ABSTRACT

FERRITER, KYLE ROBERT. Query and Storage Optimization of Genetic Variant Data via Structural and Semantic Compression and Indexing. (Under the direction of Frank Mueller.)

As genomics and personalized medicine are becoming ubiquitous around the world, and datasets are growing at an accelerating pace to fulfill the needs of the medical industry and its patients, so too grows the amount of data needing to be stored and used both within institutions and by individuals on their personal computing devices. This data must be retained long term but also needs to be quickly and cheaply accessible, two goals which in many ways compete with each other.

After genomic sequencing is performed, genomic analytics requires the ability to query large datasets for information about a particular genetic sequence at certain positions. Series of queries are combined in order to perform correlation analysis and comparative analysis in order to track genetic variation over time and between organisms, as well as the effect those variations have on the physical manifestation of traits in organisms.

Traditional database technologies are not well suited for this use case because they do not take into account usage patterns or structure of the data. These traditional databases, when storing genomic files and servicing queries on them, place significant demand on system resources and take significant amounts of time to return results. To address these problems, the bioinformatics community has developed industry-standard methods.

However, these existing industry-standard methods for compressing and indexing genetic variant files primarily use generic compression schemes, which have the benefit of being robust, reliable, and in widespread use, but novel schemes for compression as well as indexing that better take into account usage patterns as well as the structure and semantics of the data itself can lead to even better turnaround time for queries.

This work contributes a novel line-based run-length partial-compression technique for variant genotype data that performs well on large sample sets, and several novel indexing strategies. Each is accompanied by a comparison between it and the industry standard. The evaluation of these contributions are performed on two different storage technologies and two different modern filesystems, and the difference between these for storing genetic variant data and the novel indexes is also evaluated and discussed. When evaluated using the 1000 Genomes Project VCF dataset, the compression technique achieves a compression ratio of over 96%, and this compression technique, combined with the indexing strategies, leads to an approximate speedup of 2X in turnaround time on both single variant and range-based variant lookups.

Query and Storage Optimization of Genetic Variant Data via Structural and Semantic Compression and Indexing

by
Kyle Robert Ferriter

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2020

APPROVED BY:

Guoliang Jin	Steffen Heber
Amir Bahmani External Member	Frank Mueller Chair of Advisory Committee

DEDICATION

To my parents for their years of support and encouragement.

To all of the medical professionals, support staff, and volunteers on the front lines of fighting disease.

BIOGRAPHY

Kyle Ferriter grew up in Chapel Hill, North Carolina. After graduating from Carrboro High School, he attended North Carolina State University in Raleigh, NC, where he received a Bachelor of Science in Computer Science with a minor in Mathematics in 2016. Afterwards, he took half of a year off to hike the Appalachian Trail and spend quality time in nature away from most technology.

After returning to Raleigh, he began work as a software developer with the Renaissance Computing Institute (RENCI) at UNC Chapel Hill, where he became exposed to the field of bioinformatics and medical data cloud platforms. His parents both work in medical-related fields and he began to wish he had heeded their advice in taking some more biology and chemistry courses alongside his computer science curriculum.

He began a Master of Science graduate program in Computer Science at North Carolina State University in 2018 while a research assistant with RENCI under the advising of Dr. Claris Castillo. He interned with the Stanford Center for Genomics and Personalized Medicine (SCGPM), and he began research study under the advising of Dr. Frank Mueller in the summer of 2019. After the completion of his degree, he will begin work as a software engineer with the Broad Institute in Cambridge, MA.

ACKNOWLEDGEMENTS

I would first like to thank my advisor Dr. Frank Mueller for working with me on this research, and giving his help and guidance.

I would also like to thank Dr. Steffen Heber, and Dr. Guoliang Jin for their instructive courses I took with them at NC State University, and Dr. Amir Bahmani, and Dr. Cuiping Pan for their project advising during my collaboration with SCGPM. And I thank them all for taking part in my thesis committee.

I also want to thank others who gave me opportunities and encouragement to research and pursue new ideas over the past several years, including Dr. Claris Castillo, Steve Cox, Fan Jiang, Ray Idaszak, and Dr. Sarah Heckman.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1

INTRODUCTION

Over the last decade, and increasingly so in recent years, genomic datasets have rapidly grown in size, and this trend is expected to continue and accelerate into the future. What drives this trend is the increased use of genetic sampling, sequencing, and analysis as genetics becomes more integrated into common health and medical processes ranging from emergency room or hospital visits to simple periodic checkups. Researchers specializing in areas such as precision medicine, rare diseases, and infectious diseases, are placing increased demands on data infrastructure. These demands include the precision with which data can be semantically queried from storage, speed with which it can be utilized in work flows, and cost reductions enabling wider use.

In order to offload storage and computational resources from researchers' personal machines, genetic and other forms of large medical data are being moved into the cloud. The advantages are clear: it can scale to store a practically unbounded amount of data, it provides a single point of reference for teams of researchers, which need to read and in some cases write to the same dataset, it can help ensure better security and privacy of sensitive data, and the data is easily accessible by processing pipelines already being run on cloud instances or clustered environments.

However, this move to the cloud is not without disadvantages. One such disadvantage is the cost associated with storing data long-term. There are costs also associated with storing data on local disks or in an on-premise data center, but the durability, availability, and scalability of managed cloud cloud storage comes at a premium cost. For this reason, data compression at sufficiently high ratios is a critical feature of big-data tooling.

Another disadvantage is the cost of compute resources in cloud environments. These costs, in many cases, can significantly outweigh storage costs. By reducing demand on CPU, memory, and network bandwidth, the same amount of work can be performed with fewer cloud compute

resource allocations, thus lowering the cost. In a clustered environment this also means freeing up time slots for other jobs and possibly enabling cluster scale-down.

1.1 Motivation

For genomic data, data warehousing services have grown in popularity due to their ability to manage and provide an SQL or similar query API to large amounts of data. However services like these emulate traditional tabular database systems. The downside to this is that it is difficult to take into consideration the particular schematics and patterns of the data. Database systems whose schemas support object nesting and structured records can attempt to achieve better efficiency; however, in practice, unavoidable downsides remain:

1. The cost of provisioning or running a service like this is expensive. Fees charged to managed service clients are significant.

2. Even when fast, accessing data is slower than native I/O of optimally domain-structured and indexed files. Part of this is due to I/O bandwidth, but another factor is the file seek performance and minimization of total data read from storage, both of which can suffer on non-local storage connections.

3. Immense time and memory demands. Highly intelligent database engines can optimize queries and query plans, but without knowing or targeting a specific use case these will still load very large sets of data into memory which can only be filtered after the fact during a later stage of execution. Large portions of the memory footprint may be dedicated entirely to the discarded data, which can be considered *wasted memory*. The fact that this memory still must be provisioned from a cluster or available in a bare metal machine allocated to the query means that this is a wasted *cost* as well.

4. These systems must necessarily throw out some of that domain structure information in order to conform the data to a generic schema. One crucial example is sorting, which is commonly utilized in genetic data files, but which is nonexistent in most generic database systems unless a very expensive sort operation is performed. Due to the large size of the data, in some cases it is not feasible to sort the data within a table because all available working memory is exhausted, even within working memory arrays ranging into the hundreds of gigabytes.

   Storage technologies have over recent years trended away from spinning HDD devices, towards several tiers of SSD technologies, and away from SATA storage interfaces towards NVME, which supports higher speeds. These local storage devices can be leveraged in clustered environments by providing a working space for swapping data objects in and out of main memory, and for providing backing storage for networked filesystems, which are also often used for working object storage or intermediate data workspaces, but the access latency and

I/O bandwidth of these compared to main memory is inferior for servicing random access patterns on large scale datasets compared to native intra-host I/O operations.

Because these forms of medical data are intended to be used throughout the lifespan of an individual in order to achieve the best positive health outcomes, the data needs to be stored long term in a cheap manner, but also be readily accessible to low-latency sequential and random access patterns. On top of these features, storage technologies and formats must also support high concurrency and resource sharing by placing limited demands on read I/O, CPU, and memory resources. To do this, two mechanisms are used:

1. The first mechanism is compression. The core goal of this mechanism is to reduce the size of data in storage without losing information. Techniques often include combining duplicate values, or otherwise encoding digital symbols in the uncompressed data into fewer bits in the compressed form. The benefit of compression generally comes from reduced overall processing time, because in many cases, especially with large data objects common in big data use cases, a significant portion of overall turnaround time is spent on I/O. I/O operations include but are not limited to: reading a data object out of storage into memory, writing from memory into storage, and transferring between memory arrays over a network, which is common in clustered environments and frameworks such as Spark [1] and HDFS [8]. For these I/O operations, not only is there some cost associated with storage, but depending on the circumstances, there may be a direct cost associated in performing the I/O itself, as is common when reading or writing crosses a cloud region boundary. Though reduced storage and transfer costs are a benefit on their own, compression algorithms must seek to also maintain the time benefit and not cancel it out by requiring significant amounts of processing to perform the decompression once the compressed data is read into memory.

2. The second mechanism is indexing. This creates a way to find relevant data within a larger dataset, without needing to read through the entire set. In traditional database systems, a table index takes a column (or field), and for each value or set of contiguous values, stores pointers to table records, mapped to each value in a structure like a search tree or hash table. After indexing, if a database client wishes to select records, based on a value in a column which is indexed, instead of searching the entire column, the database engine can quickly filter to a set of table records which meet the criteria, without reading any records from the table itself. The primary advantage of indexing is usually considered significantly reduced I/O bandwidth and read operation turnaround time, as filtering is done at least partially within the smaller index instead of the table, but corollary benefits to this I/O decrease is reduced CPU and memory demand because less data is being read into main memory and processed.

This document covers these two mechanisms, within the specific application of genetic data, and more specifically a terminal format of genomic sequencing pipelines often used in gene research and genetic variation research. On the topic of compression, existing techniques are described. The

work contributes a different strategy, which takes into account particular known constraints and patterns of the input data. On the topic of indexing, existing techniques are briefly covered. The work then contributes new indexing strategies, which can again take into consideration certain value constraints of the data, features of the proposed compression technique, and also use case patterns associated with genetic data.

1.2 Hypothesis

> A novel encoding technique of genetic variant data can be devised based on structure- and semantic-aware compression and indexing such that response times for common queries outperform those on existing semantic-agnostic storage formats.

1.3 Contributions

This work makes the following contributions:

- A novel row-based run-length partial compression technique is developed that performs comparably well to block-based full compression techniques on large genomic datasets.
- Three row-based indexing strategies for genetic variant data stored in row-compressed format are devised that leverage both key binning and a novel use of sparse file offsets for indexing.
- A comparison is provided of NVME and SATA SSD technologies, and EXT4 and XFS filesystems, for the storage and retrieval of indexed and compressed genomic data, in both dense and sparse files.

CHAPTER

2

BACKGROUND

Genomics is the field of science dedicated to the study of genomes, which are sequences of DNA within cells. Human DNA is split across 23 chromosome pairs, 22 of which are relatively symmetric autosomes and one is a pair of sex chromosomes, which are not the same in males and females. There are other places that DNA and other biomolecular sequences exist within humans, but they are much smaller and other than perhaps mitochondrial DNA, these are not the primary focus of information format optimization within genomics or sequence bioinformatics; because of their small size they do not pose significant problems regarding digital storage and retrieval of information.

Within each pair of chromosomes, one comes from the genetic father and one from the genetic mother. The human chromosomal sequences are made up of micromolecules represented symbolically by the letters A, T, C, and G. Individual molecules in the sequence are referred to as *bases*. DNA is comprised of two strands of base molecules wound around each other in a double helix pattern, now a widely recognized image throughout the general population. A in one strand is paired with T in the other strand at the same location, and similarly C and G are paired with each other. Due to this fact, one strand is able to be inferred by knowing the contents of the other strand, and thus only one of the strands needs to be recorded. By convention, one strand is chosen as the forward, and the other is chosen as the reverse.

The contents of these sequences control many biological characteristics. The value of knowing the genetic makeup of an individual is that correlation analyses can be run between a person's physical characteristics and their sequence characteristics, especially focusing on places where an individual differs from a majority of the global or localized genetic populations. Using these types of analyses, physical ailments or physical responses to stimuli like medication can be linked to genetic traits, and such linkages when obtained with high confidence scores can be used predictively

with other individuals who have similar genetic traits. The process of attaching rich semantic information to genomes is known as *annotation*. To date, many such analyses have been performed, and annotation datasets are growing quickly alongside the raw genetic datasets. This predictive power informs processes like precision medicine and preventative healthcare, which given accurate sequence information and annotations can lead both to improved health outcomes and reduced cost, by means of both early detection, and reducing time and medical resources by tailoring treatments and care to a particular individual [11] [21].

Genetic *sequencing pipelines* are used to obtain and record the genetic makeup of an individual. The beginning of these pipelines is to obtain physical sequence fragments, and record their contents digitally. Next is to assemble the digital fragment data into a single high quality representation of the organism's overall sequence. Various methods of taking fragments and assembling them into a complete sequence are established in the genomic academic and industrial communities, and continue to be an area of research and improvements. The human genome is just over 3 billion base pairs long, so data structures and algorithms must be used to speed up the process of assembling the fragments, instead of naive solutions such as exhaustively searching for the best-match place for fragment in the overall sequence. Sequencing processes generally include many layers of redundancy in order to make sure that each position is read with high confidence, and to ensure that as many gaps are covered as possible. A metric used here is *read depth*, which affects the quality and thoroughness (minimizing gaps) of the sequencing [19]. The depth refers to the average number of reads that overlap each base position. For example, a read depth of 4 means that the sum of the lengths of the reads divided by the length of the whole genome is 4.

The Broad Institute maintains a list of "Best Practices" for sequencing pipelines, which describe the recommended steps [7]. The major computational steps are summarized below.

1. Assembling short sequences of input reads into an overall sequence, determining at which position each short read belongs. This often includes aligning to a reference (∼3 billion bases in human genome, with a file size of ∼3.1GB) by finding the most similar region based on some inexact matching algorithm. This creates a SAM (Sequence Alignment Map) file, or a binary or compressed BAM/CRAM equivalent.

2. Deduplication of overlapping regions (using read/coverage depth).

3. Sorting the records by chromosome and position (all data is sorted from now on).

4. Quality score normalization.

5. Variant calling. This determines where, and how, a sample differs from the reference, and creates a Variant Call Format (VCF) file.

6. Filtering by quality scores. Some reads are of low quality, which happens because they are obtaining data from an imperfect physical sample. If, for one sample, many reads at a particular location are low quality, a threshold can be used to filter those out because it is not valid to make an assertion about the content of that location on that genome.

7. Annotation of positions or specific alternate bases.

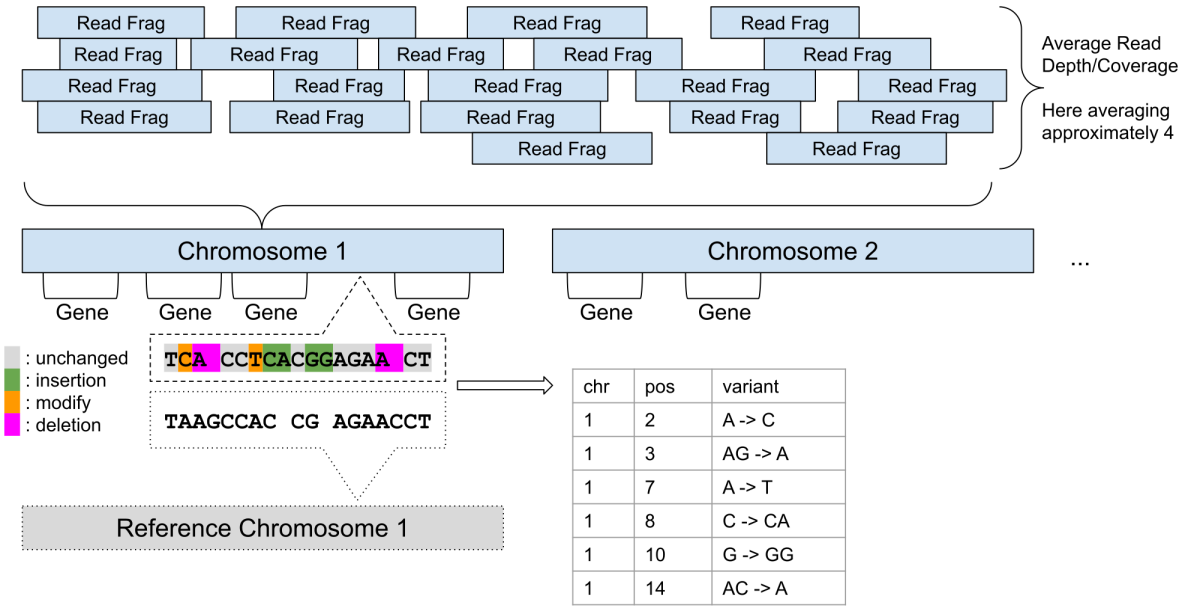

Figure 2.1 Genome Sequencing Pipeline Overview

Figure 2.1 visualizes this process in a vastly simplified manner. At the top are a number of reads; these are digital text representations of DNA fragments read in from a physical sample through a sequencing machine. These fragments are then assembled and matched to positions on the genome through some method. Several methods for this exist, and this is a computationally hard problem, so bioinformatics algorithms and data structures are crucial in performing this in an efficient manner. In one popular method, a reference genome, and a suffix tree constructed from it, are used to determine where subsequences of read fragments occur in the reference, using a best-score optimization algorithm because the match may not be exact. In a standard sequencing read there must be overlaps in reads, by the pigeonhole principle, because the sum of their lengths is multiples times the length of the full genome. The average overlap per position is known as the coverage, shown here to be approximately 4. In a sufficiently random sample of reads, the overlap is spread throughout the genome sequence and not concentrated to any particular region, leading to improved reliability for the whole genome. In shallow reads, gaps can and often do exist, meaning that the sample's genome is not fully sequenced, and these gaps can also recorded in the VCF file as unknown bases. In such a shallow read containing gaps, the gaps can be left in, or statistical methods can be used to infer bases from similar existing samples [13].

