WHITE PAPER

GenomicsDB: Storing Genome Data as Sparse Columnar Arrays

Authors

Kushal Datta, Karthik Gururaj, Mishali Naik, Paolo Narvaez, Ming Rutar: Intel Health and Life Sciences

Abstract

GenomicsDB is a storage technology for genomic variants and likelihoods. Using high-level APIs provided in C++, Java*, and Python*, users can both write and read variant records to and from GenomicsDB shared-nothing instances in parallel using multiple processes in a Single Process Multiple Data (SPMD) manner. GenomicsDB uses columnar sparse arrays where samples are mapped to rows and genome positions or sites of variants are mapped to columns. These columns are partitioned in a shared-nothing fashion across thousands of machines, enabling the joint genotyping workflow in Broad Institute's genome analyzer toolkit (GATK) to scale to 100,000 samples and beyond. This allows bioinformaticians to achieve analysis results with high statistical confidence. The low-level storage format enables faster and more efficient retrievals from disk compared to the use of files. Additionally, using libraries optimized for Intel® architecture to compress data on disk, GenomicsDB cumulatively achieves orders of magnitude improvement in performance compared to existing tools. In addition, the generalized multi-dimensional array model provides flexibility for GenomicsDB to be extended to other types of genome data.

Table of Contents

Introduction

The field of genetics has advanced remarkably since 1953, the year Watson and Cricks first described DNA's double helix structure (Watson, 1953). Today, high-throughput sequencing machines provide entire human genome sequences within a few days, with high yield and at low cost. As a result, millions of patients are increasingly sequenced across the globe every year (CoreGenomics, 2016). Their genome data is used in predicting the type, nature, and progression of genetic diseases, rare diseases, diabetes, and cancer (Raheleh Rahbari, 2016) (Yuan Yuan, 2014). These advances have the potential to radically transform health care as we know it. In the near future, genetic data in conjunction with phenotype data will be quintessential in precision medicine (Ashley, 2016), where targeted drug combinations will be synthesized for each individual patient.

Genetic mutations—more generally called variants—can affect a single nucleotide or span multiple nucleotides. Variation at a single genome position can be due either to a single nucleotide polymorphism (SNP) or single nucleotide variant (SNV). An example of an SNP is A is replaced by T. A mutation can also span multiple sites such as in INDELs (insertions and deletions) or structural variants where a sequence of nucleotides are altered. The Genome Analyzer Toolkit (GATK) is a set of tools consisting of methods and algorithms to call germline variants or somatic mutations like SNPs or INDELs from the raw sequencing reads called out of the sequencers (McKenna A, 2010) (DePristo, 2011) (Van der Auwera, 2013). The GATK Best Practices Pipeline is a workflow script available from the Broad Institute. It processes reads to call variants in two phases. The first phase, Single Sample Variant Calling, occurs independently for each sample. It takes as input FASTQ files produced by the sequencing machine and aligns the reads against a reference

genome using a combination of heuristics and the Smith-Waterman algorithm. The result of alignment is a sequence alignment/map (SAM) or binary SAM (BAM) file. Reads in BAM files are then ordered according to the kmers or single continuous sequence of reads, and duplicates are annotated to eliminate them from the later stages of analysis. This is followed by a variant-calling step—for example, the GATK Haplotype caller uses De Bruijn graph traversals and a pair-wise HMM (Hidden Markov Model) to compute likelihoods of variants for each sample. These variants are eventually written in a variant call format (VCF) file (Specification, 2016). gVCF is a derivative of the VCF format containing allelic expressions from both reference and non-reference blocks of the genome.

Once the variants for each sample are obtained, researchers are typically interested in analyzing variants from many samples jointly. Such genome-wide analysis studies (GWAS) could be for investigating the association between variants and specific diseases (George MF, 2016) or for boosting the confidence of called variants by joint genotyping (Li, 2011). All GWAS that deal with a large set of samples face a common set of challenges:

- **Variant data is large and growing**. A typical whole human genome sample contains about three million variants that are roughly a few hundred megabytes in size. However, gVCF files that are used in joint genotyping are an order of magnitude larger (a few gigabytes), as they also store likelihoods. With more individuals being sequenced, this data explodes quickly. A scalable system is required to store variant data from hundreds of thousands or more samples.
- **Scalable and efficient retrievals are needed**. As variant/likelihood data grows, so does the need for computing resources to process them. For example, joint genotyping requires access to data from every sample in a cohort for computing the posterior likelihoods for a given genomic position or interval. Such tasks generate heavy demand on data storage and retrieval systems.
- **Efficient transformations are needed**. While the storage system may use a format optimized for storage and retrieval, tools in GWAS can expect data in specific formats. For example, the GATK joint genotyping tool expects data in the VCF format. Each VCF record contains data from all the samples in the cohort and reorganized fields that are consistent with the order of alleles in the combined VCF record. Efficient design of data structures and mechanisms is required so that these transformations do not prohibitively add overhead in the analysis tools.

Further, some of the common practices widely followed by the bioinformatics community for dealing with variant data from multiple samples and their shortcomings are listed below.

- **Using a scalable file system or ObjectStore**. Using per-sample indexed VCF/gVCF files stored in a scalable file system such as Lustre*, Ceph*, or Hadoop* Distributed File System (HDFS) or object storage system such as Amazon S3 or Google Cloud Storage does not solve the problem. The issue is that querying a given genomic position would open N files/objects (where N is the number of samples), which imposes heavy demands on the system. Additionally, the user may have to combine/transform the data from all samples to feed into the analysis tools.
- **Creating a combined, indexed VCF/gVCF**. While this approach minimizes the number of open files/objects, the VCF format is extremely inefficient for storing data from many samples, primarily because it requires that every record have data (nulls at the minimum) for every sample in the cohort. Creating this combined file is itself a time-consuming, memory-intensive process. In addition, importing new samples to a cohort is equivalent to destroying the existing file and recreating it from scratch.
- **Using a database engine**. Databases are designed to share data across multiple readers efficiently. Frameworks such as Gemini (Umadevi Paila, 2013) and CellBase (Marta Bleda 2012) use database engines that are significantly better than flat files. However, it is unclear whether the database engines selected are efficient for storing and retrieving variant data. For example, MongoDB is a flexible document storage system, but it is unclear whether it is efficient for retrieving variants in a cohort for a given position/interval. Additionally, such frameworks also need to transform the data both for import and to feed downstream analysis tools.

Our approach is to use GenomicsDB, an efficient columnar storage manager for variants. Our key contributions in this work are as follows:

- **A high-performance array storage manager (TileDB)** to store and query variant data from a large number of samples efficiently on disk
- **A fast and efficient C++ library for importing large VCF/gVCF** files from many samples into TileDB
- **A fast and efficient C++ library to extract data from TileDB** and feed into genomic analysis tools such as GATK
- **An interface to Apache Spark*** that allows users to process large datasets in a distributed manner

The following sections discuss these contributions in detail.

