VariantStore: A Large-Scale Genomic Variant Search Index

Prashant Pandey[1], Yinjie Gao[1], and Carl Kingsford*[1]

[1]Computational Biology Department, School of Computer Science, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA

December 24, 2019

Abstract

The ability to efficiently query genomic variation data from thousands of samples is critical to achieve the full potential of many medical and scientific applications such as personalized medicine. We present VariantStore, a system for efficiently indexing and searching millions of genomic variants across thousands of samples. We show the scalability of VariantStore by indexing genomic variants from the TCGA-BRCA project containing 8640 samples and 5M variants in ≈ 4 Hrs and the 1000 genomes project containing 2500 samples and 924M variants in ≈ 3 Hrs. Querying for variants in a gene takes between 2 milliseconds to 3 seconds using memory only $\approx 10\%$ of the size of the full representation. As a baseline, VariantStore outperformed VG toolkit by $3\times$ in terms of memory-usage and construction time and uses 25% less disk space although VG toolkit does not support variant queries.

Advanced sequencing technology and computing resources have led to large-scale genomic sequencing efforts producing genomic variation data from thousands of samples, such as the 1000 Genomes project [1–3], GTEx [4], and The Cancer Genome Atlas (TCGA) [5]. Analysis of genomic variants combined with phenotypic information of samples promises to improve applications such as personalized medicine, population-level disease analysis, and cancer remission rate prediction. Although numerous studies [6–12] have been performed over the past decade involving genomic variation, the ability to scale these studies to large-scale data available today and in the near future is still limited.

Contrary to the scale of the size of genomic variation data available there is no large-scale indexing and

*To whom correspondence should be addressed: carlk@cs.cmu.edu

querying system that can support efficient variant queries. Traditional database solutions, such as SQL and NoSQL, have proven prohibitively slow to store and query collections of variants [13–15].

On an individual sample, the typical result of sequencing, alignment, and variant calling is a collection of millions of sample-specific variants. A variant is identified by the position in the chromosome where it occurs, an alternative sequence, a list of samples that contain the variant, and phasing information. The standard file format to report these variants is the variant call file (VCF) [16].

A common task is to identify all the samples with a given pattern of variants or identify all samples that have variants in a given gene. These tasks are translated into variant queries which require finding variants or samples with variants between two positions in a chromosome. Positions can be specified in either the reference coordinate or a sample coordinate depending on the query because variants can appear at different coordinates in a sample sequence compared to the reference sequence.

To effectively use variant information from many samples, medical and scientific applications must often answer many instances of one or more of the following types of queries:

1. Find the closest variant to position X for all samples in the reference coordinates.
2. Find the sequence between positions X and Y for sample S in the reference coordinates.
3. Find the sequence between positions X and Y for sample S in the sample coordinates.
4. Find all variants between positions X and Y for sample S in the reference coordinates.
5. Find all variants between positions X and Y for sample S in the sample coordinates.
6. Find all variants between positions X and Y for all samples in the reference coordinates.

These queries can be further used as building blocks for more complicated queries, such as finding samples with more than a given number of variants between two positions or count the number of variants for each sample between two positions.

Genomic variation data are also used for read alignment in order to avoid mapping biases that arise when mapping reads to a single high-quality reference sequence [17–21]. VG toolkit [19] is the most widely used tool to represent genomic variation. It encodes genomic variants from multiple samples in a graph, called a variation graph. A variation graph is a sequence graph where each node represents a sequence and a set of nodes through the graph, known as a path, embeds the complete sequence corresponding to the reference

or a sample. Each node on a path is assigned a position corresponding to the location of the node in the coordinate of the reference or a sample sequence the path embeds. Reads are aligned to sequences in the variation graph by following paths in the graph.

The variation graph representation in VG toolkit is designed to optimize read alignment and uses sequence-based indexes for alignment. It can not be directly used for variant queries that require an index based on the position of variants in multiple sequence coordinates. VG toolkit stores each sample path as a list of nodes in the graph and maintains a separate index corresponding to the coordinates of the reference and samples. Storing a separate list of nodes for each sequence impedes the scalability of the representation for storing variation from thousands of samples. Moreover, variants are often shared among samples, so storing a list of nodes for each sample path introduces redundancy in the representation. Finally, the VG toolkit representation does not store phasing information contained in VCF files, which is required in many analyses.

We present VariantStore, a system for efficiently indexing and querying genomic information (genomic variants and phasing information) from thousands of samples containing millions of variants. VariantStore builds a variant-oriented index by mapping each variant to the list of samples that contain the variant. Variants are indexed based on the positions where they occur. It supports querying variants occurring between two positions across a chromosome based on the reference or a sample coordinate. VariantStore can scale to tens of millions of variants and thousands of samples and can efficiently scale out-of-RAM to storage devices to enable memory-efficient construction and query.

We encode genomic variation in a directed, acyclic variation graph and build a position index (a mapping of node positions to node identifiers) on the graph to quickly access a node in the graph corresponding to a queried position. Each node in the variation graph corresponds to a variant and stores a list of samples that contain the variant along with the position of the variant in the coordinate of those samples. The inverted index design allows one to quickly find all the samples and positions in sample coordinates corresponding to a variant. It also avoids redundancy that otherwise arises in maintaining individual variant indexes for each sample coordinate and scales well in practice when the number of samples grows beyond a few thousand.

To perform index construction and query efficiently in terms of memory, we partition the variation graph into small chunks (usually a few MBs in size) based on the reference coordinates and store variation graph

nodes in these chunks. During construction, an active chunk is always maintained in memory in which new nodes are added, and once it reaches its capacity we compress and serialize it to disk and create a new active chunk. The nodes in and across chunks are ordered based on the reference coordinate since they are created based on variants in the VCF file which are themselves ordered by the reference coordinate. During a query, we only load the chunks in memory that contain the nodes corresponding to the query range.

To efficiently scale to thousands of coordinate systems (or samples), we maintain the position index only on the reference coordinate. The position index maps positions in the reference sequence where there is a variant to nodes corresponding to those variants in the variation graph. To lookup a position using a sample's coordinate system, we first lookup the node corresponding to the position in the reference coordinate. We then traverse the sample path from the node in the graph to determine the appropriate node in the sample coordinate. A node with a given position in a sample coordinate is often close to the node in the reference coordinate with the same position.

Results

Our evaluation of VariantStore is based on four parameters: construction time, query throughput, disk space, and peak memory usage.

To calibrate our performance numbers, we compare the construction performance and disk space of VariantStore against the variation graph representation in VG toolkit [22]. VG toolkit provides a succinct representation of the variation graph and enables mapping reads to variation graph at the scale of the human genome. VG toolkit is not designed for answering variant queries and therefore is not optimized for it.

