those where $\geq$ 85% of the selected Omicron genomes either have the variant or the variant status is unknown.

This is, in part, related to the specific software and data analysis pipeline used to identify nucleotide variants and construct a VCF file.

This issue is related to issues encountered when constructing inverted indexes of English text, where a search for a word like "swim" should include both "swim" as well as variants of the word like "swims", "swimming", or "swam" [8]. One solution here is to identify the root of the word to index ("swim" for the word "swimming") and index this root alongside the encountered word (*e.g.,* index "swim" and "swimming"). Then at query time the query term can also be broken down into its components and root word to match the different variations (*e.g.,* a query for "swimming" could match documents containing "swim" or vice versa). This technique is termed **stemming** [8] and is used in many information retrieval systems including major search engines [129; 130]. Stemming could be adapted to solve the issue with complex or contiguous nucleotide-level variants. These contiguous variants, like `AT→GC`, would be split up into their root variants `A→G` and `T→C` and these get indexed alongside the contiguous entry `AT→GC`.

5.3.2 Matching nucleotide to amino-acid variants

A related issue arises when matching nucleotide-level variants to the amino-acid level effects they cause within a genome. The current implementation handles this through the SnpEff software [100], which operates using nucleotide-level variants in a VCF [75] file. If there are two separate records in a VCF file for variants that affect the same amino acid, then the variation impact, as predicted by *SnpEff*, will be incorrect (it will predict the amino-acid level change for each VCF record independently and will not merge them together).

As an example, consider a codon change in some DNA sequence, `CAT → GAA`, which corresponds to an amino acid change `His → Glu`. The nucleotide changes may be recorded separately in a VCF file as: `C → G` and `T → A`. Running SnpEff on such a VCF file would return the amino acid changes for each nucleotide variant separately: `His → Asp` (`CAT → GAT`) and `His → Gln` (`CAT → CAA`). In none of these entries is there listed the correct amino acid change (`His → Glu`). I have observed this when working with SARS-CoV-2 data, which requires corrections to the listed amino-acid changes after using GDI to produce summaries of the nucleotide-level and amino-acid level changes between genomes.

One possible solution in this case would be to look for alternative software for variant effect predictions to take this into account (such as `bcftools csq`, part of BCFTools [99]). However, this relates to an additional problem, where I have assumed there is a many-to-one correspondence between a nucleotide-level variant and an amino-acid change in the GDI data model (many nucleotide variants can result in the same amino acid change). While this works in many cases, the above example is a case which breaks this assumption (the same nucleotide variant may result in different amino acid changes depending on surrounding variation). Hence, a re-design of the underlying data model for GDI may be required to account for many-to-many correspondences between nucleotide-level variation and amino-acid level variation.

5.3.3 Multiple sequence alignment

The method I implement in GDI to construct a multiple sequence alignment for use in building a maximum-likelihood phylogeny is to first generate the consensus sequences for every genome, and then append every consensus sequence together into a single file. This will construct a valid alignment where each column (site) in the alignment consists of homologous nucleotides (Section 3.4.1). However, there are two cases which could cause issues with the alignment: **insertions** and **complex variants**.

With **insertions**, the issue is that they could lead to regions in the multiple sequence alignment where there are ambiguities. As an example, consider two insertions at the same position (with respect to some reference genome) like the following: **Sample 1:** `reference:5:A:ATA` and **Sample 2:** `reference:5:A:AG`. Since the insertions are of different sizes, there is no information above which lets us know how these two insertions should be aligned with each other. This is shown in Figure 5.4.A.

With **complex (multiple nucleotide) variants**, the consensus sequences may not always correspond to the optimal alignment. Figure 5.4.B shows an example of such a situation. As this type of situation could occur in the multiple sequence alignments constructed by GDI, it is possible these could have an impact on the phylogenies produced from these alignments.

One solution for both of these cases would be to construct a multiple sequence alignment using software designed for this purpose (such as MAFFT [110]), which will handle these ambiguities by taking into account all sequences/genomes in the collection. However, this would also require a greater amount of time to run.

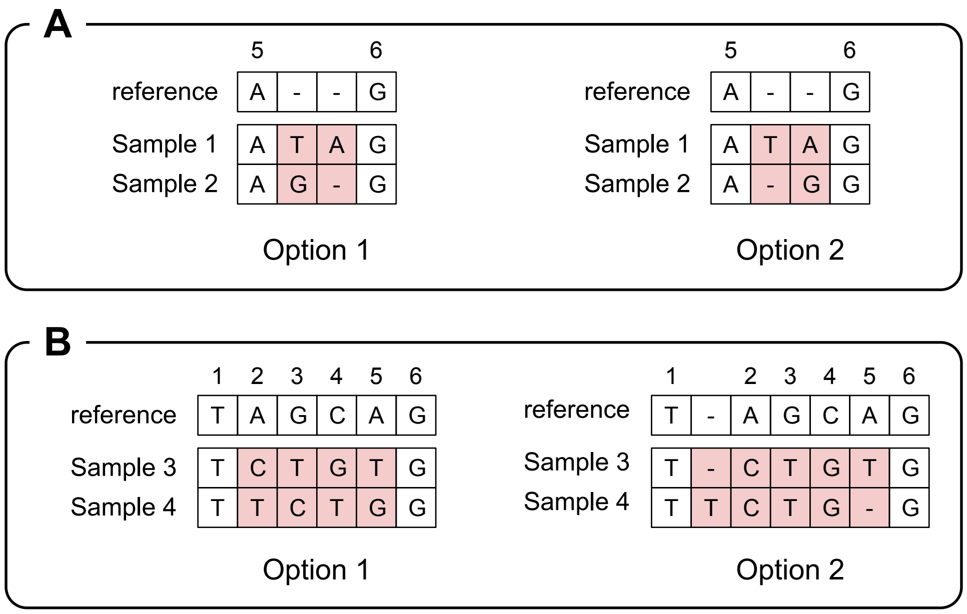


Figure 5.4: Two issues that could arise from generating a multiple sequence alignment (MSA) from consensus sequences to a reference genome. **(A) Insertion limitations.** Here, two insertions are depicted with different lengths: **Sample 1:** `reference:5:A:ATA` and **Sample 2:** `reference:5:A:AG`. This leads to ambiguities in how the insertions should be aligned—shown as *Option 1* and *Option 2*. **(B) Complex variant limitations.** Here, two complex multiple nucleotide variants are depicted of equal lengths—**Sample 3:** `reference:2:AGCA:CTGT` and **Sample 4:** `reference:2:AGCA:TCTG`. The alignment shown for *Option 1* would be returned by default if it is assumed that the consensus sequences, when concatenated, give a good alignment. However, *Option 2* shows a different method of aligning these two genomes in this regions which requires the insertion of gaps `-`. Thus, even with complex variants of equal length, the consensus sequences may not always produce the optimal alignment.

5.3.4 Roaring bitmap limitations

Another limitation is due to the use of Roaring bitmaps [104] to store collections of genomic sample identifiers rather than relying on tables within a relational database scheme. Roaring bitmaps were chosen as opposed to linking features to samples in a relational database table due to the significant reduction in the number of rows required for the inverted index (scales according to *features* as opposed to *samples* $*$ *features*, Section 3.2.2).

However, choosing this method means that some of the built-in mechanisms provided by many relational database software (such as cascading of queries, transactions, rollbacks, deletions, etc) cannot be used as-is. Hence, at this current moment, there is the potential for the database to be left in an inconsistent state if GDI was to crash during the middle of indexing genomes (specifically, you could end up in a scenario where features from some genomes are not properly added to the table storing the inverted index even though the samples themselves are recorded in the database).

An additional limitation comes out of the use of Roaring bitmaps, mainly that they can only store sets of 32-bit integers. Hence, the maximum number of genomes that can be stored in a single index is $2^{32} \approx 4$ billion.

Finally, the argument I made for choosing Roaring bitmaps compared to modeling feature-sample relations directly in the relational database (Section 3.2.2) does not account for all the major influential factors. I justified the choice of Roaring bitmaps since it dramatically reduces the number of rows required for storing the inverted index (from tens of billions to tens of thousands for SARS-CoV-2). However, reducing the number of rows falls under the larger goal of faster query times for the inverted index. The query time can be impacted by factors other than the number of rows, such as the number of sample identifiers in a Roaring bitmap, the number of samples in a *Samples* table in the relational database, and the amount of time for join operations between identifiers in a Roaring bitmap and sample names in the relational database. Properly accounting for the resource usage of all these factors could influence the justification for the use of Roaring bitmaps. In the future, additional work could be done to account for all of these factors, or to perform empirical experiments to measure resource usage in different scenarios to aid in future design decisions.

5.3.5 Gene-level clustering

While GDI supports clustering for both nucleotide variant and kmer features, it lacks the support for clustering using gene-level (MLST) features. This functionality could be provided by clustering gene-level data using a mechanism similar to that of kmers (computing pairwise distances based on gene-allele pairs and performing hierarchical clustering). Additionally, inspiration could be drawn from the hierarchical clustering (HierCC) defined in Enterobase [65], which clusters genomes at multiple distance thresholds. Providing support for clustering by genes as part of GDI would complete one of the most notable elements of GDI that is currently lacking when working with genomic features.

5.3.6 Reference genome mismatches for the data simulation evaluations

In Section 4.1, I evaluated GDI using data simulations of nucleotide variation and sequence reads constructed according to the structure of an initial phylogenetic tree used as input to the software Jackalope [116]. I simulated nucleotide variants by making use of a simulated reference genome (consisting of two sequences with a combined length of approximately 20 kbp). However, the initial input phylogenetic tree includes a leaf which represents another reference genome, which was used previously to construct this tree. I will refer to these as the **simulation reference** and **tree reference** respectively (Figure 5.5).

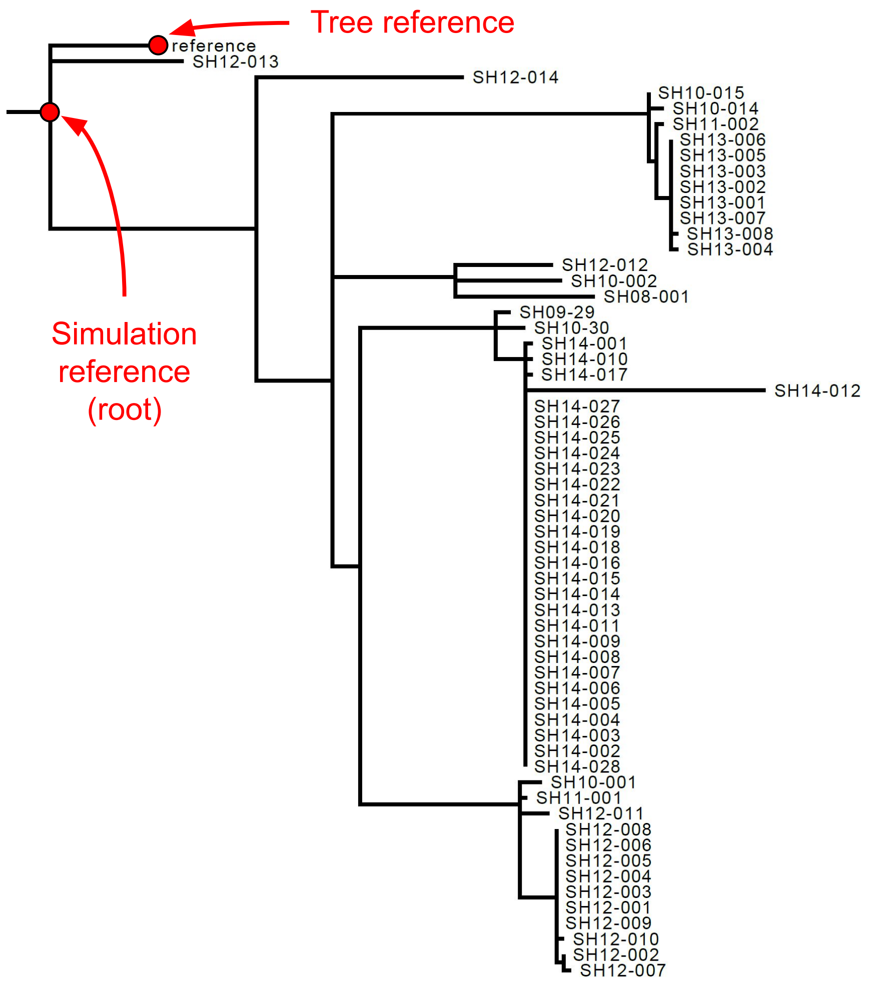


Figure 5.5: Two different reference genomes in the data simulation evaluations are labeled in the initial input phylogeny to Jackalope [116]. The **tree reference** is the reference genome that was included as part of the phylogenetic tree used to simulate nucleotide variants. The **simulation reference** is the location on the tree where the simulated reference genome used as input to Jackalope would be placed (the root of the tree). This shows that there is a small distance between the simulation reference and the tree reference.

During the data evaluation, I identified nucleotide variants with respect to the simulated reference genome, which makes it easier to compare variants I detect in GDI to those simulated by Jackalope. I removed the reads simulated from the tree reference genome from the final set of results. However, when comparing the two phylogenetic trees together using Robinson-Foulds and Kendall-Colijn distances, I am comparing the **tree reference** in the initial input phylogeny with the **simulation reference** in the constructed phylogeny. These two reference genomes are not identical, and I hypothesize that this leads to an increased distance between the initial phylogenetic tree used for simulation and the constructed trees from GDI (Figure 5.6).

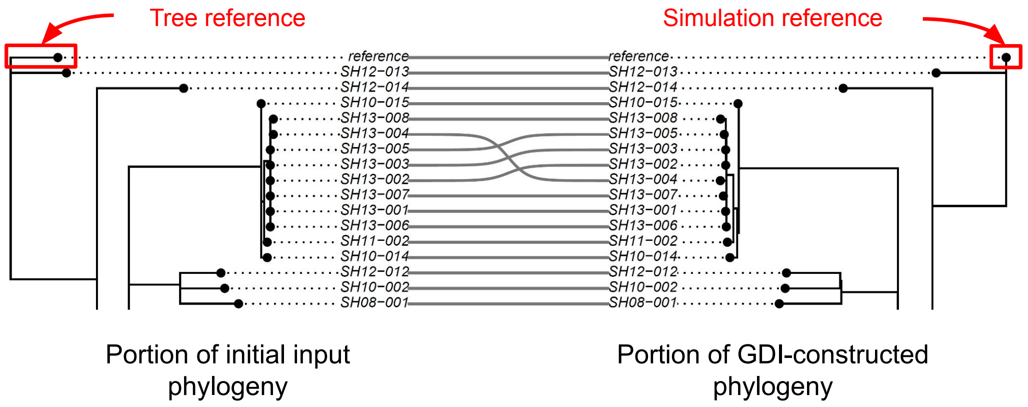


Figure 5.6: This shows the resulting difference in the branch lengths leading to the reference leaf for the initial input phylogeny for Jackalope (left) and the GDI-constructed phylogeny (right, constructed from the scenario of a read coverage of 20 and transition rate of 0.2). I hypothesize that this is a consequence of a mismatch in the particular reference genome I use for comparing phylogenies (the **tree reference** and the **simulation reference**). This figure is a portion of the phylogenetic tree comparison depicted in the appendix (Figure B.3).

While a mismatch between the reference genomes being compared in both scenarios affects the resulting trees, the reference mismatch impacts only one of the 60 leaves in each tree, and the mismatch occurs consistently for all evaluation scenarios. Additionally, since the Robinson-Foulds distance metric only compares the topology of trees [121] and since the topology appears unaffected by the mismatch in reference genomes (Figure 5.6 and Figure B.3), I conclude that the Robinson-Foulds distances are likely unaffected by the reference mismatch. The Kendall-Colijn metric [123] does incorporate branch lengths into its distance calculations, hence this measure may be affected by the reference mismatch. However, for the Kendall-Colijn distances, I have set $\lambda = 0.5$, which means both branch lengths and topology contribute equally to the distance calculations. Hence, due to all of these reasons, I conclude that it is still appropriate to perform relative comparisons of distances between trees with each other (as presented in Figure 4.3 and Figure 4.6). However, these distance measures could be improved by either directly comparing the tree reference with the one produced by Jackalope (that I removed prior to loading the data in GDI), or by removing the reference genome leaf from both trees altogether.

5.3.7 Uncontrolled variables for read data evaluation

In Section 4.3, I evaluated GDI by using real-world WGS data from four different organisms. I measured resource usage for indexing differing amounts of samples across all four organisms (such as run time presented in Figure 4.18 or peak memory usage presented in Figure 4.20). However, disentangling the influence of different variables on resource usage (*e.g.*, reference genome length, organism, number of features) proved challenging. This lead to an inability to properly explain the influence of all these variables on resource usage.

A better approach would have been to make use of simulated data, which would have provided an opportunity to better control and isolate the influence of each of these variables. This would have been similar to the initial set of data simulations I performed for evaluation (Section 4.1) and thus a better choice for read data evaluation may have been to expand this section with additional data to test different scenarios (such as the influence of reference genome length, or number of samples).

5.4 Future work

Some of the limitations discussed previously highlight different opportunities for future work on improving GDI. This includes better support for querying overlapping nucleotide variants and matching nucleotide variants to amino acids. A number of other possible directions would include the following.

5.4.1 Index tools

This would consist of a suite of command-line tools (or a Python API) for working with individual indexes. This could include commands to `merge` two indexes together, `split` indexes apart (perhaps by organism). Or a `package` command to construct an archive (e.g,. a zip file) of a genomics index for re-distribution. These commands would be useful in situations where multiple indexes were constructed separately and now there is a desire to join them together (or the opposite, perhaps split one large index apart into multiple smaller ones to improve speed).

5.4.2 Development of a web application

Web interface

Additionally, another area of improvement would be to develop a web-based interface for interacting with and visualizing the data stored in GDI. This interface could be used both for locally processed data or it could be deployed as a publicly available web-application for specific datasets to provide a method to explore the data and query for specific subsets of the data to visualize clusters and summary statistics. One possible dataset could be SARS-CoV-2, which is readily available and could be compared to existing applications for summarizing this data [49; 48; 3]. However, the benefit here is that the web application could be expanded to include data from a variety of organisms as it is not limited to SARS-CoV-2. Initial work on developing a web application has already been performed by Kimia Shadkami as part of her studies at the University of Manitoba. Kimia developed the code to integrate GDI's Python-based API into a web application using Flask (`https://palletsprojects.com/p/flask/`) and D3 (`https://d3js.org/`) and has made the code available on GitHub (`https://github.com/kimiashadkami/UI-for-a-Microbial-Genomics-Database`). This web application loads data from an existing genomics index and provides an interface to query for specific features and visualizing summary statistics of the set of samples containing those features. This application could be extended to provide additional querying and visualization capabilities, or even provide the capability to upload and index data through a web-based interface.

Web API

Additionally, a web-based API could be constructed as an additional interface alongside the Python API and CLI. This would primarily be targeted at developers who wish to interact with GDI to execute queries. This would provide a mechanism to integrate GDI into other applications by only operating at the level of this web-based API, which could provide a mechanism for remote users to integrate data indexed by GDI into their local software without having to use the Python API or CLI.

As an example, the software IRIDA [131] provides a web-based platform for the storage, management, and analysis of microbial WGS data. A web-based API for GDI could provide the capability to both load WGS data stored by IRIDA into GDI as well as execute queries on this data in GDI and display the results in IRIDA—all without having to adapt the Python-based API of GDI to the Java-based IRIDA application. One possible method for implementing this would be to make use of the BioThings SDK (`https://biothings.io/`) to construct a web-based API for accessing data stored in GDI.

