GenomicsDB

Scott Hill

May 10, 2024

GENOMICSDB DOCS

1 Overview **1**
- 1.1 Concepts + Terminology . . . 1

2 Supported Storage **3**
- 2.1 Local / Cluster . . . 3
- 2.2 Cloud . . . 3

3 Building + Installing **5**
- 3.1 Bash . . . 5
- 3.2 Docker . . . 5
- 3.3 CLI Tools . . . 6

4 Import / ETL **7**
- 4.1 VCF/gVCF . . . 7
  - 4.1.1 Organizing your data . . . 7
- 4.2 CSV . . . 8
  - 4.2.1 Preliminaries . . . 8
  - 4.2.2 CSV format . . . 8
  - 4.2.3 Organizing your data . . . 9
- 4.3 Multi-node setup . . . 9

5 Common Issues **11**

6 APIs **13**
- 6.1 Overview . . . 13
  - 6.1.1 GenomicsDB Protobuf Documentation . . . 13
- 6.2 C++ . . . 22
  - 6.2.1 API Reference . . . 22
- 6.3 Python . . . 25
  - 6.3.1 API Reference . . . 25
- 6.4 Java . . . 25
  - 6.4.1 API Reference . . . 25

7 CLI Tools **33**
- 7.1 vcf2genomicsdb_init . . . 33
- 7.2 vcf2genomicsdb . . . 33
- 7.3 gt_mpi_gather . . . 33
- 7.4 create_genomicsdb_workspace . . . 35
- 7.5 consolidate_genomicsdb_array . . . 36
- 7.6 vcf_histogram . . . 36
- 7.7 vcfdiff . . . 36

8 Example Python notebook connecting to GenomicsDB on Azure Blob storage 37

9 Import / ETL Examples 41

10 Using GenomicsDB with GATK 43

Index 45

CHAPTER

ONE

OVERVIEW

GenomicsDB is a highly performant scalable data storage written in C++ for importing, querying and transforming genomic variant data.

1.1 Concepts + Terminology

This section outlines basic GenomicsDB concepts and terminology used throughout this documentation.

- *Cell*: Cell of an array (for example: A[i][j] in a 2-D array)
- *Workspace*: A directory on the file system under which multiple GenomicsDB arrays can be stored. The underlying array stores some metadata files under the workspace directory. All GenomicsDB import/create programs assume that if the specified workspace directory exists, then it's a valid workspace directory. If the user points to an existing directory which doesn't contain the workspace metadata, the create/import programs will exit with an exception. If the directory doesn't exist, then the programs will initialize a new workspace correctly.
- *Array*: Name of a GenomicsDB array
- Given a workspace and array name, the framework will store its data in the directory <*workspace*>/<*array*>.
- *Column-major* and *row-major* ordering: Denotes the order in which cells are stored on disk.
  - Column major storage implies that cells belonging to the same column are stored contiguously on disk (cells belonging to the same column but different rows are sorted by row id). When cells are stored in column major order queries such as "retrieve all cells belonging to genomic position *X*" are fast due to high spatial locality. However, queries such as "retrieve cells for sample *Y*" are relatively slow. For GenomicsDB, we expect the former type of queries to be more frequent and hence, by default all arrays are stored in column major order (even when partitioning by rows across GenomicsDB instances).
  - Row major ordering implies that cells belonging to the same row are stored contiguously on disk.
- *Bulk importing and incremental importing*: *Bulk importing* of data implies that all the data is loaded at once into GenomicsDB and the array is never modified later on. *Incremental importing* implies that the array can be modified by adding new samples/CallSet over time. GenomicsDB support both modes of operation - however, repeated incremental updates (for example, incrementally adding one sample/CallSet at a time, ~1000 times) may lead to reduced performance during querying. If you are interested in why this occurs, please read the section on writing arrays to understand the concept of fragments and how updates are performed. Ideally, users should perform incremental imports a few times (~10s of times) per array and each import should contain a large number of samples.

CHAPTER

TWO

SUPPORTED STORAGE

GenomicsDB has support for several filesystems.

2.1 Local / Cluster

- *POSIX*: Tested on Centos 6/7 and Ubuntu 20.04

2.2 Cloud

- *AWS*: Support for S3 and EMRFS
- *GCP*: Support for GCP
- *Azure*: Support for Azure Blob storage and Azure Data Lake Storage Gen2

CHAPTER

THREE

BUILDING + INSTALLING

This section refers to building and installing the core GenomicsDB executables and tools. For including published artifacts in your code, see the appropriate *API docs* and Examples sections.

3.1 Bash

The script *install_prereqs.sh* is meant to work with Docker, but should work on any Linux distribution from a bash shell. Note that *install_prereqs.sh* requires sudo-able access.

```
# Clone GenomicsDB and change to scripts directory
git clone https://github.com/GenomicsDB/GenomicsDB && cd GenomicsDB/scripts/

# Install the prerequisites to build GenomicsDB and create a genomicsdb_prereqs.sh in
↪$HOME
sudo prereqs/install_prereqs.sh
source $HOME/genomicsdb_prereqs.sh
cmake -DCMAKE_INSTALL_PREFIX=$HOME /path/to/GenomicsDB # simplest

# Build and install the include/lib/bin files to $HOME
make && make install
```

3.2 Docker

Docker images for GenomicsDB are published to GitHub Container Registry. This page shows the image versions available. Below we show some examples of using the docker image to interact with GenomicsDB executables.

Examples:

To ingest vcfs from */path/to/vcfs* to */path/to/workspace* (both paths refer to paths on the host machine)

```
host> docker run -v /path/to/vcfs:/opt/vcfs -v /path/to/where/workspace/should/be/
↪created:/opt/workspace_parent -u $( id -u $USER ):$( id -g $USER ) -it ghcr.io/
↪genomicsdb/genomicsdb:v1.4.4
docker> vcf2genomicsdb_init -w /opt/workspace_parent/workspace -S /path/to/vcfs -n 0
docker> vcf2genomicsdb /opt/workspace_parent/loader.json
```

To query a GenomicsDB workspace, and return the results as a VCF/gVCF:

```
host> docker run -v /path/to/workspace:/opt/workspace -u $( id -u $USER ):$( id -g $USER␣
↪) -it ghcr.io/genomicsdb/genomicsdb:v1.4.4
# run with no arguments for help
docker> gt_mpi_gather
# Create a query.json file based on help
docker> gt_mpi_gather -j /path/to/query.json --produce-Broad-GVCF
```

3.3 CLI Tools

The CLI tools are compiled during the build process and placed in <build-dir>/tools.

CHAPTER

FOUR

IMPORT / ETL

GenomicsDB supports importing genomics data in several common formats, and with a variety of methods.

GenomicsDB can ingest data in VCF, BCF, gVCF and CSV formats. When importing VCF/BCFs or gVCFs, you may need to run a few preprocessing steps before importing.

The primary and suggested method of importing is to use the native vcf2genomicsdb importer tool (see *CLI Tools*). There is also a Java API for importing (see here).

4.1 VCF/gVCF

The import program can handle block compressed and indexed VCFs, BCFs and gVCFs. For brevity, we will only use the term VCF.

4.1.1 Organizing your data

- All your VCFs must be block compressed and indexed. Bcftools is one good option for compressing and indexing.
- The VCF format allows you to have multiple lines with the same position (identical chromosome+pos). The import program can NOT handle such VCFs. Make sure you use tools like bcftools to collapse multiple lines with the same position into a single line. Example command:

```
bcftools norm -m +any [-O <output_format> -o <output>] <input_file>
```

- **In a multi-node environment, you must decide:**
  - How to *partition your data in GenomicsDB <#Multi-node-setup>*
  - **How your VCF files are accessed by the import program:**
    - On a shared filesystem (NFS, Lustre etc) accessible from all nodes.
    - If you are partitioning data by rows, your files can be scattered across local filesystems on multiple machines; each filesystem accessible only by the node on which it is mounted. Currently, such a setup isn't supported for column partitioning.