Data. We use 1000 Genomes Phase 3 data [23] and three of the biggest projects from TCGA in terms of the number of samples, Ovarian Cancer (OV), Lung Adenocarcinoma (LUAD), and Breast Invasive Carcinoma (BRCA), for our evaluation. 1000 Genomes data contains more variants compared to the TCGA data but TCGA data contains more samples. 1000 Genomes data contains a separate VCF file for each chromosome containing variants from thousands of samples. The number of samples in each file is ≈ 2.5K. The variants in 1000 Genomes project are based on GRCh37 reference genome. The TCGA data contains a separate VCF file for each sample. The OV, LUAD, and BRCA projects contain 2436, 2680, and 4319 VCF files containing both normal and tumor variants respectively. For each project in TCGA, we first merged VCF

System	Time	Disk space	Peak RAM	Peak RAM Agg.
Dataset		1000 Genomes		
VariantStore	3 Hrs 25 mins	41 GB	8.8 GB	153 GB
VG-toolkit	11 Hrs 10 mins	50 GB	37 GB	450 GB
Dataset		TCGA (OV)		
VariantStore	1 Hr 5 mins	3.4 GB	1.1 GB	17.45 GB
VG-toolkit		11 GB*		
Dataset		TCGA (LUAD)		
VariantStore	1 Hr 20 mins	3.5 GB	2.3 GB	36.05 GB
VG-toolkit		12 GB*		
Dataset		TCGA (BRCA)		
VariantStore	4 Hrs 36 mins	4.2 GB	3.2 GB	53.21 GB
VG-toolkit		14 GB*		

Table 1: Time, space, peak RAM, and peak RAM (aggregate) to construct a variation graph and position index on the 1000 Genomes and TCGA (OV, LUAD, and BRCA) data using VariantStore and VG toolkit. We constructed all 24 chromosomes (1 – 22 and X and Y) in parallel. The time and peak RAM reported is for the biggest chromosome (usually chromosome 1 or 2). The space reported is the total space on disk for all 24 chromosomes. The peak RAM (aggregate) is the aggregate peak RAM for all 24 processes. *Space for the XG index that does not contain any path information.

files using the BCF tool merge command [24] and created a separate VCF file for each chromosome. The variants in the TCGA project are based on GRCh38 reference genome.

Index construction. The total time taken to construct the variation graph representation and index includes the time taken to read and parse variants from the VCF file, construct the variation graph representation and indexes, and serialize the final representation to disk. For VariantStore, the reported time includes the time to create and serialize the position index. The space reported for VariantStore is the sum of the space of the variation graph representation and position index.

For VG toolkit, creating a variation graph representation with multiple coordinate systems (or sample path annotations) requires creating two indexes, XG and GBWT index. The XG index is a succinct representation of the variation graph without path annotations that allows memory- and time-efficient random access operations on large graphs. The GBWT (graph BWT) is a substring index for storing sample paths in the variation graph. We first create the variation graph representation using the "construct" command including all sample path annotations. We then create the XG index and GBWT index from the variation graph representation to create an index with all sample path annotations in the variation graph. For VG toolkit, the reported time includes the time to create and serialize the XG and GBWT indexes. The space reported for

VG toolkit is the sum of the space of the XG and GBWT indexes. VG toolkit could not build GBWT index on TCGA data even after running for more than a day. We only report space for the XG index (which does not contain any sample path annotations) for TCGA data.

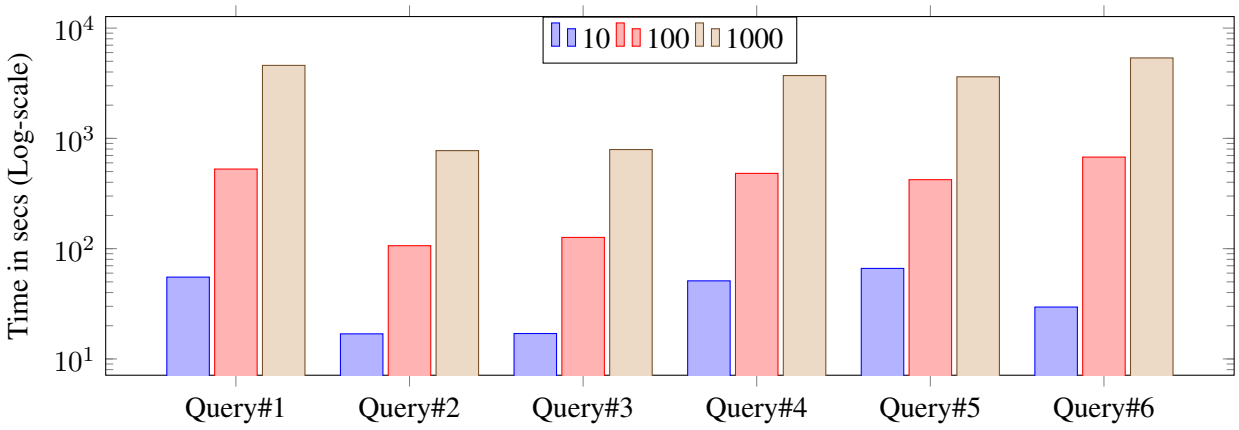
For both VariantStore and VG toolkit, we created 24 separate indexes, one each for chromosomes $1 - 22$, and X and Y. Each of these indexes were constructed in parallel as a separate process. We report the time taken for construction as the time taken by the process that finishes last. For disk space, we report the total space taken by all 24 indexes. For peak memory usage, we report the highest individual and aggregate peak RAM usage for all processes.

The performance of VariantStore and VG toolkit for constructing the index on the 1000 Genomes and TCGA data is shown in Table 1. VariantStore is $3\times$ faster, takes 25% less disk space, and $3\times$ less peak RAM than VG toolkit. For the TCGA data, VG toolkit could not build GBWT index embedding all sample paths. However, even the space needed for the XG index (not embedding sample paths) is $\approx 3.3\times$ larger than the VariantStore representation containing all sample paths.

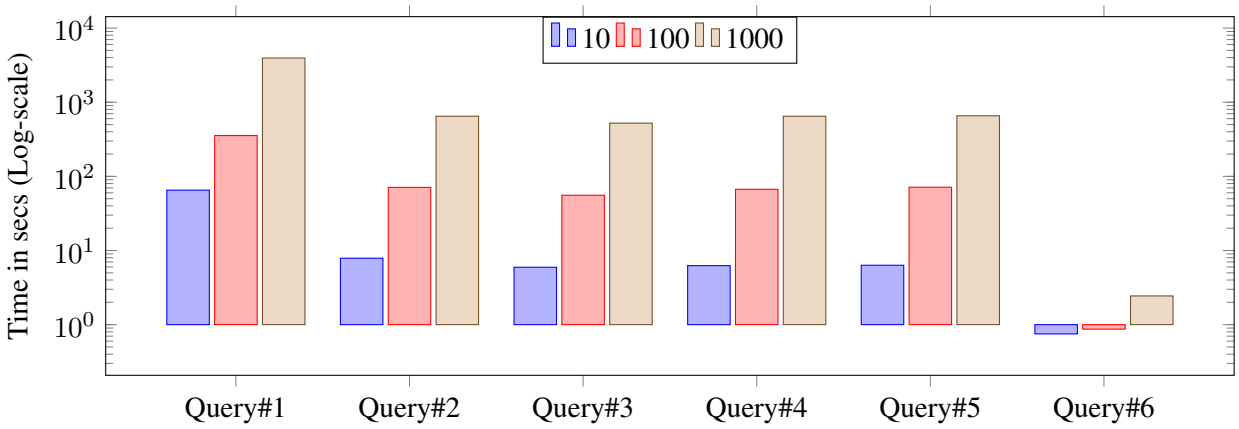
Query throughput. We measured the query throughput for all six queries mentioned above. To show the robustness of query efficiency on different data we evaluate on two different chromosome indexes from two different projects. We chose Chromosome 2 which is one of the bigger chromosomes and Chromosome 22 which is one of the smaller ones. We evaluate query time on chromosome indexes from 1000 Genomes and TCGA LUAD data.

To perform queries, we specify a pair of positions in the reference or a sample coordinate system depending on the query type and optionally a sample name. Query parameters, such as the length of the queried region and density of variants in that region, affect the query timing. Therefore, we performed three sets of query benchmarks containing 10, 100, and 1000 queries and report the aggregate time. For each query in the set, we uniformly randomly pick the start position across the full chromosome length. The size of the query range is set to $\approx$ 42K bases which is approximately the length of a typical gene. Picking multiple query regions uniformly randomly across the chromosome provides a good coverage of different regions across the chromosome.