5.4.3 Sequence typing and classification

One large feature lacking from GDI, but found in some other similar software, is the ability to assign sequence types (lineages) to a collection of genomic samples (the **Lin.** column in Table 4.4). When the sequence types are defined ahead of time (and used to train a classifier) then this would be a form of classification since the goal is to group the genomes into particular sequence types (*i.e.,* classes; see Section 2.1.4).

Given the capability of GDI to join together different kinds of features alongside different sorts of data related to genomes (such as phylogenetic trees or metadata, Figure 3.5), this could be implemented as a type of query within the GDI Python API, which returns a set of genomes that are part of a particular sequence type.

In fact, under this system, a **sequence type** could be defined as some *named combination of queries*. This could all be encapsulated into an abstract class, `SequenceTyper`, which maps the sequence type to the queries defining it. The `SequenceTyper` class could have concrete implementations depending on the typing (*i.e.,* classification) system used. This would provide a common API for implementing different types of classification systems that could use any number of combinations of genomic features (nucleotide variants, kmers, genes) or could be based on phylogenetic trees or metadata that is attached to a query.

Listing 5.1 shows an example implementation of this `SequenceTyper` class and how it would be used. It shows how to select only a particular set of genomes belonging to a single sequence type, but this could be extended to also return a collection of queries associated with each defined sequence type. Or, alternatively, to return a table that matches each genomic sample with the associated sequence type.

I had already begun work on such a system, which makes use of the Scorpio [132] method for classifying SARS-CoV-2 genomes according to the presence or absence of collections of nucleotide variants. The SARS-CoV-2 typer is implemented as a class in Python, `ExperimentalSARSCov2ConstellationsTyper`, and works similar to the code in Listing 5.1. However, this implementation was never completed and

Listing 5.1 An example of sequence typing implemented as a combination of queries

```python
# Extends abstract class SequenceTyper
class MySequenceTyper(SequenceTyper):

    # The sequence typing method which classifies genomes selected by
    # the input "query" to a particular type "name" (by executing
    # a series of sub-queries)
    def isa_type(self, name: str, query: SamplesQuery) -> SamplesQuery:
        if name == "type1":
            # "type1" is shorthand for the below set of queries
            return query.hasa("mlst:ecoli:adk:100") & query.hasa("ref:20:A:T")
        elif name == "type2":
            # "type2" is shorthand for the below set of queries
            return query.hasa("mlst:ecoli:adk:5")

# Create a new instance of the above defined MySequenceTyper
typer = MySequenceTyper()

# Assume query is some defined query selecting some genomes in an index
query

# query1 is a query that selects all genomes in the database
# that are of type "type1" according to the sequence typing implementation
query1 = typer.isa_typer("type1", query)

# query2 is a query that selects all genomes in the database
# that are of type "type2" according to the sequence typing implementation
query2 = typer.isa_typer("type2", query)
```

is left for future work.

Chapter 6

Conclusion

Microbial whole-genome sequencing has been increasingly used as a key part of routine surveillance of infectious diseases or outbreak investigations. WGS provides a wealth of information over previous techniques, down to individual nucleotide variants within the microbes causing an illness. This information can be used to track the spread of diseases and cluster microbial genomes based evolutionary distances. However, managing and interpreting the large amount of data generated from WGS experiments introduces unique challenges, which I introduced in Chapter 1.

One challenge is large-scale data analysis—with worldwide collections of microbial genomes already reaching millions of individuals. This data is continually being produced and uploaded to archives, which requires methods to dynamically process and integrate this data into existing reporting systems. Another challenge is the identification and interpretation of collections of genomes across a variety of feature-types—from individual nucleotide variants to collections of genes. Analysis methods also need to account for missing information—caused by poor-quality sequencing data, leading to uncertainty about the exact genetic content. The genomics data also needs to be integrated with epidemiological metadata or externally provided information in order to properly interpret the results and identify clusters of related microbes.

In Chapter 3, I presented my solution—the Genomics Data Index (GDI)—to address these challenges by drawing on inspiration from the fields of information retrieval and microbial genomics introduced in Chapter 2. I divided this software up into a number of components: **(1) data analysis** to break apart microbial WGS into a variety of genomic features; **(2) indexing** to construct an inverted index of genomic features; **(3) querying** to provide a mechanism to search for and select subsets of genomes; **(4) clustering** to group genomes into clusters based on shared features; and **(5) visualization** to provide built-in capability to visualize large collections of genomes. I provided both a command-line interface and a Python API for constructing and querying genomics indexes.

In Chapter 4, I evaluated this software using three datasets. I simulated a set of genomes and show that GDI is capable of accurately detecting nucleotide variants and clustering genomes over a wide range of input data types and divergences. I next used a real-world dataset of SARS-CoV-2 genomes, where I measured the relationship between the number of samples indexed and the run time, memory usage, and disk usage. I also compared clustering using both kmers and nucleotide variants and demonstrated that nucleotide variant clustering is the method which is most accurately able to group together genomes with their assigned lineages. I next evaluated the software using a collection of WGS read data from four different bac-

terial species. I evaluated differences in run time, memory, and disk usage across all four species. I also clustered the bacterial species using both kmers and nucleotide variants and demonstrated that kmer-clustering is able to group the bacteria by species, but is unable to differentiate between closely related bacteria within a species, whereas a maximum-likelihood approach is better suited to the task. Finally, I compared and contrasted my software to other existing software packages.
In Chapter 5, I examined the limitations of GDI and provided recommendations for the best ways to process different types of data. I also presented two different applications of this software to investigate viral variants from the COVID-19 pandemic. These included an analysis of the SARS-CoV-2 Delta variant in Canada during 2021 and an examination of the unique nucleotide variants in the SARS-CoV-2 Omicron variant.

Through both the evaluations and applications of GDI, I have shown the utility of an indexing system which can process and store features derived from WGS data for multiple different microbial species. I believe this software will be of great benefit to the larger scientific community for storing and processing microbial genomes and linking genomic data to contextual metadata to aid in infectious disease surveillance. This form of data analysis has become a routine part of infectious disease monitoring during the COVID-19 pandemic and will continue to increase in importance in the future.

Appendix A

Software design

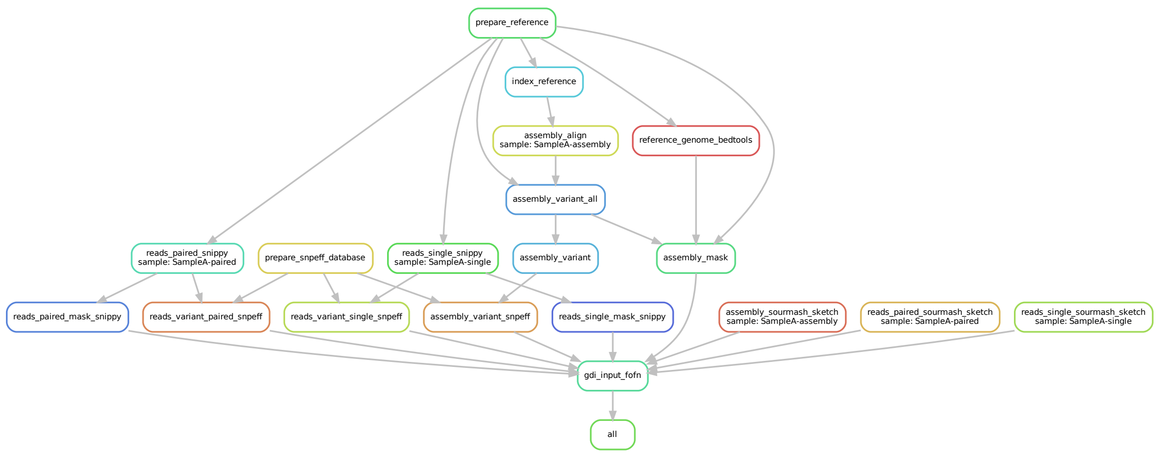


Figure A.1: A Directed Acyclic Graph (DAG) of the GDI analysis pipeline on three different samples: **SampleA-single**, **SampleA-paired**, and **SampleA-assembly**. This shows the dependencies among different stages of the analysis pipeline, which is used by Snakemake [90] to schedule jobs either sequentially (those stages on the same path in the graph) or in parallel (those stages on parallel paths in the graph). The final stage **gdi_input_fofn** is used to gather all the required files by GDI for the indexing stage (*i.e.*, VCF, BED, and sourmash sketches) into a single tab-delimited file of file-paths. This figure was constructed using the `snakemake --dag | dot -Tpdf` command as described in the Snakemake documentation.

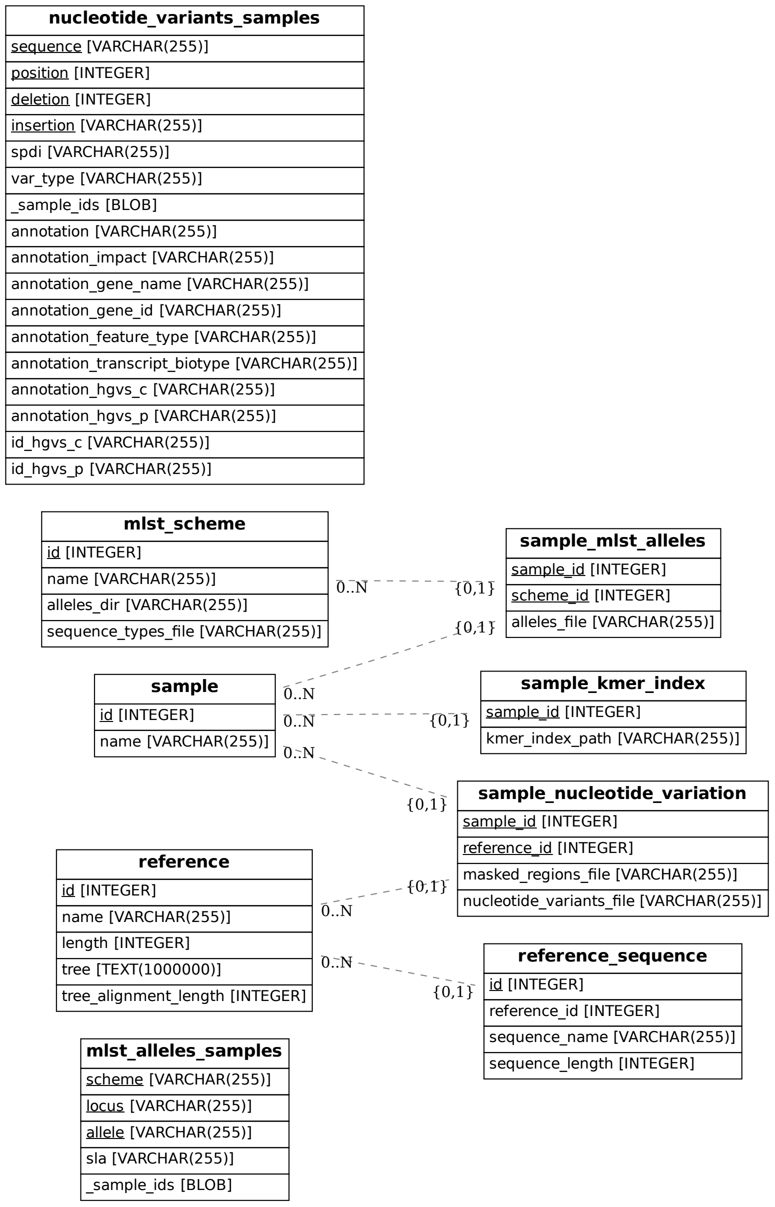


Figure A.2: An entity-relations diagram for the **Genomics Data Index** relational database. The diagram was generated using ERAlchemy version 1.2.10 [133].

Appendix B

Simulated data

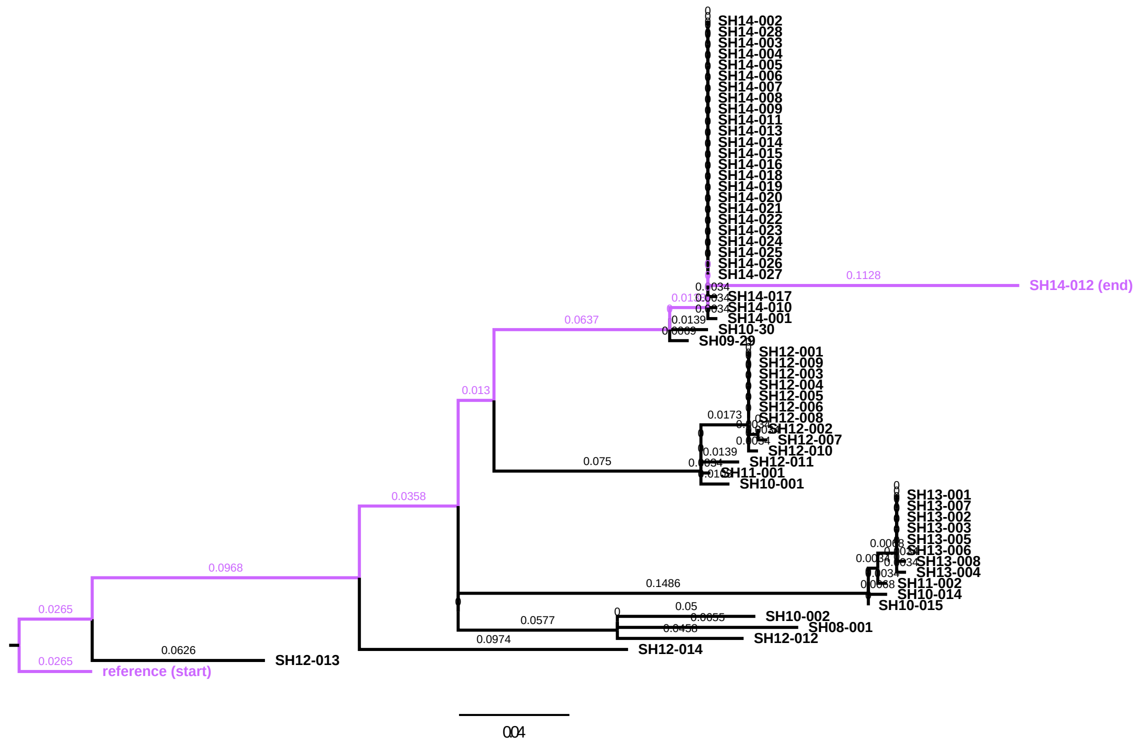


Figure B.1: The longest path to the reference genome on the tree used for simulating genomes. The longest path is highlighted in light purple with branch lengths shown. This path starts at the reference genome (*start*) and ends at SH14-012 (*end*). This is used to translate the *transition rate* to an *observed substitution divergence*. The figure was constructed using FigTree [134].

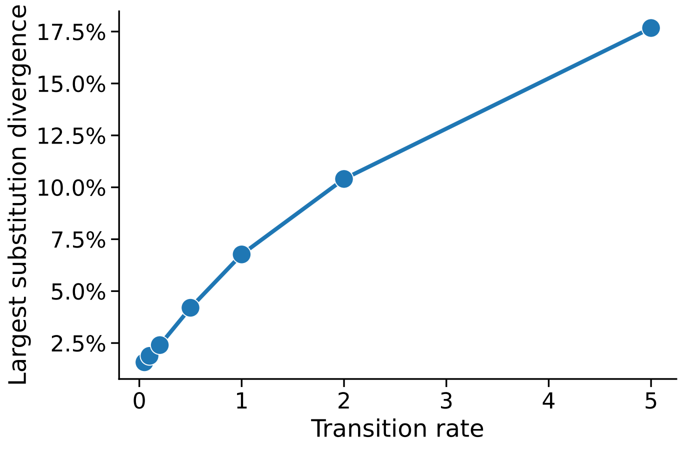


Figure B.2: A comparison of the transition rate to the largest observed substitution divergence. The transition rate is one of the parameters in the HKY85 model I used to simulate nucleotide variants (with the transversion rate set to 1/2 the transition rate). The largest observed substitution divergence is defined as the percentage of observed substitutions between *SH14-012* and the reference genome (so 0% means no substitutions were observed and 100% means the entire 19,699 bp of the simulated *SH14-012* genome is different from the reference genome). This is used to adjust the axis scales to one of observed substitution divergence from the reference genome.

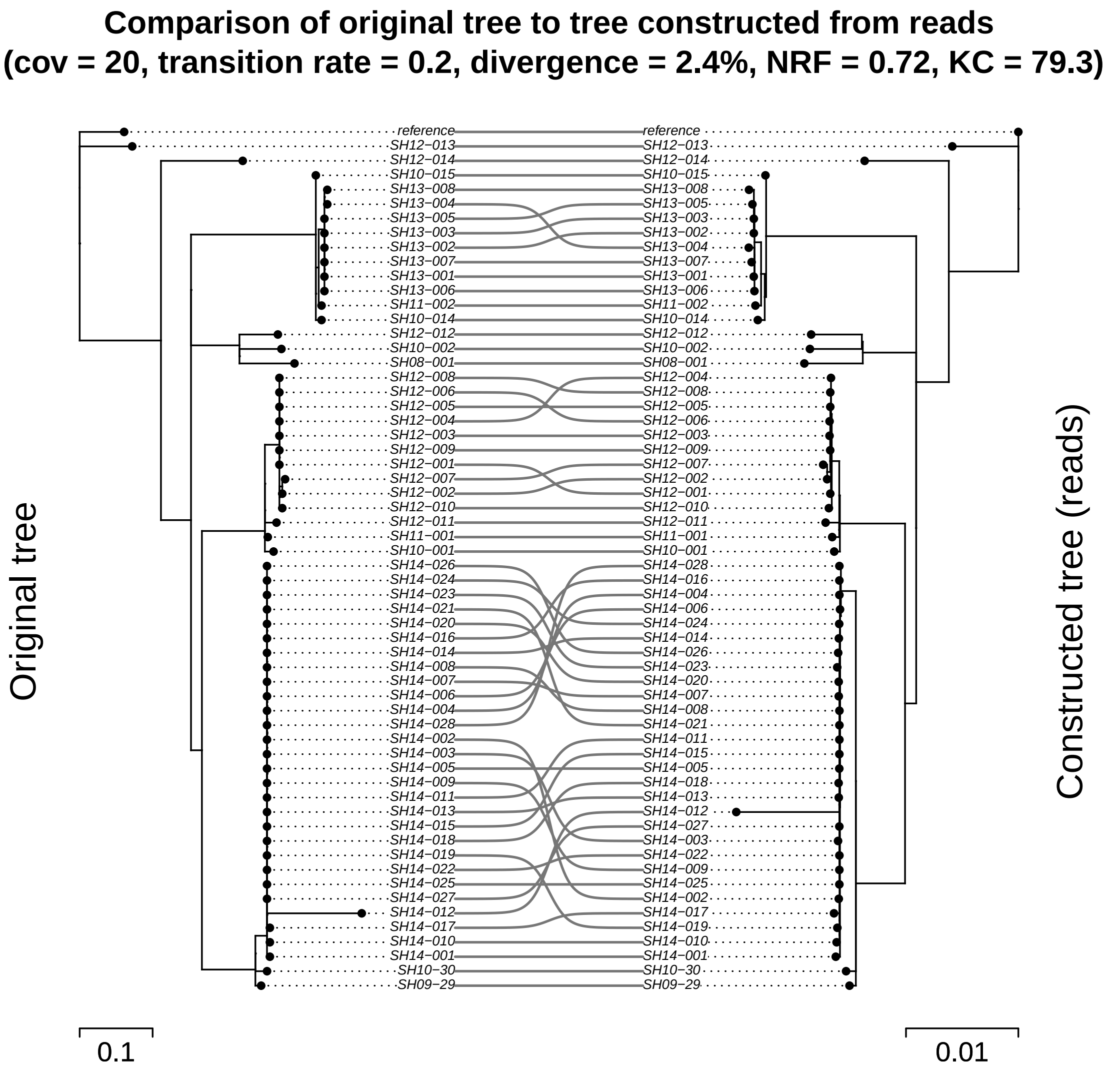

Comparison of original tree to tree constructed from reads
(cov = 20, transition rate = 0.2, divergence = 2.4%, NRF = 0.72, KC = 79.3)
Original tree
Constructed tree (reads)
0.1
0.01

Figure B.3: A comparison of the original reference tree to a tree constructed using reads with a coverage of 20X. Parameters used are listed at the top of the figure. The figure was constructed using the cophylo function from the phytools [135] package in R. NRF is the Normalized Robinson-Foulds and KC is the Kendall-Colijn distance metric. Divergence is the substitution divergence between *SH14-012* and the *reference* genome measured as a percentage difference in nucleotides to the reference genome.