After assembling the genome sequence, variant calling is performed between it and the reference. Single base modifications known as Single Nucleotide Polymorphisms (SNPs), insertions and deletions (INDELs) of bases, and other more complicated structural modifications not shown in

Table 2.1 VCF File Snippet

```
##fileformat=VCFv4.3
##FILTER=<ID=PASS,Description="All filters passed">
##INFO=<ID=AC,Number=A,Type=Integer,Description="Total number of alternate alleles">
##INFO=<ID=AN,Number=1,Type=Integer,Description="Total number of alleles">
##fileDate=20200409
#CHROM POS ID  REF  ALT QUAL FILTER  INFO    FORMAT  HG01 HG03 HG04 HG05 HG08 HG15
1  10075  rs12  A    G  100  PASS    AC=2;AN=12; GT  0|0  1|0  0|0   0|0  0|0  0|1
1  10115  rs21  G    A  100  PASS    AC=2;AN=12; GT  0|0  0|0  1|1   0|0  0|0  0|0
1  10213  rs24  C    T  100  PASS    AC=1;AN=12; GT  0|0  0|0  0|0   0|0  0|0  1|0
1  10319  rs28  C    T  100  PASS    AC=1;AN=12; GT  0|0  0|0  0|0   0|0  0|1  0|0
1  10527  rs32  C    A  100  PASS    AC=3;AN=12; GT  0|0  1|1  1|0   0|0  0|0  0|0
1  10568  rs40  C    A  100  PASS    AC=1;AN=12; GT  1|0  0|0  0|0   0|0  0|0  0|0
1  10607  rs42  G    A  100  PASS    AC=1;AN=12; GT  0|0  0|0  0|0   1|0  0|0  0|0
1  10838  rs44;rs46 GA GAA,G 100 PASS AC=1,5;AN=12; GT 2|0 0|2 0|2  0|1  2|0  0|2
```

this figure, are detected and recorded into the VCF file. This investigation looks at the output of the *variant calling* step, simply represented in the variant table in Figure 2.1.

VCF files are laid out as a mapping of genetic variation descriptions to genetic samples that may or may not contain this type of variation. This is done by storing an $N\,x\,M$ matrix of N variants and M samples. Each line is a location on the sequence, recording the way this variant position changes from the reference, and the samples which contain this variant. VCF files are written as tab separated values (TSV), so between each column is a tab character (0x09), and between each line is a newline character (0x0A). An example of a VCF file snippet is given in Table 2.1.

CHAPTER

3

EXISTING SOLUTIONS

3.1 Compression

3.1.1 BGZF (.bgzf / .vcf.gz)

The most commonly distributed compression format for VCF files is called `BGZF` [17][p. 14], also sometimes referred to as `BGZIP` after its implementation's command name. These names may be used interchangeably. This is part of the Samtools project [18] under the `htslib` software repository [9], in an executable named `bgzip`. At a high level, the strategy is to iterate through VCF data lines until 64 KiB is read, and then compress this into a GZIP block. This is done repeatedly, each time concatenating the current GZIP block to the end of the output stream of previous blocks, until the end of the input file is reached. This can achieve high throughput given the level of optimization work that has gone into the GZIP algorithm and library over many years.

GZIP has a theoretical upper bound compression ratio of approximately 99.8%, which occurs when every byte in the stream is the same and the stream is a multiple of the default GZIP block size, which is a rare scenario to encounter in real-world applications. `BGZF` used on the 1000 Genomes Project chromosome 1 VCF file achieves a compression ratio of 98.14%. This is a good ratio because the bytes are not nearly all the same, but the GZIP used under the hood of `BGZF` is still able to recognize repetitions of multi-byte symbols.

An advantage to this method is that GZIP is a mature standard with wide support in tooling, and the concatenated compression output stream of blocks constitutes a valid GZIP decompression input stream, which can be decompressed with standard utilities like the GNU `gzip` or `gunzip` commands. A downside is that lines are packed into blocks and to get one of them, the whole block

must be decompressed.

3.1.2 BCF (.bcf)

Another common format is BCF, replacing the *V* in *VCF* with *B* for *Binary*. This format more takes into consideration the original VCF format by maintaining the line and some of the column-based structure. BCF is documented on page 27 of the VCFv3 format specification [24]. It is also in the Samtools project, but under the `bcftools` repository, and it links against some files from the `htslib` codebase. The strategy is to perform some preliminary compression of lines by first encoding some textual key and label symbols as smaller lookup keys into a metadata dictionary. Then the first 8-9 columns are compressed, which contain the information about the variant itself, and then the remaining M sample columns are compressed. If the sample columns have F number of fields, where $F > 1$ (denoted by multiple field names in the `FORMAT` column), the sample columns are unpacked into F distinct text vectors, one for each field, with the internal order of samples preserved within each vector. The vectors are then concatenated and compressed in the same way a simple $F = 1$ sample vector would be. The two chunks of the line, variant description and sample genotyping, are concatenated, and then packed into a BGZF block much like the above BGZF-only strategy.

For the 1000 Genomes Project chromosome 1 VCF file, BCF achieved a compression ratio of 98.42%, compared to the `BGZF` ratio of 98.14%.

3.2 Indexing

3.2.1 Tabix

The existing standard strategy for indexing both `BGZF` and `BCF` files is *Tabix* [12]. Tabix uses both a linear index, and a 6 level binning strategy, where each level from the top down uses smaller bin sizes than the level above it. Each index entry stores a 64-bit virtual offset `voffset`, which is comprised of a 48-bit compressed block offset, `coffset`, and a 16-bit uncompressed stream offset, `uoffset`. The `coffset` is the offset within the compressed file of the start of the `BGZF` block which contains this line. The `uoffset` is the offset within the uncompressed form of that block where this line starts. The 16-bit `uoffset` is sufficient to address any location within the block, which, as mentioned in the `BGZF` section, has a maximum size of 64KiB (2^{16}).

The indexing process of Tabix iterates through the lines of the VCF file, inserting a record into a bin based on the chromosome and position of the variant record. The index file has a whole index section dedicated to each chromosome, if the VCF file it indexes contains more than one chromosome. One binning structure only contains one chromosome's records. Based on the position of the record, Tabix finds the smallest bin which it can fit that record into. It starts at the lowest level of bin layers, and if it cannot find an empty slot for an index entry, it goes to the next layer up and continues this until it finds a bin with an available slot. In theory, it is possible that it finds no available slot, but in practice, due to the nature of VCF files, this will not happen.

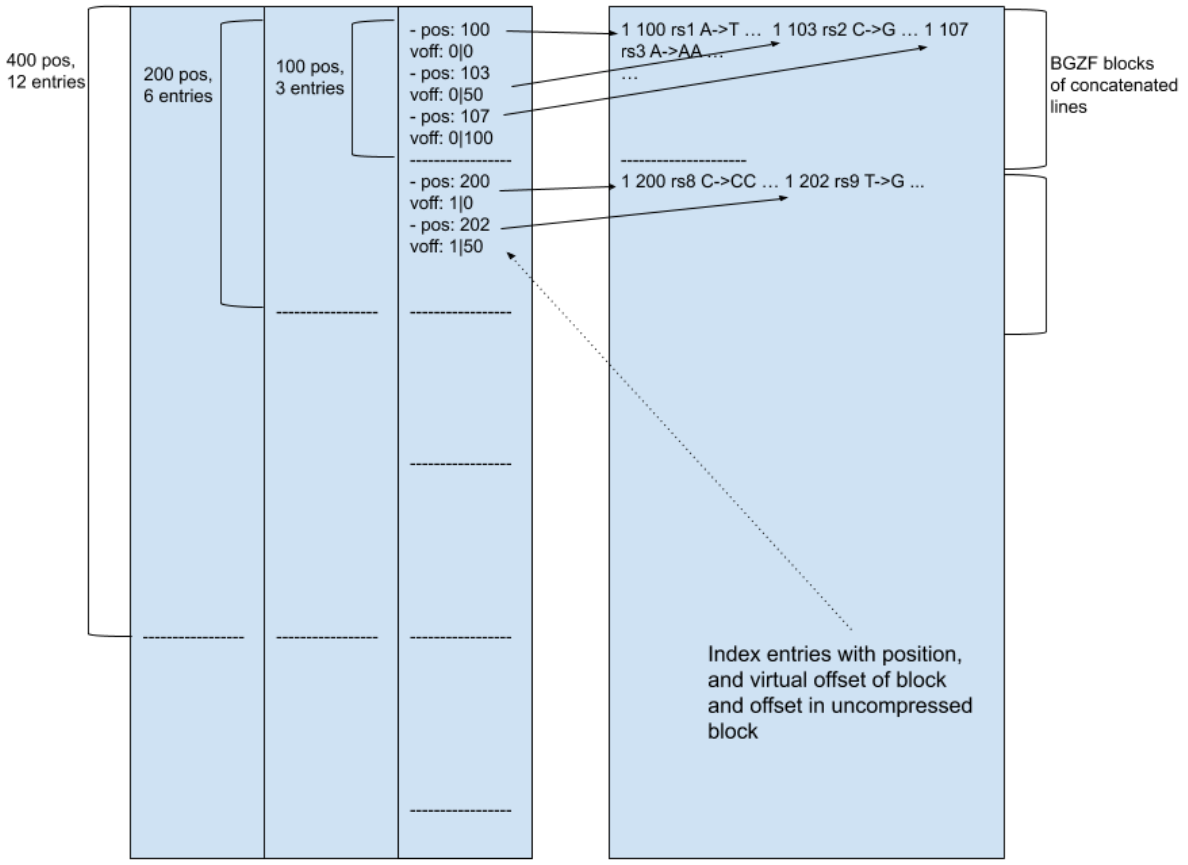

Figure 3.1 Tabix Binned Index

Figure 3.1 shows this "multilevel binned" layout. On the right hand side is the BGZF compressed file, and on the left side the multiple levels of the binned index for chromosome 1. Values are conceived only as examples. In this example, the index entries fit into the lowest level of the index, and each of the smallest bins covers 100 base positions, with space for 3 entries, and each line in the BGZF file is exactly 50 bytes long. In a real Tabix index these are much larger numbers but smaller numbers are used here to illustrate the layout.

In the linear index, Tabix stores an entry for each of the lowest-level bins in the binned index, which in the implementation are set to 16KiB regions across the chromosome, with the position and `voffset` of the first line in that bin. Tabix uses a linear index when searching larger bins, particular the largest bin covering the whole sequence range, which is likely to contain items when smaller bins overflow around variant hotspots. This significantly reduces seeks and bytes read in worst-case lookup scenarios.

CHAPTER

4

DESIGN

4.1 Compression (VCFC)

This novel compression strategy, which is coined VCFC (VCF Compressed), focuses specifically on the M sample columns within a VCF file. The justification for this is that the 9 non-sample columns are constant size and do not grow with the size of the sample set. Some benefit may be gained with compressing some of those columns, particularly the `INFO` column, which contains arbitrary-length annotation text, but even its size does not scale directly with the input size so it is not the focus of this investigation. Some fields in the `INFO` column which may contain count values, may scale in byte size on a $\log_{10}$ basis, by storing non-left-padded integers as text, but in common practice these can be assumed to have a reasonable upper bound because they are based on the size of a population.

VCFC takes advantage of underlying assumptions about the content of VCF files. These assumptions are as follows:

1. There is more than 1 sample in the file. VCF files can be created for each individual in the sample dataset, but this results in extremely high levels of redundant information and a difficult-to-query fragmentation of data across many small files. For smaller cohorts, perhaps a dozen or few dozen samples may be included, but in general the assumption is that the number of samples is on the order of a few hundred, ranging up to the many thousands or more. This juxtaposition of sample entries next to each other in the same file frequently leads to adjacent entries of identical value. Since run-length sample compression is used, a larger sample set will yield a better compression ratio.

2. There is a restricted character set. The overall character set is UTF-8. Sample columns have well-defined data fields within the VCF specification [24]. These fields in turn, for the most part, have well-defined types and even more restricted character sets than UTF-8 or even than ASCII, which is a subset of UTF-8. These restrictions can be used to reduce the number of bits needed to represent a symbol comprising one or more bytes.

3. The alternate bases for a given variant are ordered in decreasing order of frequency. At a position on the genome, the most frequent variant is listed first, proceeding down to the variant with the lowest frequency. This constraint is not imposed by the VCF specification but is a common, reliable, and most importantly computationally useful convention that can be used by processing utilities.

   Consider an example in Fig 4.1. Headers are shown for ease of comprehension. Position 10075 within chromosome 1 has 3 recorded variant alternates from the base `A` in the reference. In descending order of frequency they are `T`, `AA`, `G`. The order is verifiable by counting the appearance of those 1-based indexes in the sample columns. It is common practice to include the `AC` field in the `INFO` column, which also has these counts at the same indexes as the `ALT` column. In the example, the variant to `T` at `ALT` index 1 appears 3 times in the sample set.

```
#CHROM POS ID   REF ALT    QUAL FILTER INFO           FORMAT HG1 HG3 HG4 HG7
1    10075 rs12 A   T,AA,G 100  PASS   AC=3,2,1;AN=8; GT     3|0 1|2 1|1 2|0
```

Figure 4.1 VCF Line Example

   An important note is that this `ALT` sorting convention is so useful in alleviating future sorting operations that it warrants inclusion in the specification itself. If a VCF file does not meet this specification, a file reader can re-index the alternate bases for a variant in order to impose it, without having altered the informational content of the file.

4. The samples do not differ from the genome reference file in most positions in most samples. The implication of this is that the sample genotype value 0 is the most common case, meaning that this copy of the chromosome (humans are diploid, meaning two copies of each chromosome, which can independently vary) has, at this position, the base listed in the `REF` column.

   Because individuals within a species (and even across species) share so much of their DNA, this is a safe assumption, and the abundance of samples with genotype `0|0` can be utilized in compression schemes.

In addition to the above assumptions, for the purpose of this investigation, a constraint is imposed that only genotype values are considered, and only in the format of two non-negative

whole numbers separated by a bar character (`[0-9]+|[0-9]+`). This refers to the `GT` field for sample columns, specified in the `FORMAT` column. Other information about a sample can be included and specified in the `FORMAT` column, but those are not considered. The work subject to this investigation could be expanded to handle those cases by grouping by field type and compressing each set of contiguous field values individually in order to achieve more contiguous regions of similar values, which is how the BCF format [24][p.29] handles this.

For VCFC compression, the scheme used is very simple. It only compresses the sample columns. The reason for this is that it is the primary way that data size scales with more input data in the form of sample genomes. A species genome is bounded in size, and a thorough dataset with high coverage will include most variants, so including more input data is unlikely to add many more lines to the N variant lines in the file, but rather adds more columns to the M sample columns. Instances of human genetic variation are on the order of several hundred million, and are unlikely to increase far above a constant multiple of that in the very near future, barring some significant intraspecies genetic drift. The 1000 Genomes Project [22] has published nearly 90 million variants. On the other hand, the M sample columns are theoretically unbounded and depend directly on the amount of input data; as more samples are added, the size of the uncompressed dataset experiences a linear increase.

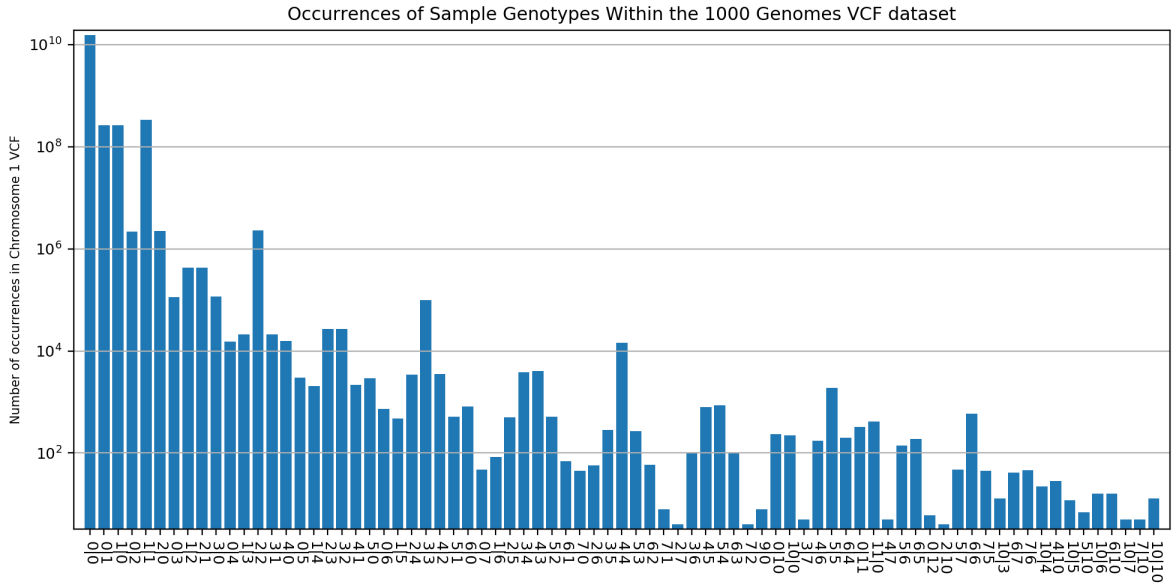

Figure 4.2 Genotype value frequencies 1000 Genomes Project Phase 3 Chromosome 1 all-sample VCF (log scale)

Figure 4.2 shows the occurrences of various sample genotype values within the 1000 Genomes Project chromosome 1 VCF file. Genotype values with fewer than 4 occurrences were excluded, for brevity and readability. The y-axis is a log scale, so `0|0` vastly dominates in frequency, but those

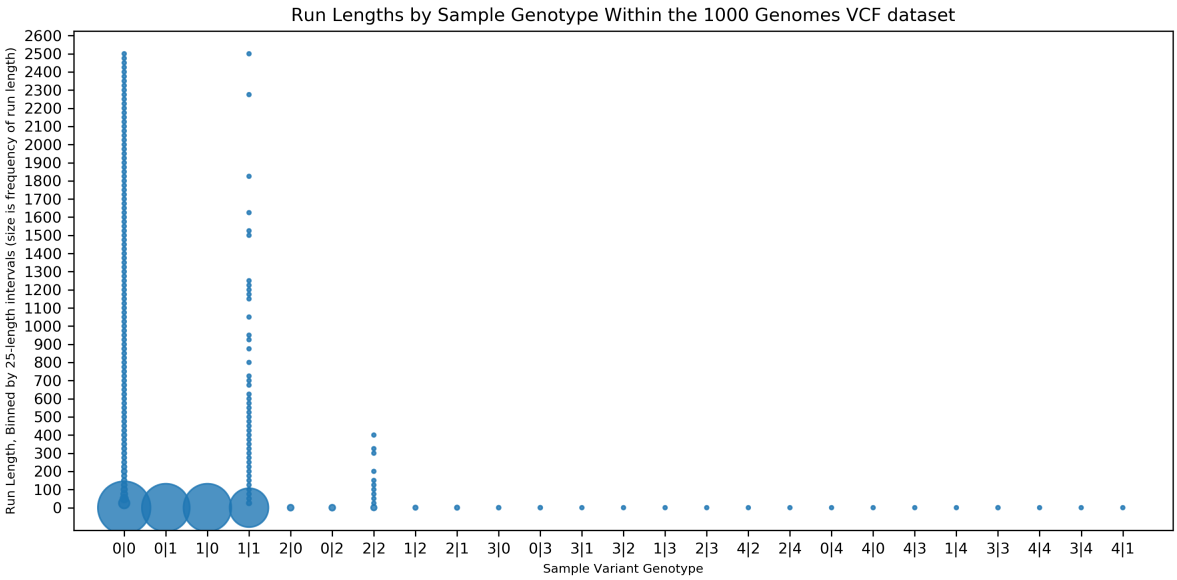

Figure 4.3 Run Lengths by Sample Genotype in 1000 Genomes Project Phase 3 Chromosome 1 all-sample VCF

with `0|1`, `1|0`, `1|1` are also highly frequent compared to all other values. This analysis was used in restricting the compression in this strategy to only compressing those 4 genotype values. An interesting phenomenon, also visible in this figure, is that symmetric allele pairs tend to be more common, on average, than mismatched alleles, meaning the two copies of the chromosome in the sample tend to have same variant at a particular location, even among relatively rare variant alternates like `5|5` and `6|6`.