4.2 CSV

Importing CSVs into GenomicsDB

4.2.1 Preliminaries

- You need libcsv while compiling.
- License terms: We use libcsv to parse CSV files. libcsv is licensed under the GNU Library or Lesser General Public License version 2.0 (LGPLv2). So, if you are re-distributing binaries or object files, they may be subject to LGPLv2 terms. Please ensure that any binaries/object files you distribute are compliant with LGPLv2. You can disable libcsv usage by not setting the USE_LIBCSV and LIBCSV_DIR flags during compilation. However, your binaries/executables will not be able to import CSV files into GenomicsDB.

4.2.2 CSV format

Given a variant call at a specific position in the genome for a particular CallSet/sample, the CSV file format supported by GenomicsDB contains one line describing the call. Essentially, each line in the CSV describes one cell in the GenomicsDB array.

The exact format of the CSV is shown below:

```
<row>,<begin_column>,<end_column>,<REF>,<concatenated_ALT>,<QUAL>,<FILTER_field_
↪specification>[,<other_fields_specification>]
```

Fixed fields:

- row (mandatory, type: int64): Row number in GenomicsDB for this sample/CallSet
- begin_column (mandatory, type: int64): Column in GenomicsDB at which the variant call begins
- end_column (mandatory, type: int64): Column in GenomicsDB at which the variant call ends (inclusive).
- REF (mandatory, type:string): Reference allele at the given position.
- concatenated_ALT (mandatory, type: string): Concatenated alternate alleles separated by the character '|'. For example, if a given call has two alternate alleles TAG and TG, then the concatenated string would be TAG|TG.
- QUAL (optional, type: float): Represents the QUAL field in a VCF. It can be left empty implying that the value was missing for this call.
- FILTER_field_specification (mandatory, type: variable length list of integers): Each element of the FILTER field list is an integer representing the FILTERs that this call failed (similar to how a BCF represents FILTERs). The first element of this field is an integer displaying the number of elements in the list. A value of 0 indicates that the list is empty.

Additional fields can be optionally specified * <other_fields_specification>: The format depends on the type of field:

- String type fields (fixed or variable length strings): The field should contain the string - an empty token indicates that the field is missing.
- Fixed length field (int or float): The field should contain exactly N elements where N is the length of the field (fixed constant). One or more tokens may be left empty to indicate that those elements are missing.
- Variable length field (int or float): The first element of this field should be an integer denoting the number of elements in the field for this call. It should then be followed by the elements of this field. An empty or missing field can be specified by setting the first element (field length) to 0 - no other elements should follow an empty field.

Example

The following line contains 2 fields in addition to the fixed fields:

- SB: Fixed length field of 4 integers
- PL: Variable length field of integers

```
2,1857210,1857210,G,A|T,894.77,0,,,,,6,923,0,599,996,701,1697
```

The line specifies the variant call for row id 2, beginning at column 1857210 and ending at 1857210. The REF allele is 'G' and the call has 2 alternate alleles 'A' and 'T' (SNVs). The QUAL value is 894.77 and there are no FILTERs specified (hence FILTER field length = 0). The SB field is missing - denoted by the 4 empty tokens. The PL field consists of 6 integers - the length appears first (since PL is a variable length field) followed by the elements [923,0,599,996,701,1697].

Special fields

- GT is represented in the CSV as a variable length list of integers - each element of the list refers to the allele index (0 for reference allele, 1 for the first alternate allele and so on). The length of the list represents the ploidy of the sample/CallSet and must be specified in the CSV line (since GT is treated as a variable length list).

4.2.3 Organizing your data

- All CSV files imported into a GenomicsDB array must respect the number and order of fields as defined in the vid_mapping_file.
- The import program cannot handle CSV files where multiple lines have the same value of row and begin_column - this restriction is similar to that imposed on loading VCFs. Consolidate these multiple lines into a single line to continue.

4.3 Multi-node setup

GenomicsDB can be setup to store variant data across multiple partitions of an array. All the data belonging to one partition of an array lives on a single filesystem. Thus, by creating multiple partitions, users can store data possibly across multiple hosts/nodes in a cluster. Array partitioning is useful when the data to be stored and queried is very large and cannot fit within a single machine/node. Or the user might wish to store array partitions in different nodes so that downstream queries and analysis can be run in a distributed manner for scalability and/or performance.

The user must decide how to partition data across multiple nodes in a cluster:

- How many nodes should be used to store the data?
- How many partitions should reside on each node? A single node can hold multiple partitions (assuming the node has enough disk space).
- What mode should be used for partitioning the data? Two modes of partitioning are supported by various import/query tools.
  - Row partitioning: In this mode, for a given sample/CallSet (row), all the variant data resides in a single partition. Data belonging to different samples/CallSets may be scattered across different partitions.
  - Column partitioning: In this mode, for a given genomic position (column), all the variant data across all samples/CallSets resides in a single partition. Data is partitioned by genomic positions.

Which partitioning scheme is better to use is dependent on the queries/analysis performed by downstream tools. Here are some example queries for which the 'best' partitioning schemes are suggested.

- Query: fetch attribute X from all samples/CallSets for position Y (or small interval [Y1-Y2])
  - **Row-based partitioning**
    - For single position queries (or small intervals), partitioning the data by rows would likely provide higher performance. By accessing data across multiple partitions that may be located in multiple nodes in parallel, the system will be able to utilize higher aggregate disk and memory bandwidth. In a column based partitioning, only a single partition would service the request.
    - Simple data import step if the original data is organized as a file per sample/CallSet (for example VCFs). Just import data from the required subset of files to the correct partition.
    - Con(s). A final aggregator may be needed since the data for a given position is scattered across machines. Some of the query tools we provide use MPI to collect the final output into a single node.
- Query: run analysis tool T on all variants (grouped by column position) found in a large column interval [Z1-Z2] (or scan across the whole array)
  - **Column-based partitioning**
    - The user is running a query/analysis for every position in the queried interval. Hence, for each position, the system must fetch data from all samples/CallSets and run T. Partitioning by column reduces/eliminates any communication between partitions. For a sufficiently large query interval, the aggregate disk and memory bandwidth across multiple nodes can still be utilized.
    - No/minimal data aggregation step as all the data for a given column is located within a single partition.
    - Con(s). Importing data into GenomicsDB may become complex, especially if the initial data is organized as a file per sample/CallSet.

CHAPTER

FIVE

COMMON ISSUES

1. Verify that your JSON configuration files are syntactically correct before invoking GenomicsDB tools

   The GenomicsDB tools assume that you provide a syntactically correct JSON - always validate your JSON files using tools such as *json_verify* before invoking GenomicsDB tools.

   ```
   cat <json_file> | json_verify
   ```

2. I have prepared my JSON files correctly, yet I get an exception with the following message:

   ```
   Could not open vid mapping file "~/<directory>/vid_mapping_file.json OR
   Could not open callsets file "~/<directory>/callset_mapping_file.json
   ```

   The character '~' is interpreted by shells (bash, tcsh) as the home directory of the user - it is NOT interpreted by file I/O system calls invoked by the GenomicsDB executables/libraries. Hence, you must specify the path of the file without special characters that are interpreted by shells.

   Correct examples:

   ```
   /home/<user>/<directory>/vid_mapping_file.json OR
   ../../<directory>/vid_mapping_file.json
   ```

3. I get an exception with the following message:

   ```
   Unhandled overlapping variants at columns <col1> and <col2> for row <row>
   ```

   GenomicsDB cannot deal with overlapping variants within a single sample. Workarounds exist for dealing with overlapping deletions and gVCF reference blocks (intervals with <NON_REF> as the only alternate allele). When overlapping variants which are neither deletions nor reference blocks are found in the input VCF file, then the above exception is thrown. Try to use bcftools to collapse multiple lines with the same position into a single line. An example bcftools invocation is shown below:

   ```
   bcftools norm -m +any [-O <output_format> -o <output>] <input_file>
   ```

4. I have both *row_partitions* and *column_partitions* in my loader JSON file and I get an exception message:

   ```
   Cannot have both "row_partitions" and "column_partitions" simultaneously in the␣
   ↪JSON file
   ```

   A GenomicsDB array can be partitioned by rows or columns but not both simultaneously.

