Indexing Genomic Variants

Table of Contents:

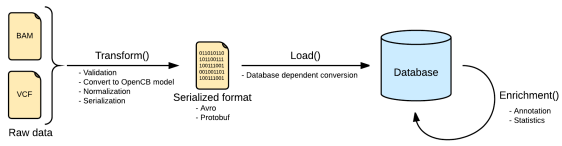
Overview

An index pipeline is the process of ingesting data into an OpenCGA-Storage backend. We define a general pipeline that is used and extended for the supported bioformats like variants and alignments. This pipeline is extended by additional steps of enrichment.

This concept is represented in Catalog to help the tracking of this status in different files.

Index

Indexing data pipeline consists in two steps, first transform and validate the input raw data into an intermediate format, and second, load it into the selected database. The input file format is VCF, accepting different variations like gVCF or aggregated VCFs



Transform

Files are converted Biodata models. The metadata and the data are serialized into two separated files. The metadata is stored into a file named *<inputFileName>.file.json.gz* serializing in json a single instance of the biodata model VariantSource, which mainly contains the header and some general stats. Along with this file, the real variants data is stored in a file named *<inputFileName>.variants.avro.gz* with a set of variant records described as the biodata model Variant.

VCF files are read using the library HTSJDK, which provides a syntactic validation of the data. Further actions on the validation will be taken, like duplicate or overlapping variants detection.

By default, malformed variants will be skipped and written into a third optional file named *<inputFileName >.malformed.txt* . If the transform step generates this file, a curation process should be taken to repair the file. Otherwise, the variants would be skipped.

All the variants in the transform step will be normalized as defined here: Variant Normalization. This will help to unify the variants representation, since the VCF specification allows multiple ways of referring to a variant and some ambiguities.

Load

Loading variants from multiple files into a single database will effectively merge them. In most of the scenarios, with a good normalization, merging variants is strait forward. But in some other scenarios, with multiple alternates or overlapping variants, a more complex merge is needed in order to create a consistent database. This situations can be solved when loading the file configuring the merge mode, or *a posteriori* in the aggregation operation.

Loading process is dependent on the implementation. Here you can see some specific information for the two implemented back-ends.

Options

referenceGenome

Reference genome in FASTA format used during the normalization step for a complete left alignment.

normalizationSkip

Do not execute the normalization process. **WARN:** INDELs will be stored with the context base.

gvcf

Hint to indicate that the input file is in gVCF format.

family

Indicate that the files to be loaded are part of a family. This will set loadHomRef to YES if it was in AUTO and execute 'family-index' afterwards.

loadHomRef

Load HOM_REF genotypes. (yes, no, auto)

Default

auto

loadSampleIndex

Build sample index while loading. (yes, no, auto)

Default

auto

loadSplitData

Indicate that the variants from a group of samples is split in multiple files, either by **CHROMOSOME** or by **REGION**. In either case, variants from different files must not overlap.

loadMultiFileData

Indicate the presence of multiple files for the same sample. Each file could be the result of a different vcf-caller or experiment over the same sample.

loadArchive

Load archive data. (yes, no, auto)

Default

auto

excludeGenotype

Do not include the genotype information.

includeSampleData

Index including other sample data fields (i.e. FORMAT fields). Use "all", "none", or CSV with the fields to load.

Default

all

postLoadCheck

Execute post load checks over the database

Default

auto

Storage Engines

OpenCGA offers two different implementations for the StorageEngine that use two different backend databases, each of one with particular properties.

MongoDB

The MongoDB implementation stores all the variant information in one centralised collection, with some secondary helper collections. In order to merge correctly new variants with the already existing data, the engine uses a stage collection to keep track of the already loaded data. In case of loading multiple files at the same time, this files will first be written into this stage collection, and then, moved to the variants collection, all at the same time.

Using this stage collection, the engine is able to solve the complex merge situations when loading the file, without the need of an extra aggregation step. Therefore, this storage engine does not implement the aggregation operation. Depending on the level of detail required, the merge mode can be configured when loading the files.