Figure 4.3 shows the distribution of genotype run lengths among the sample columns, i.e. the number of samples with a same genotype that are adjacent to each other in the VCF file. The data points were grouped by run length ranges of 25 in order to be more readable. The size of the circles are a linear relation to the number of runs of each length. Genotypes `0|0`, `0|1`, `1|0`, `1|1`, and `2|2` are by far the most frequent values and several of them extend in run lengths up into the several hundreds or a few thousand. The maximum length of a run in the VCF file is based on the number of samples. For this dataset that number is 2504.

Using the assumptions listed, and analysis shown in Figures 4.2 and 4.3, we take advantage of the prevalence of genotype values of `0|0`, `0|1`, `1|0`, `1|1`, and apply run length compression to each of these, with each sequence of 3 bytes treated as a symbol to be compressed. We additionally get a benefit of discarding white space within the runs, so we reduce some number of 4 byte symbols down to 1 byte. Genotyping values other than these four are ignored and left uncompressed in the file. This is acceptable because they not as common and even on the occasion when they do occur, they comprise only short runs or none at all. Those listed for compression are often in runs of tens or hundreds at a time in large multi-sample VCF files.

Table 4.1 shows how runs of consecutive genotype values are encoded and thereby compressed.

The underlined value is the flag which is a non overlapping prefix with the other run flags, and the remaining bits store the length of the run. Instead of reserving 2 bits for each flag, which would evenly distribute it across the 4 possible genotype values, the decision was made to use 1 bit for the `0|0` case and 3 for the others, giving the most frequent case the ability to compress longer runs, as in that case 7 bits remain for the length encoding while 5 bits remain for the others. Figure 4.2 shows the vast disparity between the more common alternate allele genotype indexes and those that are less common. The graph uses a log scale, so the genotype values shown in Table 4.1 that are much more common are the prioritized focus for optimizations in the compression scheme involving the variable length flag and run length bits.

The visual in Figure 4.4 is meant to aid understanding the compression strategy, using the same input simplified VCF as shown in Chapter 2. In the top VCF, samples are highlighted based on their value and whether or not they are compressed. In the bottom, the compressed genotype values are removed entirely and replaced with the corresponding byte containing a flag and run length bits from Table 4.1. In this example, the bytes needed to represent the sample columns went from 184 bytes, to 43 bytes in the compressed version. In larger VCF files, this ratio increases drastically as runs become more common and lengthy.

Table 4.1 VCFC Run Compression Flags

Sample Genotype Value	Flag and Length bits
0|0	`0`XXXXXXX
0|1	`101`XXXXX
1|0	`110`XXXXX
1|1	`100`XXXXX

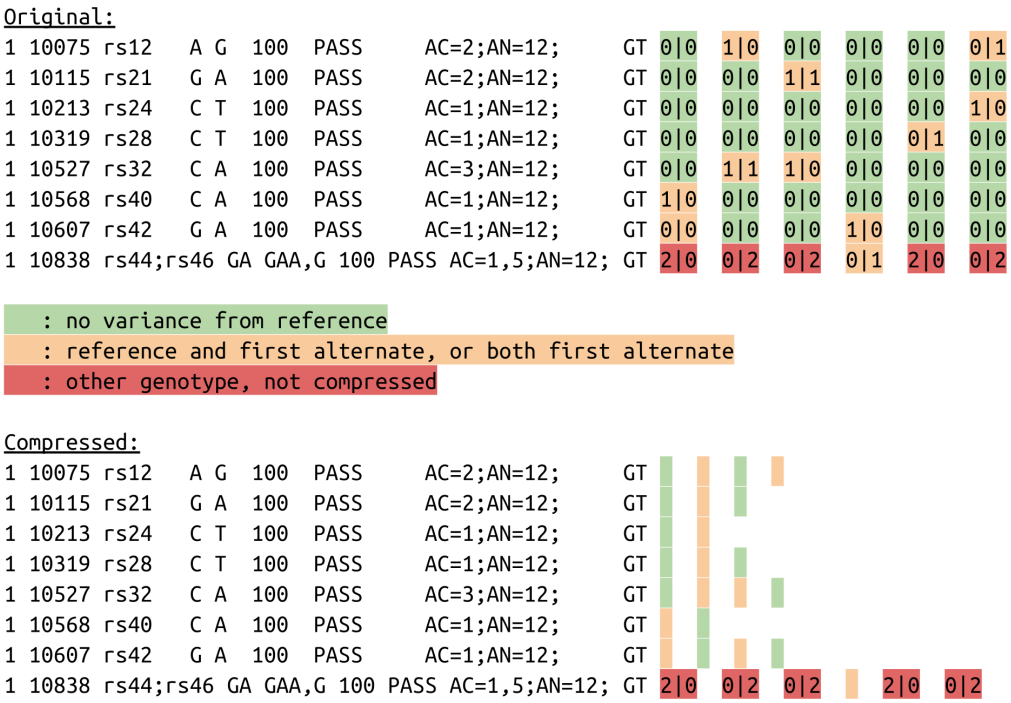

Figure 4.4 VCFC Example Visual

4.2 Line Based Indexing

The strategies for indexing used by Tabix and BGZF discussed in Chapter 3 are a hybrid approach between block-based indexing and line-based indexing. The index points to the compressed block that the requested line is in, and the offset within the decompressed block where it begins. So while a reader does know the location of a line after decompressing the required blocks, it must still read and decompress the entire block that the line is contained within.

One method of recording indexes for individual records is exhaustive indexing, in which an index entry for each record in the set is stored. This can lead to large index sizes, but for some scenarios, for example indexed columns with high value cardinality, the decreased lookup time is worth the cost of increased index size.

Another method is sparse indexing. In this method, index entries are stored for records in the dataset, except for those which are omitted based on some condition, for example, a key value does not map to legitimate data or a key value is deemed redundant with existing keys. A particular form of sparse indexing is binned indexing, in which key values are grouped into bins of similar values based on some comparison function. Comparison functions vary and certain ones are better for certain types of data, and can include strategies such as hashing, integer range binning, and exact value grouping (such as in SQL `GROUP BY` statements). Since, in our case, the file is sorted by position, the index can use this to its advantage and just skip some number of lines in the file which are close to an existing index entry, and not include those in the index. As a result, the index becomes smaller in a direct relation to how many lines are skipped between each entry, but the index can still direct a reader to a location in the file which is relatively close to the first record requested by the reader. There is a trade-off between these two factors:

- The time it takes to read through the index to locate the appropriate entry to service a query; and
- the skip distance the index entry places the reader inside the file, ideally minimizing seek time from that point to where the actual position is that was requested.

Because we are utilizing line-based indexing instead of raw byte-stream based indexing, where a global optimum between these tradeoffs can be found (at least for that particular storage technology depending on the seek overhead), the length of the lines in the input data determines the optimal balance between these two factors. The same is true for BGZF and Tabix, and both the compression block sizes and the index bin sizes used in that technique.

In this section, three strategies for performing both purely line-based based indexing are featured, taking advantage of the purely line-based compression of VCFC. These are listed and described below.

4.2.1 Sparse File Offset-As-Index

In this first strategy, for each variant, a positive integer offset is computed based on the chromosome name and its position. This enables the byte offset of a line within the file, if it exists, to be computed in constant time.

Sparse files are supported in many modern filesystems, such as EXT4 and XFS. A sparse file is a file in which some sections of the file are addressable bytes within the file, but do not actually allocate blocks within the filesystem. A filesystem block is a contiguous, non-overlapping sequence of storage device sectors. For example, a common scenario is for a storage device to have sectors of 512 bytes, and a filesystem to have a block size of 4096 bytes. This means any operation to the storage device will incur a read or write of at least 512 bytes, and any operation to the filesystem will incur a read or write of at least 4096 bytes, in this case 8 sectors. Filesystems generally use a block size larger than the sector size because they assume most files are larger than one sector. Grouping them into blocks of larger sizes is much better for performance, by reducing function calls and by reducing I/O operations (IOPS) to the storage device, which for non-flash storage may incur an expensive head seek operation. This batching of IOPS is especially beneficial when the device and kernel are direct memory access (DMA) enabled so that the storage device can be pointed to a sequence of addresses in memory to read from or write to directly without needing the CPU to issue additional read or write instructions to the device.

In a sparse file, filesystem blocks which *would* contain all zeros are simply not allocated on the hardware device. This region is referred to as a *hole*. The reported *nominal file size* is the maximum file offset minus the minimum file offset, but the *real file size* is the number of blocks allocated to the file times the size of the filesystem block. For *dense* files these are approximately the same, but for sparse files they can be very different.

Deallocating empty blocks is one way to save space, but it only can be exploited when the empty regions are at least one block large. If an empty region of a file is less than the size of a filesystem block, or overlaps into two filesystem blocks but does not reach the edges, those blocks will still need to be allocated because there is real (non-zero) data in them. Remember the filesystem block is the minimum size of an operation that the filesystem supports, it cannot allocate part of a block and leave the rest sparse. In the worst case scenario, we may have a file in which one byte of each block is written by an application, and the rest of the block is empty. For each one byte, one would need to allocate an entire block minus that byte of real data. This additional wasted allocation is referred to as *internal fragmentation*.

In dense files, it is often the case that the last block of a file is not completely filled with data. Any application reading the file will encounter the end-of-file (EOF) before reaching the end of that final block. Because of this, it is often the case that even a completely dense file in which all written non-zero bytes are contiguous still has some small amount of internal fragmentation. In that case, the nominal file size is less than the occupied storage size in blocks.

Table 4.2 shows this overhead due to internal fragmentation in one representative (1000 Genomes Chr 22) VCF file, using two different EXT4 block sizes. In the top row with 4KiB block size, the sparse

Table 4.2 Sparse File Internal Fragmentation Overhead

FS Block Size	Compressed VCFC Size	Sparse VCFC Nominal Size	Sparse VCFC Allocated Size	Percent of file that is actual data (not fragmentation)
4096 (EXT4)	360792064 (345M)	840661418206 (782G) ratio: 782G/345M = 2397.5x	4533911552 (4.2G) ratio: 4.2G/345M = **12.56x**	file: 4533911552 data: 360792064 1-(4533911552-360792064)/ 4533911552 **= 7.95%**
1024 (EXT4)	*(same as above)*	*(same as above)*	1229811712 (1.1G) ratio: 1.1G/345M = **3.40x**	file: 1229811712 data: 360796160 1-(1229811712-360796160)/ 1229811712 **= 29.33%**

equivalent of the compressed file was 12.56 times larger than the contiguous compressed file, the majority of which is purely due to internal fragmentation. In the second row, the filesystem block size was set to 1KiB. Because the block size is smaller, there is less room for fragmentation to occur.

For XFS, sparse files are handled differently. For an identical sparse offset-as-index version of the chromosome 22 VCFC file, EXT4 only allocated 9522024 blocks, while XFS allocated 84825032 blocks, an approximately 8.9x increase. A possible cause of this is that XFS uses a more aggressive, preemptive allocation of filesystem space [20], which may be poorly suited to a usage in which the overwhelming majority of the file is sparse and the majority of data writes distributed within the sparse file are relatively short (less than 16KiB)

Each line in the sparse file has an additional header placed at the start, which has the byte offset from this line to the previous line and the byte offset from this line to the next line. This is used to quickly and reliably traverse between lines of the sparse file, e.g., when performing a range-based lookup in the file. In the file described in Table 4.2, this overhead constituted just 16MiB of the 3979MiB of overhead from converting from a contiguous compressed file to the sparse representation of the same file. Sparse regions between data lines in the sparse file may be very large, so to avoid seeking through all of those bytes, these line headers provide a way to seek over them and skip the sparse region.

The first step of performing a query using the sparse offset-as-index method is to translate the query start position into an offset in the file. This is done using a linear equation based on several parameters.

Given a coordinate query: chrN:Start-End, one can translate this to a byte offset. In order to maximize sparse file efficiency and ease of traversal, this must be aligned to the start of a filesystem block.

1. First, the chromosome N is translated to `{1..22, X, Y, M}, X=23, Y=24, M=25`.

2. Next, assume the max length of a chromosome (L). Since all unused locations will be sparse one can overshoot and pick a value higher than necessary. This may result in translated coordinate offsets beyond the actual file length. It is relatively safe to assume that the maximum human chromosome length is 250 million base pairs. We set it to 300 million to be safe.

3. Next, a multiplication factor (F) is defined. This ensures two consecutive variant lines do not overlap in the sparse file, and depends on the input. If compressed lines are long, F must be higher, as F must be a multiple of the FS block size (B) that is larger than the longest line in the file.

   We set F to 16KiB, with $B = 4096$.

   See Figure 4.5 for a better understanding of why this is needed. This figure shows the byte length of compressed lines within a VCF file from the 1000 Genomes Project in compressed form using VCFC. With a block size of 4096 and lines written to the file aligned to the start of filesystem blocks, if two lines that are longer than 4096 bytes have consecutive variant positions, say 100 and 101, their offset translation would cause the second line to overlap with the bytes from the first line extending past 4096 bytes long. This factor allows us to space these lines out so that this overlapping does not happen. The actual value depends on the input data and how long its compressed lines are. Setting it higher than it needs to be is safe, and can avoid having to tweak based on any practically foreseeable input, as long as it does not overflow a `long` integer type in the offset calculation.

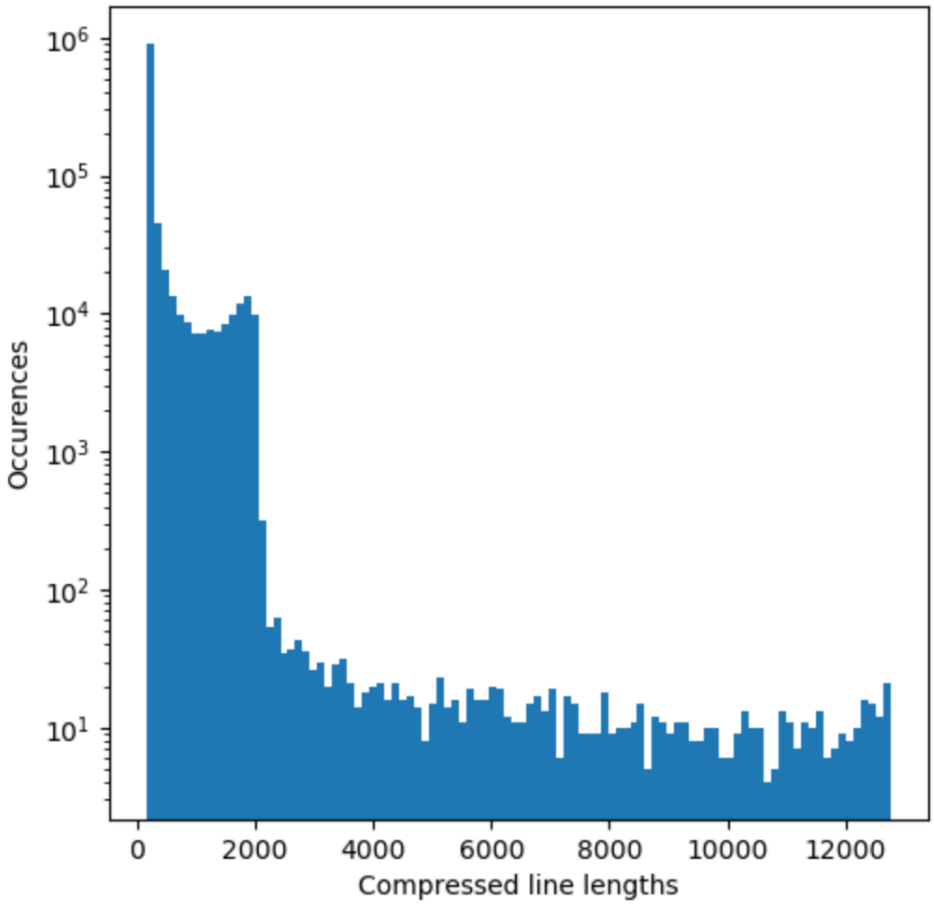

Figure 4.5 VCFC Compressed line lengths in 1000 Genomes Project chromosome 22

The equation for computing byte offsets for the sparse offset-as-index strategy are shown below:

$$\text{General form of byte interval: } [F(N \cdot L + S), F(N \cdot L + (E+1)) - 1]$$

As an example, let the following file parameters be:

- $L = 300000000$,
- $F = 16384$.

Consider the query `2:1000000-2000000`, meaning chromosome 2 from base positions 1 million to 2 million.

That query range is used to compute a sparse offset range:

$$Start = 16384(2 \cdot 300000000 + 1000000) = 9846784000000$$
$$End = 16384(2 \cdot 300000000 + (2000001)) - 1 = 9863168016383$$

The section of the sparse file where variants meet this location query is $Range$ bytes long, where $Range = End - Start$.

$$Range = 9863168016383 - 9846784000000 = 16384016383$$

This $Range$ covers 1 million $\cdot$ F bytes of the file.