To measure the query throughput, we only load the position index and graph topology in memory and the variation graph representation is kept on disk before performing the query. Keeping the variation graph



(a) Time for 10, 100, and 1000 queries on Chromosome 22 index in VariantStore for 1000 Genomes data.

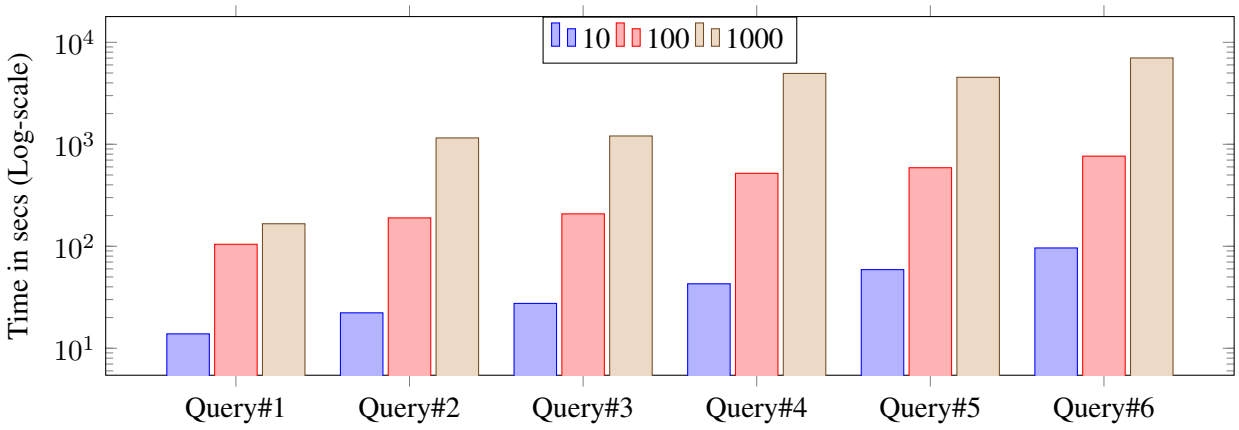


(b) Time for 10, 100, and 1000 queries on Chromosome 22 index in VariantStore for TCGA LUAD data.

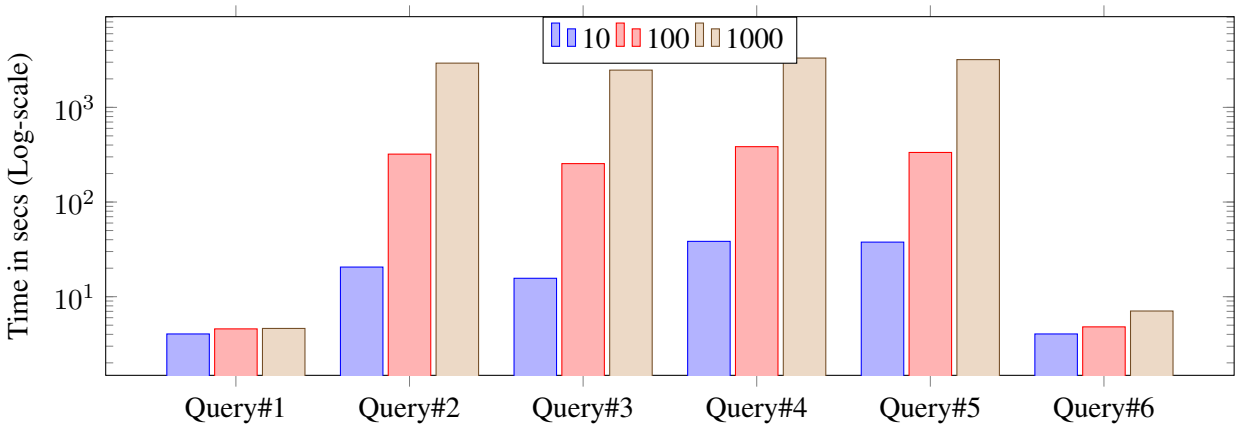
Figure 1: Time reported is the total time taken to execute 10, 100, and 1000 queries. For all queries the query length is fixed to $\approx 42K$. Query#1: Closest variant, Query#2: Seq in Ref coordinate, Query#3: Seq in Sample coordinate, Query#4: Sample vars in Ref coordinate, Query#5: Sample vars in Sample coordinate, Query#6: All vars in Ref coordinate

representation on disk keeps the peak memory usage low and all disk accesses are performed during the query to load appropriate node chunks.

The query throughput on 1000 Genomes data is shown in Figures 1a and 2a. For all query types, the aggregate time taken to execute queries increases linearly with the number of queries. Finding the sequence corresponding to a sample in a region takes less time compared to finding variants in a region. Finding the sequence takes less time because it involves traversing the sequence specific path in the region and reconstructing the sequence. However, finding variants in a given region takes more time because it involves an exhaustive search of neighbors at each node in the region to determine all the variants that are contained by a given sample or all samples.



(a) Time for 10, 100, and 1000 queries on Chromosome 2 index in VariantStore for 1000 Genomes data.



(b) Time for 10, 100, and 1000 queries on Chromosome 2 index in VariantStore for TCGA LUAD data.

Figure 2: Time reported is the total time taken to execute 10, 100, and 1000 queries. For all queries the query length is fixed to $\approx 42K$. Query#1: Closest variant, Query#2: Seq in Ref coordinate, Query#3: Seq in Sample coordinate, Query#4: Sample vars in Ref coordinate, Query#5: Sample vars in Sample coordinate, Query#6: All vars in Ref coordinate

The query throughput on TCGA data is shown in Figures 1b and 2b. The TCGA data has twice as many samples compared to 1000 Genomes data but there are fewer variants. This makes the variation graph much sparser and queries in the position index become more expensive compared to traversing the graph between two positions. Finding all variants is the fastest query because the variation graph is very sparse and most position ranges are empty. Traversing the graph to find the sequence or variants for a sample takes similar amount of time. Finding the closest variant from a position takes the most amount of time because it involves performing multiple position-index queries to determine the closest variant.

For both 1000 Genomes and TCGA LUAD data, finding the closest variant query is faster for Chromosome 2 because variants in Chromosome 22 are more dense compared to Chromosome 2 which makes it faster to