Merge Mode

For each variant that we load we have to check if the it already exists in the database, and, in that case, merge the new data with the existing variant. Otherwise, create a new variant.

We may find two types of overlapping variants: Variants from different sources that are in the same position with same reference and alternate (same variant), and variants that are not same but their coordinates (over the reference genome) are overlapping.

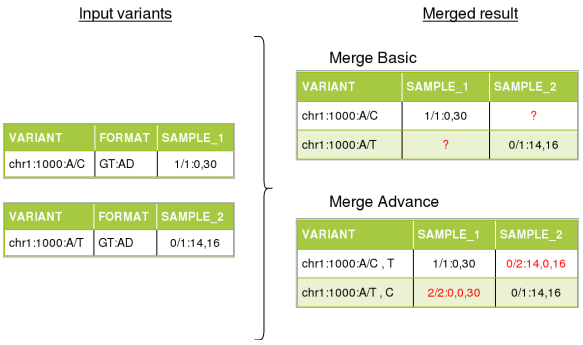
At this point is when we define two modes of merging variants:

- **Basic merge**. Only merging variants from different sources that are the same.
- **Advanced merge**. In addition to the basic merge, add the overlapping variants as secondary alternates and rearrange genotypes.

It is expected to have more unknown values for basic merge than for advanced merge.

Example

In the next figure we can see an example of merging multiple variants, from different single-sample files.



Input variants

VARIANT	FORMAT	SAMPLE_1
chr1:1000:A/C	GT:AD	1/1:0,30

VARIANT	FORMAT	SAMPLE_2
chr1:1000:A/T	GT:AD	0/1:14,16

Merged result

Merge Basic

VARIANT	SAMPLE_1	SAMPLE_2
chr1:1000:A/C	1/1:0,30	?
chr1:1000:A/T	?	0/1:14,16

Merge Advance

VARIANT	SAMPLE_1	SAMPLE_2
chr1:1000:A/C , T	1/1:0,30	0/2:14,0,16
chr1:1000:A/T , C	2/2:0,0,30	0/1:14,16

On the left, the input files. On the right we can see the merge result, depending on the merge mode, differences in red.

For basic mode, there will be unknown values for certain positions. We can not determine if the value was missing (./.), reference (0/0), or a different genotype. The output value for unknown genotypes can be modified by the user when querying. By default, the missing genotype (./.) will be used.

In the advanced mode, the variants have gained a secondary alternate, and the field AD (Allele Depth) has been rearranged in order to match with the new allele order.

Performance advantages

Loading new files will be much faster with basic merge mode. Is is because we don't need now to check if the variant overlaps with any other already existing variant. We only need to know if the variant exists or not in the database, which takes a significant amount of time in advance mode.

How to configure the merge mode

The merge mode is defined when the first file is loaded, and can not be changed.

From the command line we should add **--merge advanced** or **--merge basic**

Hadoop - HBase

The storage engine implementation for Hadoop is based on Apache HBase and Apache Phoenix. When loading a file, it will be stored (by default, entirely) in the **archive table**, and the variants (everything but the reference blocks) will be stored in the **variants table**, using a **basic merge mode**. Also, from each variant (unless otherwise specified) only samples with non homozygous reference (HOM_REF, 0/0) genotype will be loaded.

To obtain an advanced merge, including all the overlapping variants and the reference blocks, see the aggregation operation.

Most of the common queries will go to the variants table, but in case of requiring some extra information, the archive table can be also queried. There is also a third table that contains a secondary index for samples, to allow instant queries by genotype.

Table Naming Policy

- **Variants table**
  `<namespace>:<db-name>_variants`
- **Archive table**
  `<namespace>:<db-name>_archive_<study-id>`
- **Sample index table**
  `<namespace>:<db-name>_sample_index_<study-id>`
- **Metadata table**
  `<namespace>:<db-name>_meta`

Table compression algorithm

HBase supports multiple table compression algorithms natively. Compression algorithms can be configured for each of the tables. By default, SNAPPY compression is used.