The program can then seek directly to the $Start$ offset in the sparse compressed file. If the location seeked to is not a sparse file hole, then the first position in the range does have a variant in the file, and one can use the line headers to jump from this line to the next, and continue through to the end of the query range. However, if this position initially seeked to (computed from `2:1000000`) does not have a variant in this VCF file, the reader will be placed inside a sparse hole, that is, unallocated blocks of zeros where no data resides. Because this hole in the file may be very large, seeking forward or backward to the closest data line could be extremely time consuming. However, reasonably recent versions of the Linux kernel provide a mechanism for handling this with additional `whence` values for the `lseek` function, documented in its manual page [14], which are also usable in the `libc` `fseek` function. These are `SEEK_DATA` and `SEEK_HOLE`. If `lseek` is called with `SEEK_DATA` and the offset of the current position in the file, the kernel will update the file offset to be the start of the next block that contains data, if one exists, effectively jumping past all of the following sparse blocks to the first non-sparse block. If the current block contains data, the file offset is left unchanged. Since our factor parameter (F) for sparse offset computation is constrained to being a multiple of the filesystem block size, all of our data lines in the sparse file are aligned to the start of filesystem blocks. So the next non-sparse block after we initially seek into a hole based on the query offset computation will be the start of the next highest data line.

4.2.1.1 Sparse Offset-as-Index Caveats

There are two caveats in this indexing technique, which are not encountered in the others. These are detailed below, along with the reasoning for why it will not significantly impact evaluations for the first problem, while a workaround is employed for the second problem.

4.2.1.1.1 Duplicate Records

One caveat for this particular technique of indexing is that there is a 1:1 relationship between a `(chromosome, position)` location tuple and a file offset. However, VCF files support multiple records for the same location tuple. For example, when a position has multiple important variants that are deemed worthy of their own identifier and information columns. One possible solution to this would be to merge those records at the same location, which is valid and constitutes a semantically equivalent VCF file. This is done by joining the alternate bases into the same alt column, now as a list separated by commas, re-indexing the sample genotypes using that list of alternate bases, and joining the other columns and information key-value pairs together in the same order as the list of alternate bases (with commas between them and dots for nonexistent values). Nonexistent values could occur when one variant has an information key-value pair for which the key does not exist in another variant record on the same chromosomal location.

These records are relatively rare within the evaluation dataset. Hence, this resolution was not pursued in this work. In the 1000 Genomes VCF dataset, approximately 0.09% of chromosome 1 variant positions are duplicate, and 0.1% of chromosome 22 variant positions are duplicate. When these records are excluded from the sparse offset-as-index versions of the VCF files, this will reduce the query time spent in the decompression phase. But the index evaluation phase remains the same since the duplicate records are contiguous in the file and map to the same index position. This also only occurs for those records that are duplicate, and within those only with the ratio of $1-\frac{R-1}{R}$, where R is the number of times a position repeats, which is low (around 2 or 3 times). For example, if a variant query range of 10k has 0.1% of its records repeated once and the duplicate records are excluded in this indexing strategy, the overall decompression time will be reduced by approximately: $0.001 \cdot (1-\frac{1}{2}) = 0.0005$, or 0.05%

4.2.1.1.2 Maximum File Size

Another caveat is that due to the multiplications used for the multiplication factor F and upper bound on chromosome length L, the numbers for file offsets grows very large and can pose problems relating to maximum file sizes of filesystems. For example, for chromosome 1, with $F = 4*4096$ and $L = 300000000$:

$\texttt{Offset}_{max} = (300000000*25)*(4*4096)-1$

$\texttt{Offset}_{max} = 122879999999999$

$\texttt{Offset}_{max} \approx 2^{46.8}$

This offset fits easily within a 64-bit integer, however EXT4 has a maximum file size constraint of 16 TiB (2^{44}) due to it supporting 2^{32} maximum blocks per file and all of our evaluations use a 4 KiB block size (2^{12}). This means EXT4 with a 4 KiB block size is unable to address file locations in the upper regions of the space. Neither the L nor the F can be reduced to fit the max offset within the maximum EXT4 file size. If the EXT4 block size is increased, the sparse file fragmentation increases and the real file size increases slightly above linearly, making that an impractical option. XFS does not have this problem due to its maximum file size of 8 exabytes (EiB) minus one ($2^{63}-1$). To work around this, we can remove the factor of 25 used for indexing to each chromosome in the file, which reduces $\texttt{Offset}_{max}$ by more than 4 powers of two. This address space will fit within 42 bits, addressable by EXT4. This means that a strict constraint for this strategy either that each chromosome be stored in a separate file, or that a filesystem is used that supports sufficiently high file sizes.

4.2.2 Contiguous external binned index

In this second strategy, an external file is used to store a linear index of bins, each containing a number of records which point to records in the compressed VCFC file. The index entries map a chromosome and position to a byte offset within the VCFC compressed file of the line where that combination occurs. The structure of each 13 byte entry is shown in Figure 4.6.

```
struct index_entry {
   uint8_t reference_name_idx;
   uint32_t position;
   uint64_t byte_offset;
};
```

Figure 4.6 VCFC External Index Entry Structure

Stored in `struct index_entry`, `reference_name_idx` is an integer translation of the chromosome name. The `position` is the position of the variant on the chromosome. And `byte_offset` is the offset in the compressed file where this chromosome and position first occur. In the case of duplicate `(chromosome, position)` value tuples or variants which span a large range of positions, each entry covered by one or more variant lines in the input points to the byte offset of the *first* variant record which met that criteria. The fact that the file is sorted by start position makes this possible. The sequence of index entries is created for a specific VCF file; the number of entries and the position difference between them vary. Instead each index represents a constant number of lines within the VCFC file. This *bin size* is configurable as an input parameter to the `create-binned-index` command.

To query the index, since the entries in the index and the data in the file are both sorted, a binary

search implementation is employed to rapidly find the first index entry which is equal to or less than the start of the requested query. That byte offset is then seeked to in the data file, and lines are decompressed. Based on the bin size, there may be some number of lines which must be filtered past before reaching the start of the query.

4.2.3 Sparse external index

This third and final index combines beneficial aspects of both of the previous index strategies. It uses an external file as the index with the same entry layout as the contiguous external binned index (see Figure 4.6, and it uses sparse offset-based indexing into itself. One way to think about it is that the index file indexes the compressed data file, and the sparse offsets of index entries index the index file itself.

Using an external file vastly simplifies file layout as compared to the Sparse Offset-As-Index technique, and enables much faster indexing operations because only the small external file must be written or updated, instead of the compressed file containing all of the data. Even if updates to the index can be done in place without a full rewrite of the file, if the index is stored within a large file, kernel or userspace library (e.g. libc, C++ `iostream` standard library) I/O caching can increase time and I/O bandwidth cost on the larger file (where the index is interspersed with data) compared a smaller file containing only the index, because more unused file data is read into memory.

CHAPTER

5

EXPERIMENTAL FRAMEWORK

5.1 Implementation Correctness

The evaluation of correctness for the VCFC compression and indexing techniques was performed through thorough experimentation.

The first correctness experiment validates the VCFC compression and decompression. A set of VCF files from the 1000 Genomes Project were compressed with VCFC, then decompressed with VCFC, and the output of the decompression was checked against the original VCF file with the `diff` line differencing utility.

The second experiment focuses on the indexing implementations. It samples a variety of VCF queries across the possible query space and validates that they matched the expected output. This was done using Tabix as the reference query servicing implementation. The same query, for example `22:17000000-20000000` (chromosome 22 from positions 17 million to 20 million), was performed using both Tabix and our indexing strategies, and the output line count was obtained and the complete output was hashed with `sha256sum`. The same was then performed with Tabix, and the line counts and hashes were compared. A series of queries were performed in order to cover equivalence classes and boundary conditions which included cases like querying before or after the first and last lines in the file, and querying for lines that do not exist.

It is possible that the VCFC implementation and Tabix returns, for the same query, different output variants of the same number of lines and with the same SHA256 hash. However, the VCFC compression and decompression was thoroughly tested with exact output value matching, so any individual lines that are output are assumed to be correctly decompressed lines from the input VCF. Any erroneous output from the index queries would mean that one or more lines were entirely

included or excluded, altering a minimum of a few hundred bytes in the output. The chance of this large of a byte-wise difference in output producing a line count match and a hash collision, on multiple different regions of the index, is so exceedingly small that it is assumed to have not occurred.

For a small number of variants, our binned external index and sparse external index will include them in results, while Tabix will not. The reason for this is that we include some structural variants that overlap a query range if they provide an `SVLEN` key-value pair in the `INFO` column. If that integer value is present, its absolute value minus 1 is added to the start position of the variant, and this is treated as the range of the variant (see the VCF specification [24], section 3). For other structural variants, we use the `END` key-value pair in the `INFO` column to obtain the start and end position of the variant. This is the same way that Tabix computes this, so we obtain the same results as Tabix for these queries. The impact of the inclusion of structural variants with ranges provided by `SVLEN` is that some queries used in the evaluation may return slightly more results in the VCFC binned external index and sparse external index implementations than in the Tabix or VCFC sparse offset-as-index implementations, leading to very slightly increased times for those two indices. In the 1000 Genomes Project chromosome 22 VCF file, 127 out of 1,103,547 variants (0.0115%) are structural variants that use the `SVLEN` field. To evaluate these relatively rare cases, the first 3 fields of the output lines (chromosome, position, id) were extracted and compared using a line differencing utility against the output of the same query from Tabix. Output lines in our output that were not in Tabix output were then cross-referenced with their corresponding `INFO` column to verify that they constituted a case of the `SVLEN` position overlap described above.

The Sparse Offset-as-Index strategy was validated using a mix of hash and line-wise output differencing as well. Due to caveats described in Section 4.2.1.1.1, some mismatches, in which multiple lines existed at the same position but were not reported as such, were ignored in the results. As discussed previously, the time impact of these discrepancies did not make up a significant portion of the differences in average query performance over the large experimental data series for this particular index implementation.

5.2 Systems Evaluation

This section describes the methods and systems used to evaluate the performance along with other measurements performed and included in Chapter 6.

The implementation is written in C++11, compiled using the GNU C++ compiler and libraries version `4.8.5` on CentOS 7 with Linux kernel `4.10.13`. For timing evaluations, optimization level 3 was enabled with `-O3`. No other nonstandard build flags were applied.

The evaluations were performed on the the ARC cluster [2] at North Carolina State University. The two cluster instances are described in Table 5.1. The implementation is single-threaded so only one hyperthreaded logical core was used for execution. Only one execution was run at a time to minimize CPU context switching and cache contention and to eliminate I/O bandwidth contention.

Table 5.1 ARC Cluster nodes used in evaluation experiments

Cluster Node 1 (c80):

- CPU: Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz
  16 cores, 32 threads. 20MiB L3 cache.
- Storage: SAMSUNG MZPLK1T6HCHP-00003 (NVME), 1.6 TB

Cluster Node 2 (c29):

- CPU: Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz
  8 cores, 16 threads. 11MiB L3 cache.
- Storage: Samsung SSD 860 EVO (SATA), 250 GB

On each storage device, two partitions were created, each filling approximately half of the storage device. One partition was initialized to contain an EXT4 filesystem, and the other with an XFS filesystem. Both filesystems used a block size of 4 KiB on 512 byte physical device sectors. The filesystem block size is the minimum size of a contiguous set of sector-aligned bytes on the storage device that are allocated, read, and written at a time by the filesystem.

Kernel disk caching was left enabled. For comparative timing-related evaluations which seek to incorporate I/O performance, to minimize impact of disk caching and cache retention, the kernel filesystem and read/write buffers were flushed before each run using the following method:

```
echo 3 | sudo tee /proc/sys/vm/drop_caches
```

Each test scenario used in timing-related evaluations was run 10 times and averaged. For some tests which report result sets of a small number of data points, standard deviation is included in the reporting of results. In tests which graph several hundred or more data points, each averaged over 10 runs, the standard deviation of each averaged data point is on the order of a few hundred microseconds, with some ranging into the several milliseconds. The sparse file queries had higher standard deviation, with more being on the order of milliseconds, rather than hundreds of microseconds. Since these evaluations used 200 data points per graph data series, and the measured standard deviations within the 10 runs for each data point are much lower than the deviation between data points and the average trend of each series, and also less than the difference in averages between data series, the standard deviations of each data point are not reported, as the trend lines are deemed to be accurate representations of each data series.

Some tests involved aggregating a large sampling of queries across the range of the input file in order to evaluate a different independent variable, for example, index bin size, which provides additional amortization of environmental impacts on runtime. Given these mechanisms employed during execution, outliers shown in timing results are assumed to be true positive outliers.

5.3 Evaluation Data

The data used to evaluate the new compression and indexing strategies included both synthetic and real-world VCF objects.

Synthesized VCF files were created by generating N variants and M samples, with reference bases and alternate bases chosen at random, with up to A alternate bases with decreasingly likely probabilities, and with the same A for each variant in the VCF file. Since in the real world variants do not all have the same number of alternatives, this does not accurately mimic a real VCF file and may underestimate compression ratios because a large number of variants have either 1 or 2 alternate base sequences which are variant from the reference. This may underestimate the compression ratio because the synthetic files have shorter run lengths of sample genotypes in each variant line. But this method does mimic a more complex case than simply having one alternate. The configuration used for testing was to have 3 alternates for each variant position, with probabilities `[0.90, 0.08, 0.02]`, in that order, since the alternates are defined to be listed in decreasing order of frequency, that is, alternate 1 is more common than alternate 2, which is more common than alternate 3, and so on. There is in theory no limit to the number of alternates that can occur at a variant location. These synthetic files were only used for running high speed correctness evaluations, rather than sub-sampling from an existing real-world VCF file.

Real-world VCF files were also used for evaluating correctness in final stages of development of each solution, as well as for measuring storage and time metrics. The input files were from the 1000 Genomes Project [22] Phase 3 all-sample VCF files, which contain 2504 samples and approximately 89 million variants. Exhaustive evaluations were performed on the whole set of 1-22 and X chromosomes with the constraints discussed in Chapter 4. Additional non exhaustive, but still representative, tests were performed with either chromosome 1 or chromosome 22, as roughly representing the endpoints of large and small autosomal (non-sex) chromosomes, respectively.

CHAPTER

6

RESULTS

6.1 Compression

6.1.1 Compression Ratio

The compression ratio for VCFC was 96.87% on the 1000 Genomes Project chromosome 1 VCF file. This compares to the 98.14% for BGZF and 98.42% for BCF. This is a good outcome for a compression ratio. Remember that VCFC only performs compression on the sample columns, and within sample columns only compresses those with alternate alleles 0 and 1. This means that the relatively simple strategy of compressing runs of only high frequency genotype values achieved nearly the same compression ratio as performing both GZIP on a VCF file (BGZF), and performing GZIP on a slightly binary translated and additionally line-compressed form of a VCF file (BCF). The advantage is that much of the compression value is preserved, but the variant site and description columns are not compressed at all. After reaching a variant line during a query operation, if only the variant information is desired, no decompression takes place as it can read that part of the line in plaintext directly from the compressed file. This fact is also useful when performing filters on the chromosome and position, as those columns are not compressed and records can be filtered without doing any decompression, and by only reading the first 10-15 bytes of a line.

6.1.2 Compression Time

In addition to compression ratio, compression time is also of interest. Compression times for the 1000 Genomes Project chromosome 22 VCF file are shown in Table 6.1 in seconds. Each experiment was performed 10 times, and the arithmetic mean and standard deviation are reported. The time to

read the uncompressed file from storage is one factor in the compression process, along with the compression time and the write time, but since these experiments read the exact same file from storage, the compression algorithm itself mostly impacts the compression time and the write time (the algorithm compression ratio affects the output file size and the implementation affects the pattern of write calls to the device). For this reason, for each storage device, the time to read each uncompressed VCF input file from uncached storage is also included, labelled `cat` because it was performed with the following command:

```
cat $vcf_file > /dev/null
```

Table 6.1 Compression Times for 1000 Genomes Project chromosome 22 VCF

	EXT4 NVME	XFS NVME	EXT4 SATA	XFS SATA
BGZF	127.29 ± 0.50	126.97 ± 0.56	126.91 ± 0.19	126.90 ± 0.12
BCF	238.43 ± 0.52	238.27 ± 0.38	226.70 ± 0.29	226.62 ± 0.22
VCFC	192.96 ± 0.82	194.49 ± 1.73	176.38 ± 1.0	175.56 ± 2.01
cat	6.15 ± 0.24	5.88 ± 0.26	23.73 ± 0.08	20.42 ± 0.05

Refer to Table 6.1. For the simple read time, NVME performed significantly better than SATA, and EXT4 and XFS on NVME were essentially equivalent. On SATA, XFS read the input file 3.3 seconds faster than EXT4. This is interesting because the BGZF, BCF, and VCFC times for EXT4 and XFS on the SATA device were very close to each other. This means that the read performance was not a bottleneck in the overall compression process.

In the first column, the BCF compression took 1.9X as long as BGZF, and VCFC took 1.5X as long as BGZF. BCF performs two phases of compression, leading to more computational work. This trend also applies for XFS in the second column. VCFC also had a much higher standard deviation than either of the other two compression algorithms. This can be attributed to the fact that the writes performed by VCFC are on a line-by-line basis, making them smaller and issuing more of them than the block-based writes by BGZF and BCF. This can increase the device latency in responding to each operation, whereas in the former two cases the I/O bandwidth is the more significant factor when writing the output file. Since these operations were the only active processes on the device, the available bandwidth should have remained constant between all values in a column. On XFS, the standard deviation for VCFC was roughly 2X that of the same file being compressed on the same node on EXT4. Given that the standard deviation of XFS read times was very close to the standard deviation of EXT4 read times, this suggests the XFS write times have a much higher standard deviation than both the read times on XFS and the write times on EXT4. This may be due to the XFS speculative block preallocator [20] suffering in performance due to the jagged write length patterns of VCFC. To confirm this, further investigation is needed, in particular to understand why the write performance was so highly variable even between identical file writes.

For BGZF, the performance across all devices and filesystems is approximately the same, within one standard deviation of each other. However, for BCF and VCFC we see interesting results. Despite the read of the input file being multiple times faster on the NVME device, the overall compression time for BCF and VCFC was significantly faster on SATA. The NVME device was 5.2% slower for BCF compression and 9.5% slower for VCFC compression. These were run on different nodes, but the NVME node had a 20 MiB L3 cache vs. only 11 MiB on the SATA node. This is counterintuitive, as one would expect that compression from and to the NVME device would be faster. The only plausible explanation might be that the small writes resulted in *erase-before-write* operations at the block level of the NVME device within the Flash Translation Layer.

6.2 Indexing

This section covers several sets of performance related queries performed against `BGZF`, `BCF`, and `VCFC` files with corresponding indexes described in Chapter 4. It also covers performance relating to the creation of the index itself, and profiling of the novel `VCFC` binned index under different tuning parameter values.

6.2.1 VCFC Binned Index Profiling

This section covers time profiling of different key phases of the index query operation for the binned index. The reason this index is profiled separately is because of the input tuning parameter, which controls the *bin size*. The operation phases investigated are searching through the index, seeking from the start of the bin in the compressed file, and decompressing lines in the compressed data file.