Sparse Columnar Arrays

A VCF file is sorted by genome positions; the shaded box in Figure 1 shows two VCF records. The first record signifies that an insertion of an allele TA for chromosome 20 is found at position 17960594. In the reference genome, the site contained only the T nucleotide. The rest of the record contains genotype, quality, and likelihood scores for the read. The second record signifies a deletion at position 17986032, where the allele contains only nucleotide A instead of TA as found in the reference. While the human genome is 3.2 billion characters long, mutations such as these make up only two to five percent of the total length. For this reason, variant data is sparse in nature; it is stored in GenomicsDB using a sparse two dimensional (2D) array data model. Figure 1 includes a schematic representation of the model. Here, rows correspond to a person or sample and columns correspond to genome positions. The two example mutations from above will be stored in columns (17960594 - 1) or 17960594 and (17986032 - 1) or 17986031, respectively. The subtraction is required because positions in a VCF file are numbered starting from 1, but column indices in GenomicsDB are numbered starting from 0. Each cell in the array contains multiple fields to store genotypes, alternate alleles, and metrics such as the quality and likelihood scores as they appear in the example VCF records. Each cell also

contains an attribute called END that signifies where the variant terminates. A typical column-range query of the array reports all the samples whose intervals intersect with the given range. Similarly, for queries on samples (shown in the shaded box in Figure 1), intervals from required positions are returned by reading the corresponding row of the array.

```
chr20    17960594    .    T    TA,<NON_REF>    444.73    .
BaseQRankSum=0.489;ClippingRankSum=-
0.900;DP=26;ExcessHet=3.0103;MLEAC=1,0;MLEAF=0.500,0.00;MQRankSum=0.437;RAW_
MQ=93600.00;ReadPosRankSum=-1.003    GT:AD:DP:GQ:PGT:PID:PL:SB
0/1:14,12,0:26:99:0|1:17960576_G_A:482,0,798,530,837,1368:8,6,4,8

chr20    17986032    .    AT    A,<NON_REF>    480.73    .
BaseQRankSum=1.045;ClippingRankSum=0.920;DP=55;ExcessHet=3.0103;MLEAC=1,0;MLE
AF=0.500,0.00;MQRankSum=-0.545;RAW_MQ=198000.00;ReadPosRankSum=0.009
GT:AD:DP:GQ:PL:SB    0/1:29,24,0:53:99:518,0,657,605,729,1334:16,13,10,14
```

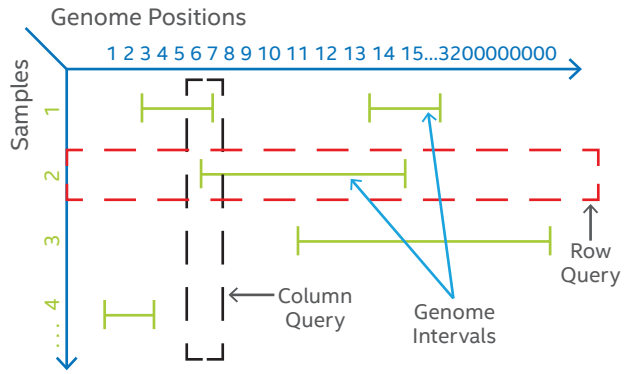

Figure 1. Variant data as a sparse 2D array.

To store sparse arrays, we use TileDB—a columnar storage manager specifically designed for sparse multi-dimensional arrays (Papadopoulos, 2016). TileDB lets users define the data type and order of the cells of an array. Our reasoning for using TileDB includes the following factors.

- **Variant data is sparse**. TileDB is faster than HDF5 or SciDB (other array stores) or relational SQL databases such as Vertica* for sparse data. It can also read and write in parallel using message passing interface (MPI and MPI-IO) libraries in C++.
- **Variant data can be stored as a 2D array**. This approach uses rows for samples and columns for genomic positions. Users can retrieve data by querying sub-regions (or subarrays) by providing ranges in the global address space on the dimensions of the array.
- **TileDB uses columnar mechanisms**. It stores each attribute of the array in a different file so that only the files corresponding to the queried fields are traversed. This architecture reduces the number of disk accesses relative to row-ordered storage systems (such as PostgreSQL*).
- **Users can specify the store order of the cells in the array**. For example, they may choose to use store orders such as row-ordered, column-ordered, or Hilbert-ordered. The cell order is followed to store cells sequentially on disk. GenomicsDB stores cells in column-major order. Our expectation is that the most common type of query will be to retrieve variant data for a cohort given a genomic position/interval. Storing cells in column-major order enables TileDB to quickly retrieve contiguous blocks of data from disk.
- **TileDB enables efficient storage and retrieval**. TileDB stores and retrieves from the storage disk contiguous sequences of cells of an array in units called tiles. In a tile, the cells appear in the cell ordering mentioned above. The number of cells per tile is a configurable parameter independent of the number of samples in the cohort. Each tile is compressed before storing, and disk offsets for all tiles are tracked for fast retrieval during queries. Users can tweak the number of cells per tile to match common query patterns. For example, a user who wishes to perform a single query over a large genomic interval might find that using a high cell count gives best performance. A different use case would be a system that receives a large number of single genomic position queries—each query for a different position of the genome. Using smaller tiles would also prevent the system from running out of memory. Note that using compression routines optimized for Intel architecture in TileDB provides approximately 20 percent additional improvement in write times compared to the un-optimized library. Detailed experiments and results are presented in the Evaluation section.

GenomicsDB Architecture and Interfaces

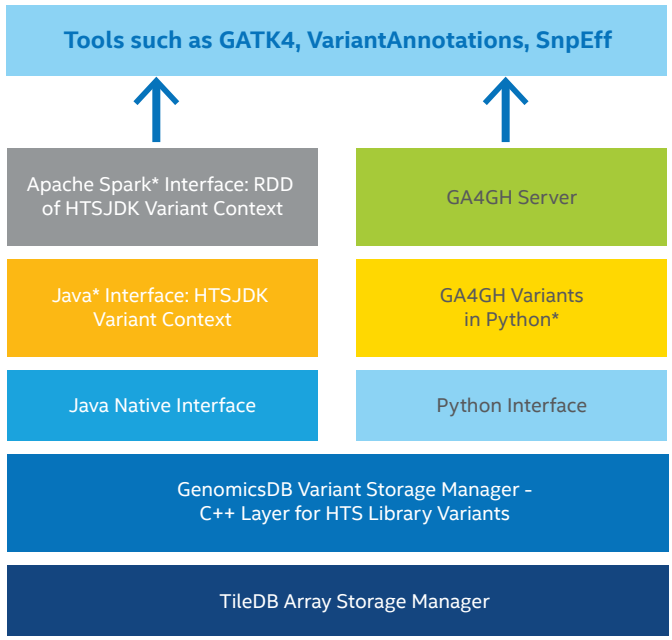

Figure 2. The different layers in GenomicsDB software stack.