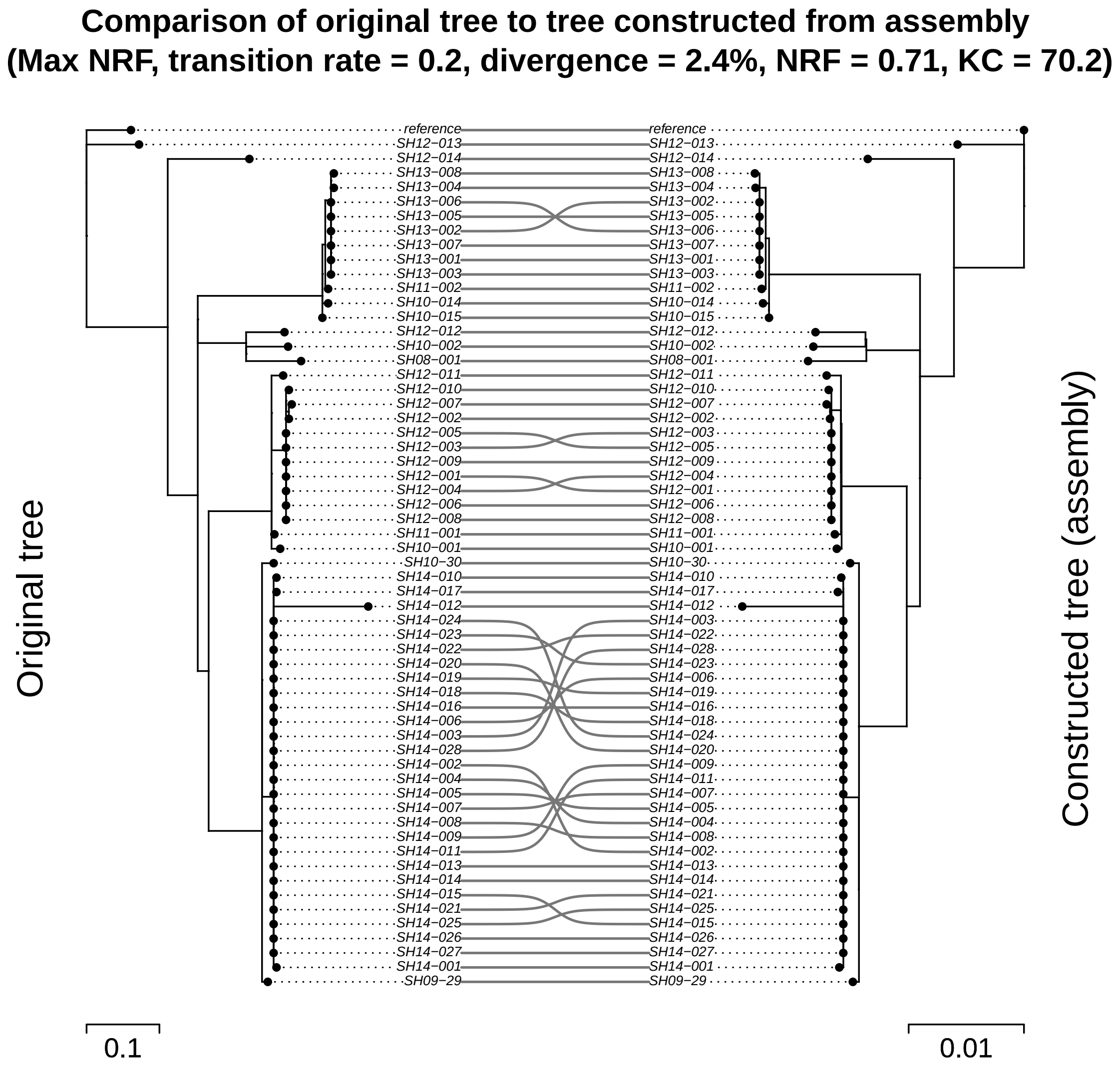


Figure B.4: A comparison of the original reference tree to a tree constructed using assemblies. Parameters used are listed at the top of the figure. The tree was chosen to be the tree which produced the maximum Normalized Robinson-Foulds distance (and so worst-performing tree) out of 6 trials with the same parameters. The figure was constructed using the cophylo function from the phytools [135] package in R. NRF is the Normalized Robinson-Foulds and KC is the Kendall-Colijn distance metric. Divergence is the substitution divergence between *SH14-012* and the *reference* genome measured as a percentage difference in nucleotides to the reference genome.

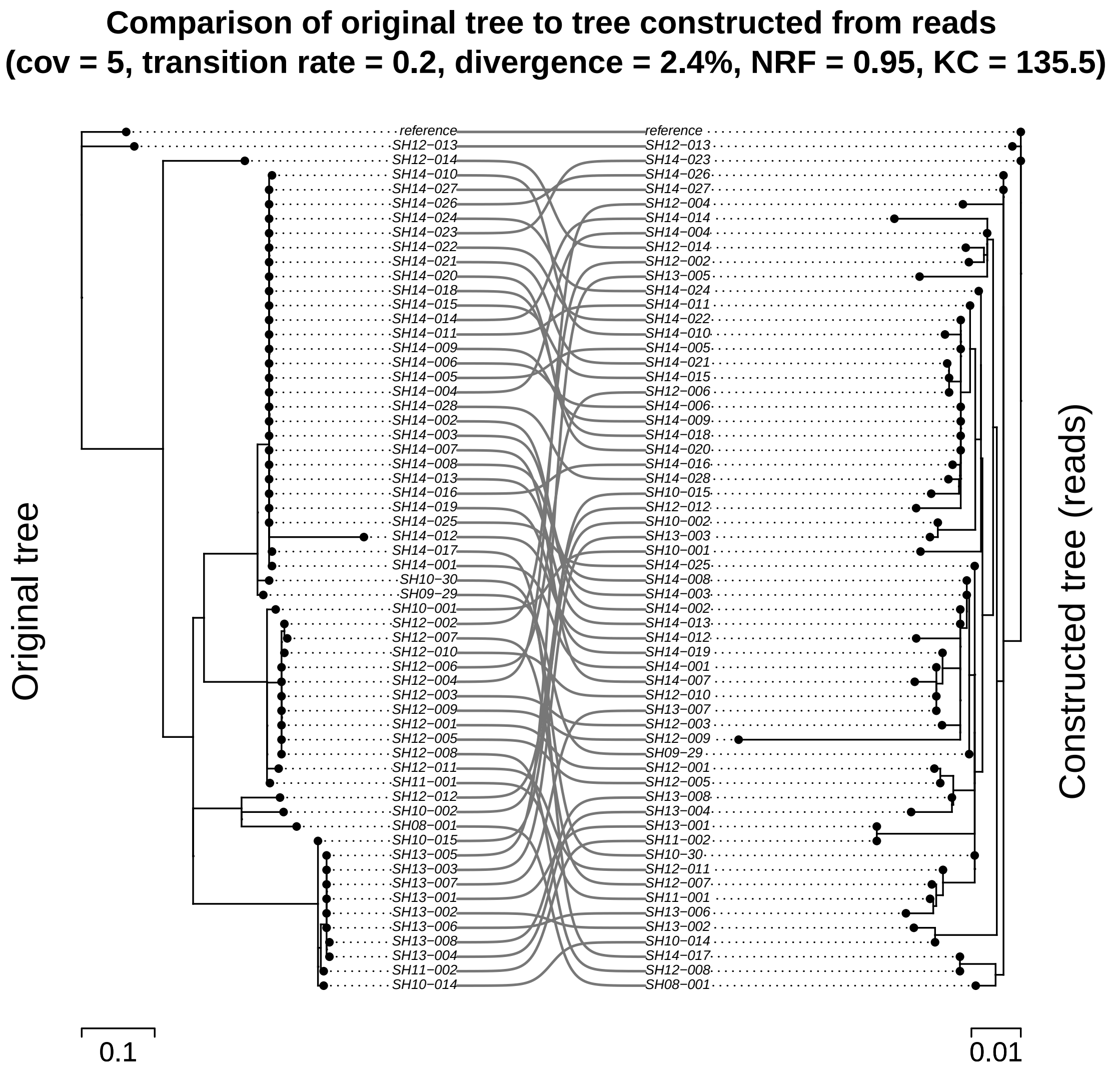


Figure B.5: A comparison of the original reference tree to a tree constructed using reads with a coverage of 5X. Parameters used are listed at the top of the figure. The figure was constructed using the cophylo function from the phytools [135] package in R. NRF is the Normalized Robinson-Foulds and KC is the Kendall-Colijn distance metric. Divergence is the substitution divergence between *SH14-012* and the *reference* genome measured as a percentage difference in nucleotides to the reference genome.

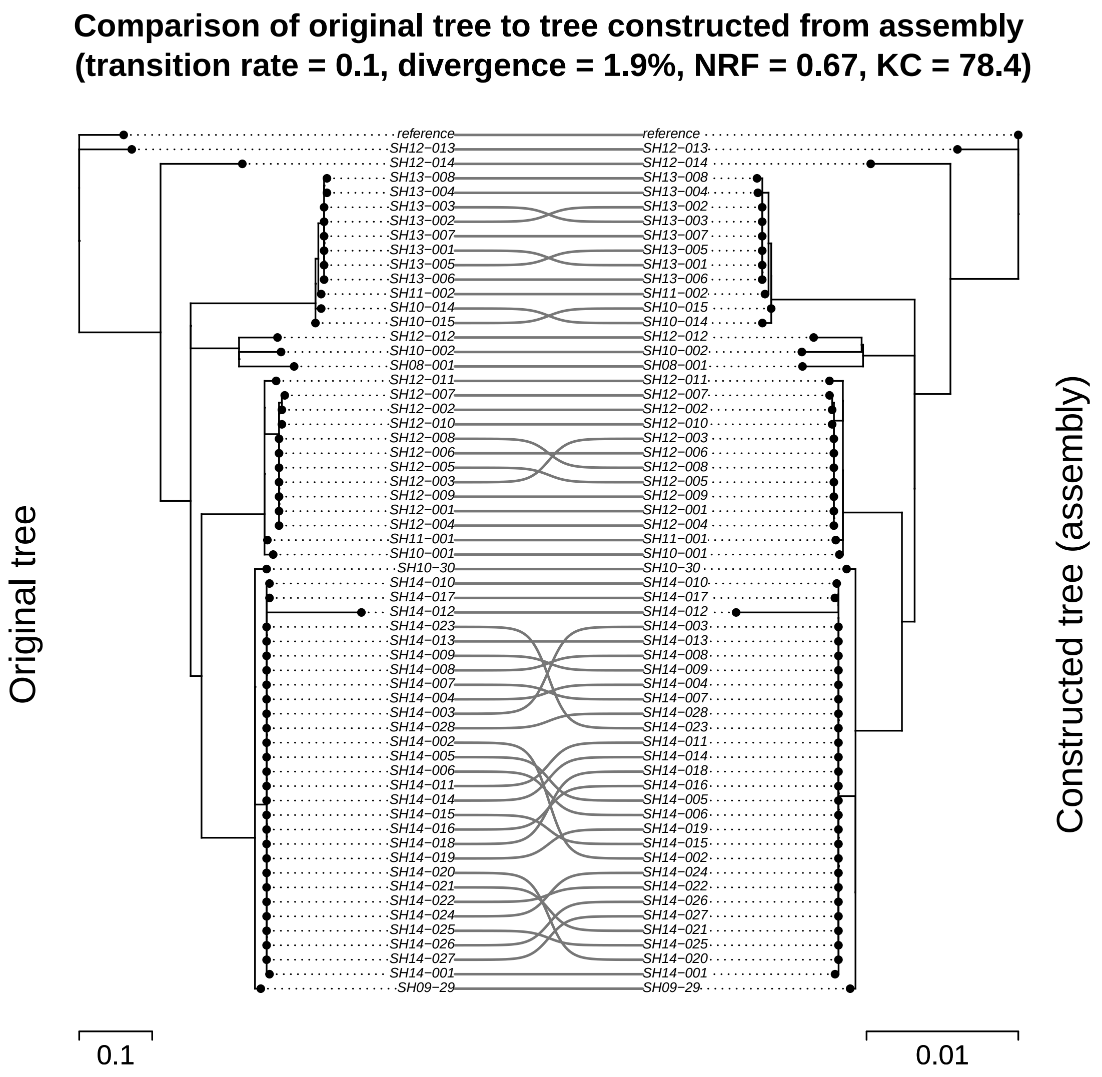

Comparison of original tree to tree constructed from assembly
(transition rate = 0.1, divergence = 1.9%, NRF = 0.67, KC = 78.4)
Original tree
Constructed tree (assembly)
0.1
0.01

Figure B.6: A comparison of the original reference tree to a tree constructed using assemblies and with a divergence of 1.88% (transition rate of 0.1). Parameters used are listed at the top of the figure. The figure was constructed using the cophylo function from the phytools [135] package in R. NRF is the Normalized Robinson-Foulds and KC is the Kendall-Colijn distance metric. Divergence is the substitution divergence between *SH14-012* and the *reference* genome measured as a percentage difference in nucleotides to the reference genome.

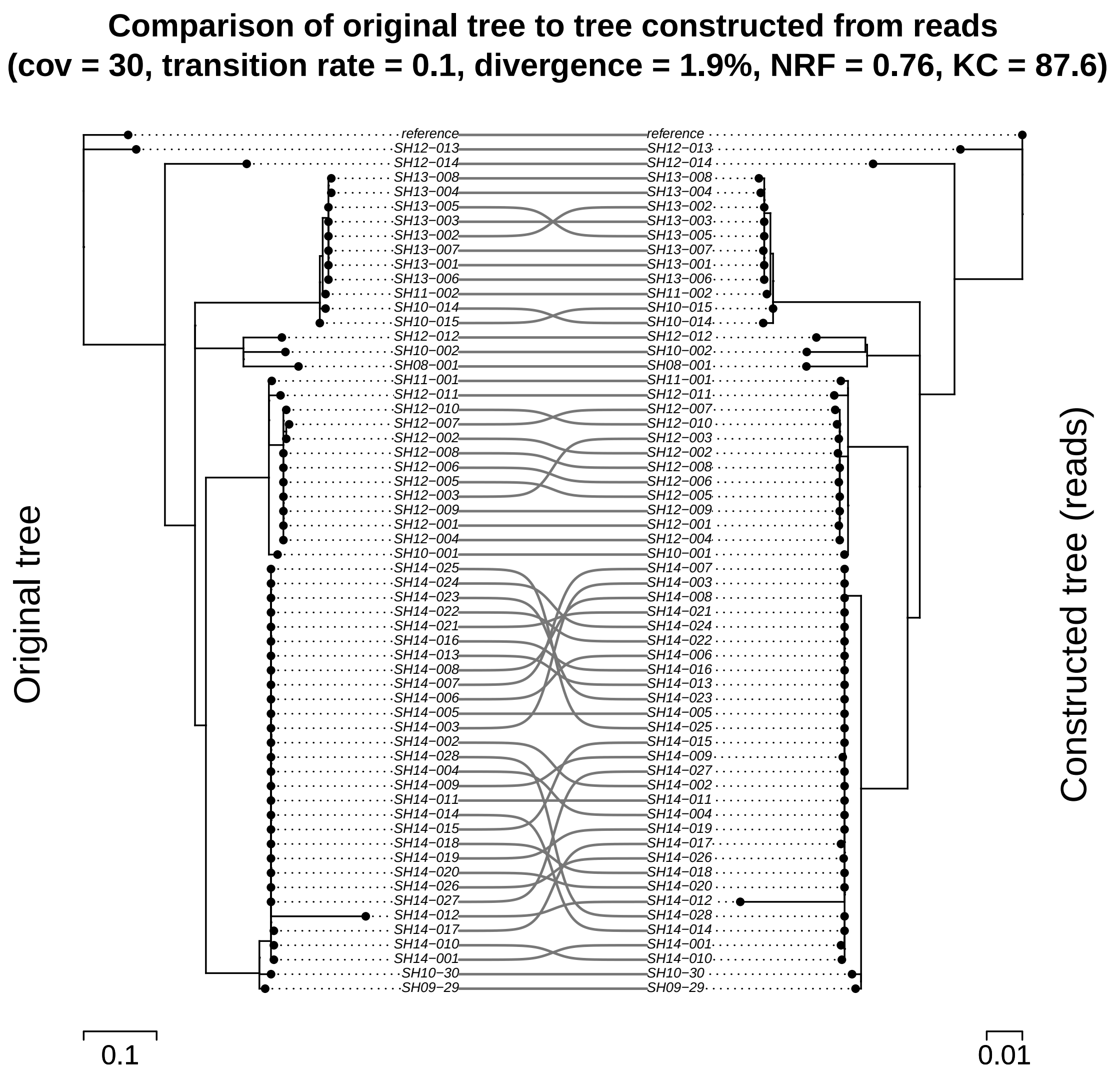


Figure B.7: A comparison of the original reference tree to a tree constructed using reads and with a divergence of 1.88% (transition rate of 0.1). Parameters used are listed at the top of the figure. The figure was constructed using the cophylo function from the phytools [135] package in R. NRF is the Normalized Robinson-Foulds and KC is the Kendall-Colijn distance metric. Divergence is the substitution divergence between *SH14-012* and the *reference* genome measured as a percentage difference in nucleotides to the reference genome.

