INTERNATIONAL ORGANISATION FOR STANDARDISATION
ORGANISATION INTERNATIONALE DE NORMALISATION
ISO/IEC JTC1/SC29/WG11
CODING OF MOVING PICTURES AND AUDIO

ISO/IEC JTC1/SC29/WG11 MPEG2020/M52159
January 2020, Brussels, BE

Source	ISO/IEC JTC1/SC29/WG11
Status	Input Document
Title	**Proposal of a Unified File Format for the Coding of Genomic Annotations**
Authors	Shubham Chandak (Stanford University), Patrick Y.H. Cheung (Royal Philips)*, Qingxi Meng (University of Illinois at Urbana-Champaign), Mikel Hernaez (Center for Applied Medical Research at University of Navarra, UIUC), Idoia Ochoa (Tecnun at University of Navarra, UIUC)

* Corresponding Author

1. Introduction

Several biological studies produce genomic annotation data such as mapping statistics, quantitative browser tracks, variants, genome functional annotations, gene expression data and Hi-C contact matrices. These are currently represented in different formats such as VCF, BED, WIG, etc., leading to issues with interoperability and the need for frequent conversions between formats in order to visualize this data. Furthermore, the lack of a single format has stifled the work on compression algorithms and has led to the widespread use of suboptimal compression algorithms based on gzip (e.g., BCF [1], BigWig [2], etc.) that do not exploit the significant structure present in these formats. These algorithms do not exploit the fact the annotation data typically comprises of multiple fields (attributes) with different statistical characteristics and instead compress them together. Thus, while these algorithms support efficient random access with respect to genome position, they do not allow extraction of specific fields without decompressing all the attributes. While there have been some works [3,4] on decomposing the data into attributes and compressing them independently, there is limited adoption due to lack of standardization. These works are unable to achieve optimal compression because of reliance on a small set of standard compressors and are limited to only one type of annotation data. Furthermore, the existing solutions lack in support for features such as selective encryption and ability to link multiple annotation datasets with each other and with sequencing data. Many of these specialized solutions are based on disk-based array management tools like TileDB and HDF5 which provide a good base framework but lack several high-level features like support for metadata, linkages and attribute-specific indexing.

We propose a unified file format capable of storing the annotation data, while allowing support for functionalities such as fast query, random access, multiple resolutions (zooms), selective encryption, authentication, access control and traceability. The format achieves significant compression gains over gzip by separating different attributes of the data and allowing the use of specialized compressors for these. Finally, the format supports metadata and linkages to the sequencing data associated with the annotations as well as linkages to other annotation data from the same study, allowing seamless integration with the existing MPEG-G file format for sequencing data [5].

The format represents the data as a multidimensional array (tables) with each cell consisting of multiple attributes. A single file can contain multiple such tables to support multiple resolutions of the same data. Each cell in the table consists of multiple attributes that are compressed separately for improved compression and selective access to attributes. The framework supports a variety of compressors specialized for different data types. The format also supports compression of one attribute using other attributes as side information/context. The format also supports embedding a compressor executable within the file format itself, with appropriate security protections. For multidimensional arrays, the format also supports additional dimension-specific attributes that share the same value for all cells across a dimension.

To achieve efficient random access, the array is divided into chunks. The chunks can be of a fixed size or variable size, with an option to use the same chunks for all attributes or not. An index allows fast access to any given position in the data by only decompressing the corresponding chunk. To support fast random access based on the values of certain attributes, one can also include attribute-specific indexes. The format also provides a mechanism for sharing of codebooks or statistical models needed for decompression across chunks.

Finally, the format consists of protection (access control) information at multiple levels in the hierarchy that allows fine-grained security settings. Similarly, the metadata and attributes allow an effective way to link different types of annotation data as well as sequencing datasets. The format can be used as a standalone file or as part of an MPEG-G file. Overall, the proposed format provides a standardized framework with sufficient flexibility to achieve state-of-the-art compression performance on a variety of data types by incorporating the appropriate compression techniques for the attributes in question.

2. File Format and Technology

In this section, we first introduce some terminology and then describe the different components of the file format following a top-down approach. The various features described above are described in the relevant components of the file. In Section 2.8, we discuss the integration into a MPEG-G file, linkages and access control. In Section 2.9 we describe the decompression process which illustrates several of the advantages of the file format. Finally, Section 2.10 discusses methods to open the file in edit mode, allowing efficient updates to parts of the file.

2.1 Terminology

Annotation file: The top-level structure which can consist of multiple tables along with some additional metadata and protection information. For example, the multiple tables can be used to store the data at multiple resolutions.

Table: Each table is an independent entity, storing an array consisting of different attributes.