5. I have setup all my JSON files correctly, but the import program finishes almost immediately without importing any data from my VCFs:

   There could be many reasons, but here are the common issues we have seen users running into:

- *Contig/chromosome names don't match in the vid_mapping_file and the input VCFs*: The contig/chromosome names in the VCF and the vid mapping JSON file MUST match EXACTLY. For example, if the vid file has a contig named _"1"_ (as per the 1000 genomes naming convention) while the VCF has a contig named _"chr1"_ (as per the UCSC convention), GenomicsDB will ignore all data corresponding to _"chr1"_.

6. I have setup all my JSON files correctly, but the import program doesn't load data for some of the samples:

   There could be many reasons, but here are the common issues we have seen users running into:

   - *Incorrect value(s) of idx_in_file in the callset_mapping_file*: Note that *row_idx* is the globally unique value of the TileDB row index corresponding to a given sample/CallSet. *idx_in_file* is useful mostly for multi-sample VCFs and specifies the index of the sample in a given VCF. For single sample VCFs, this field should be 0 (or omitted altogether).
   - *Incorrect partition bounds in the loader JSON or incorrectly specified partition index in the command line*: Please re-check your partition bounds in the loader JSON.

7. I see an incorrect cell order found error as:

```
$ vcf2genomicsdb loader.json
terminate called after throwing an instance of 'VCF2GenomicsDBException'
what():  VCF2GenomicsDBException : Incorrect cell order found - cells must be in␣
↪column major order. Previous cell: [ 0, 114111 ] current cell: [ 0, 114111 ]
Aborted
```

   The error occurs if alleles at the same position span across multiple lines, for example:

```
chrX    114112  .       TCT     T       999     PASS    .       GT:DP:GQ:MIN_DP:PL ␣
↪    0/0:0:0:0:0,0,0
chrX    114112  .       TCT     TTT     999     PASS    .       GT:DP:GQ:MIN_DP:PL ␣
↪    0/0:0:0:0:0,0,0
```

   The fix is to run bcftools norm as described above in 3. which will merge the alleles as

```
chrX    114112  .       TCT     T,TTT   999     PASS    .       GT:DP:GQ:MIN_DP:PL ␣
↪    0/2:0:0:0:0,0,0,0,0,0
```

8. I see an error message:

```
Cannot open VCF/BCF file <path.vcf.gz>
```

   even when the file and its index exist. What's going on?

   If you are importing data from many files (>1000), then it's likely that you are hitting the limit on the number of open files set in your machine(s). Find out how to increase the limit.

CHAPTER

SIX

APIS

6.1 Overview

In addition to the native C++ interface, GenomicsDB has support/bindings for Python, Java, and R. The data structure returned from a GenomicsDB query depends on the API or CLI tool being used and the method invoked. Similarly, the method of passing in query field information depends on the route chosen.

The various APIs take in the same query fields in one form or another, so check the appropriate reference for your use case. If you are new to GenomicsDB, we recommend starting with the *gt_mpi_gather* CLI tool to familiarize yourself with the concepts.

All the APIs support taking configuration options as protobuf. Here's the protobuf documentation:

6.1.1 GenomicsDB Protobuf Documentation

genomicsdb_callsets_mapping.proto

CallsetMappingPB

Field	Type	Label	Description
callsets	*SampleIDToTileDBIDMap*	repeated	

SampleIDToTileDBIDMap

Field	Type	Label	Description
sample_name	*string*	required	
row_idx	*int64*	required	
idx_in_file	*int64*	required	
stream_name	*string*	optional	
filename	*string*	optional	

genomicsdb_coordinates.proto

ContigInterval

Field	Type	Label	Description
contig	*string*	required	
begin	*int64*	optional	
end	*int64*	optional	

ContigPosition

Field	Type	Label	Description
contig	*string*	required	
position	*int64*	required	

GenomicsDBColumn

Field	Type	Label	Description
tiledb_column	*int64*	optional	
contig_position	*ContigPosition*	optional	

GenomicsDBColumnInterval

Field	Type	Label	Description
tiledb_column_interval	*TileDBColumnInterval*	optional	
contig_interval	*ContigInterval*	optional	

GenomicsDBColumnOrInterval

Field	Type	Label	Description
column	*GenomicsDBColumn*	optional	
column_interval	*GenomicsDBColumnInterval*	optional	

TileDBColumnInterval

Field	Type	Label	Description
begin	*int64*	required	
end	*int64*	required	

genomicsdb_export_config.proto

AnnotationSource

Field	Type	Label	Description
filename	*string*	required	
data_source	*string*	required	
attributes	*string*	repeated	
is_vcf	*bool*	optional	Default: true
file_chromosomes	*string*	repeated	

ExportConfiguration

Field	Type	Label	Description
workspace	*string*	required	
reference_genome	*string*	optional	
array_name	*string*	optional	
generate_array_name_from_partition_bounds	*bool*	optional	Default: true
query_column_ranges	*GenomicsDBColumnOrIntervalList*	repeated	Only one of the follow
query_contig_intervals	*ContigInterval*	repeated	
query_row_ranges	*RowRangeList*	repeated	Only one of the follow
query_sample_names	*string*	repeated	
attributes	*string*	repeated	
query_filter	*string*	optional	QueryConfiguration -
vcf_header_filename	*string*	optional	
vcf_output_filename	*string*	optional	
vcf_output_format	*string*	optional	
vid_mapping_file	*string*	optional	
vid_mapping	*VidMappingPB*	optional	
callset_mapping_file	*string*	optional	
callset_mapping	*CallsetMappingPB*	optional	
max_diploid_alt_alleles_that_can_be_genotyped	*uint32*	optional	Other configuration
max_genotype_count	*uint32*	optional	
index_output_VCF	*bool*	optional	
produce_GT_field	*bool*	optional	
produce_FILTER_field	*bool*	optional	
sites_only_query	*bool*	optional	
produce_GT_with_min_PL_value_for_spanning_deletions	*bool*	optional	
scan_full	*bool*	optional	
segment_size	*uint32*	optional	Default: 10485760

Table 1 – continued from previous page

Field	Type	Label	Description
combined_vcf_records_buffer_size_limit	*uint32*	optional	
enable_shared_posixfs_optimizations	*bool*	optional	Default: false
bypass_intersecting_intervals_phase	*bool*	optional	Default: false
spark_config	*SparkConfig*	optional	
annotation_source	*AnnotationSource*	repeated	
annotation_buffer_size	*uint32*	optional	Default: 10240

GenomicsDBColumnOrIntervalList

Field	Type	Label	Description
column_or_interval_list	*GenomicsDBColumnOrInterval*	repeated	

QueryConfiguration

Simple query configuration for GenomicsDB::query_variant_calls for the class initialized with ExportConfiguration below

Field	Type	Label	Description
array_name	*string*	optional	
generate_array_name_from_partition_bounds	*bool*	optional	Default: true
query_column_ranges	*GenomicsDBColumnOrIntervalList*	repeated	Only one of the following two fields must be defined query_contig_intervals is recommended for use
query_contig_intervals	*ContigInterval*	repeated	
query_row_ranges	*RowRangeList*	repeated	Only one of the following two fields must be defined
query_sample_names	*string*	repeated	
attributes	*string*	repeated	
query_filter	*string*	optional	