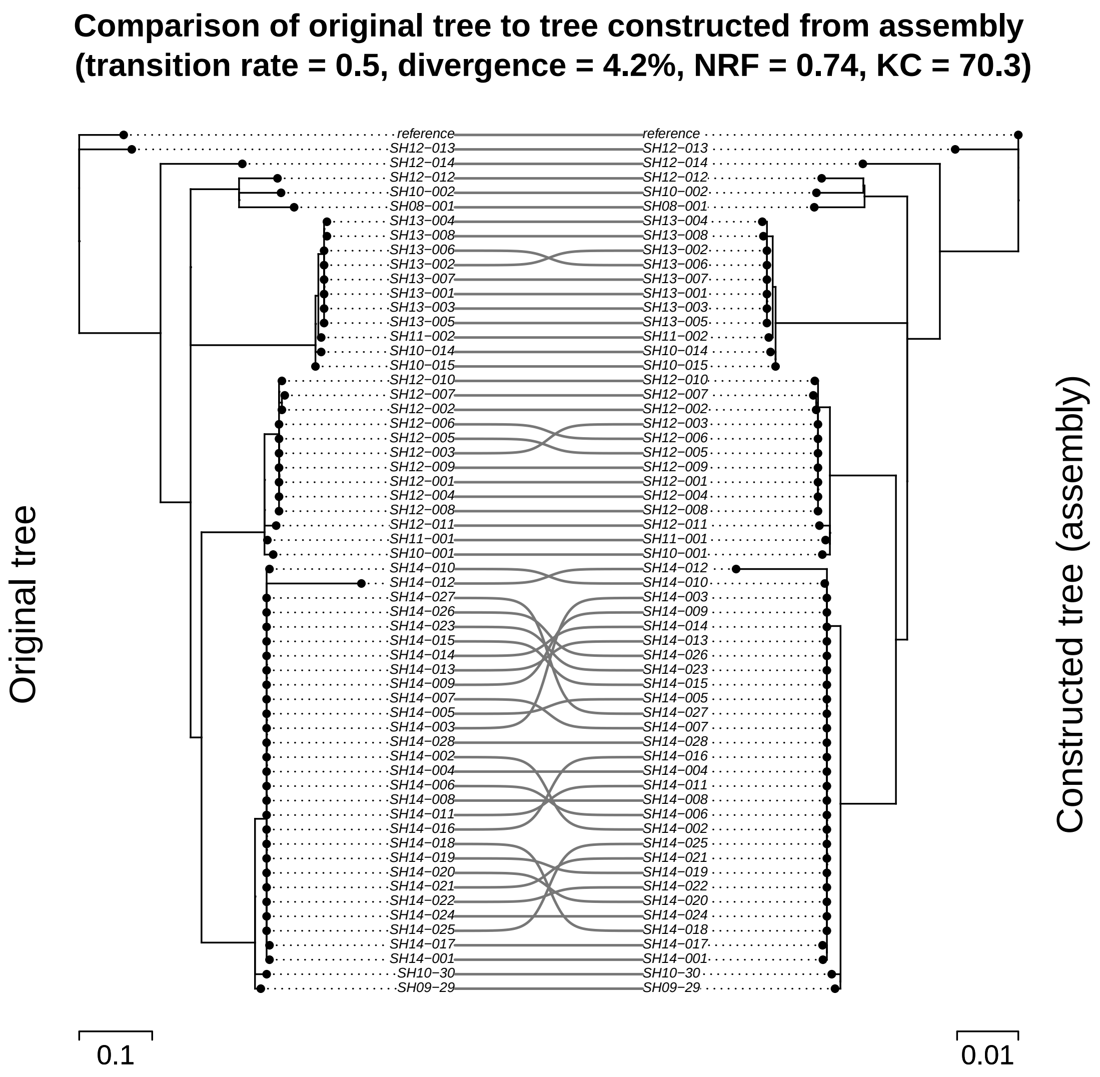

Comparison of original tree to tree constructed from assembly
(transition rate = 0.5, divergence = 4.2%, NRF = 0.74, KC = 70.3)
Original tree
Constructed tree (assembly)
0.1
0.01

Figure B.8: A comparison of the original reference tree to a tree constructed using assemblies and with a divergence of 4.20% (transition rate of 0.5). Parameters used are listed at the top of the figure. The figure was constructed using the cophylo function from the phytools [135] package in R. NRF is the Normalized Robinson-Foulds and KC is the Kendall-Colijn distance metric. Divergence is the substitution divergence between *SH14-012* and the *reference* genome measured as a percentage difference in nucleotides to the reference genome.

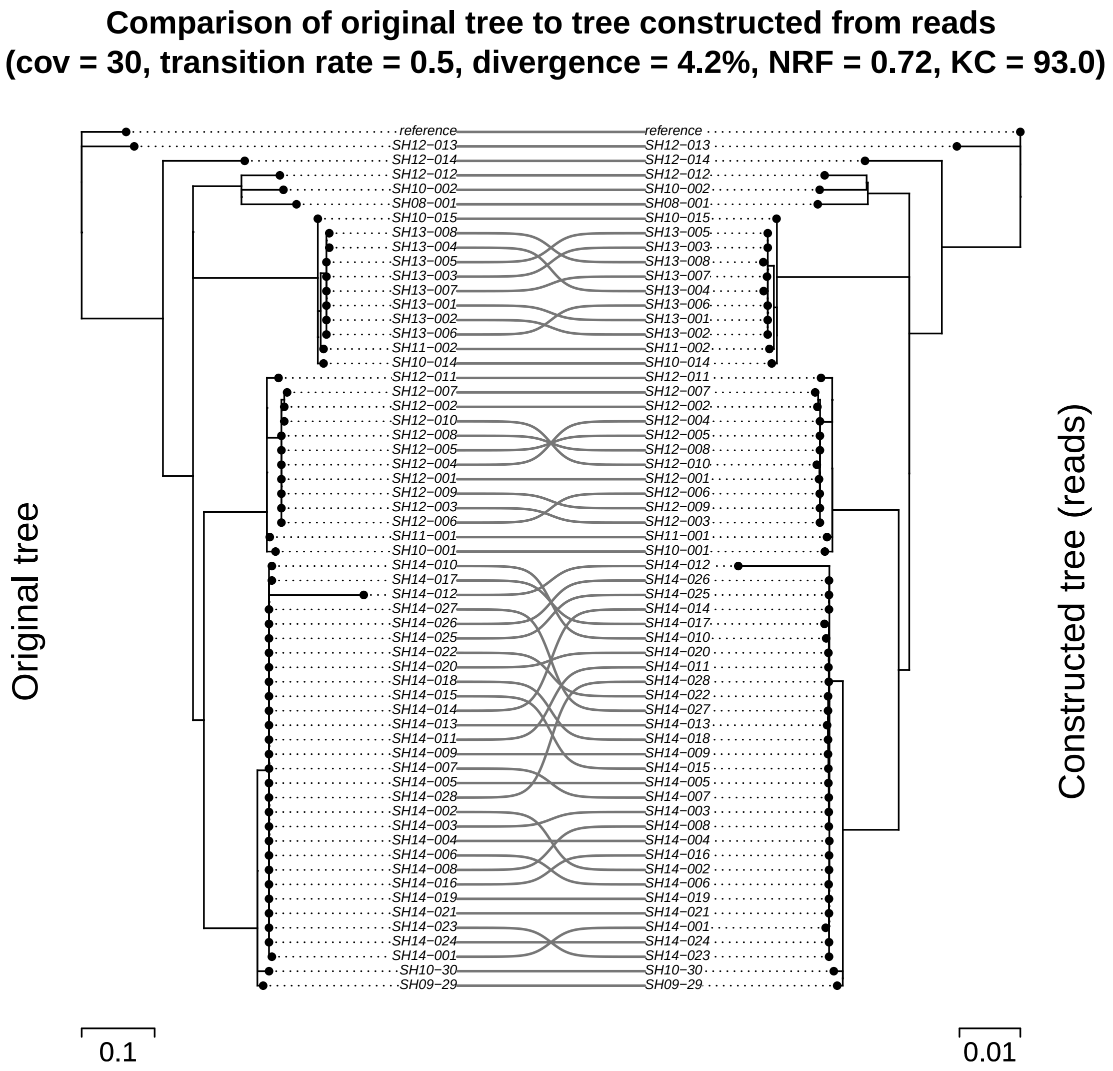


Figure B.9: A comparison of the original reference tree to a tree constructed using reads and with a divergence of 4.20% (transition rate of 0.5). Parameters used are listed at the top of the figure. The figure was constructed using the cophylo function from the phytools [135] package in R. NRF is the Normalized Robinson-Foulds and KC is the Kendall-Colijn distance metric. Divergence is the substitution divergence between *SH14-012* and the *reference* genome measured as a percentage difference in nucleotides to the reference genome.

Appendix C

SARS-CoV-2 evaluation

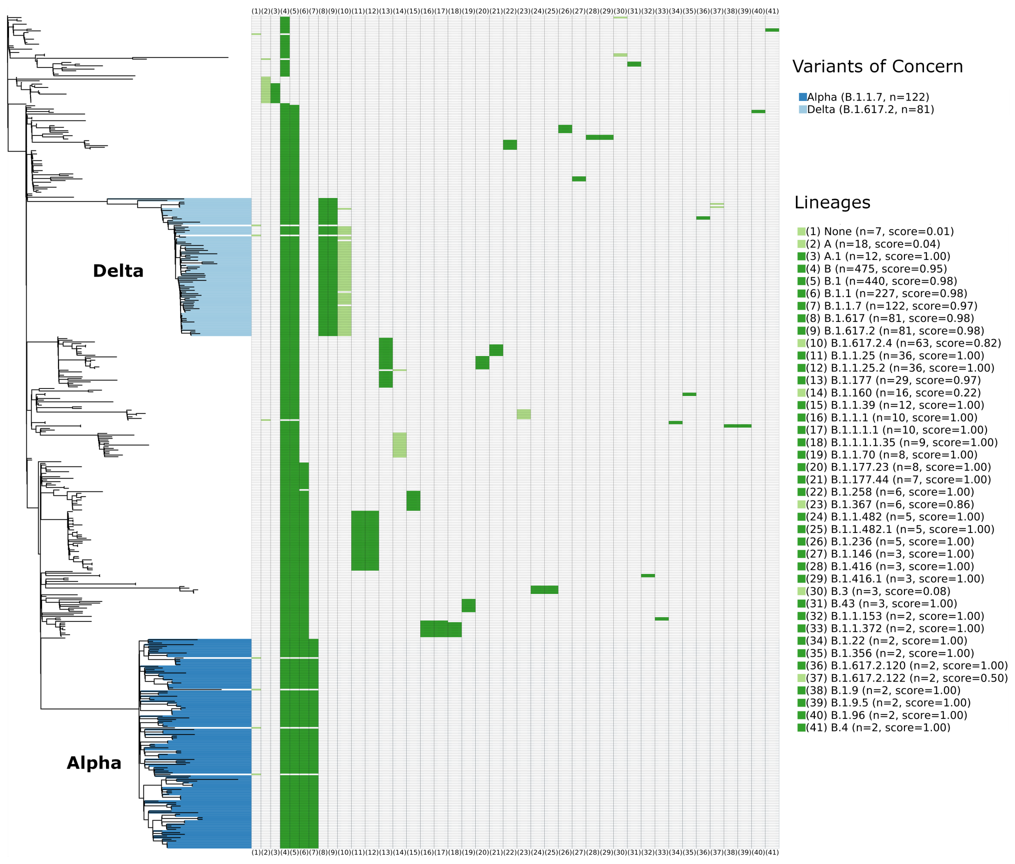


Figure C.1: A tree constructed using the Augur pipeline [3] on 500 SARS-CoV-2 genomes. The dendrogram is shown at the left (branch length in units of substitutions/site) with those SARS-CoV-2 genomes belonging to either the **Alpha** or **Delta** lineages coloured in shades of blue. The right displays a heatmap of the Pangolin lineages each genome belongs to, with one lineage per column and only including lineages with at least 2 genomes in the tree. Each lineage is assigned a score based on how many additional genomes are included underneath the most-recent common ancestor of all genomes within an individual lineage (scores range from 0 to 1, with 1 being the best possible score indicating that all genomes belonging to the particular Pangolin lineage are within a single monophyletic clade). The bars within the heatmap are coloured if the corresponding genome belongs to that lineage, with dark green indicating scores > 0.9 and light green indicating scores ≤ 0.9. There are 7 genomes which were unable to be assigned a lineage and these show up under the `(1) None` column in the heatmap. The median score across all lineages here is 1.00.

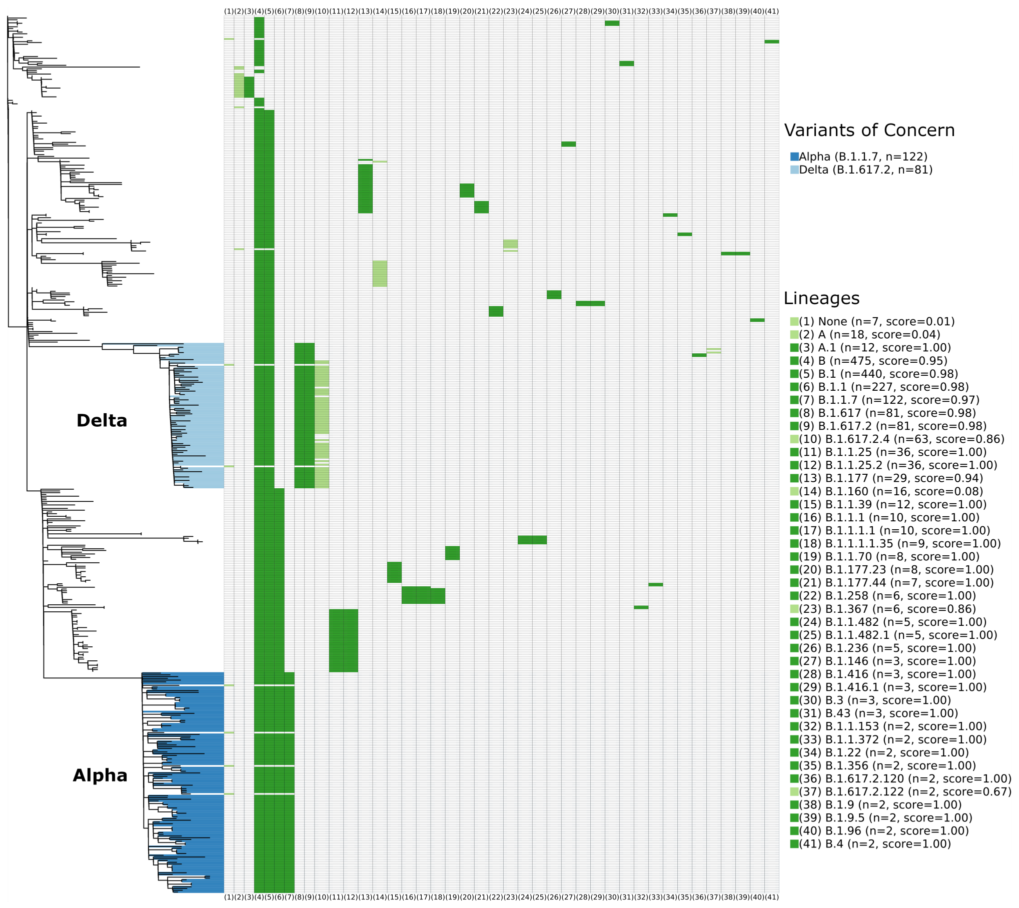


Figure C.2: A tree constructed using the maximum-likelihood method (using iqtree [17]) on 500 SARS-CoV-2 genomes indexed by GDI. The dendrogram is shown at the left (branch length in units of substitutions/site) with those SARS-CoV-2 genomes belonging to either the **Alpha** or **Delta** lineages coloured in shades of blue. The right displays a heatmap of the Pangolin lineages each genome belongs to, with one lineage per column and only including lineages with at least 2 genomes in the tree. Each lineage is assigned a score based on how many additional genomes are included underneath the most-recent common ancestor of all genomes within an individual lineage (scores range from 0 to 1, with 1 being the best possible score indicating that all genomes belonging to the particular Pangolin lineage are within a single monophyletic clade). The bars within the heatmap are coloured if the corresponding genome belongs to that lineage, with dark green indicating scores > 0.9 and light green indicating scores ≤ 0.9. There are 7 genomes which were unable to be assigned a lineage and these show up under the `(1) None` column in the heatmap. The median score across all lineages here is 1.00.

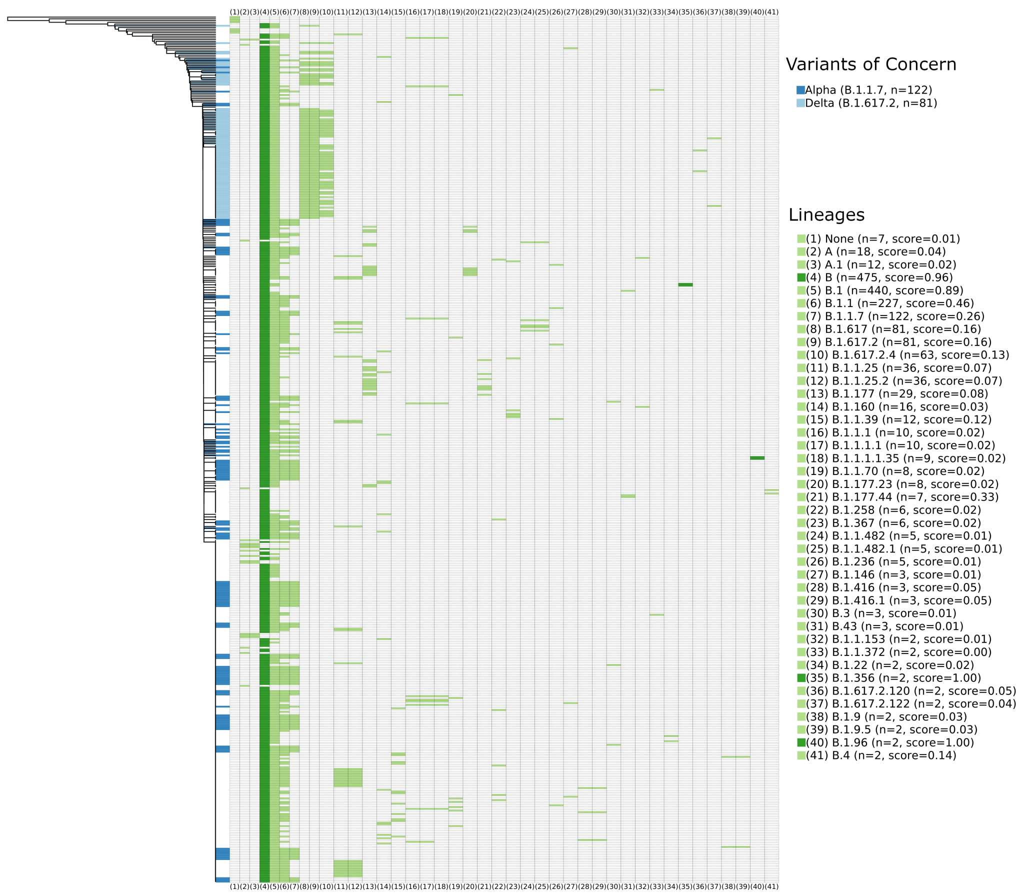


Figure C.3: A hierarchical cluster (shown as a dendrogram) constructed using pairwise sourmash [54] distances with a kmer size $k = 31$. The dendrogram is shown at the left with those SARS-CoV-2 genomes belonging to either the **Alpha** or **Delta** lineages coloured in shades of blue. The right displays a heatmap of the Pangolin lineages each genome belongs to, with one lineage per column and only including lineages with at least 2 genomes in the tree. Each lineage is assigned a score based on how many additional genomes are included underneath the most-recent common ancestor of all genomes within an individual lineage (scores range from 0 to 1, with 1 being the best possible score indicating that all genomes belonging to the particular Pangolin lineage are within a single monophyletic clade). The bars within the heatmap are coloured if the corresponding genome belongs to that lineage, with dark green indicating scores > 0.9 and light green indicating scores ≤ 0.9. There are 7 genomes which were unable to be assigned a lineage and these show up under the `(1) None` column in the heatmap. The median score across all lineages here is 0.034.