In the following figures, the index search phase (blue) represents how long the program takes to find the appropriate bin within the index. For example, if the query is looking for a line with a position field of P, this phase is the time it takes to search through the index to find the bin which contains P. With smaller bin sizes, there are directly proportionally more bins to search, and so the portion of the overall query servicing time spent on index search is higher. However, the index search implementation uses a binary search algorithm, which only increases the number of bins searched in a logarithmic fashion. In all cases, the index search phase was relatively small compared to the other phases.

One phase which was intentionally omitted from each profile graph is the metadata reading phase. The VCF metadata lines must be read in each query in order to determine the schema of the file. This phase was omitted because the overhead is a constant time for every query.

Each bin size was evaluated with a uniform distribution of 200 position queries across the chromosome 22 VCF file, and each query was evaluated 10 times and averaged. These tests are meant to focus primarily on computational query phases, and not I/O performance of the storage device, so the kernel disk page cache was not flushed between each run. For this reason, the times in this section are not directly comparable to times in the later index timing sections.

6.2.1.1 Single Variant Queries

Figure 6.1 and Figure 6.2 both show time profiles for queries performed with different bin sizes used in the index on Node 1, on EXT4 and XFS, respectively. Figure 6.3 and Figure 6.4 show the same test suite run on Node 2.

The x-axis of each of these graphs shows the bin sizes of each set of queries. The x-axis is not a linear scale, with a smaller interval between lower x-axis values than between higher x-axis values. This was necessary because differences between bin sizes are more visible at the lower end, due to the inversely proportional relationship between bin size and total number of bins (e.g. for a total input size N, $\frac{N}{5}$ is more different from $\frac{N}{10}$ than $\frac{N}{800}$ is from $\frac{N}{900}$). Each bar includes the average time profile across the 200 queries, each of which itself was averaged over 10 runs. This high level of amortization across 2000 runs for each bar provides a high degree of certainty that the profiles visible are not significantly impacted by environmental factors, making them statistically meaningful. The y-axis of each graph shows the time in milliseconds. Each bar represents the cumulative time of the profile phases for one bin size, and within a bar, each colored portion represents one phase of the time profile.

These graphs show a clear relationship between the bin size and the amount of time spent searching from the start of that bin to the actual requested record. It may seem to be universally beneficial to just pick a smaller bin size, however if the size of the index is an important factor to be considered in a usage scenario, then that may not be the case. The size of the index file is inversely proportional to the bin size, so the lowest time query shown in these figures will have a much higher index file size than the slower queries. For the 1000 Genomes Project chromosome 22 VCF file, the VCFC binned index with bin size 10 is 1.4 MiB, and with bin size 100 is 140 KiB, and with bin size 1000 was 16 KiB. Thus, it would make sense to pick a bin size with sufficiently acceptable query performance, not necessarily the best, in order to balance the trade-off between performance and storage size. All of the graphs follow the same trend. Referring to Figure 6.1, a bin size around 100 should be acceptable, as it performs well but does not increase the index size as much as a bin size of 20 would.

In Figure 6.3 for EXT4 on Node 1 and Figure 6.4 for XFS on Node 1, we observe no discernible difference in times between EXT4 and XFS. The index search drops off steeply and the trend becomes almost flat after reaching a bin size of around 20, decreasing slowly over the rest of the graph. The seek phase increases roughly linearly within each set of bin size step increments (4-100, 100-250, 250-1000). Likewise, for Node 2, the graphs for EXT4 and XFS were essentially the same. Between the two nodes there was a notable difference. On Node 1, the lowest bin size had an average profile time of 2.17 milliseconds while in Node 2, the lowest bin size had a profile time of 2.47 milliseconds. This is accounted for entirely by the index search phase, which is significantly slower on the SATA SSD than on the NVME SSD.

This is because the binary search algorithm used to search the index loads and caches index entries into memory on demand instead of loading them all up front. Because the index is one flat layer and all entries are the same size, an entry address within the file is accessible in a random-access

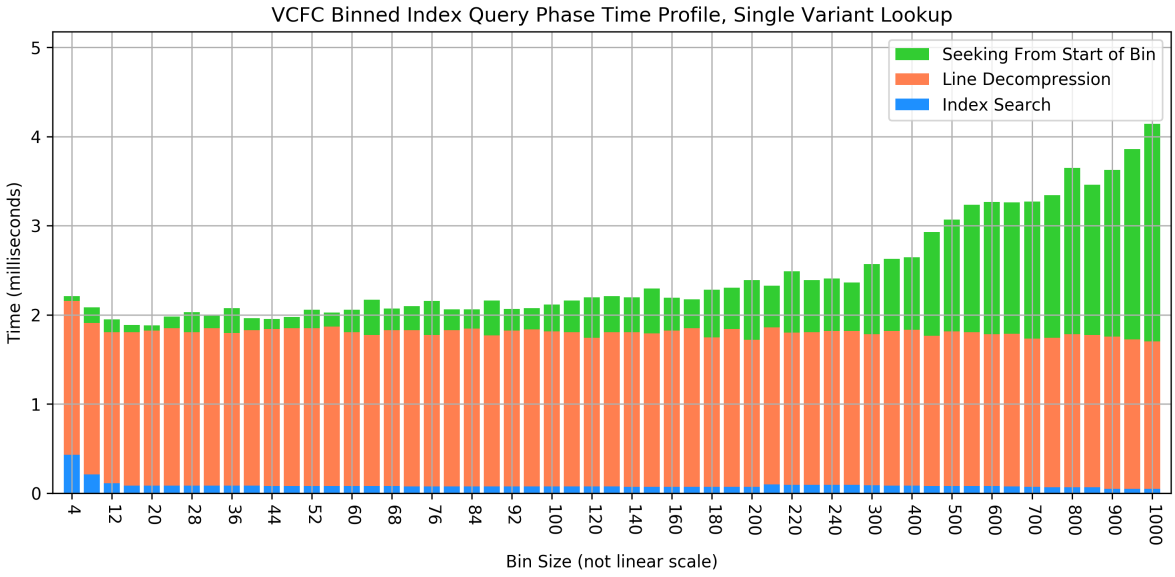

Figure 6.1 Single variant query time profile related to index bin size (Node 1, NVME, EXT4)

pattern. The downside to using this technique is that there are a larger number of seek operations and short reads from the file, which has can have a higher overhead. As discussed previously, SATA does not service these short I/O operations as well as NVME does.

In Figure 6.2 and Figure 6.4, we see the difference in index search between the SATA SSD and NVME SSD, particularly at bin sizes 4, 8, 12 before the polynomial dropoff converges to very close to zero. Figure 6.3 and Figure 6.4 show no difference in the binned index time profile for EXT4 or XFS on Node 2. For dense files and identical patterns of read calls, EXT4 and XFS performed the same.

Another observation is that for the higher bin sizes in these time profiles, the SATA SSD performs slightly better (0.1-0.2 milliseconds). This is due to the line decompression phase decreasing at higher bin sizes. This could be due to the Linux kernel using different readahead caching policies for the two different storage devices. Linux will adaptively set the readahead cache policy based on device statistics and application behavior. The NVME device has better IO operation servicing, and during the *seeking from start of bin* phase before reaching *line decompression*, small reads are being performed until reaching the desired line to start decompressing. During this phase, the SATA drive might place more data into the readahead cache than the NVME drive, which benefits it when it finishes seeking and finally reaches the line decompression phase.

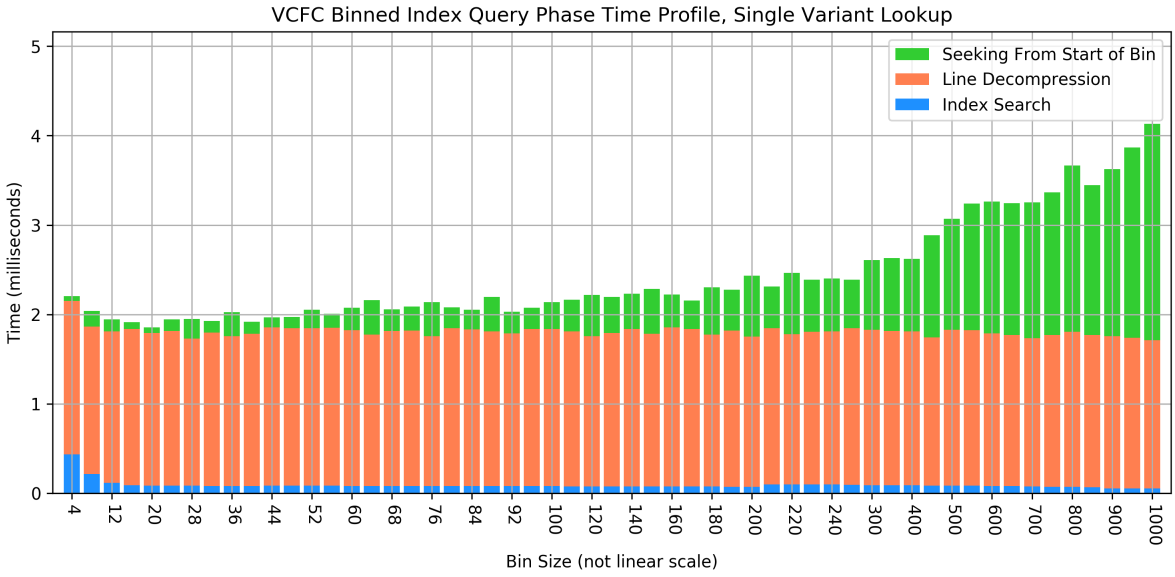

Figure 6.2 Single variant query time profile related to index bin size (Node 1, NVME, XFS)

6.2.1.2 Range-Based Variant Queries

The figures in this section show the same time profiles based on bin sizes as in the time profiles in the Single Variant section above, but this time using queries ranging 5,000 base positions extending past each start position. It takes every query performed in the single variant lookups, and obtains an end position for the query by adding 5,000 the start position of the query. Figure 6.5 and Figure 6.6

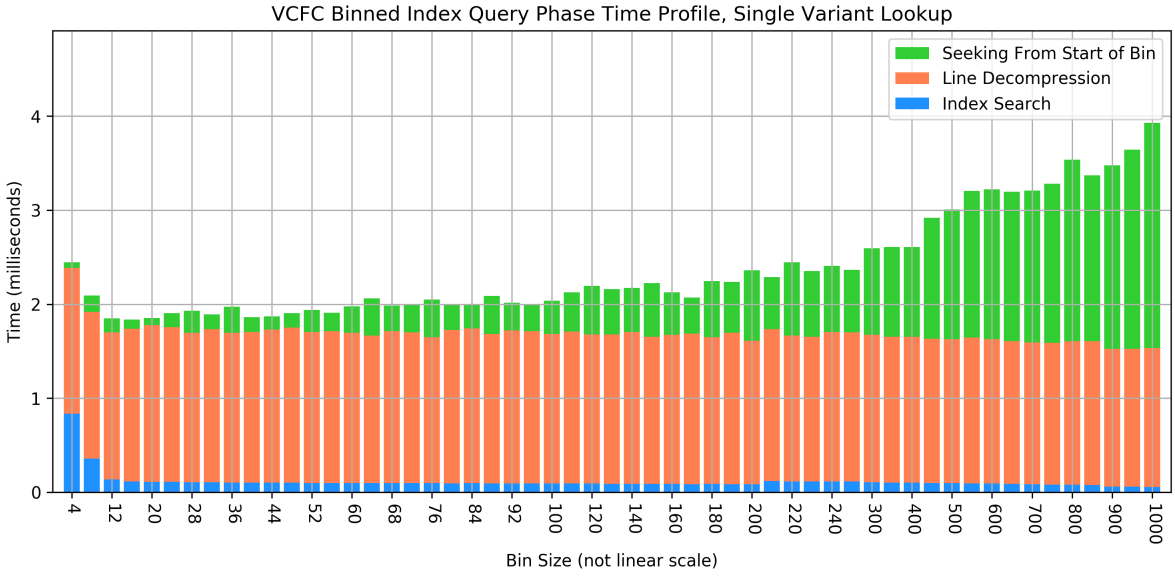

Figure 6.3 Single variant query time profile related to index bin size (Node 2, SATA, EXT4)

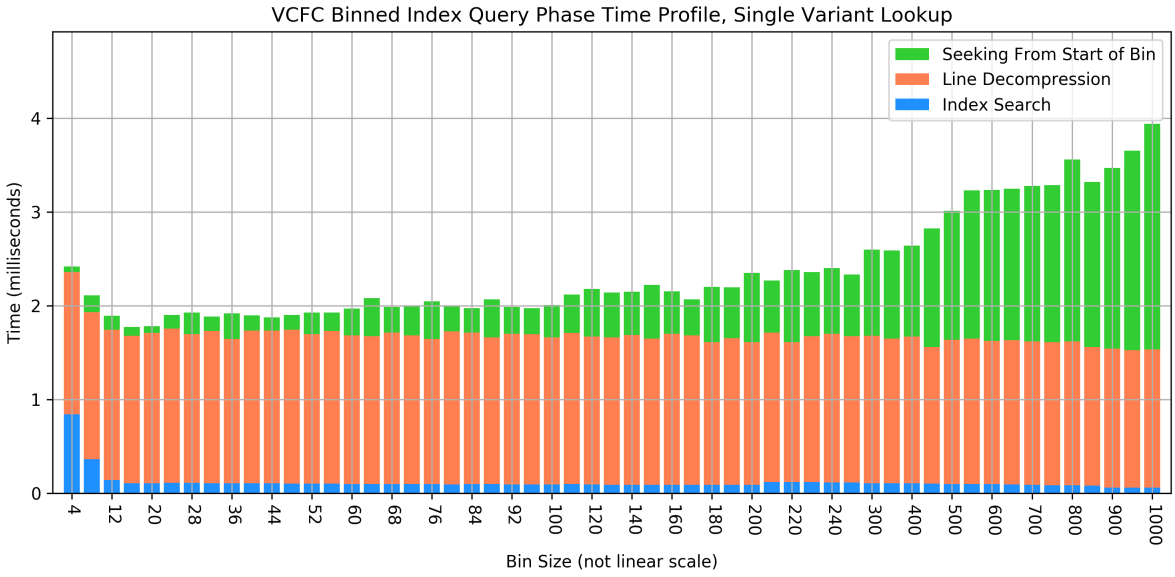

Figure 6.4 Single variant query time profile related to index bin size (Node 2, SATA, XFS)

are results obtained on Node 1 with EXT4 and XFS, respectively. And Figure 6.7 and Figure 6.8 were performed on Node 2, using EXT4 and XFS, respectively.

The y-axis scale here is larger than in the single variant lookup profiles, because the overall query time was larger due to more variants being decompressed and output. The phase for seeking from the start of the bin followed the same pattern as in the single variant lookup profiles, and the index search was also essentially constant (which is difficult to see because it comprises such a small portion of the time, shown at the bottom of each bar). In these queries, as opposed to the single variant lookups, the phase spent in line decompression was a much more significant portion of the overall time.

In Figure 6.5 and Figure 6.6, we observe no difference in profile times between EXT4 and XFS on Node 1. Similarly, in Figure 6.5 and Figure 6.6 we observe no difference in profile times between EXT4 and XFS on Node 2. This can be explained by the essentially equivalent read performance between XFS and EXT on the same device (see Table 6.1).

However, there was a time difference between the nodes. As in the single variant queries in the previous section, the SATA SSD performs worse than the NVME device during the index search phase, particularly at low bin sizes, which involve more small index entries needing to be read and searched. Also, as in the single variant lookups, the line decompression performs better on the SATA SSD than on the NVME SSD, particularly at high bin sizes.

In addition, the NVME SSD times range from approximately 6ms to 9.2 ms, while the SATA times range from 5.8 ms to 8.5 ms. This could again be due to the bin size affecting the pattern of read calls to the device during the short reads in the seeking phase, which then affects cache contents available during the decompression phase.

There are some clearly better and worse bin sizes. The queries used to evaluate did not test every

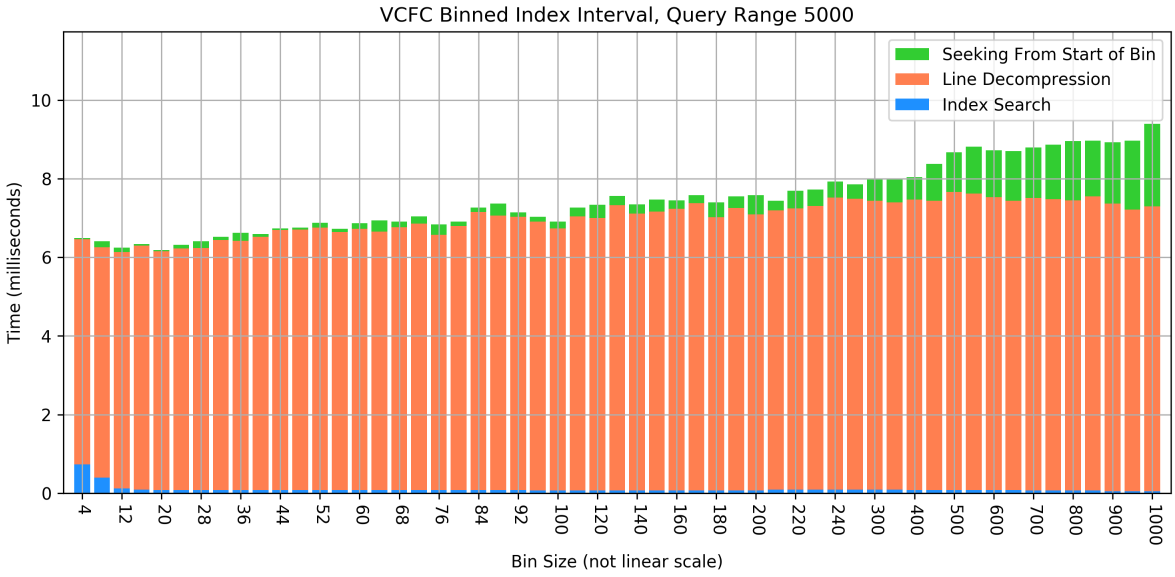

Figure 6.5 Range variant query time profile related to index bin size (Node 1, NVME, EXT4)

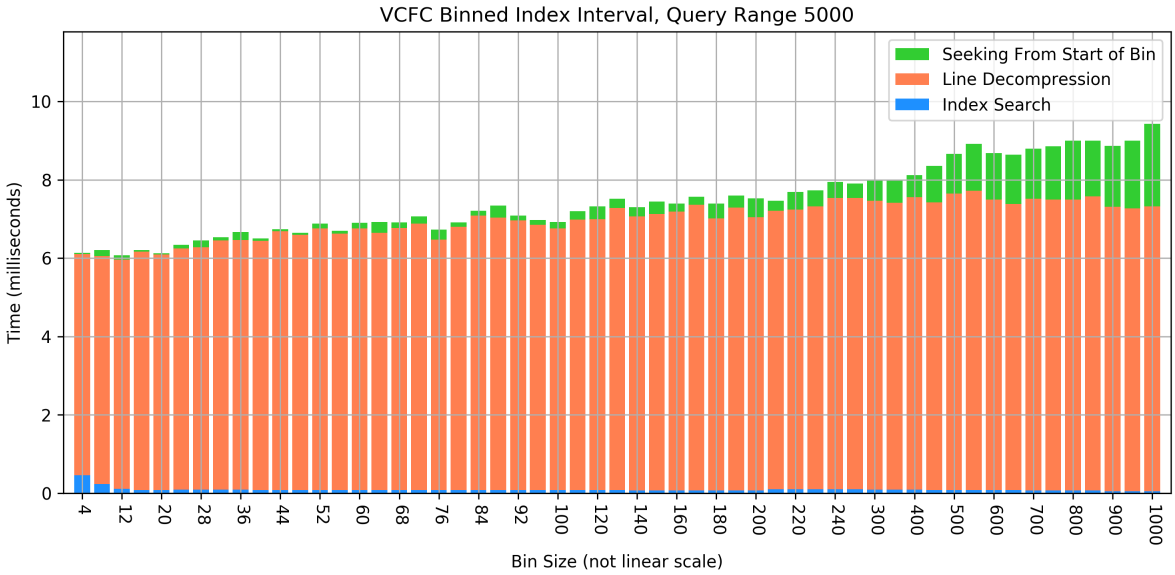

Figure 6.6 Range variant query time profile related to index bin size (Node 1, NVME, XFS)

possible query, but rather a smaller uniformly distributed sampling of possible queries. Results may be affected by bin size choices for the queries, such that the average modulo of the query result's line position within the file was lower for some bin sizes than for others. This would lead to those bin sizes performing better for that set of queries but not necessarily on average across every possible single query. One example supporting this explanation is that there is a slight local maximum around bin size 500 and a local minimum around bin size 100, which is a trend present in all four of the range-based query graphs.