Our key design goal is to achieve scalability. In a single node, GenomicsDB can use concurrent processes and utilize multiple processor cores. In a distributed environment, we scale by running multiple processes over multiple nodes in a cluster. Figure 2 shows the logical building blocks of GenomicsDB. TileDB is the core library, which stores the two dimensional genome array to disk. GenomicsDB encapsulates the TileDB arrays and provides read and write methods for genomics data structures. TileDB and GenomicsDB core layers are written in C++ (shown as the bottom two layers in Figure 2). To facilitate reading and writing variants from genome analysis tools such as GATK, we provide interfaces in Java, Scala*, and Python. These interfaces can be used to extract or import data from downstream genome analysis tools such as VariantAnnotation (Valerie Obenchain, 2014) or SnpEff (Cingolani, 2012), among others. The Java interface provides feature reader objects (shown in orange). It connects to the underlying C++ core by means of the Java Native Interface. Bookkeeping structures containing input configuration values and file names are passed as protocol buffer objects from Java to C++. Both Java and C++ layers expose two critical methods: import and query. The import method is invoked with a list of VCF or gVCF files and writes the VCF records to disk. The query is invoked with a list of genomic intervals to query and returns data formatted as JSON variant records or as multi-sample combined VCF records, or as VariantContext objects in Java. A VariantContext object is a Java object that contains variant data combined from all samples in the queried cohort at the given genomic position/interval. The Scala API can be used from Apache Spark for distributed computing when dealing with a large amount of data, as described in Appendix A. The Python API is aligned with the Global Alliance for Genomics and Health (GA4GH, 2017) standardized interface for genomics. Discussion of the GA4GH interface is beyond the scope of this document; it is covered in detail in the GenomicsDB user manual.

Importing Variants to GenomicsDB

A VCF file can contain data from one or multiple samples. The core GenomicsDB import method as shown in Figure 3 first parses VCF records from the input VCF files into intermediate buffers. One or more threads can read VCF files in parallel, shown as blue VCF reader circles in the figure. These readers pass the VCF records to writer processes by filling entries in the TileDB buffer. The TileDB buffer is directly passed to the TileDB write interface, which then serializes the data as array cells to disk. Users can control the number of parallel readers and sizes of intermediate and TileDB buffers via input configuration variables. Importing can also be done in batches. For example, a cohort of 1,000 samples can be imported in two batches containing 500 samples each.

The core import method can be used in a SPMD fashion to parallelize writes across multiple GenomicsDB partitions. A separate process is used to write a GenomicsDB partition, as shown in the figure. The user needs to specify the number of processes and the column ranges to be written to each partition. For example, four SPMD processes can be used to write data from positions 0-200M, 200M-400M, 400M-1.5B, and 1.5B-3.2B to four different GenomicsDB partitions in parallel. The processes can run on independent machines based on the host configuration of the multi-process library used (e.g., MPICH2 or Open MPI). Internally, GenomicsDB creates unique row indices for each sample. Row indices should always be coherent across all partitions. One-based genome positions are mapped to zero-based column indices of TileDB.

Tile size is a critical control variable in GenomicsDB import. This determines how many contiguous bytes TileDB will read from disk at one time. It also determines the unit of compression—the number of bytes compressed or decompressed together during write or read in TileDB. Too-small tiles can cause small compression blocks, reducing

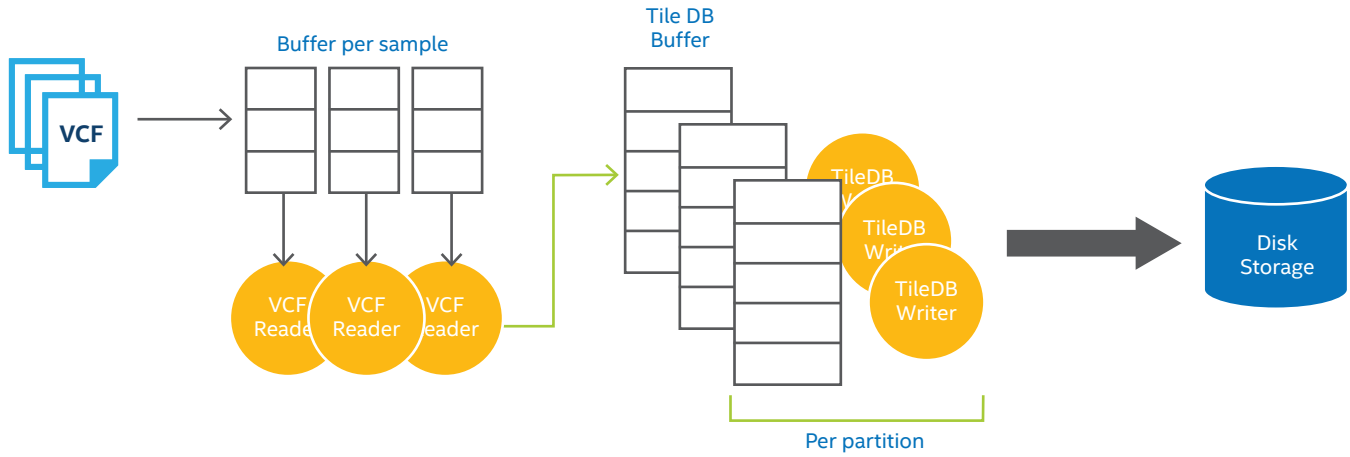

Figure 3. Data flow in GenomicsDB import.

the effectiveness of the compression algorithms, as well as excessive disk accesses even for sequential reads. On the other hand, large tiles can cause large disk reads even for smaller random reads. The optimal tile size depends on the use case, query types, and system configuration; we recommend that users perform some combinatorial experiments to determine the optimal size for their use case. In the case of the joint genotyping workflow in GATK, our evaluations suggest that a few kilobytes is optimal.

Batching is a powerful technique in GenomicsDB enabled by the fragmentation mechanism in TileDB. It allows users to import data in smaller numbers iteratively when the number of samples grows very large (i.e., a million or more samples), or when new samples are added to an existing cohort a few months after the initial array is created. For a TileDB array, each batch is written to a new fragment. This architecture makes adding new batches fast; update depends only on the new batch size and not the pre-existing data in the array. However, during query, the read algorithm traverses all fragments and provides data in column-major order. As the number of fragments increases, the query performance drops. Hence, users must run a consolidate operation at intervals to squash all fragments into one. Use of consolidation can be controlled via input configuration. If enabled, it is called at the end, after all fragments are written.

Partitioning in GenomicsDB helps us to achieve scalability. GenomicsDB stores variant data in multiple partitions or shards which may be stored in different machines. Currently, GenomicsDB supports two partitioning strategies:

- **Row partitioning**, where each partition contains a subset of rows (or samples) but contains the data from all genomic positions for the subset of samples
- **Column partitioning**, where each partition contains a subset of contiguous genomic positions but contains data from all samples for the subset of columns

Column partitioning is the more widely used of these two strategies, because it allows cohort analysis for a given genomic interval/position to read all data off a single partition without having to gather data from multiple partitions that may be scattered across multiple machines. Each partition of GenomicsDB is stored as a distinct TileDB array.

Mapping Data is maintained in a relational database that stores the relationships between string sample names and 64-bit TileDB rows. Relational systems include strong consistency mechanisms, allowing us to keep the mappings consistent across multiple concurrent writers, avoiding implementation of these mechanisms ourselves. The mappings include the following:

- The type of the various INFO, FORMAT and FILTER fields from the VCF header
- A combination function (median, concatenation or mean) used while creating a combined VCF from all samples
- The mapping between the range of 1-based genome positions of the chromosomes to the corresponding 0-based TileDB column positions

Querying GenomicsDB

The query algorithm takes as input a list of column and row intervals representing the samples in the cohort and the genomic positions of interest. Each interval can be processed in parallel. For each column interval, we first determine the intersecting variant calls, as shown in Figure 1. However, there can be partial matches, shown more precisely in Figure 4. Scenario (4) is the simplest, where the allele intersects both right and left boundaries of the query column interval. We return the intersecting portion. In the cases of both scenario (1) and scenario (2), a left sweep from left boundary of column interval is required to find the partial intersections. In scenario (3), a right sweep from right boundary is required. This implies that for each intersection of alleles, the query method finds both end and start positions. The end position is the END value specified in the gVCF format.