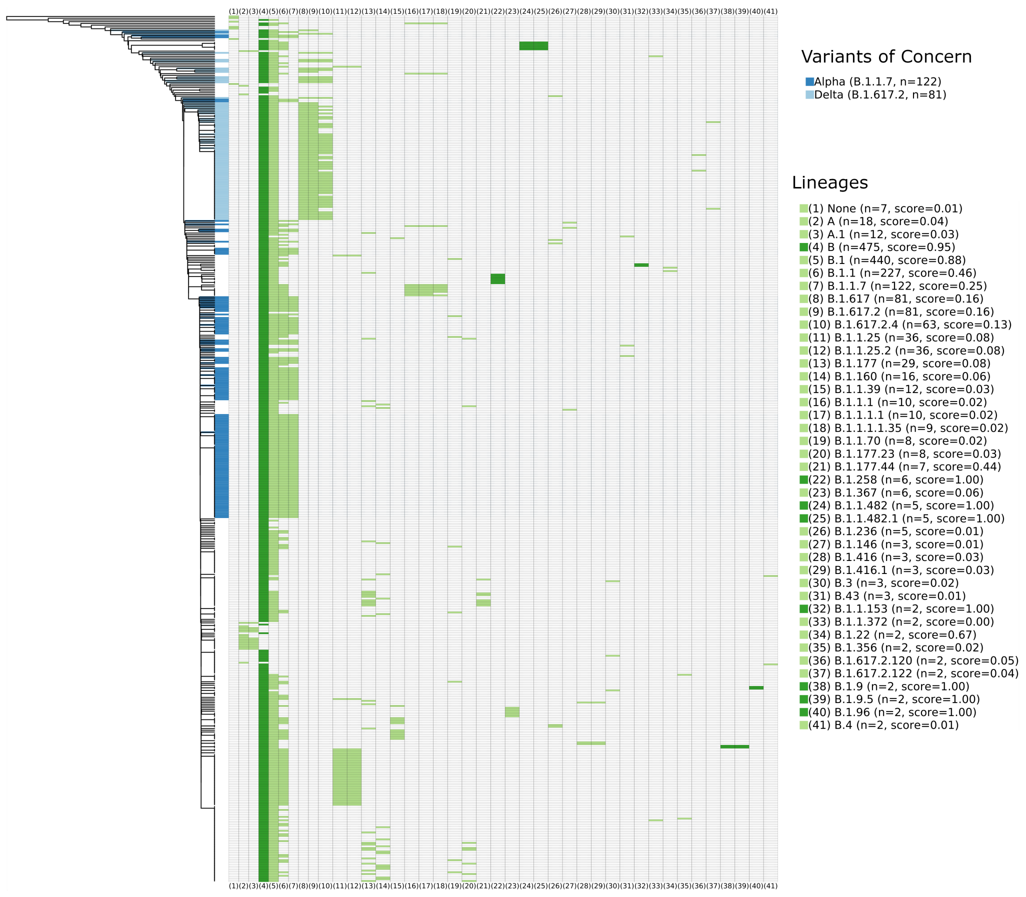


Figure C.4: A hierarchical cluster (shown as a dendrogram) constructed using pairwise sourmash [54] distances with a kmer size $k = 51$. The dendrogram is shown at the left with those SARS-CoV-2 genomes belonging to either the **Alpha** or **Delta** lineages coloured in shades of blue. The right displays a heatmap of the Pangolin lineages each genome belongs to, with one lineage per column and only including lineages with at least 2 genomes in the tree. Each lineage is assigned a score based on how many additional genomes are included underneath the most-recent common ancestor of all genomes within an individual lineage (scores range from 0 to 1, with 1 being the best possible score indicating that all genomes belonging to the particular Pangolin lineage are within a single monophyletic clade). The bars within the heatmap are coloured if the corresponding genome belongs to that lineage, with dark green indicating scores > 0.9 and light green indicating scores ≤ 0.9. There are 7 genomes which were unable to be assigned a lineage and these show up under the `(1) None` column in the heatmap. The median score across all lineages here is 0.058.

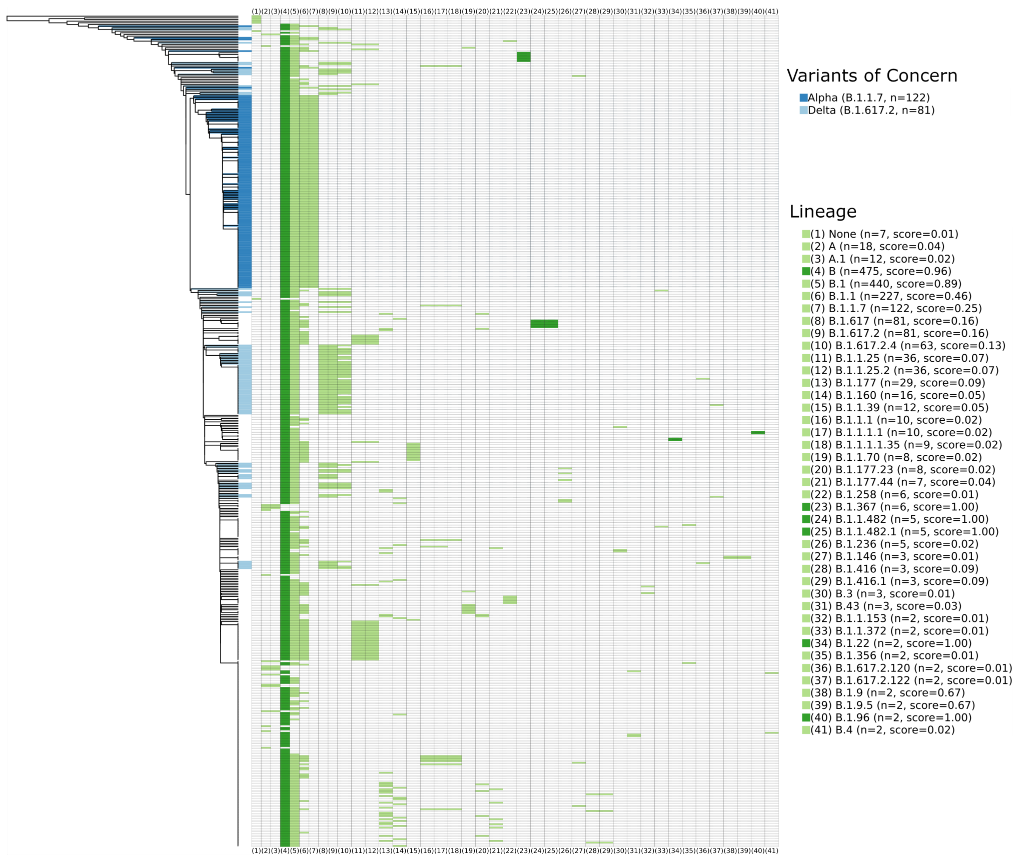


Figure C.5: A hierarchical cluster (shown as a dendrogram) constructed using pairwise sourmash [54] distances with a kmer size $k = 71$. The dendrogram is shown at the left with those SARS-CoV-2 genomes belonging to either the **Alpha** or **Delta** lineages coloured in shades of blue. The right displays a heatmap of the Pangolin lineages each genome belongs to, with one lineage per column and only including lineages with at least 2 genomes in the tree. Each lineage is assigned a score based on how many additional genomes are included underneath the most-recent common ancestor of all genomes within an individual lineage (scores range from 0 to 1, with 1 being the best possible score indicating that all genomes belonging to the particular Pangolin lineage are within a single monophyletic clade). The bars within the heatmap are coloured if the corresponding genome belongs to that lineage, with dark green indicating scores > 0.9 and light green indicating scores ≤ 0.9. There are 7 genomes which were unable to be assigned a lineage and these show up under the `(1) None` column in the heatmap. The median score across all lineages here is 0.049.

Appendix D

Reads

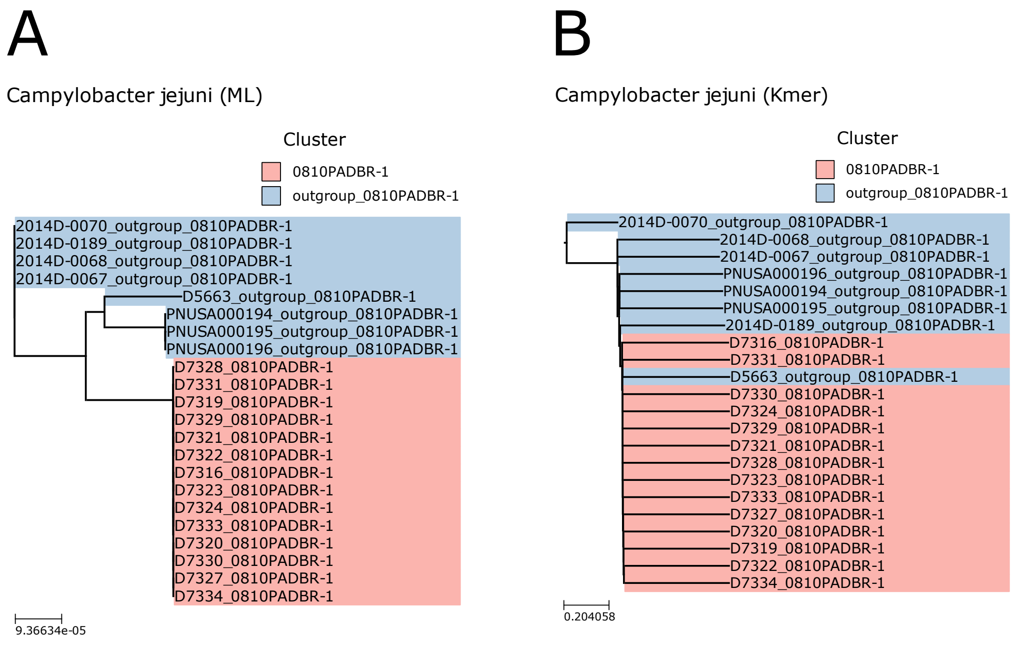


Figure D.1: A comparison between two trees constructed following the analysis and indexing of 22 *Campylobacter jejuni* genomes (processed from read data). The dataset is derived from a group of epidemiologically-linked bacterial isolates from an outbreak event (shown in red) alongside a group of outgroup genomes not linked to the outbreak (shown in blue). **(A)** A phylogenetic tree constructed using a maximum-likelihood method using nucleotide variants, which groups isolates derived from the outbreak into a monophyletic clade (reference genome not shown). **(B)** A tree depicting a single-linkage hierarchical clustering of shared kmers (derived from sourmash [54]), which does not group the isolates derived from the outbreak into a monophyletic clade.

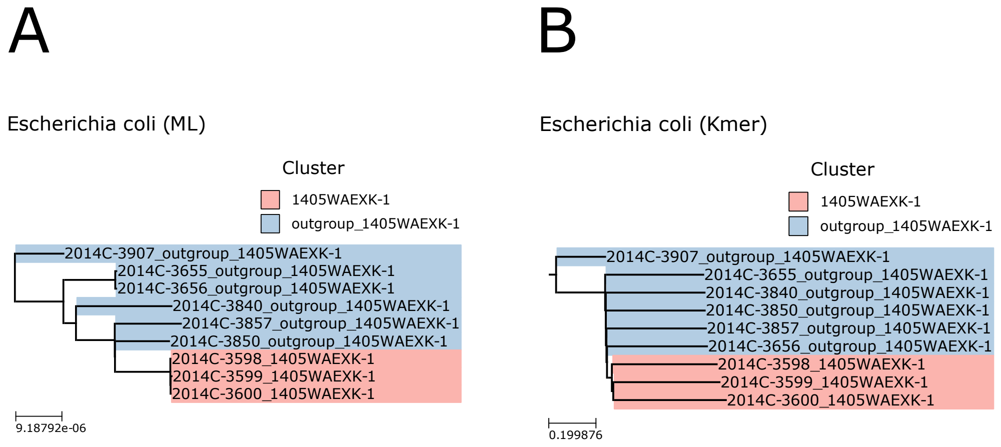

A
Escherichia coli (ML)
Cluster
1405WAEXK-1
outgroup_1405WAEXK-1
2014C-3907_outgroup_1405WAEXK-1
2014C-3655_outgroup_1405WAEXK-1
2014C-3656_outgroup_1405WAEXK-1
2014C-3840_outgroup_1405WAEXK-1
2014C-3857_outgroup_1405WAEXK-1
2014C-3850_outgroup_1405WAEXK-1
2014C-3598_1405WAEXK-1
2014C-3599_1405WAEXK-1
2014C-3600_1405WAEXK-1
9.18792e-06
B
Escherichia coli (Kmer)
Cluster
1405WAEXK-1
outgroup_1405WAEXK-1
2014C-3907_outgroup_1405WAEXK-1
2014C-3655_outgroup_1405WAEXK-1
2014C-3840_outgroup_1405WAEXK-1
2014C-3850_outgroup_1405WAEXK-1
2014C-3857_outgroup_1405WAEXK-1
2014C-3656_outgroup_1405WAEXK-1
2014C-3598_1405WAEXK-1
2014C-3599_1405WAEXK-1
2014C-3600_1405WAEXK-1
0.199876

Figure D.2: A comparison between two trees constructed following the analysis and indexing of 9 *Escherichia coli* genomes (processed from read data). The dataset is derived from a group of epidemiologically-linked bacterial isolates from an outbreak event (shown in red) alongside a group of outgroup genomes not linked to the outbreak (shown in blue). **(A)** A phylogenetic tree constructed using a maximum-likelihood method using nucleotide variants, which groups isolates derived from the outbreak into a monophyletic clade (reference genome not shown). **(B)** A tree depicting a single-linkage hierarchical clustering of shared kmers (derived from sourmash [54]), which does group the isolates derived from the outbreak into a monophyletic clade.

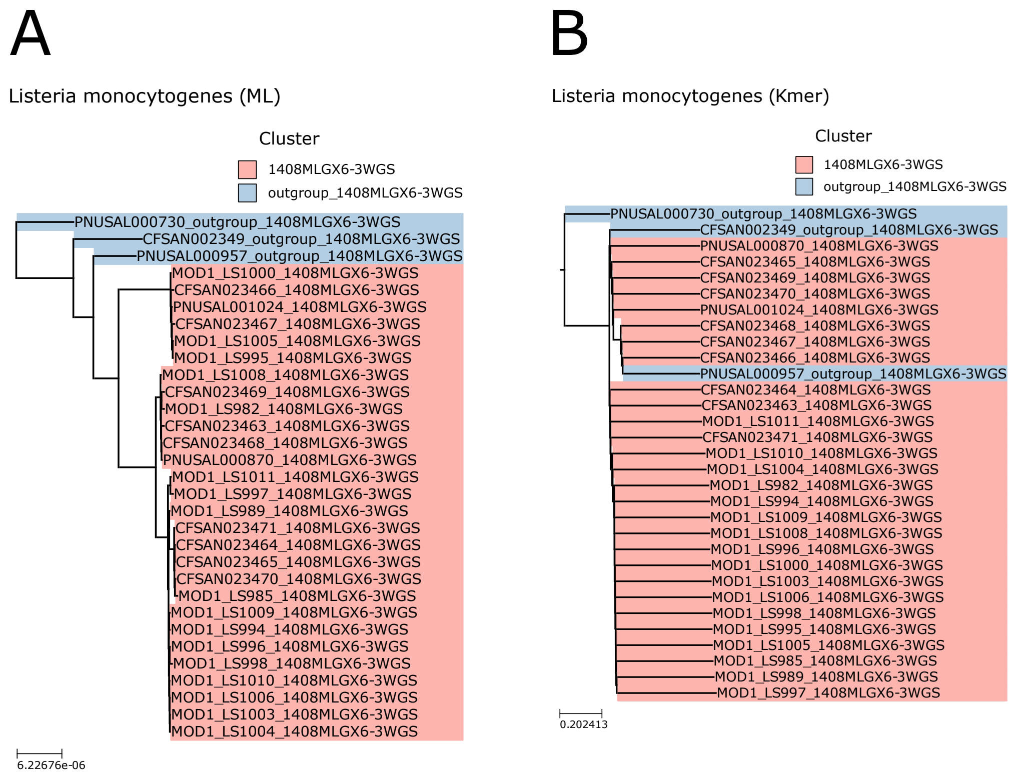

A
Listeria monocytogenes (ML)
Cluster
1408MLGX6-3WGS
outgroup_1408MLGX6-3WGS
PNUSAL000730_outgroup_1408MLGX6-3WGS
CFSAN002349_outgroup_1408MLGX6-3WGS
PNUSAL000957_outgroup_1408MLGX6-3WGS
MOD1_LS1000_1408MLGX6-3WGS
CFSAN023466_1408MLGX6-3WGS
PNUSAL001024_1408MLGX6-3WGS
CFSAN023467_1408MLGX6-3WGS
MOD1_LS1005_1408MLGX6-3WGS
MOD1_LS995_1408MLGX6-3WGS
MOD1_LS1008_1408MLGX6-3WGS
CFSAN023469_1408MLGX6-3WGS
MOD1_LS982_1408MLGX6-3WGS
CFSAN023463_1408MLGX6-3WGS
CFSAN023468_1408MLGX6-3WGS
PNUSAL000870_1408MLGX6-3WGS
MOD1_LS1011_1408MLGX6-3WGS
MOD1_LS997_1408MLGX6-3WGS
MOD1_LS989_1408MLGX6-3WGS
CFSAN023471_1408MLGX6-3WGS
CFSAN023464_1408MLGX6-3WGS
CFSAN023465_1408MLGX6-3WGS
CFSAN023470_1408MLGX6-3WGS
MOD1_LS985_1408MLGX6-3WGS
MOD1_LS1009_1408MLGX6-3WGS
MOD1_LS994_1408MLGX6-3WGS
MOD1_LS996_1408MLGX6-3WGS
MOD1_LS998_1408MLGX6-3WGS
MOD1_LS1010_1408MLGX6-3WGS
MOD1_LS1006_1408MLGX6-3WGS
MOD1_LS1003_1408MLGX6-3WGS
MOD1_LS1004_1408MLGX6-3WGS
6.22676e-06
B
Listeria monocytogenes (Kmer)
Cluster
1408MLGX6-3WGS
outgroup_1408MLGX6-3WGS
PNUSAL000730_outgroup_1408MLGX6-3WGS
CFSAN002349_outgroup_1408MLGX6-3WGS
PNUSAL000870_1408MLGX6-3WGS
CFSAN023465_1408MLGX6-3WGS
CFSAN023469_1408MLGX6-3WGS
CFSAN023470_1408MLGX6-3WGS
PNUSAL001024_1408MLGX6-3WGS
CFSAN023468_1408MLGX6-3WGS
CFSAN023467_1408MLGX6-3WGS
CFSAN023466_1408MLGX6-3WGS
PNUSAL000957_outgroup_1408MLGX6-3WGS
CFSAN023464_1408MLGX6-3WGS
CFSAN023463_1408MLGX6-3WGS
MOD1_LS1011_1408MLGX6-3WGS
CFSAN023471_1408MLGX6-3WGS
MOD1_LS1010_1408MLGX6-3WGS
MOD1_LS1004_1408MLGX6-3WGS
MOD1_LS982_1408MLGX6-3WGS
MOD1_LS994_1408MLGX6-3WGS
MOD1_LS1009_1408MLGX6-3WGS
MOD1_LS1008_1408MLGX6-3WGS
MOD1_LS996_1408MLGX6-3WGS
MOD1_LS1000_1408MLGX6-3WGS
MOD1_LS1003_1408MLGX6-3WGS
MOD1_LS1006_1408MLGX6-3WGS
MOD1_LS998_1408MLGX6-3WGS
MOD1_LS995_1408MLGX6-3WGS
MOD1_LS1005_1408MLGX6-3WGS
MOD1_LS985_1408MLGX6-3WGS
MOD1_LS989_1408MLGX6-3WGS
MOD1_LS997_1408MLGX6-3WGS
0.202413

Figure D.3: A comparison between two trees constructed following the analysis and indexing of 31 *Listeria monocytogenes* genomes (processed from read data). The dataset is derived from a group of epidemiologically-linked bacterial isolates from an outbreak event (shown in red) alongside a group of outgroup genomes not linked to the outbreak (shown in blue). **(A)** A phylogenetic tree constructed using a maximum-likelihood method using nucleotide variants, which groups isolates derived from the outbreak into a monophyletic clade (reference genome not shown). **(B)** A tree depicting a single-linkage hierarchical clustering of shared kmers (derived from sourmash [54]), which does not group the isolates derived from the outbreak into a monophyletic clade.