RowRange

Field	Type	Label	Description
low	*int64*	required	
high	*int64*	required	

RowRangeList

Field	Type	Label	Description
range_list	*RowRange*	repeated	

SparkConfig

Field	Type	Label	Description
query_block_size	*int64*	optional	
query_block_size_margin	*int64*	optional	

genomicsdb_import_config.proto

ImportConfiguration

Field	Type	Label	Description
size_per_column_partition	*int64*	required	Default: 16384
row_based_partitioning	*bool*	optional	Default: false
produce_combined_vcf	*bool*	optional	Default: false
produce_tiledb_array	*bool*	optional	Default: true
column_partitions	*Partition*	repeated	
vid_mapping_file	*string*	optional	
vid_mapping	*VidMappingPB*	optional	
callset_mapping_file	*string*	optional	
callset_mapping	*CallsetMappingPB*	optional	
treat_deletions_as_intervals	*bool*	optional	Default: true

continues on next page

Table 2 – continued from previous page

Field	Type	Label	Description
num_parallel_vcf_files	*int32*	optional	Default: 1
delete_and_create_tiledb_array	*bool*	optional	Default: false
do_ping_pong_buffering	*bool*	optional	Default: true
offload_vcf_output_processing	*bool*	optional	Default: true
discard_vcf_index	*bool*	optional	Default: true
segment_size	*int64*	optional	Default: 10485760
compress_tiledb_array	*bool*	optional	Default: true
num_cells_per_tile	*int64*	optional	Default: 1000
fail_if_updating	*bool*	optional	Default: false
tiledb_compression_type	*int32*	optional	Default: 1
tiledb_compression_level	*int32*	optional	Default: -1
consolidate_tiledb_array_after_load	*bool*	optional	Default: false
disable_synced_writes	*bool*	optional	Default: true
ignore_cells_not_in_partition	*bool*	optional	
lb_callset_row_idx	*int64*	optional	Default: 0
ub_callset_row_idx	*int64*	optional	
enable_shared_posixfs_optimizations	*bool*	optional	Default: false
disable_delta_encode_for_offsets	*bool*	optional	Default: false
disable_delta_encode_for_coords	*bool*	optional	Default: false
enable_bit_shuffle_gt	*bool*	optional	Default: false
enable_lz4_compression_gt	*bool*	optional	Default: false
reference_genome	*string*	optional	
vcf_header_filename	*string*	optional	

Partition

Field	Type	Label	Description
begin	*GenomicsDBColumn*	required	
workspace	*string*	optional	
array_name	*string*	optional	
generate_array_name_from_partition_bounds	*bool*	optional	
vcf_output_filename	*string*	optional	
vcf_header_filename	*string*	optional	
end	*GenomicsDBColumn*	optional	

genomicsdb_vid_mapping.proto

Chromosome

Field	Type	Label	Description
name	*string*	required	
length	*int64*	required	
tiledb_column_offset	*int64*	required	

FieldLengthDescriptorComponentPB

Field	Type	Label	Description
variable_length_descriptor	*string*	optional	
fixed_length	*int32*	optional	

GenomicsDBFieldInfo

Field	Type	Label	Description
name	*string*	required	
type	*string*	repeated	
vcf_field_class	*string*	repeated	
vcf_type	*string*	optional	
length	*FieldLengthDescriptorComponentPB*	repeated	
vcf_delimiter	*string*	repeated	
VCF_field_combine_operation	*string*	optional	
vcf_name	*string*	optional	useful when multiple fields of different types/length with the same name (FILTER, FORMAT, INFO) are defined in the VCF header
disable_remap_missing_with_non_ref	*bool*	optional	Default: false

VidMappingPB

Field	Type	Label	Description
fields	*GenomicsDBFieldInfo*	repeated	
contigs	*Chromosome*	repeated	

Scalar Value Types

.proto Type	Notes	C++	Java	Python	Go
double		double	double	float	float64
float		float	float	float	float32
int32	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.	int32	int	int	int32
int64	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.	int64	long	int/long	int64
uint32	Uses variable-length encoding.	uint32	int	int/long	uint32
uint64	Uses variable-length encoding.	uint64	long	int/long	uint64
sint32	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.	int32	int	int	int32
sint64	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.	int64	long	int/long	int64
fixed32	Always four bytes. More efficient than uint32 if values are often greater than 2^28.	uint32	int	int	uint32
fixed64	Always eight bytes. More efficient than uint64 if values are often greater than 2^56.	uint64	long	int/long	uint64
sfixed32	Always four bytes.	int32	int	int	int32
sfixed64	Always eight bytes.	int64	long	int/long	int64
bool		bool	boolean	boolean	bool
string	A string must always contain UTF-8 encoded or 7-bit ASCII text.	string	String	str/unicode	string
bytes	May contain any arbitrary sequence of bytes.	string	ByteString	str	[]byte

6.2 C++

- Native interface - See cpp source code
- Querying CLI tools available - see *vcf2genomicsdb*

6.2.1 API Reference

C++ Query Interface

class **GenomicsDB**

Experimental Query Interface to *GenomicsDB* for Arrays partitioned by columns Concurrency support is provided via query json files for now - see https://github.com/GenomicsDB/GenomicsDB/wiki/Querying-GenomicsDB#json-configuration-file-for-a-query https://github.com/GenomicsDB/GenomicsDB/wiki/MPI-with-GenomicsDB

Public Functions

GENOMICSDB_EXPORT **GenomicsDB**(const std::string &workspace, const std::string &callset_mapping_file, const std::string &vid_mapping_file, const std::vector<std::string> attributes = ALL_ATTRIBUTES, const uint64_t segment_size = DEFAULT_SEGMENT_SIZE)

Constructor to the *GenomicsDB* Query API workspace callset_mapping_file vid_mapping_file attributes, optional segment_size, optional Throws GenomicsDBException

GENOMICSDB_EXPORT **GenomicsDB**(const std::string &query_configuration, const query_config_type_t query_configuration_type = JSON_FILE, const std::string &loader_configuration_json_file = std::string(), const int concurrency_rank = 0)

Constructor to the *GenomicsDB* Query API with configuration json files query_configuration - describe the query configuration in either a JSON file or JSON string or protobuf binary query_configuration_type - type of query configuration, could be a JSON_FILE or JSON_STRING or PROTOBUF_BINARY_STRING loader_config_json_file, optional - describe the loader configuration in a JSON file. If a configuration key exists in both the query and the loader configuration, the query configuration takes precedence concurrency_rank, optional - if greater than 0, the constraints(workspace, array, column and row ranges) are surmised using the rank as an index into their corresponding vectors Throws GenomicsDBException

GENOMICSDB_EXPORT **~GenomicsDB**()

Destructor

GENOMICSDB_EXPORT GenomicsDBVariants query_variants (const std::string &array, genomicsdb_ranges_t column_ranges=SCAN_FULL, genomicsdb_ranges_t row_ranges={})

Query *GenomicsDB* array for variants constrained by column and row ranges. Variants are similar to GAVariant in GA4GH API array column_ranges, optional row_ranges, optional

GENOMICSDB_EXPORT GenomicsDBVariants query_variants ()

Query using set configuration for variants. Useful when using parallelism paradigms(MPI, Intel TBB) Variants are similar to GAVariant in GA4GH API

GENOMICSDB_EXPORT GenomicsDBVariantCalls query_variant_calls (const std:: string &array, genomicsdb_ranges_t column_ranges=SCAN_FULL, genomicsdb_ranges_t row_ranges={})

Query the array for variant calls constrained by the column and row ranges. Variant Calls are similar to GACall in GA4GH API. array column_ranges, optional row_ranges, optional

GENOMICSDB_EXPORT GenomicsDBVariantCalls query_variant_calls (GenomicsDBVariantCallProcessor &proce const std::string &array, genomicsdb_ranges_t column_ranges=SCAN_FULL, genomicsdb_ranges_t row_ranges={})

Query the array for variant calls constrained by the column and row ranges. Variant Calls are similar to GACall in GA4GH API. array column_ranges, optional row_ranges, optional

GENOMICSDB_EXPORT GenomicsDBVariantCalls query_variant_calls ()

Query using set configuration for variant calls. Useful when using parallelism paradigms(MPI, Intel TBB) Variant Calls are similar to GACall in GA4GH API.