- **opencga.variant.table.compression**
  Compression for the variant table.
- **opencga.archive.table.compression**
  Compression for the archive table.
- **opencga.sample-index.table.compression**
  Compression for the sample index table.

Table pre-splitting

Pre-splitting HBase tables is a common technique that reduces the number of splits and provides a better balance of the regions across the Hadoop cluster. We can configure the number of pre-splits for each of the tables.

- **opencga.variant.table.presplit.size**
  Pre-split size for the variant table.
- **opencga.archive.table.presplit.size**
  Pre-split size for the archive table.

In order to do an optimal pre-splitting, the storage engine needs to know an approximation of the number of files to be loaded. This number can be configured with:

- **expected_files_number**
  By default, 5000

Samples index table

With the Apache Solr secondary indexes we can query by any annotation field in HBase in subsecond time. But this can not help when querying by sample (or genotype).

Detailed information available here:

- Index Sample Genotypes in HBase (#838)
- Genotype index intersect in HBase (#862)
- Use variants SampleIndex when reading from MapReduce (#868)

Load options

Reduce archive data

By default, the engine writes in the archive table all the information from the variants that are reference blocks with HOM_REF genotype. This information represents, approximately between a 66% and a 90% of the original gVCF. So, reducing this part can have a big impact on the final size of the archive table. This feature can help to some installations with tied disk resources, or just because some information is not required at all for the analysis.

The fields to include can be configured using the following configuration parameter:

- **`opencga.archive.fields`**
  The default value is *all.* It can be configured with a list of fields to be included in the archive table.
    - *FILTER*
    - *QUAL*
    - *FORMAT* to include all the fields from the sample format, or *FORMAT:K1,K2, K3,...* to include specific fields
    - *INFO* to include all the fields from the file info, or *INFO:K1,K2,K3,...* to include specific fields
  - Example: `QUAL,INFO:DP,FORMAT:GT,AD,DP`
    The above line will only include the QUAL, the DP from the file attributes, the GT, AD and DP from the sample data

Load all reference variants from multi-sample files

When loading multi-sample files, from each variant only samples with non homozygous reference (HOM_REF, 0/0) genotype will be loaded. We can specify to load all the data from the samples using the following parameter:

- **opencga.variant.table.load.reference**
  The default falue is *false.* (#915)

Treat low quality reference calls as missing (no call)

Current implementation does not store reference calls in the second table *variants*. This allows to optimize disk space and improve performance. The assumption is that when a sample genotype is not present then it was a reference call since all the other genotypes including missing are stored.

The problem is that current variant callers are still far from being perfect and some variants having a reference call show a very low coverage or quality scores. So, in some use cases, users might need to confirm that *reference call* was good enough.

A simple solution for this would be treat low quality **reference** calls as **missing** calls, so they would be stored in the *variant* table in the same way than missing. By doing this users will know that not present reference calls have a good quality and there is no need to get them.

Users can configure low quality reference block in the configuration file, for instance DP<5 AND GQ<20.

- **`opencga.archive.non-ref.filter`**

  The filter is a list of key, operator, value, separated by ";".

  - Keys: `QUAL`, `FILTER`, `FORMAT:format-key`, `INFO:info-key`
  - Ops:
    - >, <, >=, <= : for numerical values
    - = for comma separated values
    - != Only for `FITLER` keys

  If a reference block does not have any of the required fields, will pass the filter, and will be treat as a missing.

  Examples:

  - `QUAL<5;FILTER=LowQual,LowGQ;FORMAT:DP<10`
  - `FILTER!=PASS`

Limitations

- You can not load two files with the same sample in the same study. See OpenCGA#158.
  There is an exception for this limitation for the scenarios where the variants were split in multiple files (by chromosome, by type, ...). In this case, you can use the parameter `--load-split-data`. SeeOpenCGA#696
- You can not index two files with the same name (e.g. */data/sample1/my.vcf.gz* and */data /sample2/my.vcf.gz*) in the same study. This limitation should not be a problem in any real scenario, where every VCF file usually has a different name. If two files have the same name, the most likely situation is that they contain the same samples, and this is already forbidden by the previous limitation.