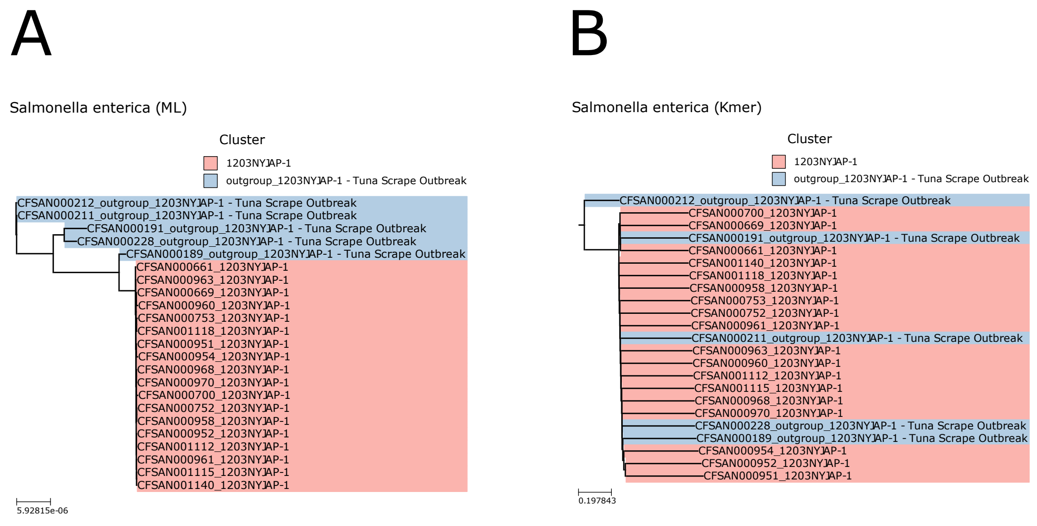


Figure D.4: A comparison between two trees constructed following the analysis and indexing of 23 *Salmonella enterica* genomes (processed from read data). The dataset is derived from a group of epidemiologically-linked bacterial isolates from an outbreak event (shown in red) alongside a group of outgroup genomes not linked to the outbreak (shown in blue). **(A)** A phylogenetic tree constructed using a maximum-likelihood method using nucleotide variants, which groups isolates derived from the outbreak into a monophyletic clade (reference genome not shown). **(B)** A tree depicting a single-linkage hierarchical clustering of shared kmers (derived from sourmash [54]), which does not group the isolates derived from the outbreak into a monophyletic clade.

Appendix E

Proof of requirements of cluster-to-tree scoring

This goes through a detailed proof of the three requirements (Requirements 1, 2, and 3) for Algorithm 3.1.

1. Since this algorithm returns the Jaccard index between two sets, then **Requirement 1** is satisfied since the Jaccard index will always be between 0 and 1.

2. To prove **Requirement 2** (highest score if and only if L are the leaves of a monophyletic clade) we can consider this statement in both directions.

    (a) Assume all genomes in L are leaves of a monophyletic clade. Then the descendants of the most recent common ancestor to L is just L; that is, $L_m = L$. From line 7 of Algorithm 3.1, the score is the Jaccard index of L and L_m, which must be 1 in this case (since the sets are equal).

(b) Now assume we have a score of $S(T, L) = 1$. Then $|L \cap L_m| = |L \cup L_m|$, which means that $L = L_m$, which means there is no additional genomes/leaves under the most recent common ancestor of L and so L consists of the leaves of a monophyletic clade.

(c) These two statements, together, prove **Requirement 2**.

3. **Requirement 3** holds in the limit as the size of the tree T increases. This is due to the following:

   (a) For a given set of leaves L, the lowest possible score is $|L|/N$ where N is the number of leaves in the tree T and where L is more than one leaf (that is $|L| > 1$). The argument is given below.

      i. More generally, for a given set of leaves L (where $|L| > 1$), the score is $|L \cap L_m|/|L \cup L_m|$ (line 7 in Algorithm 3.1).

      ii. This will achieve its lowest value when $|L \cap L_m|$ is minimized while $|L \cup L_m|$ is maximized.

      iii. Also, $L \subseteq L_m$ (L_m is the set of leaves under the most recent common ancestor to all leaves L, hence it must contain L).

      iv. So $|L \cap L_m| = |L|$ (since $L \subseteq L_m$ from above).

      v. So we can simplify the score for a given set of leaves L as $S = |L|/|L \cup L_m|$.

      vi. So to achieve the lowest possible score we need to minimize $|L|$ and maximize $|L \cup L_m|$.

      vii. $|L \cup L_m|$ is maximized for a given L when L_m contains the most number of items. This occurs when the most recent common ancestor to leaves L is the root of the tree and so $|L_m| = N$. Given that $L \subseteq L_m$ (from above), this implies that $|L \cup L_m| = N$.

viii. Therefore, for a given set of leaves L, the lowest possible score is $|L|/N$ (and is achieved when the most recent common ancestor to leaves L is the root of the tree).

(b) For the case of a single leaf $L = (l_1)$, the score will be $S(L, T) = 1$ (from lines 1-2 of Algorithm 3.1). Since this is the maximum score, we must rule out the case of a single leaf when looking for the minimum score.

(c) So, for any possible set of leaves L from tree T (where $|L| > 1$), the lowest possible score will therefore be achieved when $|L|$ is minimized (so we would minimize the value $|L|/N$).

(d) This would be achieved when we consider a pair of leaves $L = (l_1, l_2)$. So $|L| = 2$ and the minimal score would be $S(T, L) = 2/N$.

(e) Since $\lim_{N\to\infty} 2/N = 0$, then this requirement holds in the limit of an increasingly large tree T.

Bibliography

[1] E. M. Ribot, M. Freeman, K. B. Hise, and P. Gerner-Smidt, “PulseNet: Entering the Age of Next-Generation Sequencing,” *Foodborne Pathogens and Disease*, vol. 16, no. 7, pp. 451–456, Jul. 2019, doi: 10.1089/fpd.2019.2634. [Online]. Available: https://doi.org/10.1089/fpd.2019.2634

[2] B. Tolar, L. A. Joseph, M. N. Schroeder, S. Stroika, E. M. Ribot, K. B. Hise, and P. Gerner-Smidt, “An Overview of PulseNet USA Databases,” *Foodborne Pathogens and Disease*, vol. 16, no. 7, pp. 457–462, Jul. 2019, doi: 10.1089/fpd.2019.2637. [Online]. Available: https://doi.org/10.1089/fpd.2019.2637

[3] J. Hadfield, C. Megill, S. M. Bell, J. Huddleston, B. Potter, C. Callender, P. Sagulenko, T. Bedford, and R. A. Neher, “Nextstrain: real-time tracking of pathogen evolution,” *Bioinformatics*, vol. 34, no. 23, pp. 4121–4123, Dec. 2018, doi: 10.1093/bioinformatics/bty407. [Online]. Available: https://doi.org/10.1093/bioinformatics/bty407

[4] S. Khare, C. Gurry, L. Freitas, M. B Schultz, G. Bach, A. Diallo, N. Akite, J. Ho, R. TC Lee, W. Yeo, GISAID Core Curation Team, and S. Maurer-Stroh,

"GISAID's Role in Pandemic Response," *China CDC Weekly*, vol. 3, no. 49, pp. 1049–1051, 2021, doi: 10.46234/ccdcw2021.255. [Online]. Available: https://doi.org/10.46234/ccdcw2021.255

[5] C. Nadon, I. Van Walle, P. Gerner-Smidt, J. Campos, I. Chinen, J. Concepcion-Acevedo, B. Gilpin, A. M. Smith, K. M. Kam, E. Perez, E. Trees, K. Kubota, J. Takkinen, E. M. Nielsen, H. Carleton, and FWD-NEXT Expert Panel, "PulseNet International: Vision for the implementation of whole genome sequencing (WGS) for global food-borne disease surveillance," *Eurosurveillance*, vol. 22, no. 23, p. 30544, Jun. 2017, doi: 10.2807/1560-7917.ES.2017.22.23.30544. [Online]. Available: https://doi.org/10.2807/1560-7917.ES.2017.22.23.30544

[6] Á. O'Toole, E. Scher, A. Underwood, B. Jackson, V. Hill, J. T. McCrone, R. Colquhoun, C. Ruis, K. Abu-Dahab, B. Taylor, C. Yeats, L. du Plessis, D. Maloney, N. Medd, S. W. Attwood, D. M. Aanensen, E. C. Holmes, O. G. Pybus, and A. Rambaut, "Assignment of Epidemiological Lineages in an Emerging Pandemic Using the Pangolin Tool," *Virus Evolution*, p. veab064, Jul. 2021, doi: 10.1093/ve/veab064. [Online]. Available: https://doi.org/10.1093/ve/veab064

[7] World Health Organization, "Tracking SARS-CoV-2 variants," 2022, Accessed: August 24, 2022. [Online]. Available: https://www.who.int/en/activities/tracking-SARS-CoV-2-variants/

[8] S. Büttcher, C. Clarke, and G. Cormack, *Information Retrieval: Implementing*

and Evaluating Search Engines. MIT Press, 2010. [Online]. Available: https://books.google.ca/books?id=epD6AQAAQBAJ

[9] S. Kannan, P. Shaik Syed Ali, and A. Sheeza, "Omicron (B.1.1.529) – variant of concern – molecular profile and epidemiology: a mini review," *European Review for Medical and Pharmacological Sciences*, vol. 25, no. 24, pp. 8019–8022, Dec. 2021, doi: 10.26355/eurrev_202112_27653. [Online]. Available: https://doi.org/10.26355/eurrev_202112_27653

[10] I. Karsch-Mizrachi, T. Takagi, G. Cochrane, and on behalf of the International Nucleotide Sequence Database Collaboration, "The international nucleotide sequence database collaboration," *Nucleic Acids Research*, vol. 46, no. D1, pp. D48–D51, Jan. 2018, doi: 10.1093/nar/gkx1097. [Online]. Available: https://doi.org/10.1093/nar/gkx1097

[11] S. Elbe and G. Buckland-Merrett, "Data, disease and diplomacy: GISAID's innovative contribution to global health: Data, Disease and Diplomacy," *Global Challenges*, vol. 1, no. 1, pp. 33–46, Jan. 2017, doi: 10.1002/gch2.1018. [Online]. Available: https://doi.org/10.1002/gch2.1018

[12] P. Bradley, H. C. den Bakker, E. P. C. Rocha, G. McVean, and Z. Iqbal, "Ultrafast search of all deposited bacterial and viral genomic data," *Nature Biotechnology*, vol. 37, no. 2, pp. 152–159, Feb. 2019, doi: 10.1038/s41587-018-0010-1. [Online]. Available: https://doi.org/10.1038/s41587-018-0010-1

[13] A. Petkau, P. Mabon, C. Sieffert, N. C. Knox, J. Cabral, M. Iskander,

M. Iskander, K. Weedmark, R. Zaheer, L. S. Katz, C. Nadon, A. Reimer, E. Taboada, R. G. Beiko, W. Hsiao, F. Brinkman, M. Graham, and G. Van Domselaar, "SNVPhyl: a single nucleotide variant phylogenomics pipeline for microbial genomic epidemiology," *Microbial Genomics*, vol. 3, no. 6, Jun. 2017, doi: 10.1099/mgen.0.000116. [Online]. Available: https://doi.org/10.1099/mgen.0.000116

[14] C. E. Yoshida, P. Kruczkiewicz, C. R. Laing, E. J. Lingohr, V. P. J. Gannon, J. H. E. Nash, and E. N. Taboada, "The Salmonella In Silico Typing Resource (SISTR): An Open Web-Accessible Tool for Rapidly Typing and Subtyping Draft Salmonella Genome Assemblies," *PLOS ONE*, vol. 11, no. 1, p. e0147101, Jan. 2016, doi: 10.1371/journal.pone.0147101. [Online]. Available: https://doi.org/10.1371/journal.pone.0147101

[15] M. Inouye, H. Dashnow, L.-A. Raven, M. B. Schultz, B. J. Pope, T. Tomita, J. Zobel, and K. E. Holt, "SRST2: Rapid genomic surveillance for public health and hospital microbiology labs," *Genome Medicine*, vol. 6, no. 11, p. 90, Dec. 2014, doi: 10.1186/s13073-014-0090-6. [Online]. Available: https://doi.org/10.1186/s13073-014-0090-6

[16] T. Lynch, A. Petkau, N. Knox, M. Graham, and G. Van Domselaar, "A Primer on Infectious Disease Bacterial Genomics," *Clinical Microbiology Reviews*, vol. 29, no. 4, pp. 881–913, Oct. 2016, doi: 10.1128/CMR.00001-16. [Online]. Available: https://doi.org/10.1128/CMR.00001-16

[17] B. Q. Minh, H. A. Schmidt, O. Chernomor, D. Schrempf, M. D.

Woodhams, A. von Haeseler, and R. Lanfear, “IQ-TREE 2: New Models and Efficient Methods for Phylogenetic Inference in the Genomic Era,” *Molecular Biology and Evolution*, vol. 37, no. 5, pp. 1530–1534, May 2020, doi: 10.1093/molbev/msaa015. [Online]. Available: https://doi.org/10.1093/molbev/msaa015

[18] M. J. Sanderson, M. M. McMahon, and M. Steel, “Terraces in Phylogenetic Tree Space,” *Science*, vol. 333, no. 6041, pp. 448–450, Jul. 2011, doi: 10.1126/science.1206357. [Online]. Available: https://doi.org/10.1126/science.1206357

[19] E. J. Griffiths, R. E. Timme, A. J. Page, N.-F. Alikhan, D. Fornika, F. Maguire, C. I. Mendes, S. H. Tausch, A. Black, T. R. Connor, G. H. Tyson, D. M. Aanensen, B. Alcock, J. Campos, A. Christoffels, A. Gonçalves da Silva, E. Hodcroft, W. W. Hsiao, L. S. Katz, S. M. Nicholls, P. E. Oluniyi, I. B. Olawoye, A. R. Raphenya, A. T. R. Vasconcelos, A. A. Witney, and D. R. MacCannell, “The PHA4GE SARS-CoV-2 Contextual Data Specification for Open Genomic Epidemiology,” *Preprints*, Aug. 2020, doi: 10.20944/preprints202008.0220.v1. [Online]. Available: https://doi.org/10.20944/preprints202008.0220.v1

[20] L. Uelze, J. Grützke, M. Borowiak, J. A. Hammerl, K. Juraschek, C. Deneke, S. H. Tausch, and B. Malorny, “Typing methods based on whole genome sequencing data,” *One Health Outlook*, vol. 2, no. 1,

p. 3, Dec. 2020, doi: 10.1186/s42522-020-0010-1. [Online]. Available: https://doi.org/10.1186/s42522-020-0010-1

[21] J. Huerta-Cepas, F. Serra, and P. Bork, "ETE 3: Reconstruction, Analysis, and Visualization of Phylogenomic Data," *Molecular Biology and Evolution*, vol. 33, no. 6, pp. 1635–1638, Jun. 2016, doi: 10.1093/molbev/msw046. [Online]. Available: https://doi.org/10.1093/molbev/msw046

[22] Wes McKinney, "Data Structures for Statistical Computing in Python," in *Proceedings of the 9th Python in Science Conference*, Stéfan van der Walt and Jarrod Millman, Eds., 2010, pp. 56 – 61. [Online]. Available: https://doi.org/10.25080/Majora-92bf1922-00a

[23] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, C. Willing, and Jupyter Development Team, "Jupyter notebooks - a publishing format for reproducible computational workflows." in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, 2016, doi: 10.3233/978-1-61499-649-1-87. [Online]. Available: https://doi.org/10.3233/978-1-61499-649-1-87

[24] N. Tonellotto, C. Macdonald, and I. Ounis, "Efficient Query Processing for Scalable Web Search," *Foundations and Trends® in Information Retrieval*, vol. 12, no. 4-5, pp. 319–500, 2018, doi: 10.1561/1500000057. [Online]. Available: https://doi.org/10.1561/1500000057

[25] A. Saxena, M. Prasad, A. Gupta, N. Bharill, O. P. Patel, A. Tiwari,

M. J. Er, W. Ding, and C.-T. Lin, "A review of clustering techniques and developments," *Neurocomputing*, vol. 267, pp. 664–681, Dec. 2017, doi: 10.1016/j.neucom.2017.06.053. [Online]. Available: https://doi.org/10.1016/j.neucom.2017.06.053

[26] F. Nielsen, *Hierarchical Clustering.* Cham: Springer International Publishing, 2016, pp. 195–211. [Online]. Available: https://doi.org/10.1007/978-3-319-21903-5_8

[27] T. M. Ghazal, M. Zahid Hussain, R. A. Said, A. Nadeem, M. Kamrul Hasan, M. Ahmad, M. Adnan Khan, and M. Tahir Naseem, "Performances of K-Means Clustering Algorithm with Different Distance Metrics," *Intelligent Automation & Soft Computing*, vol. 29, no. 3, pp. 735–742, 2021, doi: 10.32604/iasc.2021.019067. [Online]. Available: https://doi.org/10.32604/iasc.2021.019067

[28] P. R. Wielinga, R. S. Hendriksen, F. M. Aarestrup, O. Lund, S. L. Smits, M. P. G. Koopmans, and J. Schlundt, "Global Microbial Identifier," in *Applied Genomics of Foodborne Pathogens*, X. Deng, H. C. den Bakker, and R. S. Hendriksen, Eds. Cham: Springer International Publishing, 2017, pp. 13–31, doi: 10.1007/978-3-319-43751-4_2. [Online]. Available: https://doi.org/10.1007/978-3-319-43751-4_2

[29] L. M. Prescott, J. P. Harley, and D. A. Klein, *Microbiology*, 6th ed. Dubuque, IA: McGraw-Hill Higher Education, 2005.

[30] J. Besser, H. Carleton, P. Gerner-Smidt, R. Lindsey, and E. Trees,

"Next-generation sequencing technologies and their application to the study and control of bacterial infections," *Clinical Microbiology and Infection*, vol. 24, no. 4, pp. 335–341, Apr. 2018, doi: 10.1016/j.cmi.2017.10.013. [Online]. Available: https://doi.org/10.1016/j.cmi.2017.10.013

[31] A. Souvorov, R. Agarwala, and D. J. Lipman, "SKESA: strategic k-mer extension for scrupulous assemblies," *Genome Biology*, vol. 19, no. 1, p. 153, Dec. 2018, doi: 10.1186/s13059-018-1540-z. [Online]. Available: https://doi.org/10.1186/s13059-018-1540-z

[32] P. J. A. Cock, C. J. Fields, N. Goto, M. L. Heuer, and P. M. Rice, "The sanger FASTQ file format for sequences with quality scores, and the solexa/illumina FASTQ variants," *Nucleic Acids Research*, vol. 38, no. 6, pp. 1767–1771, 2010, doi: 10.1093/nar/gkp1137. [Online]. Available: https://doi.org/10.1093/nar/gkp1137

[33] E. S. Lander and M. S. Waterman, "Genomic mapping by fingerprinting random clones: A mathematical analysis," *Genomics*, vol. 2, no. 3, pp. 231–239, Apr. 1988, doi: 10.1016/0888-7543(88)90007-9. [Online]. Available: https://doi.org/10.1016/0888-7543(88)90007-9

[34] J. T. Simpson and M. Pop, "The Theory and Practice of Genome Sequence Assembly," *Annual Review of Genomics and Human Genetics*, vol. 16, no. 1, pp. 153–172, Aug. 2015, doi: 10.1146/annurev-genom-090314-050032. [Online]. Available: https://doi.org/10.1146/annurev-genom-090314-050032

[35] A. Bankevich, S. Nurk, D. Antipov, A. A. Gurevich, M. Dvorkin, A. S.