GENOMICSDB_EXPORT GenomicsDBVariantCalls query_variant_calls (GenomicsDBVariantCallProcessor &proce const std::string &query_configuration, const query_config_type_t query_configuration_type)

Query with a configuration describing the subset for variant calls. Useful with paradigms like MPI, Intel TBB and when a *GenomicsDB* instance is cached with multiple, concurrent query_variant_calls with different subset configurations. Variant Calls are similar to GACall in GA4GH API. processor custom processor to process variant calls query_configuration protobuf export configuration as binary string, optional. If not specified, the configuration specified during class construction will be used with the query query_configuration_type Type of configuration, Currently only PROTOBUF_BINARY_STRING is supported and an exception is throwm for other types.

GENOMICSDB_EXPORT void generate_vcf (const std::string &array, genomicsdb_ranges_t column_ranges, genomicsdb_ranges_t row_ranges, const std::string &reference_genome, const std::string &vcf_header="vcf_header.vcf", const std::string &output="", const std::string &output_format="", bool overwrite=false)

Generate multi-sample vcf files from *GenomicsDB* in the Broad GVCF format for given array constrained by column/row ranges

- see https://gatk.broadinstitute.org/hc/en-us/articles/360035531812-GVCF-Genomic-Variant-Call-Format

GENOMICSDB_EXPORT void generate_vcf (const std::string &output="", const std::string &output_format="", bool overwrite=false)

Generate multi-sample vcf files from *GenomicsDB* in the Broad GVCF format using set configuration. This method is useful with parallelism paradigms (MPI, Intel TBB)

- see https://gatk.broadinstitute.org/hc/en-us/articles/360035531812-GVCF-Genomic-Variant-Call-Format

GENOMICSDB_EXPORT void generate_plink (const std::string &array, genomicsdb_ranges_t column_ranges, genomicsdb_ranges_t row_ranges, unsigned char format=7, int compression=1, bool one_pass=false, bool verbose=false, double progress_interval=-1, const std::string &output_prefix="output", const std::string &fam_list="")

Generate plink files from *GenomicsDB* for given array constrained by column/row ranges and given format to generate plink .ped and .map files. The output files are named <output_prefix>.ped and <output_prefix>.map respectively.

GENOMICSDB_EXPORT void generate_plink (unsigned char format=7, int compression=1, bool one_pass=false, bool verbose=false, double progress_interval=-1, const std::string &output_prefix="output", const std::string &fam_list="")

Generate plink files from *GenomicsDB* for given format to generate plink .ped and .map files. The output files are named <output_prefix>.ped and <output_prefix>.map respectively. This method is useful with parallelism paradigms (MPI, Intel TBB).

GENOMICSDB_EXPORT interval_t get_interval (const genomicsdb_variant_t *variant)

Utility template functions to extract information from Variant and VariantCall classes

6.3 Python

- Python bindings for GenomicsDB. Requires Python version >= *3.6*
- Built using Cython.
- Currently only supports querying (no importing).
- Data can be returned in either a numpy matrix or a list of lists.

The GenomicsDB python bindings are published on PyPI (currently experimental).

Install `genomicsdb` with the following command:

```
pip install genomicsdb
```

6.3.1 API Reference

genomicsdb module

6.4 Java

- Data can be returned in either a htsjdk VariantContext object or <other>.
- The GenomicsDB java bindings are published to Maven Central every release.
- Add GenomicsDB to your maven project with the following snippet (update the version, if necessary):

```
<!-- https://mvnrepository.com/artifact/org.genomicsdb/genomicsdb -->
<dependency>
   <groupId>org.genomicsdb</groupId>
   <artifactId>genomicsdb</artifactId>
   <version>1.4.5</version>
</dependency>
```

6.4.1 API Reference

Java Import Package

The *GenomicsDBImporter* class is the main package to import VCFs into GenomicsDB.

org::genomicsdb::importer::GenomicsDBImporter : public org.genomicsdb.importer.GenomicsDBImporterJni , public org.genomicsdb.importer.extensions.JsonFileExtensions , public org.genomicsdb.importer.extensions.CallSetMapExtensions , public org.genomicsdb.importer.extensions.VidMapExtensions

Java wrapper for vcf2genomicsdb - imports VCFs into *GenomicsDB*. All vid information is assumed to be set correctly by the user (JSON files)

Public Functions

`inline GenomicsDBImporter (final String loaderJSONFile)`

Constructor

Parameters
`loaderJSONFile` – *GenomicsDB* loader JSON configuration file

`inline GenomicsDBImporter (final String loaderJSONFile, final int rank)`

Constructor

Parameters

- `loaderJSONFile` – *GenomicsDB* loader JSON configuration file
- `rank` – Rank of this process (TileDB/GenomicsDB partition idx)

`inline GenomicsDBImporter (final ImportConfig config)`

Constructor to create required data structures from a list of GVCF files and a chromosome interval. This constructor is developed specifically for running Chromosome intervals imports in parallel.

Parameters
`config` – Parallel import configuration

Throws
`FileNotFoundException` – when files could not be read/written

`inline GenomicsDBVidMapProto.VidMappingPB getProtobufVidMapping ()`

Function to return the vid mapping protobuf object.

Returns
protobuf object for vid mapping

`inline void updateProtobufVidMapping (GenomicsDBVidMapProto.VidMappingPB vidMapPB)`

Function to update vid mapping protobuf object in the top level config object. Used in cases where the VCF header doesn't contain accurate information about how to parse fields. For instance, allele specific annotations

Parameters
`vidMapPB` – vid mapping protobuf object to use as new

`inline int addSortedVariantContextIterator (final String streamName, final VCFHeader vcfHeader, Iterator< VariantContext > vcIterator, final long bufferCapacity, final VariantContextWriterBuilder.OutputType streamType, final Map< Integer, SampleInfo > sampleIndexToInfo)`

Add a sorted VC iterator as the data source - caller must:

i. Call *setupGenomicsDBImporter()* once all iterators are added

ii. Call *doSingleImport()*

iii. Done!

Parameters

- `streamName` – Name of the stream being added - must be unique with respect to this *GenomicsDBImporter* object

- **vcfHeader** – VCF header for the stream
- **vcIterator** – Iterator over VariantContext objects
- **bufferCapacity** – Capacity of the stream buffer in bytes
- **streamType** – BCF_STREAM or VCF_STREAM
- **sampleIndexToInfo** – map from sample index in the vcfHeader to SampleInfo object which contains row index and globally unique name can be set to null, which implies that the mapping is stored in a callsets JSON file

Returns
returns the stream index

inline int addBufferStream (final String streamName, final VCFHeader vcfHeader, final long bufferCapacity, final VariantContextWriterBuilder.OutputType streamType, Iterator< VariantContext > vcIterator, final Map< Integer, SampleInfo > sampleIndexToInfo)

Add a buffer stream or VC iterator - internal function

Parameters

- **streamName** – Name of the stream being added - must be unique with respect to this *GenomicsDBImporter* object
- **vcfHeader** – VCF header for the stream
- **bufferCapacity** – Capacity of the stream buffer in bytes
- **streamType** – BCF_STREAM or VCF_STREAM
- **vcIterator** – Iterator over VariantContext objects - can be null
- **sampleIndexToInfo** – map from sample index in the vcfHeader to SampleInfo object which contains row index and globally unique name can be set to null, which implies that the mapping is stored in a callsets JSON file

Returns
returns the stream index

inline void **setupGenomicsDBImporter**()

Setup the importer after all the buffer streams are added, but before any data is inserted into any stream No more buffer streams can be added once *setupGenomicsDBImporter()* is called

Throws
IOException – throws IOException if modified callsets JSON cannot be written

inline boolean add (VariantContext vc, final int streamIdx)

Write VariantContext object to stream - may fail if the buffer is full It's the caller's responsibility keep track of the VC object that's not written

Parameters

- **vc** – VariantContext object
- **streamIdx** – index of the stream returned by the *addBufferStream()* call

Returns
true if the vc object was written successfully, false otherwise

inline boolean **doSingleImport**()

Only to be used in cases where iterator of VariantContext are not used. The data is written to buffers directly after which this function is called. See TestBufferStreamGenomicsDBImporter.java for an example