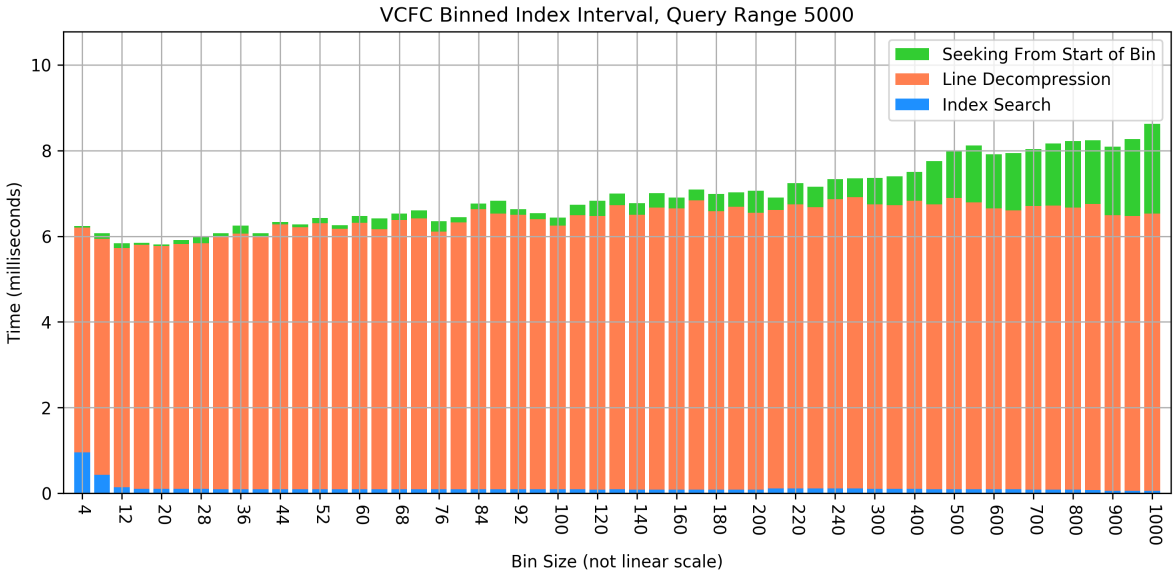

Figure 6.7 Range variant query time profile related to index bin size (Node 2, SATA, EXT4)

In Figure 6.7 and Figure 6.8, which show results for EXT4 and XFS on the SATA SSD, we observe a higher index search time than in the same query profiles on the NVME SSD device. The small bin sizes involve more short reads and seeks, which perform better on NVME than SATA.

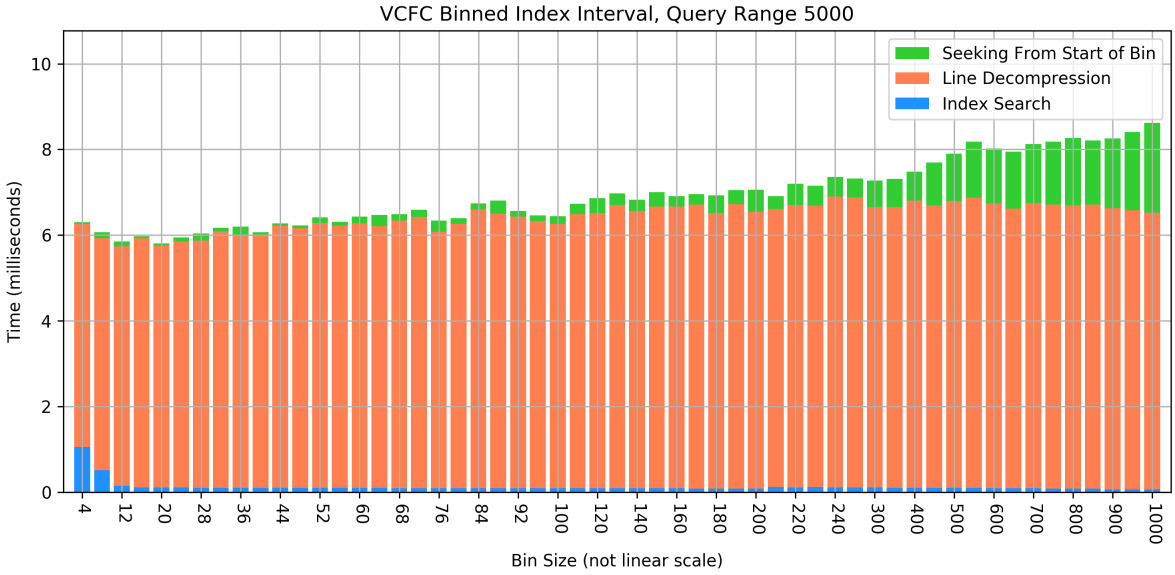

Figure 6.8 Range variant query time profile related to index bin size (Node 2, SATA, XFS)

6.2.2 Query Performance

6.2.2.1 Single Variant Queries

Figure 6.9 depicts results for Node 1 with EXT4, Figure 6.10 those of Node 1 with XFS, Figure 6.11 those of Node 2 with EXT4. Figure 6.12 those of Node 2 with XFS. Each shows the BGZIP and BCF compression with Tabix indexing, compared to the VCFC compression with a sparse offset-as-index, sparse external index, and the external binned index techniques. The near-horizontal lines are the linear trend lines for the respectively colored data series. In these single variant position queries, all three of the VCFC index strategies returned query results in roughly half of the time as Tabix for either BCF or BGZF.

In Figure 6.9 we see the two Tabix index trend lines overlapping and averaging around 375 milliseconds, and the three VCFC index trend lines overlapping just over 200 milliseconds. The VCFC Sparse External Index does slightly worse than the other VCFC indexes near the front of the chromosome VCF file, but the linear trend line has a moderate negative slope does, which implies that this index improves in comparison to the others at the end of the file. However, linear trend lines are a limited way to compare slopes between the data series, and they are so similar to each other in slope there is very little difference.

In Figure 6.10, which displays results for XFS on Node 1, we see that the Sparse Offset-as-Index is slower than the other VCFC indexes, especially compared to Figure 6.9. This suggests that this strategy performs worse on XFS than EXT4. The other indexing strategies performed approximately equivalently. Between EXT4 and XFS on Node 1.

For the SATA results on Node 2 in Figure 6.11, the variance of data points within the Tabix with

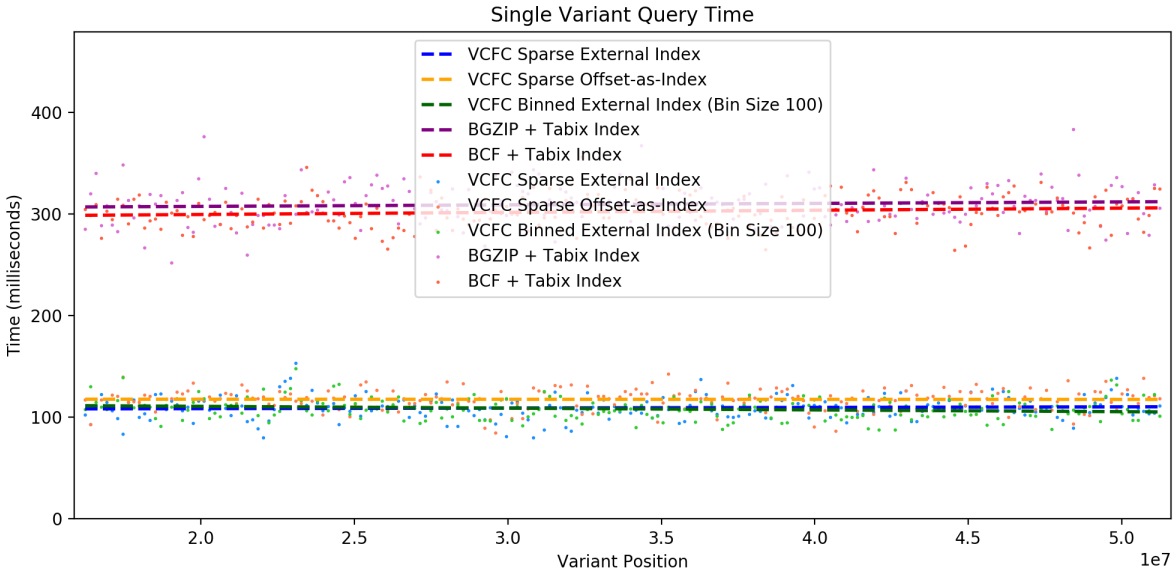

Figure 6.9 Single Variant Lookups by Position (Node 1, NVME, EXT4)

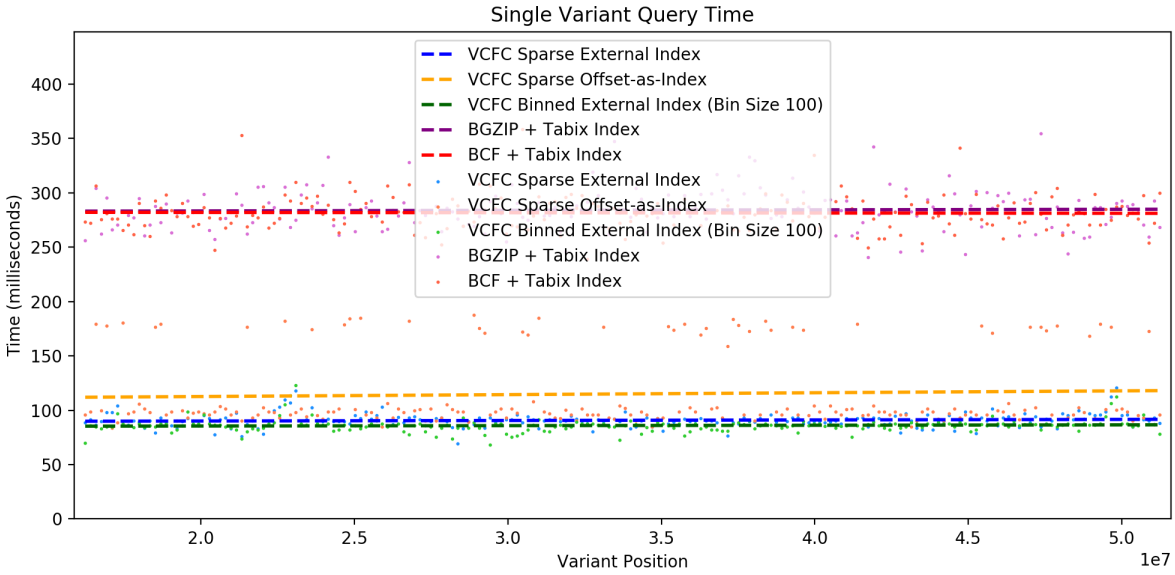

Figure 6.10 Single Variant Lookups by Position (Node 1, NVME, XFS)

BCF and Tabix with BGZIP series from each series' trend line was significantly higher compared to the absolute value of each data point, than for the same queries performed on the NVME device, shown in Figure 6.9. There are a small number of query turnaround times that were much higher (60-80%) than the trend line. This conflicts with observations of read time standard deviations from Table 6.1, which show a larger standard deviation on the NVME SSD than on the SATA SSD. However, the reads in that table are single uninterrupted reads, while these index queries involve many seek operations as well as short variable-length reads. This suggests that NVME can service a larger number of short I/O operations with a more reliable turnaround time (lower standard deviation) than SATA can. The difference between the worst-case and average case was larger on the SATA SSD than on the NVME SSD.

Figure 6.12, which displays query performance using XFS on Node 2, shows a large difference between Figure 6.11, which displays results for EXT4 on Node 2. We observe relatively similar trend lines for all indices except VCFC Sparse Offset-as-Index, which is slower on XFS. We also observe significant outliers for BGZIP + Tabix, taking 150-300 milliseconds, while the series average was between 45 and 60 milliseconds.

When comparing the Node 1 results graphs to the Node 2 results graphs we see interesting findings. The NVME SSD on Node 1 performed roughly 5X worse than the SATA SSD on Node 2. This is difficult to account for. It could be that the cache flushing mechanism used causes extremely poor performance outcomes for the NVME SSD, more so than for the SATA SSD. The SSDs were also released in different years, with the SATA SSD device being newer, which may benefit from newer and improved internal technology.

The script which flushes the cache before each run has a vastly different runtime between the NVME SSD and SATA SSD. Averaged over 50 runs, the flush-cache script takes an average of 1.09

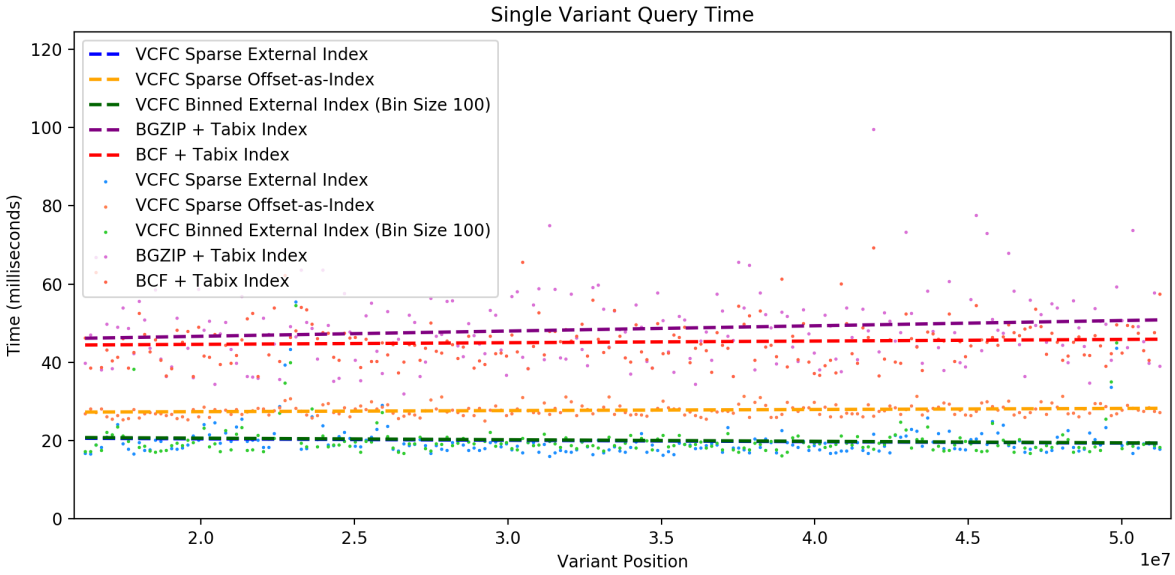

Figure 6.11 Single Variant Lookups by Position (Node 2, SATA, EXT4)

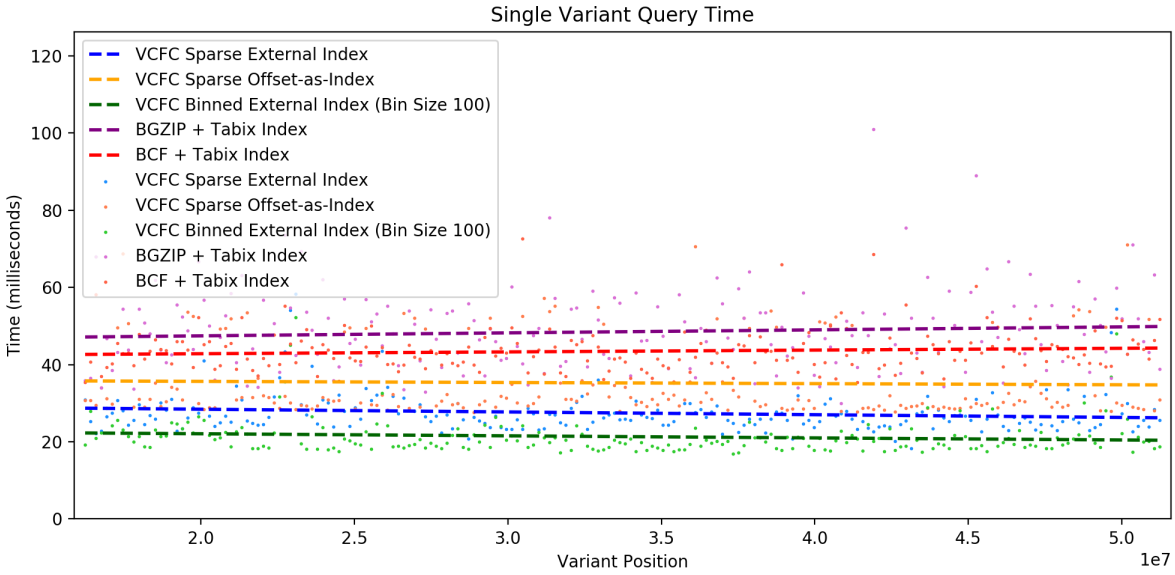

Figure 6.12 Single Variant Lookups by Position (Node 2, SATA, XFS)

seconds on the NVME SSD, while only 0.31 seconds on the SATA SSD. The NVME SSD is 1.6 TB while the SATA SSD is only 250 GB, which could play some role in this time discrepancy, however the only active processes interacting with these storage devices were the evaluation scripts, so no caching by the kernel should have been taking place outside the files we were interacting with. In the case of this 50 run evaluation of cache flush runtime, no files on either storage device were touched, so there should be no data from those devices in either kernel page cache. This suggests that there is a big difference in how caching is managed between the two devices, which happens at multiple

layers, from the standard library, to the kernel page cache, to the storage device's onboard cache. And this difference leads to a read from the NVME SSD with a cold cache being worse than a read from the SATA SSD with a cold cache.