Kulikov, V. M. Lesin, S. I. Nikolenko, S. Pham, A. D. Prjibelski, A. V. Pyshkin, A. V. Sirotkin, N. Vyahhi, G. Tesler, M. A. Alekseyev, and P. A. Pevzner, "SPAdes: A New Genome Assembly Algorithm and Its Applications to Single-Cell Sequencing," *Journal of Computational Biology*, vol. 19, no. 5, pp. 455–477, May 2012, doi: 10.1089/cmb.2012.0021. [Online]. Available: https://doi.org/10.1089/cmb.2012.0021

[36] H. Li and R. Durbin, "Fast and accurate short read alignment with Burrows-Wheeler transform," *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, Jul. 2009, doi: 10.1093/bioinformatics/btp324. [Online]. Available: https://doi.org/10.1093/bioinformatics/btp324

[37] H. Li, "Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM," *arXiv:1303.3997 [q-bio]*, May 2013, doi: 10.48550/arXiv.1303.3997. [Online]. Available: https://doi.org/10.48550/arXiv.1303.3997

[38] B. Langmead and S. L. Salzberg, "Fast gapped-read alignment with Bowtie 2," *Nature Methods*, vol. 9, no. 4, pp. 357–359, Apr. 2012, doi: 10.1038/nmeth.1923. [Online]. Available: https://doi.org/10.1038/nmeth.1923

[39] H. Li, "Minimap2: pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, Sep. 2018, doi: 10.1093/bioinformatics/bty191. [Online]. Available: https://doi.org/10.1093/bioinformatics/bty191

[40] M. W. Allard, E. Strain, D. Melka, K. Bunning, S. M. Musser, E. W. Brown,

and R. Timme, “Practical value of food pathogen traceability through building a whole-genome sequencing network and database,” *Journal of Clinical Microbiology*, vol. 54, no. 8, pp. 1975–1983, 2016, doi: 10.1128/JCM.00081-16. [Online]. Available: https://doi.org/10.1128/JCM.00081-16

[41] A. Rambaut, E. C. Holmes, Á. O’Toole, V. Hill, J. T. McCrone, C. Ruis, L. du Plessis, and O. G. Pybus, “A dynamic nomenclature proposal for SARS-CoV-2 lineages to assist genomic epidemiology,” *Nature Microbiology*, vol. 5, no. 11, pp. 1403–1407, Nov. 2020, doi: 10.1038/s41564-020-0770-5. [Online]. Available: https://doi.org/10.1038/s41564-020-0770-5

[42] J. B. Holmes, E. Moyer, L. Phan, D. Maglott, and B. Kattman, “SPDI: data model for variants and applications at NCBI,” *Bioinformatics*, vol. 36, no. 6, pp. 1902–1907, Mar. 2020, doi: 10.1093/bioinformatics/btz856. [Online]. Available: https://doi.org/10.1093/bioinformatics/btz856

[43] D. H. Spencer, B. Zhang, and J. Pfeifer, “Single Nucleotide Variant Detection Using Next Generation Sequencing,” in *Clinical Genomics.* Elsevier, 2015, pp. 109–127, doi: 10.1016/B978-0-12-404748-8.00008-3. [Online]. Available: https://doi.org/10.1016/B978-0-12-404748-8.00008-3

[44] S. Davis, J. B. Pettengill, Y. Luo, J. Payne, A. Shpuntoff, H. Rand, and E. Strain, “CFSAN SNP Pipeline: an automated method for constructing SNP matrices from next-generation sequence data,” *PeerJ Computer Science*, vol. 1, p. e20, Aug. 2015, doi: 10.7717/peerj-cs.20. [Online]. Available: https://doi.org/10.7717/peerj-cs.20

[45] L. S. Katz, T. Griswold, A. J. Williams-Newkirk, D. Wagner, A. Petkau, C. Sieffert, G. Van Domselaar, X. Deng, and H. A. Carleton, "A Comparative Analysis of the Lyve-SET Phylogenomics Pipeline for Genomic Epidemiology of Foodborne Pathogens," *Frontiers in Microbiology*, vol. 8, Mar. 2017, doi: 10.3389/fmicb.2017.00375. [Online]. Available: https://doi.org/10.3389/fmicb.2017.00375

[46] T. Seemann, "snippy: fast bacterial variant calling form ngs reads," 2015, Accessed: August 24, 2022. [Online]. Available: https://github.com/tseemann/snippy

[47] T. Dallman, P. Ashton, U. Schafer, A. Jironkin, A. Painset, S. Shaaban, H. Hartman, R. Myers, A. Underwood, C. Jenkins, and K. Grant, "SnapperDB: a database solution for routine sequencing analysis of bacterial isolates," *Bioinformatics*, vol. 34, no. 17, pp. 3028–3029, Sep. 2018, doi: 10.1093/bioinformatics/bty212. [Online]. Available: https://doi.org/10.1093/bioinformatics/bty212

[48] A. T. Chen, K. Altschuler, S. H. Zhan, Y. A. Chan, and B. E. Deverman, "COVID-19 CG enables SARS-CoV-2 mutation and lineage tracking by locations and dates of interest," *eLife*, vol. 10, p. e63409, 2021, doi: 10.7554/eLife.63409. [Online]. Available: https://doi.org/10.7554/eLife.63409

[49] G. Tsueng, J. L. Mullen, M. Alkuzweny, M. Cano, B. Rush, E. Haag, Outbreak Curators, A. A. Latif, X. Zhou, Z. Qian, E. Hufbauer, M. Zeller, K. G. Andersen, C. Wu, A. I. Su, K. Gangavarapu, and L. D. Hughes, "Outbreak.info

research library: A standardized, searchable platform to discover and explore covid-19 resources," *bioRxiv*, 2022, doi: 10.1101/2022.01.20.477133. [Online]. Available: https://doi.org/10.1101/2022.01.20.477133

[50] B. D. Ondov, T. J. Treangen, P. Melsted, A. B. Mallonee, N. H. Bergman, S. Koren, and A. M. Phillippy, "Mash: fast genome and metagenome distance estimation using MinHash," *Genome Biology*, vol. 17, no. 1, p. 132, Dec. 2016, doi: 10.1186/s13059-016-0997-x. [Online]. Available: https://doi.org/10.1186/s13059-016-0997-x

[51] A. Broder, "On the resemblance and containment of documents," in *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171).* Salerno, Italy: IEEE Comput. Soc, 1998, pp. 21–29, doi: 10.1109/SEQUEN.1997.666900. [Online]. Available: https://doi.org/10.1109/SEQUEN.1997.666900

[52] P. Jaccard, "Etude comparative de la distribution florale dans une portion des alpes et des jura," *Bull Soc Vaudoise Sci Nat*, vol. 37, pp. 547–579, 1901. [Online]. Available: https://cir.nii.ac.jp/crid/1570009750546179712

[53] L. Katz, T. Griswold, S. Morrison, J. Caravas, S. Zhang, H. den Bakker, X. Deng, and H. Carleton, "Mashtree: a rapid comparison of whole genome sequence files," *Journal of Open Source Software*, vol. 4, no. 44, p. 1762, Dec. 2019, doi: 10.21105/joss.01762. [Online]. Available: https://doi.org/10.21105/joss.01762

[54] C. Titus Brown and L. Irber, "sourmash: a library for MinHash sketching of

DNA," *The Journal of Open Source Software*, vol. 1, no. 5, p. 27, Sep. 2016, doi: 10.21105/joss.00027. [Online]. Available: https://doi.org/10.21105/joss.00027

[55] B. Goodwin, M. Hopcroft, D. Luu, A. Clemmer, M. Curmei, S. Elnikety, and Y. He, "BitFunnel: Revisiting Signatures for Search," in *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval - SIGIR '17*. Shinjuku, Tokyo, Japan: ACM Press, 2017, pp. 605–614, doi: 10.1145/3077136.3080789. [Online]. Available: https://doi.org/10.1145/3077136.3080789

[56] T. Bingmann, P. Bradley, F. Gauger, and Z. Iqbal, "COBS: A Compact Bit-Sliced Signature Index," in *String Processing and Information Retrieval*, N. R. Brisaboa and S. J. Puglisi, Eds. Cham: Springer International Publishing, 2019, vol. 11811, pp. 285–303, doi: 10.1007/978-3-030-32686-9_21. [Online]. Available: https://doi.org/10.1007/978-3-030-32686-9_21

[57] G. Labbé, P. Kruczkiewicz, J. Robertson, P. Mabon, J. Schonfeld, D. Kein, M. A. Rankin, M. Gopez, D. Hole, D. Son, N. Knox, C. R. Laing, K. Bessonov, E. N. Taboada, C. Yoshida, K. Ziebell, A. Nichani, R. P. Johnson, G. Van Domselaar, and J. H. E. Nash, "Rapid and accurate SNP genotyping of clonal bacterial pathogens with BioHansel," *Microbial Genomics*, vol. 7, no. 9, 2021, doi: 10.1099/mgen.0.000651. [Online]. Available: https://doi.org/10.1099/mgen.0.000651

[58] M. C. J. Maiden, J. A. Bygraves, E. Feil, G. Morelli, J. E. Russell, R. Urwin, Q. Zhang, J. Zhou, K. Zurth, D. A. Caugant, I. M. Feavers,

M. Achtman, and B. G. Spratt, “Multilocus sequence typing: A portable approach to the identification of clones within populations of pathogenic microorganisms,” *Proceedings of the National Academy of Sciences*, vol. 95, no. 6, pp. 3140–3145, Mar. 1998, doi: 10.1073/pnas.95.6.3140. [Online]. Available: https://doi.org/10.1073/pnas.95.6.3140

[59] M. C. J. Maiden, M. J. J. van Rensburg, J. E. Bray, S. G. Earle, S. A. Ford, K. A. Jolley, and N. D. McCarthy, “MLST revisited: the gene-by-gene approach to bacterial genomics,” *Nature Reviews Microbiology*, vol. 11, no. 10, pp. 728–736, Oct. 2013, doi: 10.1038/nrmicro3093. [Online]. Available: https://doi.org/10.1038/nrmicro3093

[60] P. Feijao, H.-T. Yao, D. Fornika, J. Gardy, W. Hsiao, C. Chauve, and L. Chindelevitch, “MentaLiST – A fast MLST caller for large MLST schemes,” *Microbial Genomics*, vol. 4, no. 2, Feb. 2018, doi: 10.1099/mgen.0.000146. [Online]. Available: https://doi.org/10.1099/mgen.0.000146

[61] H. F. Espitia-Navarro, A. T. Chande, S. D. Nagar, H. Smith, I. K. Jordan, and L. Rishishwar, “STing: accurate and ultrafast genomic profiling with exact sequence matches,” *Nucleic Acids Research*, vol. 48, no. 14, pp. 7681–7689, 07 2020, doi: 10.1093/nar/gkaa566. [Online]. Available: https://doi.org/10.1093/nar/gkaa566

[62] K. A. Jolley, J. E. Bray, and M. C. J. Maiden, “Open-access bacterial population genomics: BIGSdb software, the PubMLST.org website and their applications,” *Wellcome Open Research*, vol. 3, p.

124, 2018, doi: 10.12688/wellcomeopenres.14826.1. [Online]. Available: https://doi.org/10.12688/wellcomeopenres.14826.1

[63] K. A. Jolley and M. C. Maiden, "BIGSdb: Scalable analysis of bacterial genome variation at the population level," *BMC Bioinformatics*, vol. 11, no. 1, p. 595, Dec. 2010, doi: 10.1186/1471-2105-11-595. [Online]. Available: https://doi.org/10.1186/1471-2105-11-595

[64] N.-F. Alikhan, Z. Zhou, M. J. Sergeant, and M. Achtman, "A genomic overview of the population structure of *Salmonella*," *PLOS Genetics*, vol. 14, no. 4, p. e1007261, Apr. 2018, doi: 10.1371/journal.pgen.1007261. [Online]. Available: https://doi.org/10.1371/journal.pgen.1007261

[65] Z. Zhou, N.-F. Alikhan, K. Mohamed, Y. Fan, the Agama Study Group, and M. Achtman, "The EnteroBase user's guide, with case studies on *Salmonella* transmissions, *Yersinia pestis* phylogeny, and *Escherichia* core genomic diversity," *Genome Research*, vol. 30, no. 1, pp. 138–152, Jan. 2020, doi: 10.1101/gr.251678.119. [Online]. Available: https://doi.org/10.1101/gr.251678.119

[66] Z. Yang and B. Rannala, "Molecular phylogenetics: principles and practice," *Nature Reviews Genetics*, vol. 13, no. 5, pp. 303–314, May 2012, doi: 10.1038/nrg3186. [Online]. Available: https://doi.org/10.1038/nrg3186

[67] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison, Eds., *Biological sequence analysis: probabalistic models of proteins and nucleic acids.* Cambridge, UK : New York: Cambridge University Press, 1998.

[68] P. Kapli, Z. Yang, and M. J. Telford, “Phylogenetic tree building in the genomic age,” *Nature Reviews Genetics*, vol. 21, no. 7, pp. 428–444, Jul. 2020, doi: 10.1038/s41576-020-0233-0. [Online]. Available: https://doi.org/10.1038/s41576-020-0233-0

[69] T. H. Jukes and C. R. Cantor, “Evolution of protein molecules,” *Mammalian protein metabolism*, vol. 3, pp. 21–132, 1969.

[70] M. Kimura, “A simple method for estimating evolutionary rates of base substitutions through comparative studies of nucleotide sequences,” *Journal of Molecular Evolution*, vol. 16, no. 2, pp. 111–120, Jun. 1980, doi: 10.1007/BF01731581. [Online]. Available: https://doi.org/10.1007/BF01731581

[71] M. Hasegawa, H. Kishino, and T.-a. Yano, “Dating of the human-ape splitting by a molecular clock of mitochondrial DNA,” *Journal of Molecular Evolution*, vol. 22, no. 2, pp. 160–174, Oct. 1985, doi: 10.1007/BF02101694. [Online]. Available: https://doi.org/10.1007/BF02101694

[72] S. Tavaré *et al.*, “Some probabilistic and statistical problems in the analysis of dna sequences,” *Lectures on mathematics in the life sciences*, vol. 17, no. 2, pp. 57–86, 1986.

[73] M. Balaban, N. Moshiri, U. Mai, X. Jia, and S. Mirarab, “TreeCluster: Clustering biological sequences using phylogenetic trees,” *PLOS ONE*, vol. 14, no. 8, pp. 1–20, 08 2019, doi: 10.1371/journal.pone.0221068. [Online]. Available: https://doi.org/10.1371/journal.pone.0221068

[74] M. Ragonnet-Cronin, E. Hodcroft, S. Hué, E. Fearnhill, V. Delpech, A. J. L. Brown, and S. Lycett, "Automated analysis of phylogenetic clusters," *BMC Bioinformatics*, vol. 14, no. 1, p. 317, Dec. 2013, doi: 10.1186/1471-2105-14-317. [Online]. Available: https://doi.org/10.1186/1471-2105-14-317

[75] P. Danecek, A. Auton, G. Abecasis, C. A. Albers, E. Banks, M. A. DePristo, R. E. Handsaker, G. Lunter, G. T. Marth, S. T. Sherry, G. McVean, R. Durbin, and 1000 Genomes Project Analysis Group, "The variant call format and VCFtools," *Bioinformatics*, vol. 27, no. 15, pp. 2156–2158, Aug. 2011, doi: 10.1093/bioinformatics/btr330. [Online]. Available: https://doi.org/10.1093/bioinformatics/btr330

[76] A. H. Wagner, L. Babb, G. Alterovitz, M. Baudis, M. Brush, D. L. Cameron, M. Cline, M. Griffith, O. L. Griffith, S. E. Hunt, D. Kreda, J. M. Lee, S. Li, J. Lopez, E. Moyer, T. Nelson, R. Y. Patel, K. Riehle, P. N. Robinson, S. Rynearson, H. Schuilenburg, K. Tsukanov, B. Walsh, M. Konopko, H. L. Rehm, A. D. Yates, R. R. Freimuth, and R. K. Hart, "The GA4GH Variation Representation Specification: A computational framework for variation representation and federated identification," *Cell Genomics*, vol. 1, no. 2, p. 100027, Nov. 2021, doi: 10.1016/j.xgen.2021.100027. [Online]. Available: https://doi.org/10.1016/j.xgen.2021.100027

[77] N. A. O'Leary, M. W. Wright, J. R. Brister, S. Ciufo, D. Haddad, R. McVeigh, B. Rajput, B. Robbertse, B. Smith-White, D. Ako-Adjei, A. Astashyn, A. Badretdin, Y. Bao, O. Blinkova, V. Brover, V. Chetvernin, J. Choi,

E. Cox, O. Ermolaeva, C. M. Farrell, T. Goldfarb, T. Gupta, D. Haft, E. Hatcher, W. Hlavina, V. S. Joardar, V. K. Kodali, W. Li, D. Maglott, P. Masterson, K. M. McGarvey, M. R. Murphy, K. O'Neill, S. Pujar, S. H. Rangwala, D. Rausch, L. D. Riddick, C. Schoch, A. Shkeda, S. S. Storz, H. Sun, F. Thibaud-Nissen, I. Tolstoy, R. E. Tully, A. R. Vatsan, C. Wallin, D. Webb, W. Wu, M. J. Landrum, A. Kimchi, T. Tatusova, M. DiCuccio, P. Kitts, T. D. Murphy, and K. D. Pruitt, "Reference sequence (RefSeq) database at NCBI: current status, taxonomic expansion, and functional annotation," *Nucleic Acids Research*, vol. 44, no. D1, pp. D733–D745, Jan. 2016, doi: 10.1093/nar/gkv1189. [Online]. Available: https://doi.org/10.1093/nar/gkv1189

[78] J. T. den Dunnen, R. Dalgleish, D. R. Maglott, R. K. Hart, M. S. Greenblatt, J. McGowan-Jordan, A.-F. Roux, T. Smith, S. E. Antonarakis, P. E. Taschner, and on behalf of the Human Genome Variation Society (HGVS), the Human Variome Project (HVP), and the Human Genome Organisation (HUGO), "HGVS Recommendations for the Description of Sequence Variants: 2016 Update," *Human Mutation*, vol. 37, no. 6, pp. 564–569, Jun. 2016, doi: 10.1002/humu.22981. [Online]. Available: https://doi.org/10.1002/humu.22981

[79] W. J. Kent, C. W. Sugnet, T. S. Furey, K. M. Roskin, T. H. Pringle, A. M. Zahler, and D. Haussler, "The Human Genome Browser at UCSC," *Genome Research*, vol. 12, no. 6, pp. 996–1006, Jun. 2002, doi: 10.1101/gr.229102. [Online]. Available: https://doi.org/10.1101/gr.229102

[80] A. R. Quinlan and I. M. Hall, "BEDTools: a flexible suite of utilities for comparing genomic features," *Bioinformatics*, vol. 26, no. 6, pp. 841–842, Mar. 2010, doi: 10.1093/bioinformatics/btq033. [Online]. Available: https://doi.org/10.1093/bioinformatics/btq033