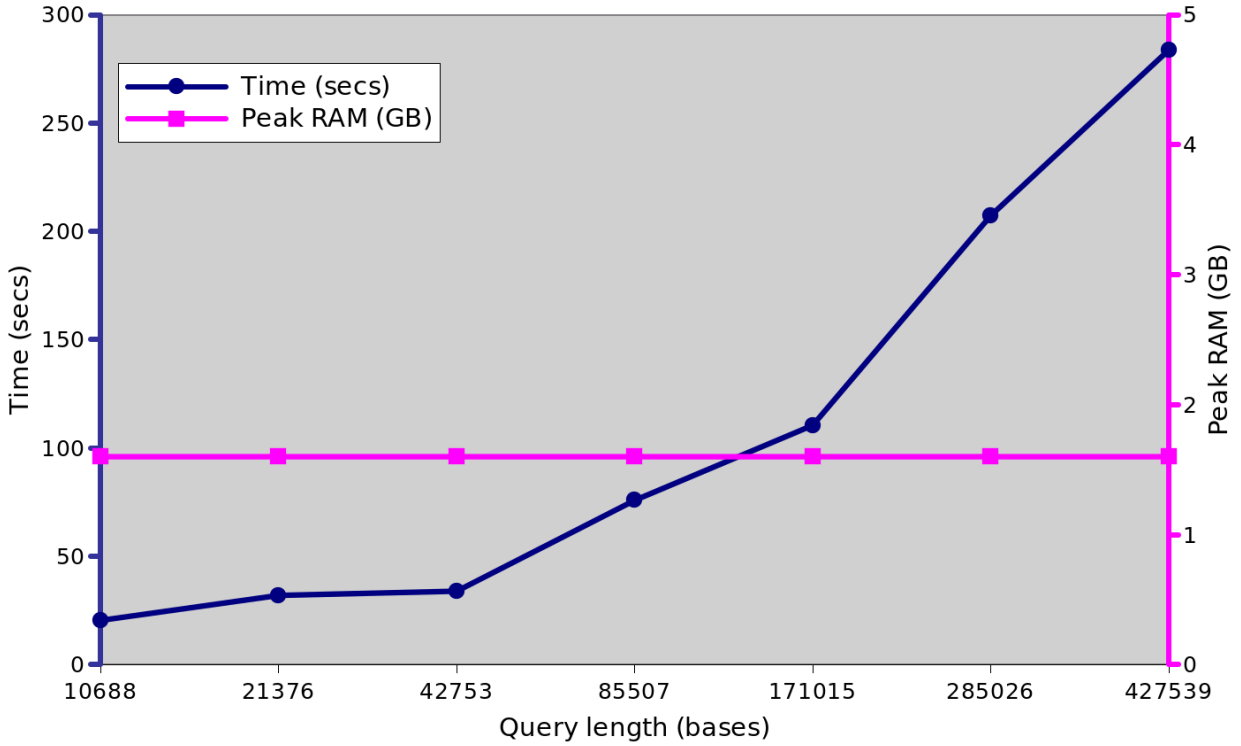


Figure 3: Time (seconds) and peak memory usage (GB) with increasing query length for "Sample sequence in the reference coordinate" (query #2) for 1000 Genomes chromosome 22 index. The time taken increases as the query length increases. But the memory usage remains constant regardless of the query length.

locate the closest variant.

The effect of query range on peak memory usage. We performed another query benchmark to evaluate the effect of size of the position range on the peak memory usage and time. For this benchmark, we chose the "Sample sequence in reference coordinate" query (query #2) because this query involves traversing the full sample path between two positions. We performed sets of 100 queries with increasing size of the position range and record the total time and peak RAM usage. For each query in the set, we uniformly randomly pick the start position across the full chromosome length.

Effect of the query range size on peak memory usage and time is shown in Figure 3. The memory usage remains constant regardless of the query length. This is because during a query we access node chunks in sequential order and regardless of the query length only load at most two node chunks in RAM at a time. This keeps the memory usage essentially constant.

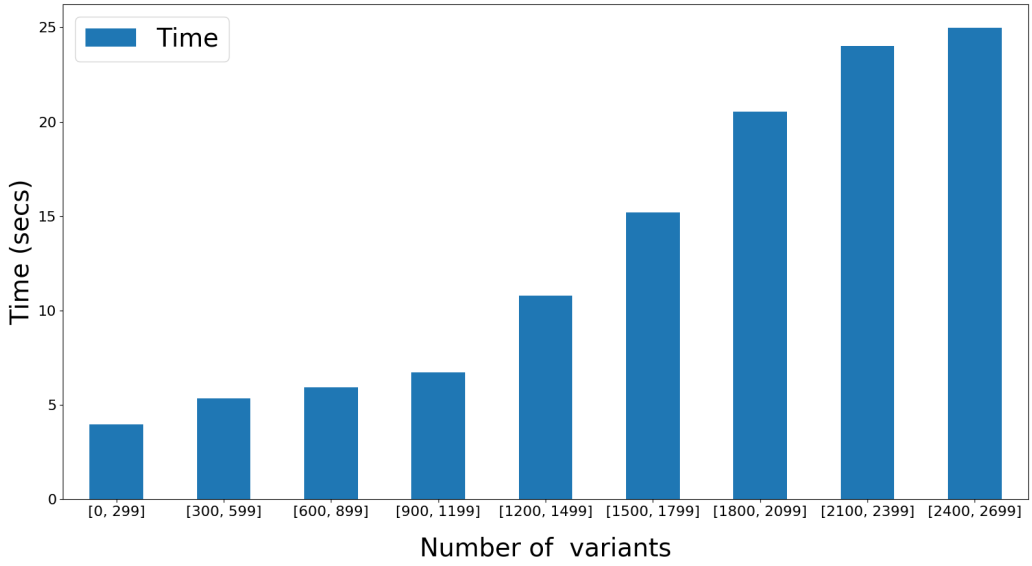


Figure 4: Time (seconds) and the number of variants in the query region for "All variants in the reference coordinate" (query #6) for 1000 Genomes chromosome 22 index. Query times are binned based on the number of variants in the query region and mean time is reported in each bin. The mean time increases as the number of variants increases in the region.

The effect of number of variants on query time. We also evaluate how the number of variants in the position range affects the query time. For this benchmark, we performed the "All variants in the reference coordinate" query (query #6) because this query involves performing a breadth-first search in the graph to determine all variants in a region and the query time depends on the number of variants in the region. To perform queries on regions with different number of variants, we chose 1000 regions with a fixed size of the position range ($\approx$ 42K bases) and start position chosen uniformly randomly across the chromosome.

Effect of the number of variants in the query region on query time is shown in Figure 4. The query time increases as the number of variants in the queried region increases. This is because when the number of variants is in a region is small the graph is sparser and faster to traverse and report all variants.

Experimental hardware. All experiments on 1000 Genomes data were performed on an Intel Xeon CPU E5-2699A v4 @ 2.40GHz (44 cores and 56MB L3 cache) with 1TB RAM and a 7.3TB HGST HDN728080AL HDD running Ubuntu 18.04.2 LTS (Linux kernel 4.15.0-46-generic) and were run using a single thread. Benchmarks for TCGA data were performed on a cluster machine running AMD Opteron Processor 6220 @ 3GHz with 6MB L3 cache. TCGA data was stored on a remote disk and accessed via NFS.

Discussion

We attribute the scalability and efficient index construction and query performance of VariantStore to the variant-oriented index design. VariantStore uses an inverted index from variants to the samples which scales efficiently when multiple samples share a variant which is often seen in genomic variation data. The inverted index design further allows us to build the position index only on the reference sequence and use graph traversal to transform the position in reference coordinates to sample coordinates.

All the supported variant queries look for variants in a contiguous region of the chromosome which allows VariantStore to partition the variation graph representation into small chunks based on the position of nodes in the chromosome and sequentially load only the relevant chunks into memory during a query. This makes VariantStore memory-efficient and scale to genomic variation data from hundreds of thousands of samples in future.

The variation graph representation in VariantStore is smaller and more efficient to construct than the representation in state-of-the-art VG toolkit. It can be further used in read alignment as a replacement to the variation graph representation in the VG toolkit.

While the current implementation does not support adding new variants or update the reference sequence in an existing VariantStore, this is not a fundamental limitation of the design. In future, we plan to extend the immutable version of VariantStore to support dynamic updates following the LSM-tree design [25].