Throws
IOException – if the import fails

Returns
true if the import process is done

inline void **executeImport**()

Import multiple chromosome interval

inline void executeImport (final int numThreads)

Import multiple chromosome interval

Parameters
numThreads – number of threads used to import partitions

inline void doConsolidate (final int numThreads)

Consolidate all intervals/arrays in a given workspace into a single fragment

Parameters
numThreads – number of threads to use to parallelize consolidation

inline long **getNumExhaustedBufferStreams**()

Returns
get number of buffer streams for which new data must be supplied

inline int getExhaustedBufferStreamIndex (final long i)

Get buffer stream index of i-th exhausted stream There are mNumExhaustedBufferStreams and the caller must provide data for streams with indexes getExhaustedBufferStreamIndex(0), getExhaustedBufferStreamIndex(1),..., getExhaustedBufferStreamIndex(mNumExhaustedBufferStreams-1)

Parameters
i – i-th exhausted buffer stream

Returns
the buffer stream index of the i-th exhausted stream

inline boolean **isDone**()

Is the import process completed

Returns
true if complete, false otherwise

inline MultiChromosomeIterator **columnPartitionIterator**(FeatureReader<VariantContext> reader)

Utility function that returns a MultiChromosomeIterator given an FeatureReader that will iterate over the VariantContext objects provided by the reader belonging to the column partition specified by this object's loader JSON file and rank/partition index

Parameters
reader – AbstractFeatureReader over VariantContext objects

Throws

- IOException – when the reader's query method throws an IOException

- `ParseException` – when there is a bug in the JNI interface and a faulty JSON is returned

Returns
MultiChromosomeIterator that iterates over VariantContext objects in the reader belonging to the specified column partition

inline void **write**()

Write to TileDB/GenomicsDB using the configuration specified in the loader file passed to constructor

inline void write (final int rank)

Write to TileDB/GenomicsDB using the configuration specified in the loader file passed to constructor

Parameters
rank – Rank of this process (TileDB/GenomicsDB partition idx)

inline void write (final String loaderJSONFile, final int rank)

Write to TileDB/GenomicsDB

Parameters

- **loaderJSONFile** – *GenomicsDB* loader JSON configuration file
- **rank** – Rank of this process (TileDB/GenomicsDB partition idx)

inline void coalesceContigsIntoNumPartitions (final int partitions)

Coalesce contigs into fewer *GenomicsDB* partitions

Parameters
partitions – Approximate number of partitions to coalesce into

Public Static Functions

static inline MultiChromosomeIterator columnPartitionIterator (FeatureReader< VariantContext > read
final String loaderJSONFile, final int partitionIdx)

Utility function that returns a MultiChromosomeIterator given an FeatureReader that will iterate over the VariantContext objects provided by the reader belonging to the column partition specified by the loader JSON file and rank/partition index

Parameters

- **reader** – AbstractFeatureReader over VariantContext objects
- **loaderJSONFile** – path to loader JSON file
- **partitionIdx** – rank/partition index

Throws

- `IOException` – when the reader's query method throws an IOException
- `ParseException` – when there is a bug in the JNI interface and a faulty JSON is returned

Returns
MultiChromosomeIterator that iterates over VariantContext objects in the reader belonging to the specified column partition

The following packages are useful for specifying the import configuration.

class **ImportConfig**

This implementation extends what is in GenomicsDBImportConfiguration. Add extra data that is needed for parallel import.

Subclassed by org.genomicsdb.model.CommandLineImportConfig

Public Functions

inline ImportConfig (final GenomicsDBImportConfiguration.ImportConfiguration importConfiguration, final boolean validateSampleToReaderMap, final boolean passAsVcf, final int batchSize, final Set< VCFHeaderLine > mergedHeader, final Map< String, URI > sampleNameToVcfPath, final Func< Map< String, URI >, Integer, Integer, Map< String, FeatureReader< VariantContext >>> sampleToReaderMapCreator, final boolean incrementalImport)

Main *ImportConfig* constructor

Parameters

- **importConfiguration** – GenomicsDBImportConfiguration protobuf object
- **validateSampleToReaderMap** – Flag for validating sample to reader map
- **passAsVcf** – Flag for indicating that a VCF is being passed
- **batchSize** – Batch size
- **mergedHeader** – Required header
- **sampleNameToVcfPath** – Sample name to VCF path map
- **sampleToReaderMapCreator** – Function used for creating sampleToReaderMap
- **incrementalImport** – Flag for indicating incremental import

template<T1, T2, T3, R>

interface **Func**

class **BatchCompletionCallbackFunctionArgument**

Java Query Package

The *GenomicsDBFeatureReader* package can be used to read from a GenomicsDB workspace.

template<T, SOURCE> org::genomicsdb::reader::GenomicsDBFeatureReader : public htsjdk::tribble::FeatureReader< T > , public org.genomicsdb.importer.extensions.JsonFileExtensions

A reader for *GenomicsDB* that implements htsjdk.tribble.FeatureReader Currently, the reader only return htsjdk.variant.variantcontext.VariantContext

Public Functions

inline GenomicsDBFeatureReader (final GenomicsDBExportConfiguration.ExportConfiguration exportConfiguration, final FeatureCodec< T, SOURCE > codec, final Optional< String > loaderJSONFile)

Constructor

Parameters

- **exportConfiguration** – query parameters
- **codec** – FeatureCodec, currently only htsjdk.variant.bcf2.BCF2Codec and htsjdk.variant.vcf.VCFCodec are tested
- **loaderJSONFile** – *GenomicsDB* loader JSON configuration file

Throws

IOException – when data cannot be read from the stream

inline Object **getHeader**()

Return the VCF header of the combined gVCF stream

Returns

the VCF header of the combined gVCF stream

inline List<String> **getSequenceNames**()

Return the list of contigs in the combined VCF header

Returns

list of strings of the contig names

inline CloseableTribbleIterator<T> **iterator**()

Return an iterator over htsjdk.variant.variantcontext.VariantContext objects for the specified TileDB array and query configuration

Returns

iterator over htsjdk.variant.variantcontext.VariantContext objects

inline CloseableTribbleIterator< T > query (final String chr, final int start, final int end)

Return an iterator over htsjdk.variant.variantcontext.VariantContext objects for the specified TileDB array and queried position

Parameters

- **chr** – contig name
- **start** – start position (1-based)
- **end** – end position, inclusive (1-based)

Returns

iterator over htsjdk.variant.variantcontext.VariantContext objects

The *GenomicsDBQuery* package can also be used to query a workspace.

class **GenomicsDBQuery**

class **Interval** : public Serializable

class **Pair** : public Serializable

class **VariantCall** : public Serializable

CHAPTER

SEVEN

CLI TOOLS

GenomicsDB contains several native command line tools. These tools are also available through GenomicsDB docker. Refer to *this link* for instructions on building and installing GenomicsDB, as well as how to work with artifacts in docker images.

7.1 vcf2genomicsdb_init

Tool for creating a GenomicsDB workspace, and initializing it with configuration json files.

Example usage:

```
vcf2genomicsdb_init -w /path/to/workspace -s <list-of-sample-vcfs>
```

7.2 vcf2genomicsdb

Tool for importing VCF files into GenomicsDB.

Example usage:

```
vcf2genomicsdb <loader-json.json>
```

7.3 gt_mpi_gather

Tool for querying GenomicsDB. Outputs results as either *VariantCalls* or *Variants*

Example usage:

```
gt_mpi_gather -j <query.json>
```

Here is an example *query.json* file:

```
{
    "workspace" : "/tmp/ws/",
    "array" : "t0_1_2",
    "query_column_ranges" : [ [ [0, 100 ], 500 ] ],
    "query_row_ranges" : [ [ [0, 2 ] ] ],
    "vid_mapping_file": "tests/inputs/vid.json",
```

(continues on next page)

(continued from previous page)

```
    "callset_mapping_file": "tests/inputs/callset_mapping.json",
    "query_attributes" : [ "REF", "ALT", "BaseQRankSum", "MQ", "MQ0",
↪"ClippingRankSum", "MQRankSum", "ReadPosRankSum", "DP", "GT", "GQ", "SB", "AD
↪", "PL", "DP_FORMAT", "MIN_DP" ]
}
```

Most of the fields are self-explanatory (take a look at the *terminology section* section). The following fields are mandatory:

- workspace (type: string or list of strings)
- array (type: string or list of strings)
- query_column_ranges : This field contains is a list of lists. Each member list contains the column ranges that are to be queried. Each element of the inner list can be a single integer, representing the single column position to be queried, or a list of size 2 representing the column range to be queried.