[81] R. K. Dale, B. S. Pedersen, and A. R. Quinlan, "Pybedtools: a flexible Python library for manipulating genomic datasets and annotations," *Bioinformatics*, vol. 27, no. 24, pp. 3423–3424, Dec. 2011, doi: 10.1093/bioinformatics/btr539. [Online]. Available: https://doi.org/10.1093/bioinformatics/btr539

[82] G. Gonzalez-Calderon, R. Liu, R. Carvajal, and J. K. Teer, "A negative storage model for precise but compact storage of genetic variation data," *Database*, vol. 2020, p. baz158, Jan. 2020, doi: 10.1093/database/baz158. [Online]. Available: https://doi.org/10.1093/database/baz158

[83] C. Raczy, R. Petrovski, C. T. Saunders, I. Chorny, S. Kruglyak, E. H. Margulies, H.-Y. Chuang, M. Källberg, S. A. Kumar, A. Liao, K. M. Little, M. P. Strömberg, and S. W. Tanner, "Isaac: ultra-fast whole-genome secondary analysis on Illumina sequencing platforms," *Bioinformatics*, vol. 29, no. 16, pp. 2041–2043, Aug. 2013, doi: 10.1093/bioinformatics/btt314. [Online]. Available: https://doi.org/10.1093/bioinformatics/btt314

[84] L. Wratten, A. Wilm, and J. Göke, "Reproducible, scalable, and shareable analysis pipelines with bioinformatics workflow managers," *Nature Methods*, vol. 18, no. 10, pp. 1161–1168, Oct. 2021, doi: 10.1038/s41592-021-01254-9. [Online]. Available: https://doi.org/10.1038/s41592-021-01254-9

[85] Anaconda, Inc., “Conda,” 2017, Accessed: August 24, 2022. [Online]. Available: https://conda.io

[86] B. Grüning, R. Dale, A. Sjödin, B. A. Chapman, J. Rowe, C. H. Tomkins-Tinch, R. Valieris, J. Köster, and The Bioconda Team, “Bioconda: sustainable and comprehensive software distribution for the life sciences,” *Nature Methods*, vol. 15, no. 7, pp. 475–476, Jul. 2018, doi: 10.1038/s41592-018-0046-7. [Online]. Available: https://doi.org/10.1038/s41592-018-0046-7

[87] Docker, Inc., “Docker,” 2022, Accessed: August 24, 2022. [Online]. Available: https://docker.com/

[88] G. M. Kurtzer, V. Sochat, and M. W. Bauer, “Singularity: Scientific containers for mobility of compute,” *PLOS ONE*, vol. 12, no. 5, p. e0177459, May 2017, doi: 10.1371/journal.pone.0177459. [Online]. Available: https://doi.org/10.1371/journal.pone.0177459

[89] E. Larsonneur, J. Mercier, N. Wiart, E. L. Floch, O. Delhomme, and V. Meyer, “Evaluating Workflow Management Systems: A Bioinformatics Use Case,” in *2018 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. Madrid, Spain: IEEE, Dec. 2018, pp. 2773–2775, doi: 10.1109/BIBM.2018.8621141. [Online]. Available: https://doi.org/10.1109/BIBM.2018.8621141

[90] F. Mölder, K. P. Jablonski, B. Letcher, M. B. Hall, C. H. Tomkins-Tinch, V. Sochat, J. Forster, S. Lee, S. O. Twardziok, A. Kanitz, A. Wilm, M. Holtgrewe, S. Rahmann, S. Nahnsen, and J. Köster,

"Sustainable data analysis with Snakemake," *F1000Research*, vol. 10, p. 33, Apr. 2021, doi: 10.12688/f1000research.29032.2. [Online]. Available: https://doi.org/10.12688/f1000research.29032.2

[91] P. Di Tommaso, M. Chatzou, E. W. Floden, P. P. Barja, E. Palumbo, and C. Notredame, "Nextflow enables reproducible computational workflows," *Nature Biotechnology*, vol. 35, no. 4, pp. 316–319, Apr. 2017, doi: 10.1038/nbt.3820. [Online]. Available: https://doi.org/10.1038/nbt.3820

[92] E. Afgan, D. Baker, B. Batut, M. van den Beek, D. Bouvier, M. Čech, J. Chilton, D. Clements, N. Coraor, B. A. Grüning, A. Guerler, J. Hillman-Jackson, S. Hiltemann, V. Jalili, H. Rasche, N. Soranzo, J. Goecks, J. Taylor, A. Nekrutenko, and D. Blankenberg, "The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2018 update," *Nucleic Acids Research*, vol. 46, no. W1, pp. W537–W544, 05 2018, doi: 10.1093/nar/gky379. [Online]. Available: https://doi.org/10.1093/nar/gky379

[93] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, and K. Wenger, "Pegasus, a workflow management system for science automation," *Future Generation Computer Systems*, vol. 46, pp. 17–35, May 2015, doi: 10.1016/j.future.2014.10.008. [Online]. Available: https://doi.org/10.1016/j.future.2014.10.008

[94] K. Voss, G. V. D. Auwera, and J. Gentry, "Full-stack genomics pipelining with gatk4 + wdl + cromwell [version 1; not peer

reviewed],” 2017, doi: 10.7490/f1000research.1114634.1. [Online]. Available: https://f1000research.com/slides/6-1381

[95] M. R. Crusoe, S. Abeln, A. Iosup, P. Amstutz, J. Chilton, N. Tijanić, H. Ménager, S. Soiland-Reyes, B. Gavrilović, C. Goble, and The CWL Community, “Methods included: Standardizing computational reuse and portability with the common workflow language,” *Commun. ACM*, vol. 65, no. 6, p. 54–63, may 2022, doi: 10.1145/3486897. [Online]. Available: https://doi.org/10.1145/3486897

[96] J. Gentry, C. Llanwarne, M. Lin, P. Magee, B. O’Connor, O. Rodeh, G. Van der Auwera, B. Chapman, and A. Prabhakaran, “OpenWDL,” 2022, Accessed: August 24, 2022. [Online]. Available: https://openwdl.org

[97] S. Bray, M. Bernt, N. Soranzo, M. van den Beek, B. Batut, H. Rasche, M. Čech, P. Cock, A. Nekrutenko, B. Grüning, and J. Chilton, “Planemo: a command-line toolkit for developing, deploying, and executing scientific data analyses,” *bioRxiv*, 2022, doi: 10.1101/2022.03.13.483965. [Online]. Available: https://doi.org/10.1101/2022.03.13.483965

[98] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin, and 1000 Genome Project Data Processing Subgroup, “The Sequence Alignment/Map format and SAMtools,” *Bioinformatics*, vol. 25, no. 16, pp. 2078–2079, Aug. 2009, doi: 10.1093/bioinformatics/btp352. [Online]. Available: https://doi.org/10.1093/bioinformatics/btp352

[99] P. Danecek, J. K. Bonfield, J. Liddle, J. Marshall, V. Ohan, M. O. Pollard,

A. Whitwham, T. Keane, S. A. McCarthy, R. M. Davies, and H. Li, "Twelve years of SAMtools and BCFtools," *GigaScience*, vol. 10, no. 2, p. giab008, Jan. 2021, doi: 10.1093/gigascience/giab008. [Online]. Available: https://doi.org/10.1093/gigascience/giab008

[100] P. Cingolani, A. Platts, L. L. Wang, M. Coon, T. Nguyen, L. Wang, S. J. Land, X. Lu, and D. M. Ruden, "A program for annotating and predicting the effects of single nucleotide polymorphisms, SnpEff: SNPs in the genome of Drosophila melanogaster strain w ^{1118} ; iso-2; iso-3," *Fly*, vol. 6, no. 2, pp. 80–92, Apr. 2012, doi: 10.4161/fly.19695. [Online]. Available: https://doi.org/10.4161/fly.19695

[101] T. Seemann, "mlst," 2022, Accessed: August 24, 2022. [Online]. Available: https://github.com/tseemann/mlst

[102] H. Li and R. Durbin, "Fast and accurate long-read alignment with Burrows–Wheeler transform," *Bioinformatics*, vol. 26, no. 5, pp. 589–595, Mar. 2010, doi: 10.1093/bioinformatics/btp698. [Online]. Available: https://doi.org/10.1093/bioinformatics/btp698

[103] E. Garrison and G. Marth, "Haplotype-based variant detection from short-read sequencing," *arXiv*, 2012, doi: 10.48550/ARXIV.1207.3907. [Online]. Available: https://doi.org/10.48550/ARXIV.1207.3907

[104] D. Lemire, G. Ssi-Yan-Kai, and O. Kaser, "Consistently faster and smaller compressed bitmaps with Roaring," *Software: Practice and Experience*, vol. 46,

no. 11, pp. 1547–1569, Nov. 2016, doi: 10.1002/spe.2402. [Online]. Available: https://doi.org/10.1002/spe.2402

[105] R. K. Hart and A. Prlić, "SeqRepo: A system for managing local collections of biological sequences," *PLOS ONE*, vol. 15, no. 12, p. e0239883, Dec. 2020, doi: 10.1371/journal.pone.0239883. [Online]. Available: https://doi.org/10.1371/journal.pone.0239883

[106] M. Bayer, "Sqlalchemy," in *The Architecture of Open Source Applications Volume II: Structure, Scale, and a Few More Fearless Hacks*, A. Brown and G. Wilson, Eds. aosabook.org, 2012. [Online]. Available: http://aosabook.org/en/sqlalchemy.html

[107] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. [Online]. Available: https://doi.org/10.1109/MCSE.2007.55

[108] S. C. Kleene, "On notation for ordinal numbers," *Journal of Symbolic Logic*, vol. 3, no. 4, pp. 150–155, Dec. 1938, doi: 10.2307/2267778. [Online]. Available: https://doi.org/10.2307/2267778

[109] S. Wintein, "On All Strong Kleene Generalizations of Classical Logic," *Studia Logica*, vol. 104, no. 3, pp. 503–545, Jun. 2016, doi: 10.1007/s11225-015-9649-5. [Online]. Available: https://doi.org/10.1007/s11225-015-9649-5

[110] K. Katoh, K. Misawa, K. Kuma, and T. Miyata, "MAFFT: a novel method for rapid multiple sequence alignment based on fast Fourier transform,"

Nucleic Acids Research, vol. 30, no. 14, pp. 3059–3066, 07 2002, doi: 10.1093/nar/gkf436. [Online]. Available: https://doi.org/10.1093/nar/gkf436

[111] J. Archie, W. H. Day, W. Maddison, C. Meacham, F. J. Rohlf, D. Swofford, and J. Felsenstein, "The newick tree format," 1986, Accessed: August 24, 2022. [Online]. Available: https://evolution.genetics.washington.edu/phylip/newicktree.html

[112] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020. [Online]. Available: https://doi.org/10.1038/s41592-019-0686-2

[113] J. R. Rideout, G. Caporaso, E. Bolyen, D. McDonald, Y. V. Baeza, J. C. Alastuey, A. Pitman, J. Morton, J. Navas, K. Gorlick, J. Debelius, Z. Xu, Llcooljohn, Adamrp, J. Shorenstein, L. Luce, W. Van Treuren, Charudatta-Navare, A. Gonzalez, C. Brislawn, W. Patena, K. Schwarzberg, Teravest, J. Reeder, Shiffer1, I. Sfiligoi, Nbresnick, D. K. D. Murray, K. Sharma, and Alexbrc, "biocore/scikit-bio: scikit-bio 0.5.7: Performance and

maintenance," Apr. 2022, doi: 10.5281/ZENODO.6403781. [Online]. Available: https://doi.org/10.5281/ZENODO.6403781

[114] P. J. Rousseeuw, "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis," *Journal of Computational and Applied Mathematics*, vol. 20, pp. 53–65, Nov. 1987, doi: 10.1016/0377-0427(87)90125-7. [Online]. Available: https://doi.org/10.1016/0377-0427(87)90125-7

[115] M. Yousefian, "CMDBench," 2020, Accessed: August 24, 2022. [Online]. Available: https://github.com/manzik/cmdbench

[116] L. A. Nell, "jackalope: A swift, versatile phylogenomic and high-throughput sequencing simulator," *Molecular Ecology Resources*, vol. 20, no. 4, pp. 1132–1140, Jul. 2020, doi: 10.1111/1755-0998.13173. [Online]. Available: https://doi.org/10.1111/1755-0998.13173

[117] S. Bekal, C. Berry, A. R. Reimer, G. Van Domselaar, G. Beaudry, E. Fournier, F. Doualla-Bell, E. Levac, C. Gaulin, D. Ramsay, C. Huot, M. Walker, C. Sieffert, and C. Tremblay, "Usefulness of High-Quality Core Genome Single-Nucleotide Variant Analysis for Subtyping the Highly Clonal and the Most Prevalent Salmonella enterica Serovar Heidelberg Clone in the Context of Outbreak Investigations," *Journal of Clinical Microbiology*, vol. 54, no. 2, pp. 289–295, Feb. 2016, doi: 10.1128/JCM.02200-15. [Online]. Available: https://doi.org/10.1128/JCM.02200-15

[118] M. Hasegawa, T.-a. Yano, and H. Kishino, "A new molecular clock of mitochondrial DNA and the evolution of hominoids," *Proceedings of the Japan*

Academy, Series B, vol. 60, no. 4, pp. 95–98, 1984, doi: 10.2183/pjab.60.95. [Online]. Available: https://doi.org/10.2183/pjab.60.95

[119] T. Saito and M. Rehmsmeier, "The Precision-Recall Plot Is More Informative than the ROC Plot When Evaluating Binary Classifiers on Imbalanced Datasets," *PLOS ONE*, vol. 10, no. 3, p. e0118432, Mar. 2015, doi: 10.1371/journal.pone.0118432. [Online]. Available: https://doi.org/10.1371/journal.pone.0118432

[120] E. Paradis and K. Schliep, "ape 5.0: an environment for modern phylogenetics and evolutionary analyses in R," *Bioinformatics*, vol. 35, no. 3, pp. 526–528, Feb. 2019, doi: 10.1093/bioinformatics/bty633. [Online]. Available: https://doi.org/10.1093/bioinformatics/bty633

[121] D. Robinson and L. Foulds, "Comparison of phylogenetic trees," *Mathematical Biosciences*, vol. 53, no. 1-2, pp. 131–147, Feb. 1981, doi: 10.1016/0025-5564(81)90043-2. [Online]. Available: https://doi.org/10.1016/0025-5564(81)90043-2

[122] K. P. Schliep, "phangorn: phylogenetic analysis in R," *Bioinformatics*, vol. 27, no. 4, pp. 592–593, Feb. 2011, doi: 10.1093/bioinformatics/btq706. [Online]. Available: https://doi.org/10.1093/bioinformatics/btq706

[123] M. Kendall and C. Colijn, "Mapping Phylogenetic Trees to Reveal Distinct Patterns of Evolution," *Molecular Biology and Evolution*, vol. 33, no. 10, pp. 2735–2743, Oct. 2016, doi: 10.1093/molbev/msw124. [Online]. Available: https://doi.org/10.1093/molbev/msw124

[124] T. Jombart, M. Kendall, J. Almagro-Garcia, and C. Colijn, "treespace: Statistical exploration of landscapes of phylogenetic trees," *Molecular Ecology Resources*, vol. 17, no. 6, pp. 1385–1392, Nov. 2017, doi: 10.1111/1755-0998.12676. [Online]. Available: https://doi.org/10.1111/1755-0998.12676

[125] R. E. Timme, H. Rand, M. Shumway, E. K. Trees, M. Simmons, R. Agarwala, S. Davis, G. E. Tillman, S. Defibaugh-Chavez, H. A. Carleton, W. A. Klimke, and L. S. Katz, "Benchmark datasets for phylogenomic pipeline validation, applications for foodborne pathogen surveillance," *PeerJ*, vol. 5, p. e3893, 2017, doi: 10.7717/peerj.3893. [Online]. Available: https://doi.org/10.7717/peerj.3893

[126] National Center for Biotechnology Information, "The NCBI Pathogen Detection Project," 2016, Accessed: August 24, 2022. [Online]. Available: https://www.ncbi.nlm.nih.gov/pathogens

[127] R.-C. Ferreira, E. Wong, G. Gugan, K. Wade, M. Liu, L. M. Baena, C. Chato, B. Lu, A. S. Olabode, and A. F. Y. Poon, "CoVizu: Rapid analysis and visualization of the global diversity of SARS-CoV-2 genomes," *Virus Evolution*, vol. 7, no. 2, 11 2021, doi: 10.1093/ve/veab092. [Online]. Available: https://doi.org/10.1093/ve/veab092

[128] C. L. Murall, R. Poujol, A. Petkau, B. Sobkowiak, A. Zetner, S. A. Kraemer, A. N'Guessan, S. Naderi, E. E. Gill, J. Fritz, F. S. Brinkman, J. Hussin, N. Prystajecky, T. Lynch, M. A. Croxen, R. McDonald, K. MacKenzie, C. Colijn, G. Van Domselaar, S. P. Otto, B. J. Shapiro,

and The Canadian COVID-19 Genomics Network (CanCOGeN) Virus Sequencing Consortium, “Monitoring the evolution and spread of delta sublineages AY.25 and AY.27 in canada,” *Virological*, 2021. [Online]. Available: https://virological.org/t/monitoring-the-evolution-and-spread-of-delta-sublineages-ay-25-and-ay-27-in-canada/767

[129] A. Uyar, “Google stemming mechanisms,” *Journal of Information Science*, vol. 35, no. 5, pp. 499–514, Oct. 2009, doi: 10.1177/1363459309336801. [Online]. Available: https://doi.org/10.1177/1363459309336801

[130] F. Şentürk and G. Gunduz, “A framework for investigating search engines’ stemming mechanisms: A case study on Bing,” *Concurrency and Computation: Practice and Experience*, vol. 34, no. 9, Apr. 2022, doi: 10.1002/cpe.6562. [Online]. Available: https://doi.org/10.1002/cpe.6562

[131] T. C. Matthews, F. R. Bristow, E. J. Griffiths, A. Petkau, J. Adam, D. Dooley, P. Kruczkiewicz, J. Curatcha, J. Cabral, D. Fornika, G. L. Winsor, M. Courtot, C. Bertelli, A. Roudgar, P. Feijao, P. Mabon, E. Enns, J. Thiessen, A. Keddy, J. Isaac-Renton, J. L. Gardy, P. Tang, The IRIDA Consortium, J. A. Carriço, L. Chindelevitch, C. Chauve, M. R. Graham, A. G. McArthur, E. N. Taboada, R. G. Beiko, F. S. Brinkman, W. W. Hsiao, and G. V. Domselaar, “The Integrated Rapid Infectious Disease Analysis (IRIDA) Platform,” *bioRxiv*, p. 381830, Jan. 2018, doi: 10.1101/381830. [Online]. Available: https://doi.org/10.1101/381830

[132] R. Colquhoun, Á. O’Toole, A. Rambaut, and B. Jackson, “Scorpio,”

2022, Accessed: August 24, 2022. [Online]. Available: https://github.com/cov-lineages/scorpio

[133] A. Benoist, “ERAlchemy,” Feb. 2018. [Online]. Available: https://github.com/Alexis-benoist/eralchemy

[134] A. Rambaut, “FigTree,” 2007, Accessed: August 24, 2022. [Online]. Available: http://tree.bio.ed.ac.uk/software/figtree/

[135] L. J. Revell, “phytools: an R package for phylogenetic comparative biology (and other things): *phytools: R package*,” *Methods in Ecology and Evolution*, vol. 3, no. 2, pp. 217–223, Apr. 2012, doi: 10.1111/j.2041-210X.2011.00169.x. [Online]. Available: https://doi.org/10.1111/j.2041-210X.2011.00169.x