6.2.2.2 Range-Based Variant Queries

The range-based variant queries shown in this section have much higher times than the single variant queries, because many more lines are decompressed and output.

For Node 1, Figure 6.13 displays results on EXT4, and Figure 6.14 displays results on XFS. For Node 2, Figure 6.15 displays results on EXT4, and Figure 6.16 displays results on XFS. These show the same query start positions as in the previous single variant queries section, but using a 5,000 base position range extending past each start position. They each show evaluations of BCF and BGZF with Tabix indexing, and VCFC with Sparse Offset-As-Index, Binned External Indexing, and Sparse External Indexing.

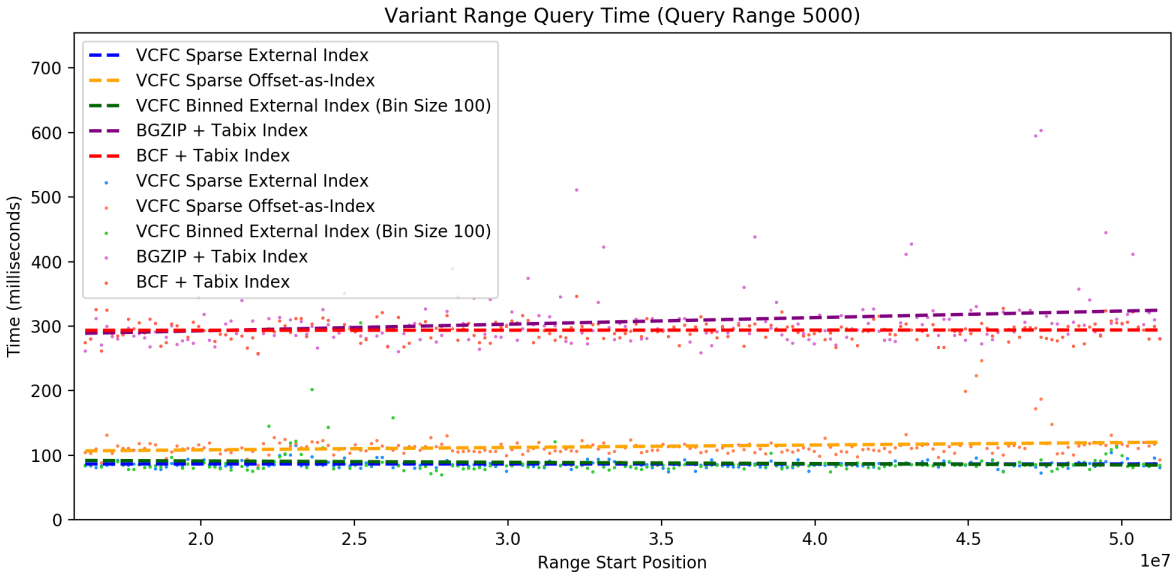

Figure 6.13 Range-Based Variant Lookups by Position (Node 1, NVME, EXT4)

For the queries on the NVME SSD device in Figure 6.13 and Figure 6.14, the VCFC Sparse External Index and Binned External index performed roughly equivalent to the single variant queries in the previous section. However, the Sparse Offset-as-Index technique performed worse. This is because in order to traverse between adjacent variant records in the file, the reader must, for each line, read an additional 16 byte header saying how far the next line is away, and then perform a file seek operation to it. This cost was not incurred in the single variant queries because only 1 line was requested. The Sparse Offset-as-Index technique performed particularly worse on XFS, increasing turnaround time by approximately 20-25%. This suggests that XFS performs significantly worse with reading and traversing sparse files than EXT4 does.

For these same Sparse Offset-as-Index queries on the SATA SSD device, the turnaround time suffered even more. This is due to, as previously discussed in the single variant query section, the

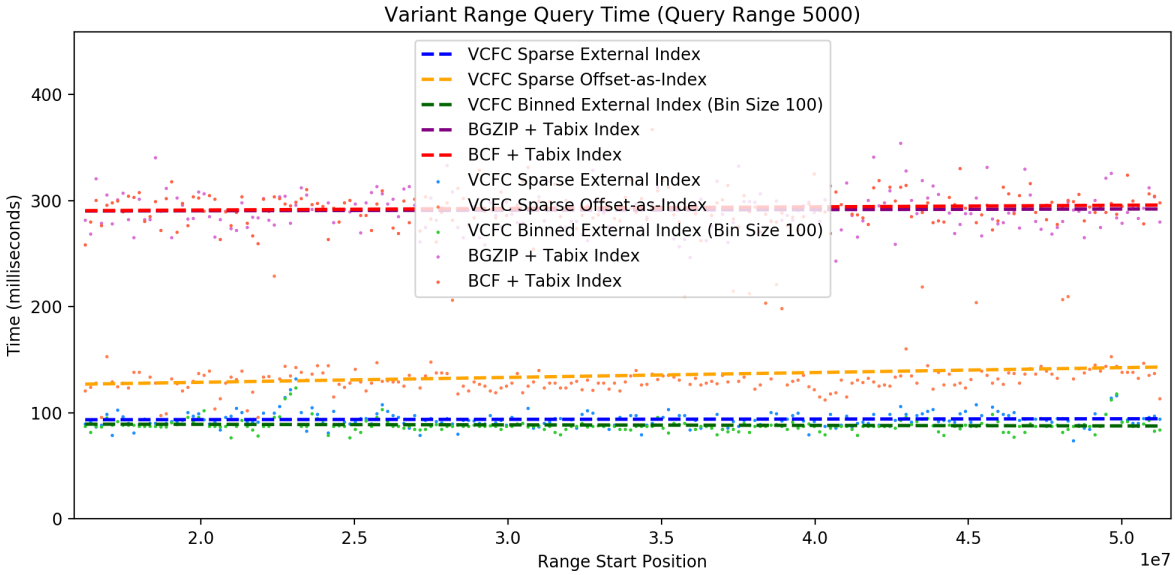

Figure 6.14 Range-Based Variant Lookups by Position (Node 1, NVME, XFS)

limitations of SATA service of frequent short I/O operations such as 16 byte reads and file seeks. The Sparse Offset-as-Index indexing strategy relies much more heavily on those than the other indexing strategies, leading to a lower SATA performance than the other indexing strategies.

The results for the the SATA SSD device are displayed in Figure 6.15 and Fig 6.16. As in the single variant queries for this device, we also observe that the two Tabix indexes had much higher variance between some data points in each series and each series trend line than on the NVME devices.

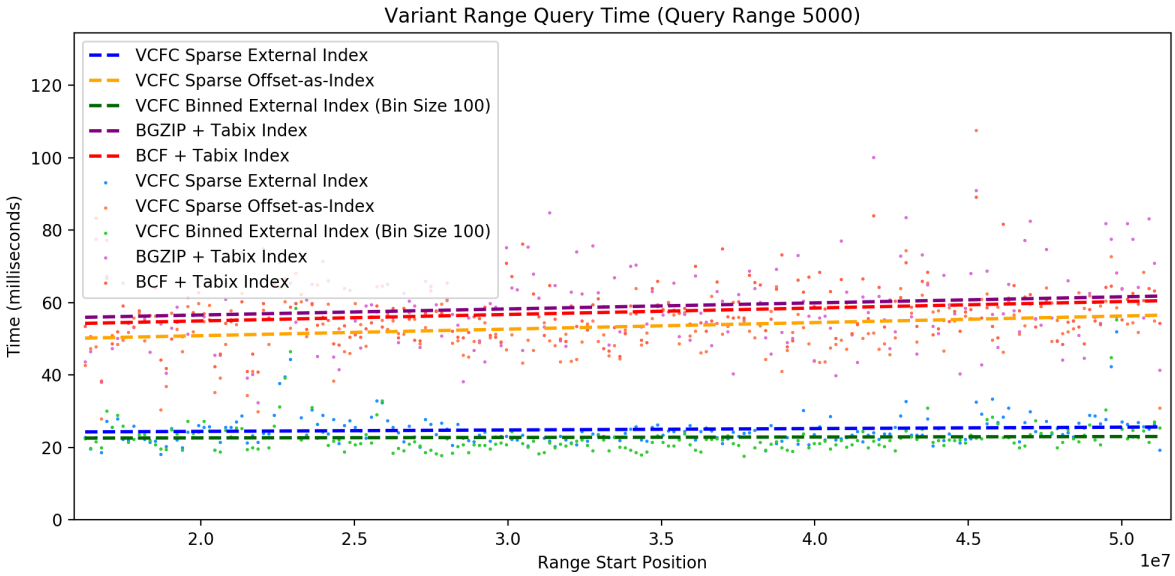

Figure 6.15 Range-Based Variant Lookups by Position (Node 2, SATA, EXT4)

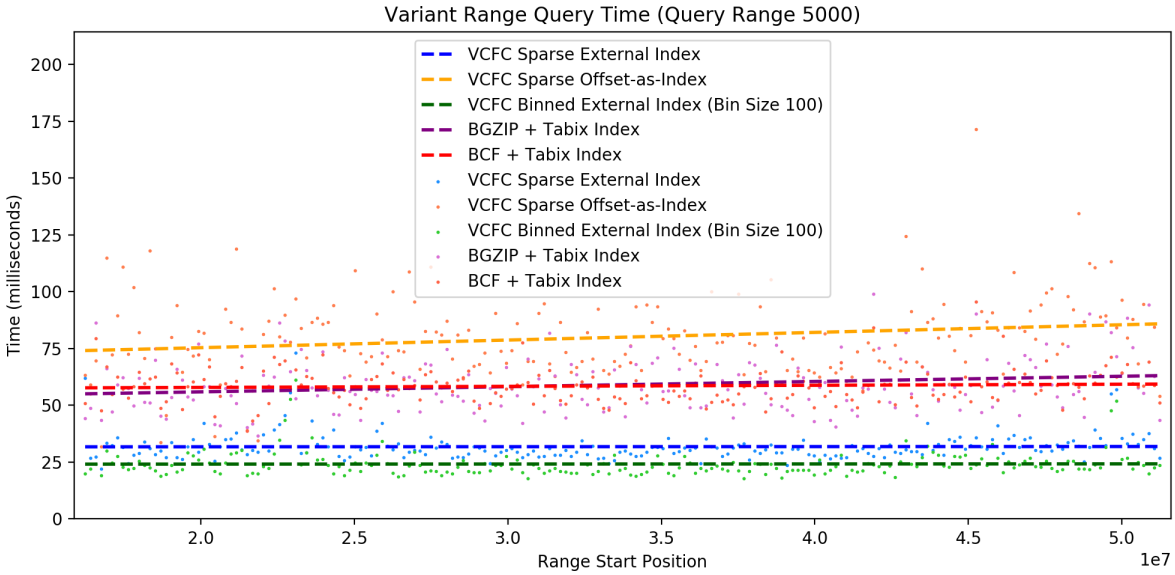

Figure 6.16 Range-Based Variant Lookups by Position (Node 2, SATA, XFS)

Between Figure 6.15 and Figure 6.16 we observe a large difference for both the VCFC Sparse External Index and the VCFC Sparse Offset-as-Index, suggesting a substantial performance cost for sparse file traversal on XFS. This large discrepancy did not exist for single variant queries because those perform no traversal of the file, they only look at one record location (see Figure 6.11 and Figure 6.12). The difference in VCFC Sparse External Index times between EXT4 and XFS was much lower than the difference between EXT4 and XFS for VCFC Sparse Offset-as-Index. This is because the Sparse Offset-as-Index traverses more sparse file regions and file holes than the Sparse External Index, making the cost of the XFS sparse file handling more prominent.

Table 6.2 Index Creation Times

	EXT4 NVME	XFS NVME	EXT4 SATA	XFS SATA
Tabix BGZF	34.38 ± 0.07	34.32 ± 0.08	35.02 ± 0.07	35.08 ± 0.13
Tabix BCF	12.22 ± 0.07	12.19 ± 0.08	11.62 ± 0.05	11.59 ± 0.02
VCFC Binned External Index	3.68 ± 0.03	3.78 ± 0.05	3.24 ± 0.04	3.26 ± 0.04
VCFC Sparse External Index	10.62 ± 1.04	12.82 ± 0.69	66.76 ± 1.01	28.97 ± 7.03
VCFC Sparse Offset-as-Index	17202	32150	> 12 hours	> 12 hours

6.2.3 Index Creation Time

Another area of interest is the time to create the index. When records in a dataset are added as is common in genomic datasets, the dataset must be re-indexed in order to reflect the changes. For both Tabix and VCFC contiguous external binned indexing, this involves re-writing the whole index file. For the sparse index strategies, since indexing is based on file offset and any unused record offsets are left as sparse file holes, if new records need to be added they can just be inserted into the file holes, without disturbing or needing to rewrite any of the rest of the file.

For evaluations of index creation time, shown in Table 6.2, the 1000 Genomes Chromosome 22 VCF file was used, along with its compressed BGZF, BCF, and VCFC versions. For VCFC binned index, indexing time for a file depends on the bin size used, and those values are shown in Figure 6.18. Tabix does not have a mechanism for specifying bin sizes or number of bin layers for its layered binning approach, so those two indexing times are static and recorded below. The VCFC Sparse Offset-as-Index and VCFC Sparse External Index also use a static indexing method, so they are also recorded below as a single value. The VCFC Binned External Index strategy uses a linear indexing strategy as opposed to a multi-level approach, so the indexing time does not vary substantially based on the bin size used, rather it largely varies based on the size of the input file that must be read during indexing; for this reason that strategy is also shown below with times based on a bin size of 100.

Table 6.2 depicts runtimes in seconds, and all filesystem block sizes are 4 KiB. The NVME and SATA drives, and processor types and cache specifications are described in Table 5.1 of Chapter 5. For the SATA device and the VCFC Sparse Offset-As-Index technique in the last row, the run time was so large that performing it numerous times to average was deemed unnecessary for a comparison to other values, which were several orders of magnitude lower.

Figure 6.17 shows the binned index creation time when entries were inserted at regular *bin size* intervals, even if that same information could be inferred from the adjacent index entries. We see observe a polynomially decreasing relation between the bin size and the time, as the number of entries written out is the size of the position range of the VCF file divided by the bin size.

Figure 6.18 shows the binned index creation time using an implementation that removed redundant information, that is, entries which contained no new information than what could be inferred during the servicing of a query from either the entry before or the entry after. This reduced the number of index entries significantly in the worst and average cases and eliminated the polynomial

relation to the bin size. The y-axis range in this figure only covers only approximately 0.25 seconds, while the y-axis in Figure 6.17 covers 1.75 seconds. The best case in the new version is not as low as the best case in the previous version (3.25 seconds). But as discussed in Section 6.2.1 (VCFC Binned Index Profiling) showing query turnaround times based on bin size), we want to keep the bin size relatively low i.e, between 50 and 300. This fact reduces the difference by only comparing the data points from Figure 6.18 against the leftmost points from Figure 6.17.

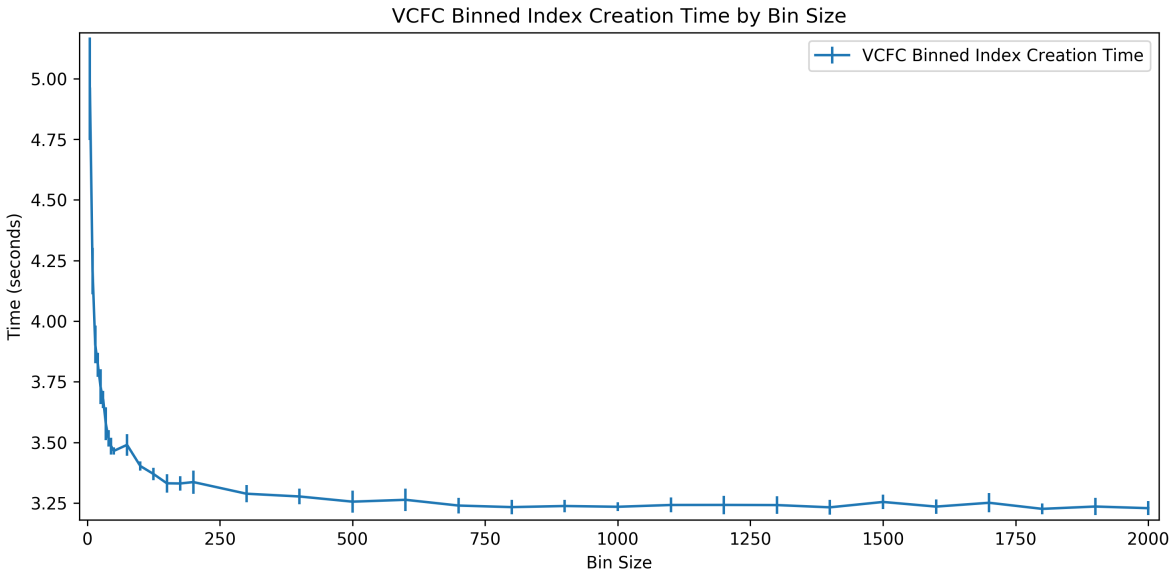

Figure 6.17 VCFC Binned Index Creation Time (Before Redundant Entry Removal)

An interesting observation during the process of writing the two sparse index files is that it is so much slower than writing the dense index files (see Table 6.2), and a substantial amount of work is being done by kernel threads. In other indexing methods, the work is performed almost entirely by the user space program process with very little work being done in kernel space. However, in the sparse index methods, the balance of work shifted into kernel space, with 2-4% core utilization by the program process, and 90%+ core utilization by a Linux kernel kworker thread. Because this reduced the core utilization of the indexing program, this significantly increased the time to write the file.