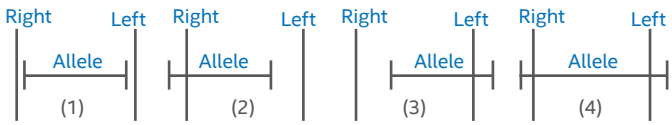

Figure 4. Different scenarios of allele intersections with column intervals. Longer bars signify the right and left boundaries of the query column interval.

Note that the tile size has high impact on the read performance, as explained in the "Importing Variants to GenomicsDB" section of this document. More data is presented in the Evaluation section, below. Interfaces to GenomicsDB query are provided in C++, Java, and Python. Users can print the query result to flat output files in JSON format or in the format of a combined VCF, or they can perform operations on that result in memory. The high-throughput query interface for Apache Spark is described in Appendix A.

Evaluation

The objective of our evaluation is to demonstrate our methodology to optimize input configuration for joint genotyping with GenomicsDB and to show how the performance of import and query operations scales with the number of samples. We have chosen 1,000 whole exome sequences from the 1,000 Genome Project (Resource, 2008-2016). To perform scale experiments, we divide the human genome into 16 position ranges. The approach is to balance bytes written per partition, using the methodology that is described in detail in Appendix B. The run time of import or query is done on 1/16th of the positions unless otherwise identified. Thus, variants from 1/16th of the human genome from all samples are either written or read in the experiment.

Our experimental setup includes a cluster of four independent dual-socket servers, each based on two Intel® Xeon® processors E5-2699 v4 @ 2.20GHz. Each server has one rotational storage device model number WDC WD30EZRX-00DC0B0 and one Intel® Solid State Drive (SSD) connected by means of an Intel® C610/X99 series chipset SATA controller. We use CentOS* 7.0 as the operating system, Intel® OTC zlib version 1.2.8 (Adler, 2013) and TileDB version 0.3.0. In all these experiments, compression with standard zlib is enabled unless otherwise identified.

Table 1. Import configuration parameters.

TYPE	PARAMETER NAME	VALUES	DESCRIPTION
Import Configuration	num_parallel_vcf_files	1:N::1	Number of concurrent VCF files readers
	compress_tiledb_array	True or False	Enables compression
	segment_size	1:N::1KB	Buffer size to store TileDB cells in a columnar fashion; this buffer is used to both compress and serialize bits from memory to storage via TileDB library
	size_per_column_partition	1:N::1B	Buffer size to store VCF records
	num_cells_per_tile	1:N::1KB	Number of array cells per tile in TileDB
	Partitions	1:1::1	Number of partitions based on genome positions or TileDB columns
Export Configuration	segment_size	1:N::1KB	Buffer size to read TileDB cells from storage to memory

The configuration parameters are listed in Table 1. The variables are grouped into import and export parameters. The second column shows the actual parameter names in GenomicsDB, and the third column shows the possible ranges of their values and units of increment (separated by double colon), with units B and KB indicating bytes and kilobytes, respectively. The third column provides a short description of each parameter.

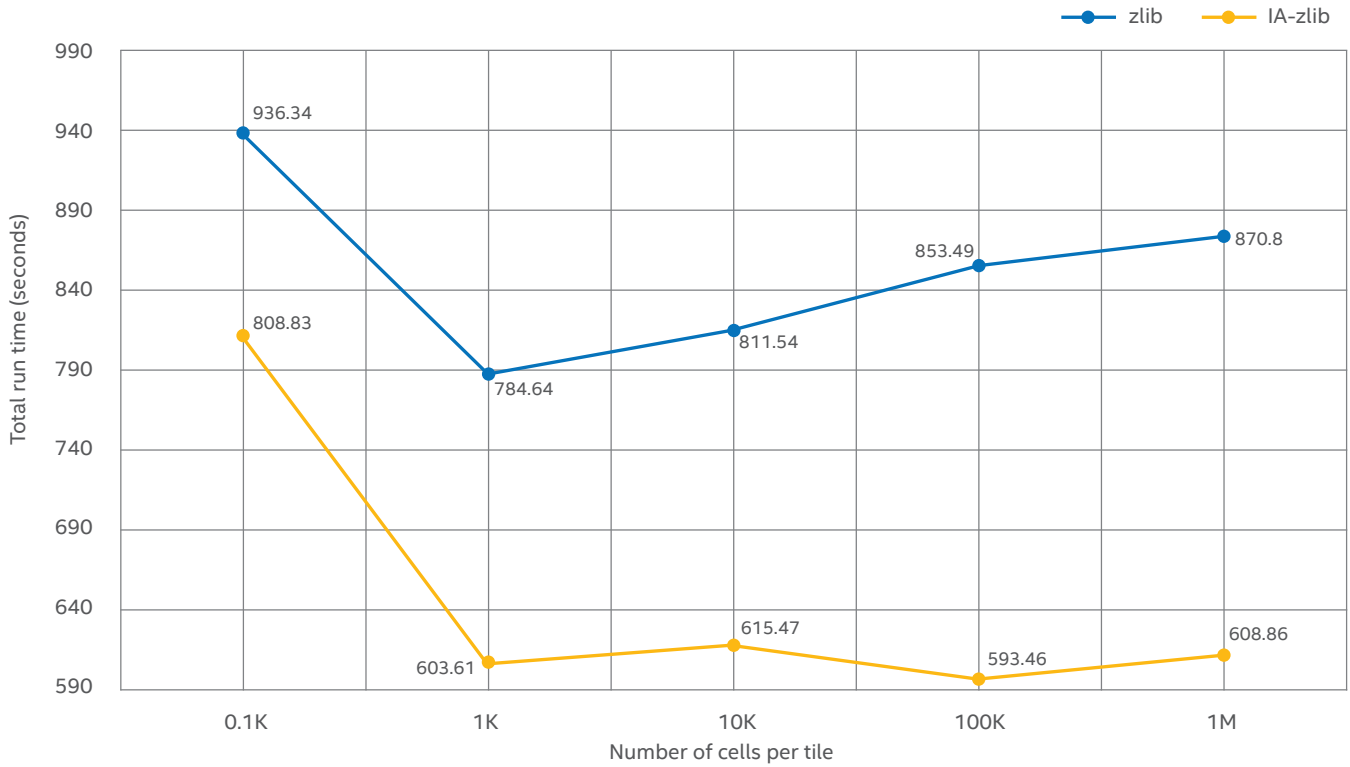

Figure 5. Total time to import 1/16th of 1,000 samples with different cells per tile.

Number of cells per tile determines the tile size in TileDB. It represents the cells of the array read sequentially from disk, not the bytes read from disk. The tile size also determines the unit of compression. Therefore, too few cells in a tile will mean fewer compressed bytes and frequent disk reads/writes. Values that are too large mean writes will compress larger buffers at a time, but reads will fetch larger chunks from disk, which might prohibitively slow down random small queries. Figure 5 demonstrates this behavior. The difference between 1 KB and 100 KB is minimal, and values in this range result in the best performance. Any values less than 1 KB or larger than 100 KB can prohibitively slow down writes. The orange line marked as IA-zlib shows the same completion time with compression optimized for Intel architecture. On average, IA-zlib yields a 24 percent improvement. The TileDB paper showed that using larger numbers of cells per tile always improved performance in synthetic benchmarks; however, when real data is read from input VCF files, we see a V-shaped curve, with 1,000 cells per tile giving the best performance. The reason behind this performance difference will be investigated.