Methods

Variation graph

A variation graph (VG) [19] (also defined as a genome graph in Kim et al. [20] and Rakocevic et al. [21]) is a directed, acyclic graph (DAG) $G = (N, E, P)$ that embeds a set of DNA sequences. It comprises of a set of nodes N, a set of direct edges E, and a set of paths P. For DNA sequences, we use the alphabet $\{$A, C, G, T, N$\}$. Each $n_i \in N$ represents a sequence $seq(n_i)$. Edges in the graph connect nodes that are followed on a path. Each sample in the VCF file follows a path through the variation graph. The embedded sequence given by the path is the sample sequence. Given that a variation graph is a directed graph edges can be

traversed in only one direction. Although, not applicable to VariantStore, an edge can also be traversed in the reverse direction when the variation graph is used for read alignment [19–21].

Each path $p = n_s n_1 \dots n_p n_d$ in the graph is an embedded sequence defined as a sequence of nodes between a source node n_s and a destination node n_d. Nodes on a path are assigned positions based on the coordinate systems of sequences they represent [26]. The position of a node on a path is the sum of the lengths of the sequences represented by nodes traversed to get to the node on the path. For a path $p = n_1 \dots n_p$, position $P(n_p)$ is $\sum_{i=1}^{p-1} |seq(n_i)|$. A node in the graph can appear in multiple paths and therefore can have multiple positions based on different coordinate systems.

An initial variation graph is constructed with a single node and no edges using a linear reference sequence and a reference genome coordinate system. Variants are added to the variation graph from one or more VCF files [16]. A variant is encoded by a node in the variation graph that represents the variant sequence and is connected to nodes representing the reference sequence via directed edges. Each variant in the VCF file splits an existing reference sequence node into two (or three in some cases) and joins them via an alternative path corresponding to the variant. For example, a substitution or deletion can cause an existing reference node to be split into three parts (Figure 5). An insertion can cause the reference node to be split into two parts (Figure 5).

Representing multiple coordinate systems in a variation graph

Representing multiple coordinates systems in a variation graph poses challenges that are not present in linear reference genomes. First, a node can appear on multiple paths at a different position on each path. Second, given that a variation graph can contain thousands of paths and coordinate systems it would be non-trivial to maintain a position index to quickly get to a node corresponding to a path and position.

Much work has been done in coming up with efficient approaches to handle multiple coordinate systems. Rand et al. [26] introduced the offset-based coordinate system. VG toolkit [22] implemented the multiple coordinate system by explicitly storing a list of node identifiers and node offsets for each path in the variation graph. They store the list of node identifiers as integer vectors using the succinct data structure library (SDSL [27]). We call this an *explicit-path representation*.

However, storing the list of nodes on each path explicitly can become a bottleneck as the number of input

Reference sequence	CAATTTGCTGATCT					
Position	Reference seq.	Alternative seq.	HG00096	HG00101	HG00103	Variant type
2	A	G	0	1	1	SUBSTITUTION
2	AATT	A	1	0	0	DELETION
6	T	TACG	0	0	1	INSERTION

Table 2: Variants ordered by the position in the reference genome for three samples (HG00096, HG00101, HG00103). Each variant has the list of samples that contain the variant.

paths increases. Moreover, nodes that appear on multiple paths are stored multiple times causing redundancy in storage. For a set of N variants and S samples the space required to store the explicit-path representation is $O(SN)$ since each variant creates a constant number of new nodes in the variation graph.

VariantStore

We describe how we represent a variation graph in VariantStore and maintain multiple coordinate systems efficiently. We then describe how we build a position index using succinct data structures.

The variation graph representation is divided into three components:

1. Variation graph topology
2. Sequence buffer
3. List of variation graph nodes

Variation graph topology. A variation graph constructed by inserting variants from VCF files often shows high sparsity (the number of edges is close to the number of nodes). For example, the ratio of the number of edges to nodes in the variation graph on 1000 Genomes data [23] is close to 1. Given the sparsity of the graph, we store the topology of the variation graph in a representation optimized for sparse graphs.

Our graph representation uses the counting quotient filter (CQF) [28] as the underlying container for storing nodes and their outgoing neighbors. The CQF is a compact representation of a multiset S. Thus a CQF supports inserts and queries. A query for an item x returns the number of instances of x in S. The CQF uses a variable-length encoding scheme to count the multiplicity of items. In our graph representation, we encode the node as the item and id of the outgoing neighbor as the count of the item in the CQF.

For nodes with a single outgoing neighbor, we store the node and its neighbor together (without an indirection) so we can access them quickly. For nodes with more than one outgoing neighbor we store outgoing

neighbors in separates lists. In the variation graph, most nodes have a single outgoing neighbor and using this compact and optimized representation we achieve cache efficient and fast traversal of the graph.

We use the version of the counting quotient filter with no false-positives to map a node to its outgoing edge(s). We store the node id as the key in the CQF and if there is only one outgoing edge we encode the outgoing neighbor id as the count of the key. If there are more than one outgoing edges we use indirection. We maintain a list of vectors where each vector contains a list of outgoing neighbor identifiers corresponding to a node. We store the node id as the key and the offset in the list of vectors (or index of the vector containing the list of outgoing neighbor ids) as the count of the key.

Sequence buffer. The sequence buffer contains the reference sequence and all variant sequences corresponding to each substitution and insertion variant. All sequences are encoded using 3-bit characters in an integer vector from SDSL library [27, 29]. The integer vector initially only contains the reference sequence. Sequences from incoming variants are appended to the integer vector. Once all variants are inserted the integer vector is bit compressed before being written to disk.

List of variation graph nodes. Each node in the variation graph contains an offset and length. The offset points to the start of the sequence in the sequence buffer and length is the number of nucleotides in the sequence starting from the offset. This uniquely identifies a node sequence in the sequence buffer.

At each node we also store a list of sample identifiers that have the variant, position of the node on all those sample paths, and phasing information from the VCF file corresponding to each sample.

Our representation of the list of samples is based on two observations. First, multiple samples share a variant and storing a list of sample identifiers for each variant is space inefficient. Instead, we store a bit vector of length equal to the number of samples and set bits corresponding to the present samples in the bit vector. Second, multiple variants share the same set of samples. We define an equivalence relation $\sim$ over the set of variants. Let $E(v)$ denote the function that maps each variant to the set of samples that have the variant. We say that two variants are equivalent (i.e., $v_1 \sim v_2$) if and only if $E(v_1) = E(v_2)$. We refer to the set of samples shared by variants as *sample class*. A unique id is assigned to each sample class and nodes store the sample class id instead of the whole sample class. This scheme is previously been employed by other colored de Bruijn graph representation tools [30–32] for efficiently maintaining a mapping from k-mers (a k-length substring sequence) to the set of samples where k-mers appear.

Phasing information is encoded using 3 bits. Position and phasing information corresponding to each sample in the list of samples is stored as tuples. Tuples are stored in the same order as the samples appear in the sample class bit vector. To retrieve the tuple corresponding to a sample a rank operation is performed on the sample bit vector to determine the rank of the sample. Using the rank output, a select operation is performed on the tuple list to determine the tuple corresponding to a sample.

Variation graph nodes are stored as protocol buffer objects using Google's open-source protocol buffers library. Every time a new node is created we instantiate a new protocol buffer object in memory. We compress the protocol buffers before writing them to disk and decompress them while reading them back in memory.

For a set of N variants and S samples where each variant is shared by P samples on average, each node contains information about P samples and storing $O(N)$ nodes (a constant number of nodes for each variant) the space required to store the variation graph representation in VariantStore is $O(NP)$. When $P = 1$ (i.e., no two samples share a variant) the space required by the variation graph representation becomes $O(N)$.

Position index In order to answer variant queries we need an index to quickly locate nodes in the graph corresponding to input positions. These position can be specified in multiple coordinate systems, i.e., in the coordinate system of the reference or a sample.