  In the above example, the process will query column range [0-100] (inclusive) and the position 500 and return all *VariantCalls* intersecting with these query ranges/positions. This field is a list of lists (and other fields are lists of strings), in order to support using GenomicsDB with MPI. Each member of the outermost list can be processed by a separate MPI rank. If MPI is not being used, only the first element of the outermost list will be processed.
- query_attributes : List of strings specifying attributes to be fetched

Optional field(s):

- query_row_ranges : Similar to *query_column_ranges* but for rows. If this field is omitted, all rows are included in the query.
- vid_mapping_file/callset_mapping_file (type: string or list of strings): Paths to JSON files specifying the *vid mapping* and *callset mapping*.

  These two fields are optional because a user might specify them in the loader JSON while creating an array and pass the loader JSON to the query tool(s) (see below). This allows the user to write many different query JSON files without repeating the information. If the *vid_mapping_file* and/or *callset_mapping_file* are specified in both the loader and query JSON files and passed to the query tool(s), then the parameter value in the query JSON gets precedence.

gt_mpi_gather returns data in a JSON format by default. Query results may be returned as VariantCalls or Variants depending on command line parameters passed by the user. Other formats (gVCF, PLINK) are also supported but these are well described elsewhere and so omitted here.

A VariantCall object contains information stored for a given sample/CallSet for a given location/genomic interval in GenomicsDB, i.e, it contains all the fields stored in the corresponding GenomicsDB array cell. A sample VariantCall formatted as a JSON is shown below:

```
{
    "row": 2,
    "interval": [ 17384, 17386 ],
    "fields": {
        "REF": "GAT",
        "ALT": [ "GT","<NON_REF>" ],
        "PL": [ 1018,0,1116,1137,1224,2361 ],
        "BaseQRankSum": [ 1.046 ]
    }
}
```

The above example shows a VariantCall object for the sample/CallSet corresponding to row 2 in the GenomicsDB at column 17384. Since the VariantCall is a deletion, it spans multiple columns and ends at column 17386 (inclusive). The fields field is self explanatory.

A Variant is a set of VariantCalls which meet the following properties:

- All VariantCalls span the exact same column interval
- All VariantCalls have the same value of REF
- All VariantCalls have the same value of ALT

A sample Variant formatted as a JSON is shown below:

```
{
    "interval": [ 17384, 17384 ],
        "common_fields" : {
            "REF": "G",
            "ALT": [ "A","<NON_REF>" ]
        },
        "variant_calls": [
        {
            "row": 0,
            "interval": [ 17384, 17384 ],
            "fields": {
                "REF": "G",
                "ALT": [ "A","<NON_REF>" ],
                "BaseQRankSum": [ -2.096000 ]
            }
        },
        {
            "row": 2,
            "interval": [ 17384, 17384 ],
            "fields": {
                "REF": "G",
                "ALT": [ "A","<NON_REF>" ],
                "BaseQRankSum": [ 1.046000 ]
            }
        }
    ]
}
```

In the above example, the Variant object corresponds to interval [17384, 17384] with “G” as the reference allele and “A”, “<NON_REF>” as the alternate alleles. It consists of two VariantCalls at rows 0 and 2.

7.4 create_genomicsdb_workspace

Tool for creating a new GenomicsDB workspace.

Example usage:

```
create_genomicsdb_workspace /path/to/workspace
```

7.5 consolidate_genomicsdb_array

Consolidate fragments from incremental imports into a single fragment.

Example usage:

```
consolidate_genomicsdb_array -w /path/to/workspace -a <name-of-array-to-
↪consolidate>
```

7.6 vcf_histogram

Create a histogram showing the number of variants per histogram bin. Can be useful in deciding how to partition the GenomicsDB workspace.

Example usage:

```
vcf_histogram <json>
```

7.7 vcfdiff

Check whether VCFs are identical

Example usage:

```
vcfdiff /path/to/vcf1 /path/to/vcf2
```

CHAPTER

EIGHT

EXAMPLE PYTHON NOTEBOOK CONNECTING TO GENOMICSDB ON AZURE BLOB STORAGE

[24]:
```
import genomicsdb
print(genomicsdb.version())
```

```
1.4.5-SNAPSHOT-855435f
```

Below we set up some (optional) environment variables in order to connect to Azure Blob Storage. We also support using environment variables to connect AWS S3 or GCS storage – in each case, we use the native cloud SDK so refer to the appropriate documentation for supported environment variables. In addition, we also support using roles or service principals for access to cloud storage.

Next, we specify the cloud URIs for the GenomicsDB workspace, callset, vid and reference file, as well as the GenomicsDB array we wish to query. Lastly, we also specify the genomic attributes we wish to query from the workspace.

The environment variables and URIs have all been redacted below.

---

[25]:
```
# set up environment variables and configuration for query
import os
storageaccount = ""
os.environ["AZURE_STORAGE_ACCOUNT"] = storageaccount
os.environ["AZURE_STORAGE_KEY"] = ""

container = ""
workspace_prefix = ""
workspace = f"az://{container}@{storageaccount}.blob.core.windows.net/{workspace_prefix}"
callset_file = f"az://{container}@{storageaccount}.blob.core.windows.net/{workspace_
↪prefix}/callset_mapping.json"
vid_file = f"az://{container}@{storageaccount}.blob.core.windows.net/{workspace_prefix}/
↪vid_mapping.json"
array = ""
attributes = ["REF", "ALT", "GT"]
```

We use the `connect_with_protobuf` function to connect to the workspace. Check out the GenomicsDB protobuf specification here