This kworker represents work performed at the filesystem layer. During the writing of the sparse files, the records are spaced out and aligned to a file offset that is a multiple of the filesystem block size. In the Sparse Offset-As-Index technique, for every variant line, between one and four 4 KiB blocks need to be allocated in the filesystem (see Figure 4.5). Both EXT4 and XFS also use the concept of *extents*: file metadata blocks that record 1 or more contiguous data blocks. EXT4 has a maximum extent size of 128 MiB and XFS has a maximum extent size of 8 GiB [5] [25]. Since our sparse files cover an offset range much larger than these maximum extent sizes, in addition to allocating blocks,

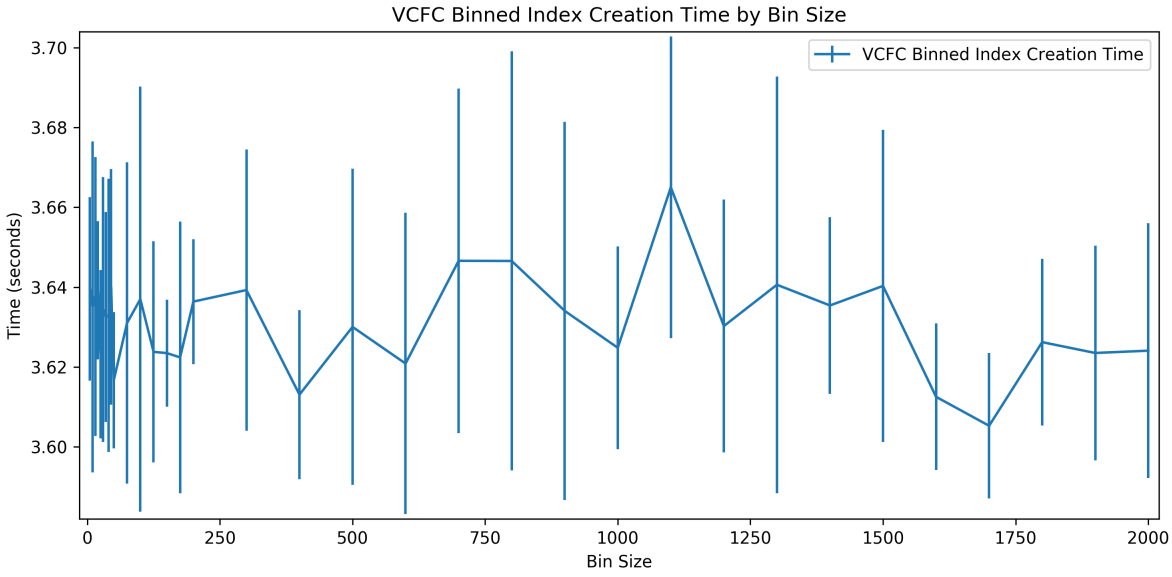

Figure 6.18 VCFC Binned Index Creation Time

at a certain interval, sparse file writes lead the filesystem to allocate an additional extent. The Sparse Offset-As-Index method is very sparse, so it must allocate many extents. It also uses a position multiplication factor (F) of 4 blocks (16 KiB), so at most, 8,192 VCF lines will fit in an EXT4 extent, and 524,288 in an XFS extent. This occurs when all VCF positions differ by one (1, 2, 3, etc.), which is uncommon. In the 1000 Genomes Project chromosome 22 VCF file, the average gap between variant start positions is 31, so during the translation of this file to a Sparse Offset-As-Index file, on average, an extent is created roughly every 264 lines for EXT4 and every 16,912 lines for XFS (see formulas in Figure 6.19).

$$\frac{blocks\text{–}per\text{–}extent}{blocks\text{–}per\text{–}position} = positions\text{–}per\text{–}extent$$
$$\frac{positions\text{–}per\text{–}extent}{positions\text{–}per\text{–}line} = lines\text{–}per\text{–}extent$$

Figure 6.19 Formula for computing Sparse Offset-As-Index VCF lines per extent

This additional bookkeeping work by the filesystem explains some of the overhead during sparse file creation. However, given the average interval between extent creations, it is unlikely to explain it in full. In particular, it does not explain the discrepancy between EXT4 and XFS sparse file write times in which EXT4 is substantially faster. In addition to the creation of extents, before a write to a file offset, the filesystem must first to check if the offset is in an existing extent. This process of checking for an extent containing the desired offset has a computational cost. EXT4 stores extents in a tree structure with a variable branching factor, but XFS uses a list. This difference in offset-to-extent resolution strategy could explain the large difference between EXT4 and XFS for the two sparse

index strategies in this work, and the difference became more prominent when putting more stress on block allocations from writing to such sparse offsets.

CHAPTER

7

RELATED WORK

The most directly related other works to this work are Tabix, BGZF, and BCF, covered fairly extensively already throughout this work. In addition, engineering work by many institutions is continually ongoing in the area of genomic storage optimization in cloud data warehousing environments.

In *Next Generation Indexing for Genomic Intervals* [10], the authors describe the challenges posed in servicing range based queries on large scale genomic data using traditional indexing strategies and existing domain-specific indexing strategies. Existing strategies require expensive file scans, or tree traversals which work well for single positional lookups but which may suffer in lookups of larger ranges of positions. The solution developed by the authors seeks to alleviate these issues by focusing on semantic information of genomic *regions*, in order for a query to quickly gain information about a particular region (marked by a start and end position) without doing a wider file scan.

Sparse indexing as a general method of mapping only a subset of key tuples has been used in a variety of database technologies [23] [15] and can also be referred to as *binning*. However, these sparse indexes are stored in dense files. To date, no technique has been developed for leveraging filesystem sparse files for key-to-offset based indexing. Technologies such as QEMU [16] can leverage sparse file support for reducing disk utilization of the backing file for guest filesystems, however distribution of data within the sparse file is based on filesystem allocations of the guest filesystem, not based on a deterministic translation of the object keys, which in a filesystem are file paths.

Samsung, along with other organizations, is working on a novel storage API layer enabling key-value SSD storage through a key-value device driver as opposed to a traditional block-based filesystem layer. This was first presented at the Storage Developer Conference (SDC) in 2017 [26] and has been described in several technical news articles [3] [4]. This technique hashes object

keys and maps them directly to physical sectors on the storage device, instead of going through a filesystem tree and a file block mapping. The advantage is that an object can be queried in a time complexity based on the length of the key (or file path) instead of via a tree walk, however a disadvantage is that the tree layout often used to usefully organize objects is lost. The scenarios discussed in the conference talk focused primarily on the most clear match, which is key-value databases such as Redis and RocksDB. Indexing could be moved from the software application layer into the hash function and corresponding hash lookup from the storage device. They showed a clear I/O performance gain for key-value databases backed by a key-value device driver over a traditional block filesystem. However they believe it may also be used in more general purpose document databases such as MongoDB and object storage such as Ceph. If the performance of such adaptations prove to be better than the traditional filesystem approach as backing stores to Ceph, this key-value storage could be integrated into cloud storage systems such as S3 and HDFS which are used in cloud compute tools such as Hadoop, Jupyter, and Spark.

CHAPTER

8

CONCLUSION

The hypothesis, that a novel encoding technique for genetic variant data devised based on structure- and semantic-aware compression and indexing can improve response times for common queries, is justified by the observed results. The evaluations show that the line-based partial compression technique for genotype data can provide comparable compression and improved index response times for common use case patterns compared to existing methods, which utilize more generic, full, block-based compression formats.

In addition, indexing techniques that place a heightened focus on minimizing I/O from storage can perform better than indexing techniques that do not, even if the former is a much simpler format that may seem intuitively to be missing search algorithm advantages of a more complex index structure. Tabix uses a more complex multi-layered indexing approach that may perform well when the index and the data file are cached into main memory, but the block-based approach leads to more data being processed during the servicing of each query.

Another finding was that the pattern of read, write, and seek calls has a significant impact on the performance of compression, indexing, and index querying software, and this impact can be dependent on the storage technology used. The NVME SSD used in evaluations has a much higher read and write bandwidth than the SATA SSD, but counter-intuitively performed worse in some evaluations. Additionally, the filesystem used for storing data files has a large impact on the read and write performance of both dense and sparse files, as well as on the difference in turnaround times between the average and worst-case inputs.

After the evaluation of using sparse files and their offsets as an indexing technique based on a semantic query to offset translation, it appears that this novel form of indexing is promising and should be an area of additional exploration, as should the general area of key-value constant-

time storage strategies based on a constant-time mapping of keys to physical storage locations. Limitations around the 1:1 mapping nature of field tuples to offsets in the Sparse Offset-as-Index technique can be addressed using a multi-layered approach in which sparse index entries point to a second-layer of objects, which contain 1 or more record entries. However, for sorted data, like the sorted VCF files in this work, the Sparse External Index addresses these fallbacks and also does not have the downside of such excessive index-creation times and query time increasing in cost as much on SATA SSD devices. Given that short read and seek I/O patterns of the Sparse Offset-as-Index strategy suffered so much on the SATA SSD compared to the NVME SSD, it can be assumed further that on a spinning disk for which seek latency is much higher, performance would be even more adversely impacted.

CHAPTER

9

FUTURE WORK

- One improvement is the size-conditional compression of the INFO column in VCF files. This paper does not consider the compression of INFO columns, which are arbitrarily-large key-value pairs of additional features of each variant not stored in other columns. A length header can be injected into the line to say how long the INFO column is, so that it can be seeked over if not needed, but its existence in the file will likely incur file reading and memory overhead consisting of some of those unused bytes, if a read-ahead buffer enabled API is used to access the file, and if the likely event that column data not needed crosses filesystem block boundary into a block which contains data that is needed, resulting in the reading of physical storage sectors that are not needed. So compressing large INFO columns would be beneficial.

  The BCF file format uses a mechanism for schema-enforcement on INFO columns which can compress the column if the key labels are long, however the amount of compression gained through this is not small. Additional work can be performed to further leverage content format constraints of the INFO column to reduce its size and improve queries on keys it contains. For example, a nested dictionary encoding of key names and a map to value offsets, can provide faster lookups on the key-value pairs, to avoid an exhaustive parsing of all keys and values when only a small number of them are requested, as long as this additional encoding does not significantly increase the size of the variant record in the file.

- In the discussion of run lengths in Chapter 4, there is an analysis of how many samples of the same genotype were adjacent to each other in the file. Another optimization for compression would be to compute similarity scores between samples, and reorder them in the column order, so that more similar samples are adjacent to each other. This would improve compression

in any scheme using row-oriented run-length compression, which VCFC, BCF, and BGZF do. Given the large number of variants, computing an pairwise similarity computation for all pairs of samples and all variants within the samples is expensive. A smaller number of important genetic markers, which can, with some statistical confidence, distinguish superpopulation or subpopulation groupings, could be used compute similarity scores. Fewer values would need to be compared, and other variants tend to correlate with these population markers, so it could be a good way to heuristically determine sample similarity.

- Other general data warehousing technologies in use in cloud-based environments often use column-oriented storage and compression. In many use cases this can provide speed-ups by reducing amount of data processed per query if the number of columns selected is kept small. However in firsthand experience, data read sizes, memory utilization, and compute time spent on servicing even filtered queries is much too large. But this does not mean that columnar storage itself is a problem. Other investigations are needed to see if more semantically-tailored columnar formats, indexing, and index-based query processing can lead to benefits over row-oriented solutions in common use case patterns for genomic data. There is likely a trade-off point, such that columnar storage always performs better when only a few columns are selected, but row-oriented storage may be better at higher column cardinality. Overall table dimensions and value repetition patterns must also be considered.

- Tabix was published alongside a technical description paper [12] in 2011. Since then, datasets have grown exponentially in size, but underlying compression and indexing strategies have not fundamentally changed. The source code for Tabix in the `htslib` repository [9] has been actively modified over the years by a number of individuals, demonstrating an active interest in improving tool support. The codebase is very robust, with changes made to improve error detection and handling, to expand input file formats accepted to a wide variety of genetic file formats, and to accept remote URLs as file paths which demonstrates a demand for the capability to use these tools in networked environments where the data is not co-located with the compute resources. Additional work can be performed to leverage the robust file and parsing support of Tabix, and to adapt the novel indexing strategies in this paper to the Tabix codebase.

  Additionally, the BCF file format described in the VCF version 3 specification [24] employs two strategies, row compression and block compression. The second stage of block compression is tied to the `BGZF` blocked-gzip strategy that Tabix depends on. By adding purely row-based indexing support to Tabix, BCF could be adapted to support a non-BGZF variant, which could perform purely row-based compression and have those rows indexed directly.

BIBLIOGRAPHY

[1] *Apache Spark*. https://spark.apache.org/.

[2] *ARC: A Root Cluster for Research into Scalable Computer Systems*. https://arcb.csc.ncsu.edu/~mueller/cluster/arc/. 2011.

[3] Billy Tallis. *Samsung Announces Standards-Compliant Key-Value SSD Prototype*. https://www.anandtech.com/show/14839/samsung-announces-standardscompliant-keyvalue-ssd-prototype. Accessed on: 2020-04-27. 2019.

[4] Chris Mellor. *Samsung's open source key:value SSD is a game-changer for unstructured apps*. https://blocksandfiles.com/2019/09/05/samsungs-potentially-groundbreaking-keyvalue-ssd/. Accessed on: 2020-04-27. 2019.

[5] *Ext4 Disk Layout*. https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout.

[6] Ferriter, K. *VCF Compression Repository*. https://github.com/theferrit32/vcf-compression/. 2020.

[7] *GATK Best Practices*. https://www.broadinstitute.org/partnerships/education/broade/best-practices-variant-calling-gatk-1.

[8] *Hadoop Distributed FileSystem*. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[9] *HTSlib*. https://github.com/samtools/htslib.

[10] Jalili, V. et al. "Next Generation Indexing for Genomic Intervals". *IEEE Transactions on Knowledge and Data Engineering* **31**.10 (2019), pp. 2008–2021.

[11] Leach, R. *The Rise of Genomic Medicine*. https://tedxgrandrapids.org/portfolio-item/the-rise-of-genomic-medicinerick-leach/. 2013.

[12] Li, H. "Tabix: fast retrieval of sequence features from generic TAB-delimited files" (2011).

[13] Li Y, Willer C, Sanna S, Abecasis G. "Genotype Imputation". *Annual review of genomics and human genetics* **10** (2009), pp. 387–406.

[14] *lseek(2) Manual Page*. http://man7.org/linux/man-pages/man2/lseek.2.html. Accessed on: 2020-04-20.

[15] MongoDB, Inc. *Sparse Indexes*. https://docs.mongodb.com/manual/core/index-sparse/. 2020.

[16] *QEMU disk image utility*.
https://www.qemu.org/docs/master/interop/qemu-img.html.
Accessed on: 2020-04-26.

[17] *SAM Specification version 1*. https://samtools.github.io/hts-specs/SAMv1.pdf.
The SAM/BAM Format Specification Working Group, 2020.

[18] *Samtools*. https://www.htslib.org/. Wellcome Trust Sanger Institute.

[19] Sims, D., Sudbery, I., Ilott, N. et al.
"Sequencing depth and coverage: key considerations in genomic analyses" (2014).

[20] Stack Exchange user ewwhite. *XFS Dynamic Speculative Preallocation*.
https://serverfault.com/a/406070. 2012.

[21] Stavropoulos DJ, Merico D, Jobling R, et al. "Whole Genome Sequencing Expands Diagnostic Utility and Improves Clinical Management in Pediatric Medicine".
NPJ Genomic Medicine (2016).

[22] The 1000 Genomes Project Consortium. *A global reference for human genetic variation*.
Nature 526, 68-74. 2015.

[23] The PostgreSQL Global Development Group. *PostgreSQL BRIN Indexes*.
https://www.postgresql.org/docs/12/brin-intro.html. 2020.

[24] *VCF specification version 4.3*.
https://samtools.github.io/hts-specs/VCFv4.3.pdf. 2019.

[25] *XFS Filesystem Structure: Data Extents*. https://xfs.org/docs/xfsdocs-xml-dev/XFS_Filesystem_Structure/tmp/en-US/html/Data_Extents.html.

[26] Yang Seok Ki. "Key Value SSD Explained – Concept, Device, System, and Standard".
Storage Developer Conference. 2017.

APPENDIX

APPENDIX

A

APPENDIX

This Appendix covers a set of useful commands and scripts used to run and collect results. The source code created for this investigation is published online in a GitHub repository[6]. Files referenced are hosted in that repository.

A.1 Compilation

The `Makefile` provides targets for 3 builds of the `main` VCFC command line. These are `release`, `debug`, and `timing`. Release suppresses all unnecessary output except for when errors occur, providing only the VCF query results on standard output. The timing build injects timers into critical code sections and prints time information to standard output along side query results. This is used for profiling the code especially higher level phases of the code flow. The debug build contains a substantial amount of debugging output and should not be used except when debugging.

There is also a `uniqc` target to build an executable by that name, which is based around the GNU `uniq` command but modified for the use cases of collecting total counts as well as counts run lengths on a line-by-line basis. These can be performed without an expensive `sort` operation beforehand that the `uniq` command relies on for overall statistics, and which can write large amounts of data to a `tmp` directory for its sorting algorithm's working intermediate memory. This executable, along with GNU `grep` and `cut` is used for the collection of genotype run length and frequency metrics.

A.2 Development

In the `dev` folder in the repository, there is an Ansible playbook called `ubuntu-playbook.yaml` which will configure an Ubuntu machine with the necessary library and header packages, and will build the `bgzip` and `tabix` commands and add them to the `PATH` interactive bash environment variable.

There is also a file called `setup-fedora.sh` which will install all of the necessary packages for Fedora/CentOS/RHEL systems.

A.3 Evaluation

The majority of the evaluation is performed in Python, with some help from shell commands for batching. The `evaluation` folder in the repository contains this code. Within that folder there is another `evaluation` folder which is a python3 module by that name, which contains common functions for running and timing `bgzip`, `bcftools`, `tabix`, and `vcfc` commands.

There is a large file named `evaluation_main.py` which contains a main function for performing a time evaluation run, as well as functions to record and save results, and, for each evaluation, a function to render a graph image for the results.

There is also a `query.py` file which contains code to perform queries for genes. This was not used in the evaluation in this paper because it was not exhaustive and uniform enough across VCF files. With a vastly more comprehensive list of genes, or if it was expanded to support SNP RSIDs as well, it could be used in place of the uniform position distributions.

A.3.1 Scripting

Inside the `evaluation` folder, there are two files used for batching of time evaluations, `run-all.sh` for running, and `graph-all.sh` for graphing.

For evaluating compression times, there is a file called `compression-times.py` which performs 10 runs of each compression program and reports the mean and standard deviation for each.