Attribute: Each cell in a table can store multiple attributes, where each attribute has a specific datatype and is compressed using a specific compressor. This allows better compression and also selective access to attributes. For example, in a genome functional annotation file, the attributes could be chromosome, start position, end position, feature ID, feature name and so on.

Dimension: Tables can be single dimensional (e.g., genome annotation data, quantitative browser tracks) or multidimensional (e.g., Variant call data, gene expression data for multiple samples are 2-dimensional). Note that single dimensional tables can also hold multiple attributes. See Figure 1 below for an illustration.

Dimension-specific attributes: When the array is multidimensional, the table might store certain dimension specific attributes along with the attributes for the main table. Each dimension specific attribute can be thought of as being part of a 1-dimensional table. For example, variant data for multiple samples can be represented using a 2-dimensional table, with the sample genotypes and sample level likelihoods being attributes for the main 2-d array, while the variant position and sample name being dimension specific attributes. See Figure 1 below for an illustration.

Chunk: The attributes in the table are compressed in rectangular chunks to allow efficient random access. The chunks can be of fixed size or variable size. An index is stored for efficiently determining the position of a specific chunk in the compressed file.

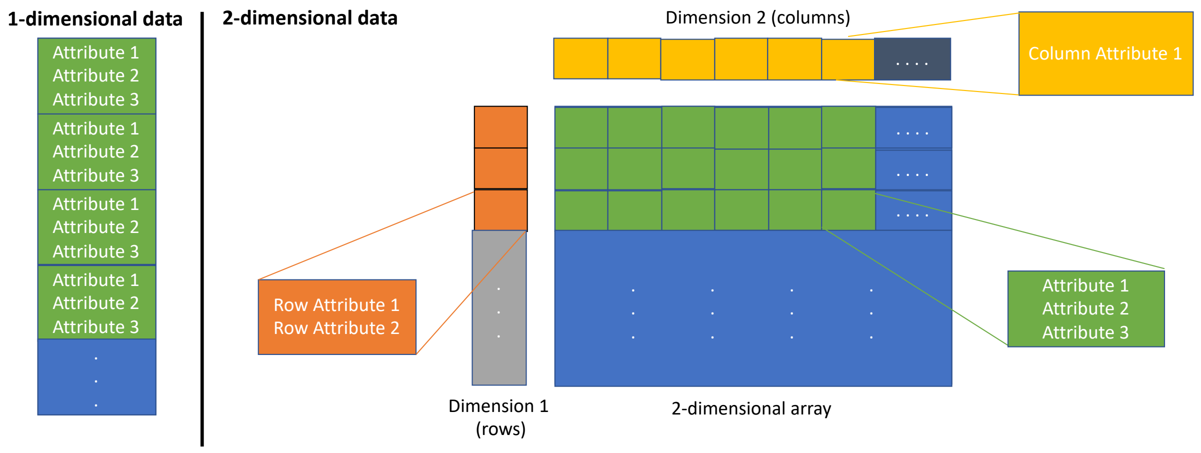


Figure 1: Illustration of 1-dimensional and 2-dimensional data with multiple attributes. The 2-dimensional data contains dimension-specific attributes in addition to the main 2-dimensional array.