---

[26]:
```
from genomicsdb.protobuf import genomicsdb_export_config_pb2 as query_pb
from genomicsdb.protobuf import genomicsdb_coordinates_pb2 as query_coords

# create the query protobuf, and point to the workspace we want to query
query = query_pb.ExportConfiguration()
query.workspace = workspace
query.array_name = array
query.attributes.extend(["REF", "ALT", "GT"])
query.callset_mapping_file = callset_file
query.vid_mapping_file = vid_file

# specify the samples we wish to query
query.query_sample_names.extend([
    '0x00922C4598840C041CB1BB19DC75231C969E9057F7E3B9CC04EE8B44E714B793_1xAFO897WL2I',
    '0x00AFE0D460A52BC28399A89E68A62D4CF2D279B56F482166B11BA7F29AFB45C9_1xGQPZ8BXWU0',
    '0x008AA255A3EF6E54E308C9B6728F9D7D60D71428A2DDD339DD6A7956DFB73B90_1x7HNHG5JNCA',
    '0x00AFE0D460A52BC28399A89E68A62D4CF2D279B56F482166B11BA7F29AFB45C9_1x6E9Z1KV502'
])

# specify the genomic intervals we wish to query
intervals = []
intervals.append(query_coords.ContigInterval(contig="1"))
intervals.append(query_coords.ContigInterval(contig="2", begin=1, end=100000000))
intervals.append(query_coords.ContigInterval(contig="5", begin=5000000, end=75000000))
query.query_contig_intervals.extend(intervals)

gdb = genomicsdb.connect_with_protobuf(query)
list = gdb.query_variant_calls()
print(*list, sep='\n')
```

```
(0, 249250620, [{'Row': 1, 'Col': 11188011, 'Sample':
↪'0x00922C4598840C041CB1BB19DC75231C969E9057F7E3B9CC04EE8B44E714B793_1xAFO897WL2I',
↪'CHROM': '1', 'POS': 11188012, 'END': 11188012, 'REF': 'C', 'ALT': '[T]', 'GT': '0/0'},
↪ {'Row': 1, 'Col': 65310488, 'Sample':
↪'0x00922C4598840C041CB1BB19DC75231C969E9057F7E3B9CC04EE8B44E714B793_1xAFO897WL2I',
↪'CHROM': '1', 'POS': 65310489, 'END': 65310489, 'REF': 'T', 'ALT': '[C]', 'GT': '0/0'},
↪ {'Row': 3, 'Col': 65310488, 'Sample':
↪'0x00AFE0D460A52BC28399A89E68A62D4CF2D279B56F482166B11BA7F29AFB45C9_1xGQPZ8BXWU0',
↪'CHROM': '1', 'POS': 65310489, 'END': 65310489, 'REF': 'T', 'ALT': '[C]', 'GT': '0/0'}
↪])
(251743127, 351743126, [{'Row': 3, 'Col': 281159492, 'Sample':
↪'0x00AFE0D460A52BC28399A89E68A62D4CF2D279B56F482166B11BA7F29AFB45C9_1xGQPZ8BXWU0',
↪'CHROM': '2', 'POS': 29416366, 'END': 29416366, 'REF': 'G', 'ALT': '[C]', 'GT': '0/0'},
↪ {'Row': 1, 'Col': 281159698, 'Sample':
↪'0x00922C4598840C041CB1BB19DC75231C969E9057F7E3B9CC04EE8B44E714B793_1xAFO897WL2I',
↪'CHROM': '2', 'POS': 29416572, 'END': 29416572, 'REF': 'T', 'ALT': '[C]', 'GT': '0/0'},
↪ {'Row': 3, 'Col': 281159698, 'Sample':
↪'0x00AFE0D460A52BC28399A89E68A62D4CF2D279B56F482166B11BA7F29AFB45C9_1xGQPZ8BXWU0',
↪'CHROM': '2', 'POS': 29416572, 'END': 29416572, 'REF': 'T', 'ALT': '[C]', 'GT': '0/0'},
↪ {'Row': 3, 'Col': 281188584, 'Sample':
↪'0x00AFE0D460A52BC28399A89E68A62D4CF2D279B56F482166B11BA7F29AFB45C9_1xGQPZ8BXWU0',
↪'CHROM': '2', 'POS': 29445458, 'END': 29445458, 'REF': 'G', 'ALT': '[T]', 'GT': '0/0'},
↪ {'Row': 1, 'Col': 281241093, 'Sample':
```

(continues on next page)

(continued from previous page)

```
↪'0x00922C4598840C041CB1BB19DC75231C969E9057F7E3B9CC04EE8B44E714B793_1xAF0897WL2I',
↪'CHROM': '2', 'POS': 29497967, 'END': 29497967, 'REF': 'G', 'ALT': '[A]', 'GT': '0/0'}
↪])
```

Results are returned as a list of tuples, where the length of the list will correspond to the number of intervals being queried. Each entry will consist of:

- Flattened start position of the genomic interval
- Flattened end position of the genomic interval
- List of variant calls represented as a dict

Some use cases may only require a flattened list of all the variant calls - the above data can be easily transformed to achieve that. Below we show an example of doing so, and also create a Pandas dataframe from the results

[27]:
```
import pandas as pd

x,y,calls = zip(*list)
flattened = [variant for sublist in calls for variant in sublist]
print(pd.DataFrame(flattened).to_markdown())
```

```
|    |   Row |        Col | Sample                                                                   ␣
↪                  |   CHROM |       POS |       END | REF    | ALT    | GT    |
|---:|------:|----------:|:------------------------------------------------------------------
↪------------------|--------:|---------:|---------:|:------|:------|:-----|
|  0 |     1 |  11188011 |␣
↪0x00922C4598840C041CB1BB19DC75231C969E9057F7E3B9CC04EE8B44E714B793_1xAF0897WL2I |      ␣
↪ 1 | 11188012 | 11188012 | C      | [T]    | 0/0   |
|  1 |     1 |  65310488 |␣
↪0x00922C4598840C041CB1BB19DC75231C969E9057F7E3B9CC04EE8B44E714B793_1xAF0897WL2I |      ␣
↪ 1 | 65310489 | 65310489 | T      | [C]    | 0/0   |
|  2 |     3 |  65310488 |␣
↪0x00AFE0D460A52BC28399A89E68A62D4CF2D279B56F482166B11BA7F29AFB45C9_1xGQPZ8BXWU0 |      ␣
↪ 1 | 65310489 | 65310489 | T      | [C]    | 0/0   |
|  3 |     3 | 281159492 |␣
↪0x00AFE0D460A52BC28399A89E68A62D4CF2D279B56F482166B11BA7F29AFB45C9_1xGQPZ8BXWU0 |      ␣
↪ 2 | 29416366 | 29416366 | G      | [C]    | 0/0   |
|  4 |     1 | 281159698 |␣
↪0x00922C4598840C041CB1BB19DC75231C969E9057F7E3B9CC04EE8B44E714B793_1xAF0897WL2I |      ␣
↪ 2 | 29416572 | 29416572 | T      | [C]    | 0/0   |
|  5 |     3 | 281159698 |␣
↪0x00AFE0D460A52BC28399A89E68A62D4CF2D279B56F482166B11BA7F29AFB45C9_1xGQPZ8BXWU0 |      ␣
↪ 2 | 29416572 | 29416572 | T      | [C]    | 0/0   |
|  6 |     3 | 281188584 |␣
↪0x00AFE0D460A52BC28399A89E68A62D4CF2D279B56F482166B11BA7F29AFB45C9_1xGQPZ8BXWU0 |      ␣
↪ 2 | 29445458 | 29445458 | G      | [T]    | 0/0   |
|  7 |     1 | 281241093 |␣
↪0x00922C4598840C041CB1BB19DC75231C969E9057F7E3B9CC04EE8B44E714B793_1xAF0897WL2I |      ␣
↪ 2 | 29497967 | 29497967 | G      | [A]    | 0/0   |
```

CHAPTER

NINE

IMPORT / ETL EXAMPLES

CHAPTER

TEN

USING GENOMICSDB WITH GATK

GenomicsDB is packaged into gatk4 and benefits qualitatively from a large user base.

GATK tools such as GenomicsDBImport, SelectVariants and GenotypeGVCFs interact with GenomicsDB workspaces. Refer to the tool index for more information on how to use these tools.

The *GenomicsDB CLI tools* can be used to modify or query workspaces created using GATK tools.

INDEX

G

GenomicsDB (*C++ class*), 22
GenomicsDB::~GenomicsDB (*C++ function*), 23
GenomicsDB::GenomicsDB (*C++ function*), 23

O

org::genomicsdb::model::BatchCompletionCallbackFunctionArgument (*C++ class*), 30
org::genomicsdb::model::ImportConfig (*C++ class*), 29
org::genomicsdb::model::ImportConfig::Func (*C++ class*), 30
org::genomicsdb::reader::GenomicsDBQuery (*C++ class*), 31
org::genomicsdb::reader::GenomicsDBQuery::Interval (*C++ class*), 31
org::genomicsdb::reader::GenomicsDBQuery::Pair (*C++ class*), 31
org::genomicsdb::reader::GenomicsDBQuery::VariantCall (*C++ class*), 32

P

PhonyNameDueToError::columnPartitionIterator (*C++ function*), 28
PhonyNameDueToError::doSingleImport (*C++ function*), 27
PhonyNameDueToError::executeImport (*C++ function*), 28
PhonyNameDueToError::getHeader (*C++ function*), 31
PhonyNameDueToError::getNumExhaustedBufferStreams (*C++ function*), 28
PhonyNameDueToError::getSequenceNames (*C++ function*), 31
PhonyNameDueToError::isDone (*C++ function*), 28
PhonyNameDueToError::iterator (*C++ function*), 31
PhonyNameDueToError::setupGenomicsDBImporter (*C++ function*), 27
PhonyNameDueToError::write (*C++ function*), 29