One way to index the variation graph, is to store an ordered mapping from position $\rightarrow$ node identifier. We can perform a binary search in the map to find the position closest to the queried position and the corresponding node id in the graph. However, given that there are multiple coordinate systems in the variation graph we can not create a single mapping with a global ordering. Keeping a separate position index for each coordinate system will require space equal to the explicit-path representation.

In VariantStore, we maintain a mapping of positions to node identifiers only for the reference coordinate system. All nodes on the reference sequence path are present in the mapping. If the queried position is in the reference coordinate system we use the mapping to locate the node in the graph. However, if the queried position is in a sample's coordinate system, we first locate the node in the graph corresponding to the same position in the reference coordinate system. Then we perform a local search by traversing the sample path from that node to determine the node corresponding to the position in sample's coordinate system. The local graph search incurs a small one-time cost because sample nodes are rarely far from a reference node and is

amortized against future searches in the sample's coordinate system.

We create the position index using a bit vector called the position-bv of length equal to the reference sequence length and a list of node identifiers on the reference path in the increasing order by their reference sequence positions. For every node in the list we set the bit corresponding to the node's position in the position-bv. There is a one-to-one correspondence between every set bit in the position-bv and node positions in the list. We store the position-bv using a bit vector and node list as an integer vector from the SDSL library [27, 29].

Variation graph construction

We construct the variation graph by inserting variants from a VCF file. Each variant has a position in the reference genome, alternative sequence (except in case of a deletion), and a list of samples with phasing information for each sample.

Based on the position of the variant we split an existing reference node that contains the sequence at that position in the graph. We update the split nodes on the reference path with new sequence buffer offsets, lengths, and node positions (based on the reference coordinate system). We then append the alternative sequence to the sequence buffer and create an alternative node with the offset and length of the alternative sequence. We then add the list of tuples (position, phasing info) for each sample.

We also need to determine the position of the alternative node on the path of each sample that contains the variant. One way to determine the position of the node for each sample would be to backtrack in the graph to determine a previous node that contains a sample variant and the absolute position of that node in the sample's coordinate system. If no node is found with a sample variant we trace all the way back to the source of the graph. We would then traverse the sample path forward up to the new alternative node and compute the position. This backtracking process would need to be performed once for each sample that contains the variant. This would slow down adding a new variant and cause the construction process to not scale well with increasing number of samples.

Instead, we construct the variation graph in two phases to avoid the backtracking process. In the first phase, while adding variants we do not update the position of nodes on sample paths. We only maintain the position of nodes on the reference path because that does not require backtracking. In the second phase, we perform

a breadth-first traversal of the variation graph starting from the source node and update the position of nodes on sample paths.

During the breadth-first traversal we maintain a delta value for each sample in the VCF file. At any node, the *delta value* is the difference between the position of the node in the reference coordinate and the sample's coordinate. During the traversal, we update sample positions for each node based on the current delta value and reference coordinate value. Algorithm 1 gives the pseudocode of the algorithm.

Algorithm 1 Fix sample positions

```
for i in Samples do
    delta[i] ← 0
for node in BFS(variation graph) do
    if IsReference(node) then
        for neighbor in node.neighbors do
            if pos[sample] = 0 then
                pos[sample] ← pos[ref] + node.len + delta[sample]
            else
                delta[sample] ← pos[sample] − (pos[ref] + node.len)
    else
        delta[sample] ← pos[sample] + pos[ref]
```

Position index construction

In the position index, we maintain a mapping from positions of nodes on the reference path to corresponding node identifiers in the graph. Node positions are stored in a "position-bv" bit vector of size equal to the length of the reference sequence and node identifies are stored in a list. To construct the position index, we follow the reference path starting from the source node in the graph and for every node on the path we set the corresponding position bit in the position-bv and add the node identifier to the list. Node identifiers are stored in the order of their position on the reference path.

Variant queries

A query is performed in two steps. We first perform a predecessor search (largest item smaller than or equal to the queried item) using the queried position in the position index to locate the node n_p with the highest position smaller than or equal to the queried position pos. The predecessor search is implemented using the rank operation on position-bv. For bit vector $B[0, \ldots, n]$, $RANK(j)$ returns the number of 1s in prefix

$B[0, ..., j]$ of B. An RRR compressed position-bv supports rank operation in constant time [27, 33]. The rank of pos in position-bv corresponds to the index of the node id in the node list. Figure 6a shows a sample query in the position index.

Based on how reference nodes are split while adding variants the sequence starting at pos will either be contained in the node n_p or the next node after n_p on the reference path. All queries are then answered by traversing the graph either by following a specific path (reference or a sample) or a breadth-first traversal and filtering nodes based on query options.

If the queried position is based on the reference coordinate system then we can directly use n_p as the start node for graph traversal. However, if the position is based on a sample coordinate then we perform a local search in the graph starting from n_p to determine the start node based on the sample coordinate.

Memory-efficient construction and query

In the variation graph representation, the biggest component in terms of space is the list of variation graph nodes stored as Google protobuf objects. These node objects contain the sequence information and the list of sample positions and phasing information. For 1000 Genomes data, the space required for variation graph nodes is $\approx 87\%$ to 92% of the total space in VariantStore. However, keeping the full list of node objects in memory during construction or query is not necessary and would make these processes memory inefficient.

To perform memory efficient construction and query, we store and serialize these nodes in small chunks usually containing $\approx$ 200K nodes (the number of nodes in a chunk varies based on the data to keep the size to a few MBs). Nodes in and across these chunks are kept in their creation order (which is roughly the breadth-first traversal order). Therefore, during a breadth-first traversal of the graph we only need to load these chunk in sequential order.

During construction, we only keep two chunks in memory, the current active chunk and the previous one. All chunks before the previous chunk are written to disk. In the second phase of the construction when we update sample positions and during the position index creation we perform a breadth-first traversal on the graph and load chunks in sequential order.

Variant queries involve traversing a path in the graph between a start and an end position or exploring the

graph locally around a start position. All these queries require bounded exploration of the graph for which we only need to look into one or a few chunks.

To perform queries with a constant memory we only load the position index and variation graph topology in memory and keep the node chunks on disk. We use the index and the graph topology to determine the set of nodes to look at to answer the query. We then load appropriate chunks from disk which contain the start and end nodes in the query range. For queries involving local exploration of the graph we load the chunk containing the start node. During the exploration, we load new chunks lazily as needed. At any time during the query, we only maintain two contiguous chunks in memory.

References

[1] 1000 Genomes Project Consortium. A map of human genome variation from population-scale sequencing. *Nature*, 467(7319):1061, 2010.

[2] 1000 Genomes Project Consortium. An integrated map of genetic variation from 1,092 human genomes. *Nature*, 491(7422):56, 2012.

[3] 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526 (7571):68, 2015.

[4] John Lonsdale, Jeffrey Thomas, Mike Salvatore, Rebecca Phillips, Edmund Lo, Saboor Shad, Richard Hasz, Gary Walters, Fernando Garcia, and Nancy Young. The genotype-tissue expression (GTEx) project. *Nature Genetics*, 45(6):580, 2013.

[5] TCGA: the cancer genome atlas program, 2019. URL https://www.cancer.gov/about-nci/organization/ccg/research/structural-genomics/tcga. [Online accessed August 2019].

[6] Ananyo Choudhury, Michèle Ramsay, Scott Hazelhurst, Shaun Aron, Soraya Bardien, Gerrit Botha, Emile R Chimusa, Alan Christoffels, Junaid Gamieldien, and Mahjoubeh Sefid-Dashti. Whole-genome sequencing for an enhanced understanding of genetic variation among South Africans. *Nature Communications*, 8(1):2062, 2017.