Number of parallel readers enables concurrent readers to read input files and populate intermediate buffers before they can be written to TileDB. Even with 1,000 input VCF files, increasing this parameter has minimal effect on total import time, because the bottleneck arises in later stages during compression.

Segment size in bytes sets the raw buffer size that TileDB uses to serialize cells to disk. When this buffer is full, compression kicks in. It appears that when varying this size from 10 KB to 10 MB, the difference is less than five percent, as shown in Figure 6. Therefore, we recommend using a few megabytes and the default value of 10 MB for the rest of the experiments. The total run times for different segment sizes are shown in Figure 6.

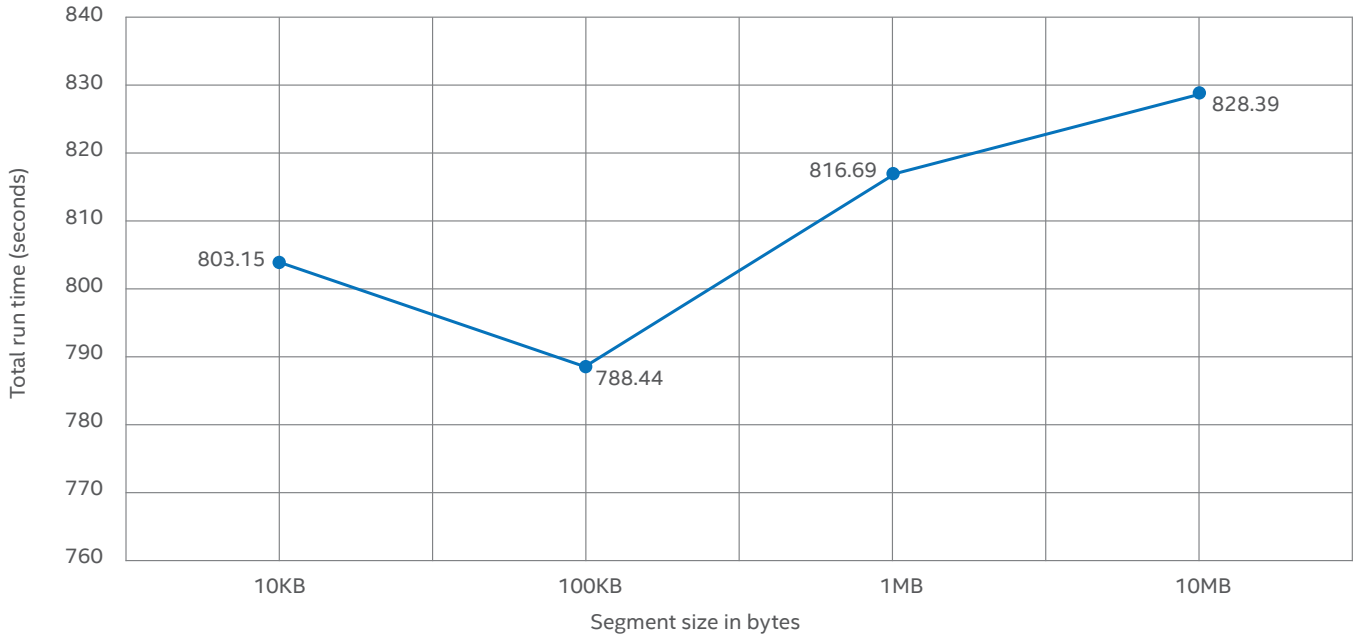

Figure 6. Run time as found with varying segment size.

Compression level also determines the trade-off between compression ratio and time. Level 0 means no compression and takes the least time, whereas 9 means maximum compression with maximum time requirements. Experimentation with levels 0, 1, 6 (the default), and 9 revealed that using level 6 provides a suitable relationship between compression ratio and time required. Comparatively, level 9 consumes significantly more processor resources without increasing the compression ratio. Total write time for all 16 partitions and raw size on disk is shown in Figure 7. The compression ratio with IA-zlib is approximately 10.0 for all configurations (i.e. 1/16, 2/16, 4/16, 8/16 and 16/16) with level 6.

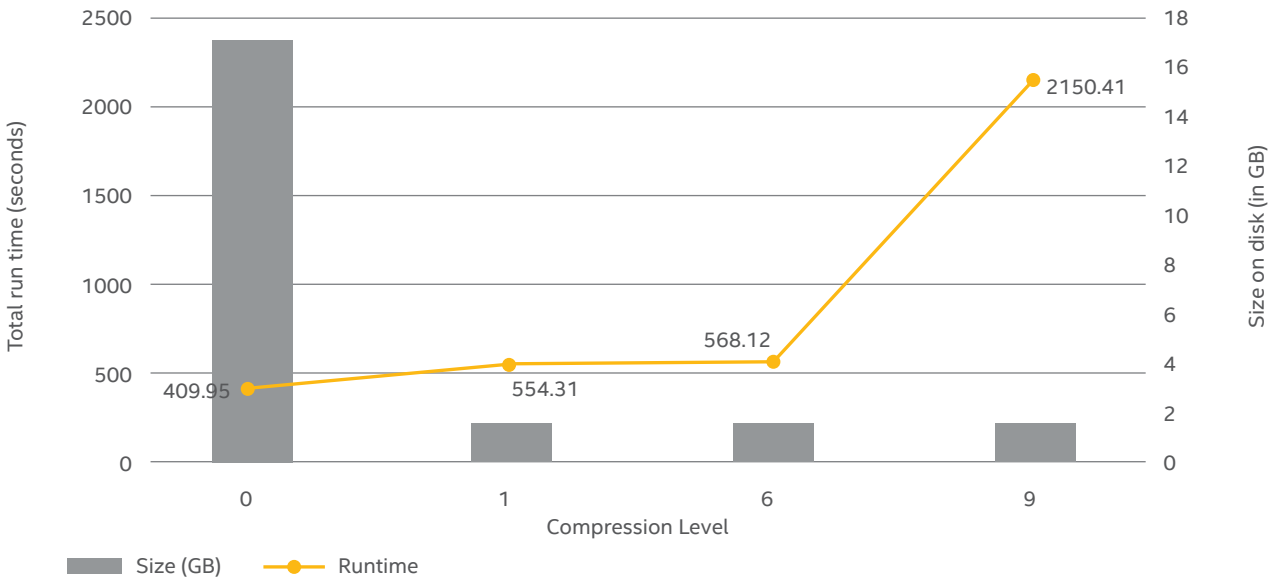

Figure 7. Run time and disk usage for different compression levels with all 16 partitions.