[7] Sebastian Roskosch, Hákon Jónsson, Eythór Björnsson, Doruk Beyter, Hannes P Eggertsson, Patrick

Sulem, Kári Stefánsson, Bjarni V Halldórsson, and Birte Kehr. PopDel identifies medium-size deletions jointly in tens of thousands of genomes. *bioRxiv*, page 740225, 2019.

[8] Cristian Groza, Tony Kwan, Nicole Soranzo, Tomi Pastinen, and Guillaume Bourque. Personalized and graph genomes reveal missing signal in epigenomic data. *bioRxiv*, page 457101, 2019.

[9] Frank W Albert and Leonid Kruglyak. The role of regulatory variation in complex traits and disease. *Nature Reviews Genetics*, 16(4):197, 2015.

[10] Karen Eilbeck, Aaron Quinlan, and Mark Yandell. Settling the score: variant prioritization and mendelian disease. *Nature Reviews Genetics*, 18(10):599, 2017.

[11] Claudia MB Carvalho and James R Lupski. Mechanisms underlying structural variant formation in genomic disorders. *Nature Reviews Genetics*, 17(4):224, 2016.

[12] Jerome Kelleher, Yan Wong, Anthony W Wohns, Chaimaa Fadil, Patrick K Albers, and Gil McVean. Inferring whole-genome histories in large population datasets. *Nature Genetics*, 51(9):1330–1338, 2019.

[13] Anthony J Brookes and Peter N Robinson. Human genotype–phenotype databases: aims, challenges and opportunities. *Nature Reviews Genetics*, 16(12):702, 2015.

[14] Joachim Kutzera and Patrick May. Variant-DB: A tool for efficiently exploring millions of human genetic variants and their annotations. In *International Conference on Data Integration in the Life Sciences*, pages 22–28. Springer, 2017.

[15] Geert Vandeweyer, Lut Van Laer, Bart Loeys, Tim Van den Bulcke, and R Frank Kooy. VariantDB: a flexible annotation and filtering portal for next generation sequencing data. *Genome Medicine*, 6(10): 74, 2014.

[16] The variant call format (VCF) version 4.1 specification, 2019. URL https://samtools.github.io/hts-specs/VCFv4.1.pdf. [Online accessed March 2019].

[17] Alexander Dilthey, Charles Cox, Zamin Iqbal, Matthew R Nelson, and Gil McVean. Improved genome inference in the MHC using a population reference graph. *Nature Genetics*, 47(6):682, 2015.

[18] Hannes P Eggertsson, Hakon Jonsson, Snaedis Kristmundsdottir, Eirikur Hjartarson, Birte Kehr,

443 Gisli Masson, Florian Zink, Kristjan E Hjorleifsson, Aslaug Jonasdottir, and Adalbjorg Jonasdottir.
444 Graphtyper enables population-scale genotyping using pangenome graphs. *Nature Genetics*, 49(11):
445 1654, 2017.

446 [19] Erik Garrison, Jouni Sirén, Adam M Novak, Glenn Hickey, Jordan M Eizenga, Eric T Dawson, William
447 Jones, Shilpa Garg, Charles Markello, Michael F Lin, Benedict Paten, and Richard Durbin. Varia-
448 tion graph toolkit improves read mapping by representing genetic variation in the reference. *Nature*
449 *Biotechnology*, 36:875–879, 2018.

450 [20] Daehwan Kim, Joseph M Paggi, Chanhee Park, Christopher Bennett, and Steven L Salzberg. Graph-
451 based genome alignment and genotyping with HISAT2 and HISAT-genotype. *Nature Biotechnology*,
452 37(8):907–915, 2019.

453 [21] Goran Rakocevic, Vladimir Semenyuk, Wan-Ping Lee, James Spencer, John Browning, Ivan J Johnson,
454 Vladan Arsenijevic, Jelena Nadj, Kaushik Ghose, and Maria C Suciu. Fast and accurate genomic
455 analyses using genome graphs. *Nature Genetics*, 51:354–362, 2019.

456 [22] Variation graph toolkit, 2019. URL https://github.com/vgteam/vg. [Online accessed March 2019].

457 [23] IGSR: the international genome sample resource, 2019. URL http://www.internationalgenome.org/
458 home. [Online accessed March 2019].

459 [24] BCF toolkit, 2019. URL https://samtools.github.io/bcftools/bcftools.html. [Online accessed March
460 2019].

461 [25] Patrick ONeil, Edward Cheng, Dieter Gawlick, and Elizabeth ONeil. The log-structured merge-tree
462 (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

463 [26] Knut D Rand, Ivar Grytten, Alexander J Nederbragt, Geir O Storvik, Ingrid K Glad, and Geir K
464 Sandve. Coordinates and intervals in graph-based reference genomes. *BMC Bioinformatics*, 18(1):
465 263, 2017.

466 [27] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play
467 with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA*
468 *2014)*, pages 326–337, 2014.

[28] Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. A general-purpose counting filter: Making every bit count. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 775–787. ACM, 2017.

[29] SDSL: succinct data structure library, 2019. URL https://github.com/simongog/sdsl-sdsllite. [Online accessed March 2019].

[30] Fatemeh Almodaresi, Prashant Pandey, and Rob Patro. Rainbowfish: a succinct colored de Bruijn graph representation. In *17th International Workshop on Algorithms in Bioinformatics (WABI 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[31] Prashant Pandey, Fatemeh Almodaresi, Michael A Bender, Michael Ferdman, Rob Johnson, and Rob Patro. Mantis: A fast, small, and exact large-scale sequence-search index. *Cell Systems*, 7(2):201–207, 2018.

[32] Fatemeh Almodaresi, Prashant Pandey, Michael Ferdman, Rob Johnson, and Rob Patro. An efficient, scalable and exact representation of high-dimensional color information enabled via de Bruijn graph search. In *International Conference on Research in Computational Molecular Biology*, pages 1–18. Springer, 2019.

[33] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms (TALG)*, 3(4):43, 2007.

Data availability

Code and data are available at https://github.com/Kingsford-Group/variantstore

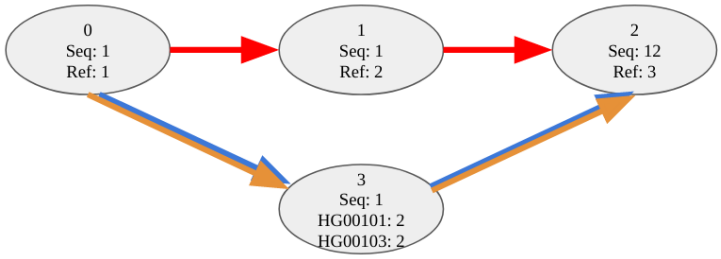
Acknowledgements

The results published here are in whole or part based upon data generated by The Cancer Genome Atlas (dbGaP accession phs000178) managed by the NCI and NHGRI. Information about TCGA can be found at http://cancergenome.nih.gov The 1000 Genomes data used for the analyses described in this manuscript

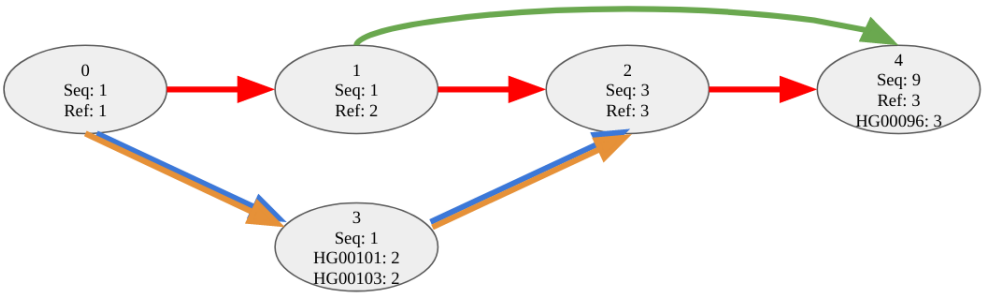
were obtained from https://www.internationalgenome.org/. This research is funded in part by the Gordon and Betty Moore Foundation's Data-Driven Discovery Initiative through Grant GBMF4554 to C.K. and by the US National Institutes of Health (R01GM122935). We would also like to thank Shawn Baker for many helpful discussions, and Guillaume Marçais and Yutong Qiu for comments on the manuscript.

Competing Interests

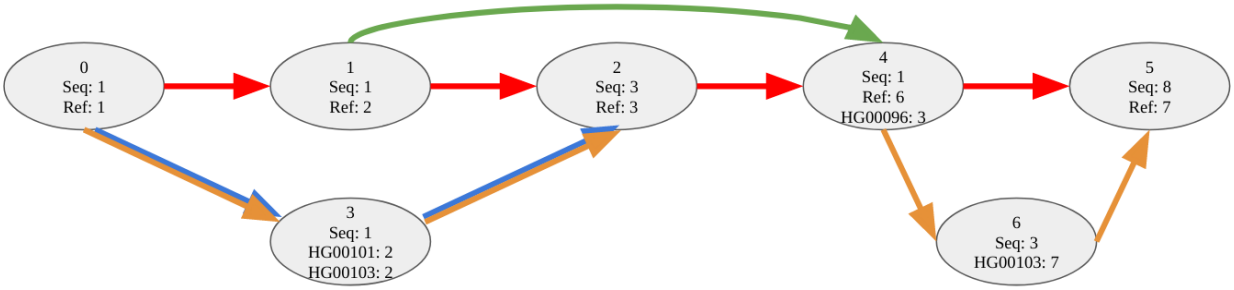
C.K. is a co-founder of Ocean Genomics, Inc.



(a) SUBSTITUTION

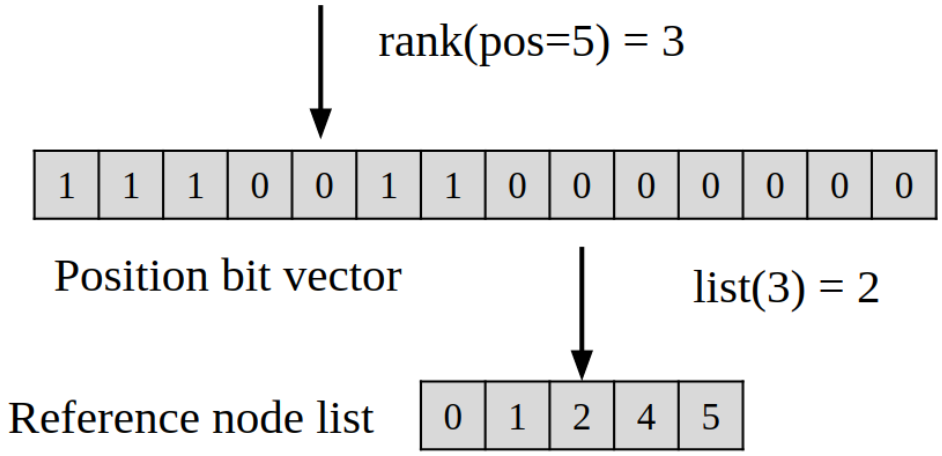


(b) DELETION

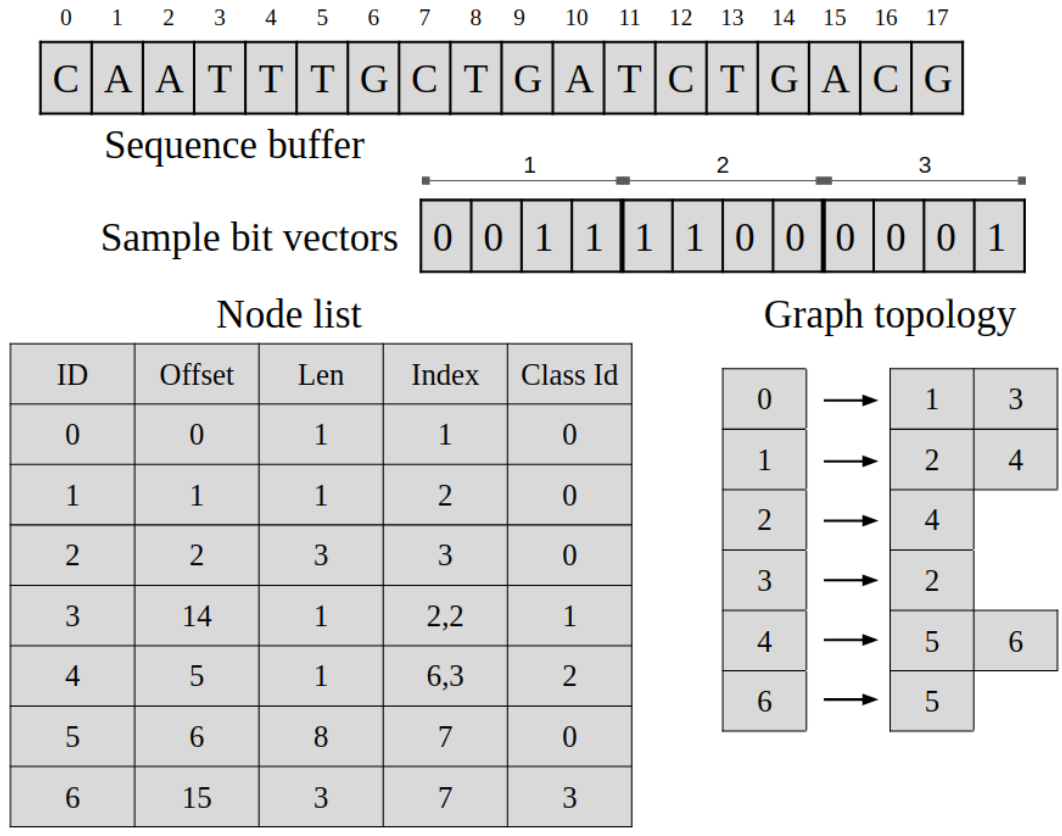


(c) INSERTION

Figure 5: A variation graph with three input samples (HG00096, HG00101, HG00103) showing the encoding of substitutions, insertions, and deletions as stated in Table 2. Fig. (a) shows the substitution, (b) shows the deletion, and (c) shows the insertion. Edges are colored (or multi-colored) to show the path taken by reference and samples through the graph. (Ref: red, HG00096: green, HG00101: blue, HG00103: brown). Samples with no variant at a node follow the reference path, e.g., sample HG00096 will follow the reference path between nodes $0 \rightarrow 1$ and $4 \rightarrow 5$. Each node contains node id, the length of the sequence it represents, and a list of samples and their positions.



(a) Position Index



ID	Offset	Len	Index	Class Id
0	0	1	1	0
1	1	1	2	0
2	2	3	3	0
3	14	1	2,2	1
4	5	1	6,3	2
5	6	8	7	0
6	15	3	7	3

(b) Variation graph representation

Figure 6: Position index and variation graph representation in VariantStore for the sample graph from Figure 5. (a) Shows the query operation for finding the node at position 5 in the sequence. (b) Phasing information is omitted from the node list for simplicity of the figure. In implementation, phasing information using three bits for each sample in each node.