Numbers of partitions are scaled in Figure 8. The data is partitioned 16 ways, following the histogram shown in Appendix B. Between each experiment, data is doubled as indicated on the X-axis. The increasing number of partitions on the X-axis also signifies the number of parallel processes being doubled at every step. While in an ideal system, the total time for each step should be the same, this is not the case in a multi-core environment with shared hardware and software resources. Contention for resources such as memory bandwidth and last-level cache adds overhead, leading to the time increase shown at each step. Using standard zlib compression, the time to import all 16 partitions is 1.5x greater than the time to import one partition. In other words, 16x data was imported by scaling to 16 simultaneous processes with a 50 percent increase in time, which demonstrates higher scalability in the import process. The data for the same scaling operation using compression optimized for Intel architecture instead of standard zlib shows that while the optimized compression offers similar scalability to standard zlib, the optimized compression is approximately 20 percent faster across all runs. The compression level was set to 6 across all runs. In addition, each data point was taken three times, as shown by the high/low lines in the figure.

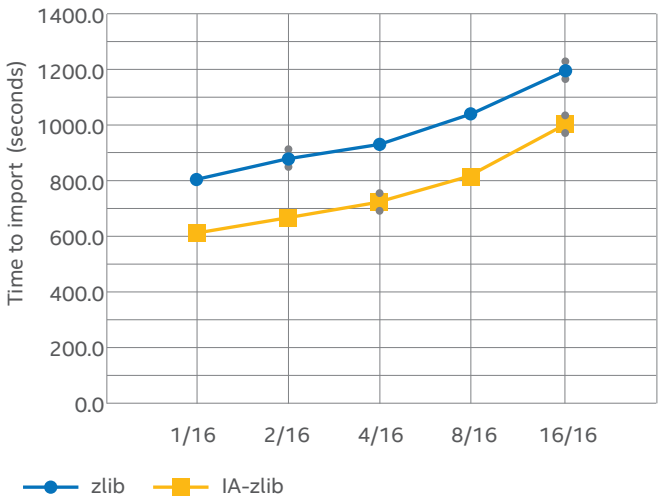

Figure 8. Comparison of import times of 1, 2, 4, 8, and 16 partitions of human genome from 1,000 samples with standard zlib and compression libraries optimized for Intel® architecture.

Reading data from GenomicsDB is critical to its success, as it affects the performance of downstream genome analysis tools. In this work, we only measure the wall-clock time of bulk sequential reads, because that is the method used by joint genotyping.

Number of cells per tile during query is shown in Figure 9. It shows that for large reads, a large number of cells per tile is conducive, and beyond 10 KB, the run times saturate. The import times with tile sizes show similar trends. We conclude from these relationships that values between 1 KB and 10 KB are optimal tile sizes when reading data from GenomicsDB during joint genotyping.

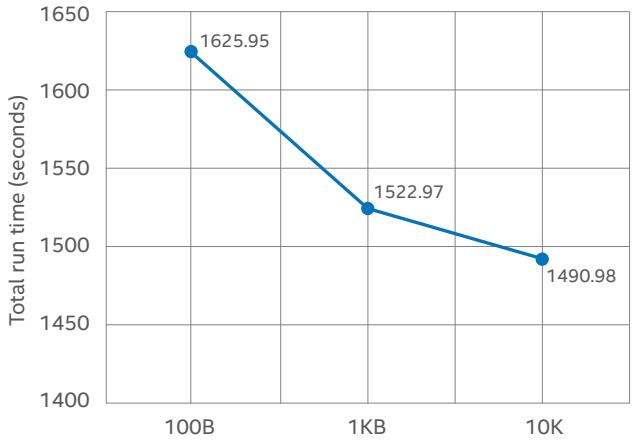

Figure 9. Read time with increasing cells per tile.

The number of partitions is doubled to show scaling of query in Figure 10. The figure shows that processor cores are efficiently utilized by the read algorithm, and it scales effectively. With each successive blue column, the data size is doubled; the corresponding orange curve shows that utilization of processor cores doubles as well. This means that that query time increases sub-linearly as data size is doubled in every run. Note that we double the number of parallel processes as the data is doubled at every step. Also, decompression consumes a very small fraction of total CPU time during queries. As a result, we see little difference between decompression times using the compression library optimized for Intel architecture compared with standard zlib.

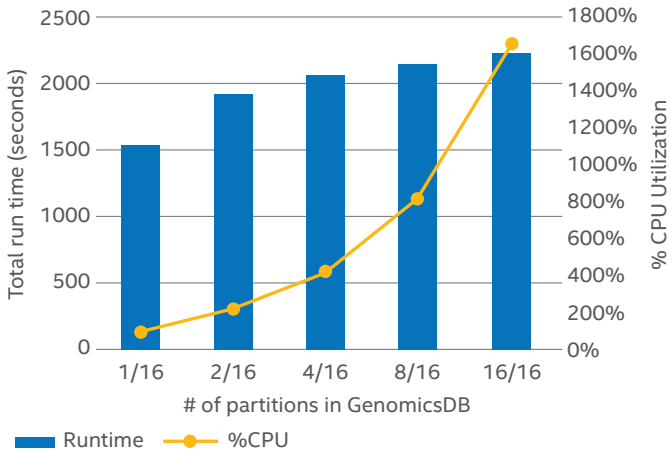

Figure 10. Read time with increasing number of partitions.

The number of samples is increased, and both import and query times are measured in Figure 11. The data is generated by duplicating variants from the 1,000 genome project. This synthetic data enables us to showcase the scalability of GenomicsDB operations up to tens of thousands of samples. We acknowledge that there can be exceptions with real life datasets that are not captured by simply duplicating data. Note that 1/16th of the human genome is imported and read here. The column ranges are given in Appendix B. The figure shows that by scaling the number of samples from 5K to 40K, the import time increased nearly linearly by 8.4x. As data increases, the compression algorithms are stressed to a greater degree. As a result, up to 33 percent of the total import time is spent in compression, creating the upward trend shown in the figure. To the best of our knowledge, however, no other data-store system matches this scalability. The read times between 5K and 40K samples increase by 13.23x. As more and more data is read into memory, we observe a remarkably high number of L3 cache misses, which saturate the memory bandwidth. This is the reason the read times increase significantly with read size. In addition, the memory used by both write and read scales linearly with the number of samples, as shown in Figure 12. Note that in this experiment, only one process is used per write and read.

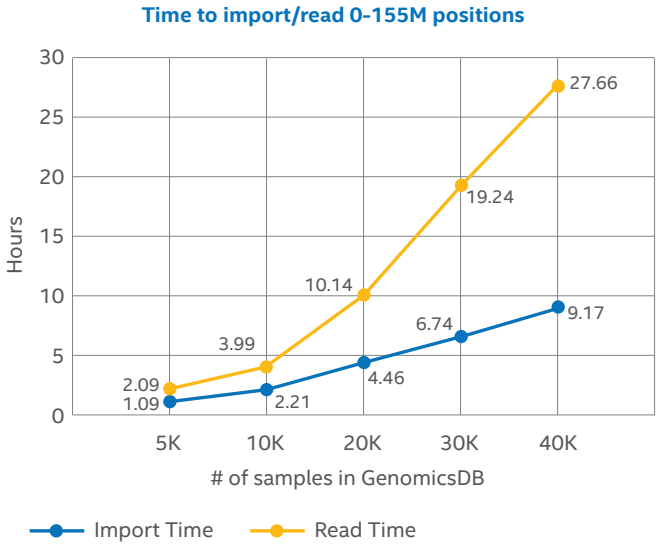

Figure 11. Scalability of GenomicsDB import and query with number of samples.

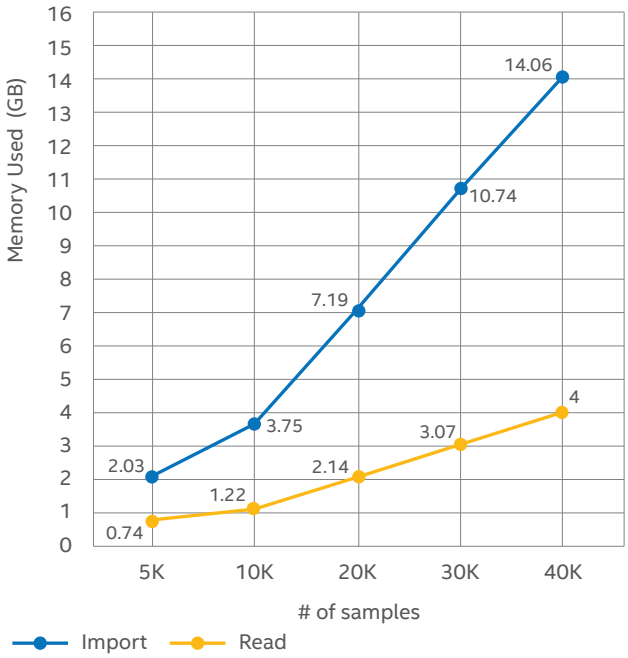

Figure 12. Memory used in a single node while importing and reading 0-155M positions from GenomicsDB for 5K, 10K, 20K, 30K, and 40K samples.

Discussion

In this paper, we have demonstrated the use of GenomicsDB as a data store for genomics likelihoods and variants. Our evaluations show how both import and query mechanisms in GenomicsDB scale with the number of samples using multiple processes. We also show how a compression library optimized for Intel architecture achieves a 20 percent improvement in write performance compared to standard zlib. With this technology, we solve the scaling problem of joint genotyping workflows in GATK with 100,000 samples and beyond. We envision that GenomicsDB can also be efficiently applied to store other genomics data structures, such as structural variants and reads in both raw (FASTQ) format and aligned (BAM) format. While these structures are serialized using text formats, they are processed as arrays. Therefore, we can reduce the cost of repeated serialization and deserialization. From the standpoint of our distributed interface, we envision a service-oriented model of GenomicsDB that provides interfaces to import, query, and move data between peer nodes dynamically and concurrently with high efficiency. We hope to address these issues in the next generation of this technology.

Acknowledgements

This work is a joint collaboration between the Broad Institute in Cambridge, Massachusetts, Intel® Health and Life Sciences, and Intel® Labs, in conjunction with the Intel® Science and Technology Center for Big Data at the Massachusetts Institute of Technology.

References

Adler, J.-l. G. (2013). *Intel Genome Kernel Library*. Retrieved from OTC Zlib Compression Library: https://github.com/Intel-HLS/GKL/tree/master/src/main/native/compression/otc_zlib.

Ashley, E. A. (2016). Towards precision medicine. *Nat Rev Genet*, 507-522.

Cingolani, P. a. (2012). A program for annotating and predicting the effects of single nucleotide polymorphisms, SnpEff: SNPs in the genome of Drosophila melanogaster strain w1118; iso-2; iso-3. *Fly*, 80-92.

CoreGenomics. (2016, 05). *How many genomes can the world sequence per year on X Ten?* Retrieved from core-genomics.blog-spot.com: http://core-genomics.blogspot.com/2016/05/how-many-genomes-can-world-sequence-per.html.

DePristo, M. A. (2011). A framework for variation discovery and genotyping using next-generation DNA sequencing data. *Nature Genetics* 43(5), 491–498.

GA4GH. (2017). *GA4GH Reference Implementation*. Retrieved from GA4GH-Server API: https://github.com/ga4gh/ga4gh-server.

Li, H. (2011). A statistical framework for SNP calling, mutation discovery, association mapping and population genetical parameter estimation from sequencing data. *Bioinformatics*.

McKenna A, H. M. (2010). The Genome Analysis Toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data. *Genome Research*, 1297–1303.

Papadopoulos, S. a. (2016). The TileDB Array Data Storage Manager. *Proc. VLDB Endow.*, 349-360.

Raheleh Rahbari, A. W. (2016). Timing, rates and spectra of human germline mutation. *Nature*, 126-133.

Resource, I. G. (2008-2016). *Using data from IGSR*. Retrieved from IGSR: The International Genome Sample Resource: http://www.internationalgenome.org/data/.

Specification, V. (2016, November 15). *Samtools Github*. Retrieved from VCF4.2: https://samtools.github.io/hts-specs/VCFv4.2.pdf.

Umadevi Paila, B. A. (2013). GEMINI: Integrative Exploration of Genetic Variation and Genome Annotations. *PLOS Computational Biology*.

Valerie Obenchain, M. M. (2014). VariantAnnotation: a Bioconductor package for exploration and annotation of genetic variants. *Bioinformatics*, 2076-2078.

Van der Auwera, G. A.-M. (2013). From FastQ data to high confidence variant calls: the Genome Analysis Toolkit best practices pipeline. *Current Protocols in Bioinformatics*, 11.10.1–11.10.33.

WATSON, J. D. (1953). Molecular Structure of Nucleic Acids: A Structure for Deoxyribose Nucleic Acid. *Nature*, 737-743.

Yuan Yuan, E. M.-M. (2014). Assessing the clinical utility of cancer genomic and proteomic data across tumor types. *Nature Biotech*, 644-652.

Zaharia, M. a. (2012). Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (pp. 2-2). Berkeley, CA: USENIX Association.

George MF et al (2016). Multiple sclerosis risk loci and disease severity in 7,125 individuals from 10 studies. *Neurol Genet*.

Marta Bleda et al (2012). CellBase, a comprehensive collection of RESTful web services for retrieving relevant biological information from heterogeneous sources. *Nucleic Acids Res*.

Appendix A – Apache Spark* Interface

Apache Spark is a distributed runtime environment based on a resilient distributed data structure (RDD) partitioned across multiple nodes (Zaharia, 2012). The embarrassingly parallel operators exposed by Spark such as map filter flatMap can occur in parallel in the nodes, whereas the combine operators such as join groupBy reduceByKey require data to be shuffled across the partitions over the network. The mechanism of the Spark interface with GenomicsDB is shown in Figure 13. The approach is based on the idea that shared-nothing GenomicsDB instances (separated by the green line in the figure) will contain data for the column ranges the instance was partitioned with. Because this is a columnar partitioning scheme, data from all samples appear in all nodes. One Spark worker is spawned for each GenomicsDB instance. Each worker reads data from its local instance, creating an RDD of VariantContexts. Similar to Spark runtime, all operators local to an instance occur locally on the GenomicsDB data, whereas any shuffle required during a combine operation uses Spark's communication layer.

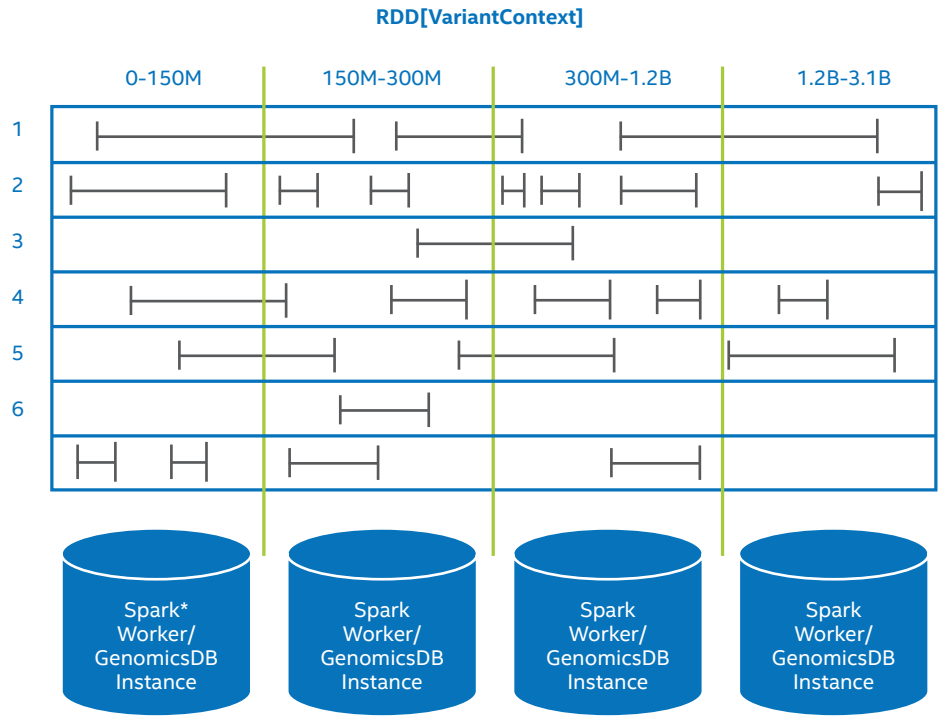

Figure 13. How the Apache Spark* interface of GenomicsDB works on partitioned data.

Appendix B – GenomicsDB Utilities

In addition to import and query, the utility tools vcfdiff and vcfhistogram in GenomicsDB provide two important capabilities. The first of these, vcfdiff, compares two VCF files, record by record. It is useful to validate combined VCF files generated from other genome analyzer tools against the ones generated by GenomicsDB. It is also used in the continuous integration and nightly build environment. The tool takes in tolerance level as an argument, which defines the threshold for floating-point comparisons. The second tool, vcfhistogram, enables us to balance the VCF records imported across multiple import processes. The distribution of alleles occurring in individuals is not uniform. Thus, when import is parallelized, it is difficult to find the column ranges such that the each writes an approximately equal number of VCF records to GenomicsDB. For example, dividing a VCF file into two processes loading 0-to-1.6B and 1.6B-to-3.2B respectively will not write the same number of bytes to GenomicsDB. This non-uniform distribution creates an imbalance in computation, creating straggler processes that significantly outrun other concurrent processes, as shown in the left chart (in blue) of Figure 14. To avoid such a scenario, vcfhistogram can be used to find out the ranges of genomic positions with uniform distribution of VCF records as shown on the right side (in orange) of the figure.

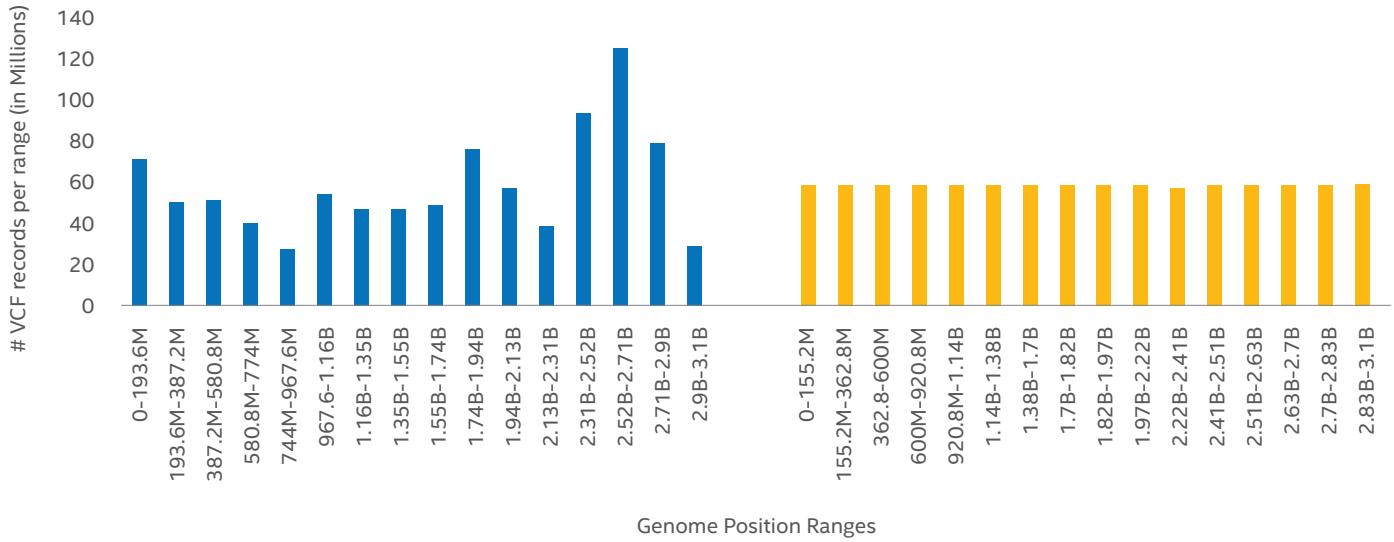

Figure 14. Distribution of VCF records per interval.

Appendix C – Solving the N+1 problem

Table 2 highlights a key benefit that a storage system such as TileDB provides over the existing methodology of performing joint genotyping with a multi-sample combined VCF file. With every update in a cohort, such as the addition or removal of individuals, analysts would have to recreate the combined VCF file. As we have emphasized before, this is an expensive operation. This is commonly referred to as the N+1 problem. In comparison, GenomicsDB handles updates gracefully by adding another TileDB fragment. The organization and method of operation of fragments are discussed in more detail in The TileDB Array Data Storage Manager (Papadopoulos, 2016). The table demonstrates the behavior quantitatively. To add the whole genome sequence of an individual or sample to an existing cohort of 1,000 samples takes a few seconds. This number does not depend on the pre-existing cohort size. Therefore, insertion time of genome from an individual will still take a few seconds, irrespective of whether the existing cohort contains 1K, 10K, or 100K samples. The insertion time also scales linearly with the number of new samples being added. Eliminating the need for creation of combined VCF files implies that a new batch can be imported rapidly and that downstream analysis tools (such as joint genotyping) can begin processing immediately.

Table 2. How GenomicsDB solves the N+1 problem.

# OF SAMPLES WRITTEN INCREMENTALLY TO GENOMICSDB	TIME TO IMPORT SAMPLES (SECONDS)	TIME TO READ ALL DATA (SECONDS)	SIZE ON DISK
1,000	819	1355	1.6 GB
1,000+1	2	1356	1.6 GB+3.7 MB
1,000+10	8.94	1411	1.6 GB+17 MB
1,000+100	78.13	1562	1.6 GB+158 MB
1,000+1,000	788	3089	1.6 GB+1.6 GB

0717/ML/MESH/PDF 336113-001US