2.2 Top-Level File Format

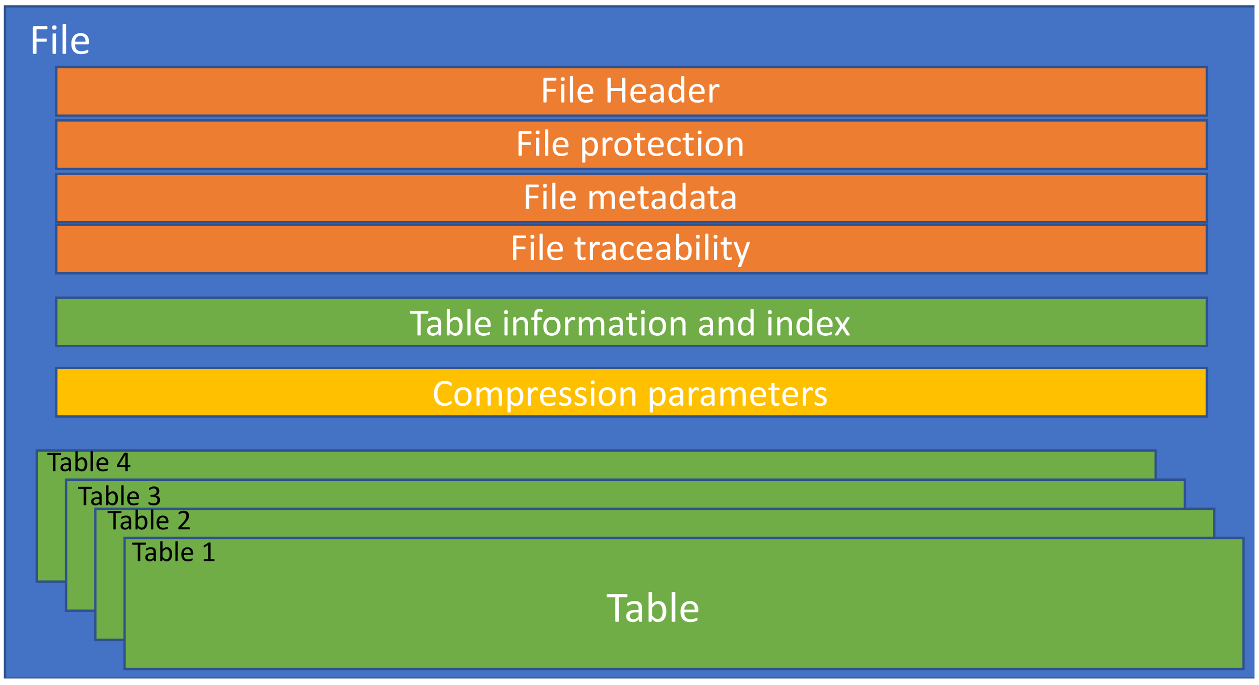


Figure 2: Illustration of top-level file format.

§T1: Annotation file

Field	Brief Description	Type
FileHeader		file_header (§T2)
FileProtectionInfo	Access control policy	gen_info (§T3)
FileMetadata	Metadata/Linkage	gen_info (§T3)
FileTraceabilityInfo	Commands used to generate data	gen_info (§T3)
nTables	Number of tables stored in file (e.g., multiple resolutions)	
For i in 1…nTables:		
TableID[i]	Unique table identifier	Integer
TableInfo[i]	Table information (e.g., resolution)	gen_info (§T3)
ByteOffset[i]	Byte offset of table i in file	Integer
nCompressors	Number of distinct compressors used in the different attributes stored later	Integer
For i in nCompressors:		
Compressor[i]		comp_info (§T4)
For i in 1…nTables:		
Table[i]		table (§T5)

§T2: File header (file_header)

Field	Brief Description	Type
FileName		String
FileType	e.g. "Variant", "Gene Expression", etc.	String
FileVersion	For keeping track of updates	String

§T3: General information structure (gen_info)

Field	Brief Description	Type
PayloadSize	To allow skipping over this	Integer
Payload	Compressed with predefined compressor (e.g., 7z)	Bytes

Description:

- The file format supports storage of protection information for **access control, metadata, versioning** and **traceability**. While these allow a generic representation compressed with 7zip, in practice one would typically use standard **JSON/XML/XACML based schemas** for this information along with standard **URI** (uniform resource identifier) notation (e.g., as done in MPEG-G part 3 [5]). See Section 2.8 for more details.
- The proposed file format stores the annotation data in **multiple tables**, where different tables can be used to store the data at different **resolutions**, among other possible applications. The basic information about the table such as the resolution level can be extracted without needing to read the whole file using the TableInfo field in some standard

JSON/XML-like format. Similarly, the byte offset of the tables in the compressed file are available to directly jump to a specific table.

- The file stores a list of compressors indexed by unique identifiers. These can be referred to in the tables, thus avoiding repeated description of compressors used in multiple tables or for multiple attributes. See Section 2.3 for details.
- Finally, the file stores the tables (Section 2.4).

2.3 Compressors

§T4: Compressor information structure (comp_info)

Field	**Brief Description**	**Type**
CompressorID	Unique compressor identifier	String
nDependencies	To allows compression of an attribute based on values of other attributes	Integer
CompressorNameList	List of compressor names (or “EMBEDDED”)	List(String)
CompressorParametersList	Parameters required for decompression	List(gen_info)

Description:
This structure stores the description of a compressor. The unique CompressorID is used within the tables to point to a compressor. The compressor name and parameters can be used for a standard compressor (listed below) or it can be used for describing the decompression mechanism within the CompressorParameters by setting CompressorName to “EMBEDDED”. Multiple compressors can be applied in sequence using a list.

The format supports compression of an attribute using other attributes as side information (as long as there is no cyclic dependency). The variable nDependencies denotes the number of dependency attributes needed for the decompression (the corresponding attributeIDs are specified in §T6). Compressors like context-based arithmetic coding can easily support the incorporation of side information. Another mechanism to incorporate the side information from other attributes is to reorder or split the values of the current attribute based on the values of the other attributes, which can bring together similar values and provide better compression. The parameters describe the dependencies used by each compressor in the list.

The attribute information structure (§T6) supports storage of additional data required for decompression, which is common to all chunks, in the variable CompressorCommonData. This can be useful for storing codebooks, dictionaries or statistical models computed from the entire data.

Non-exhaustive list of standard compressors:

- **Run length encoding:** for long runs with same value, replace by value and length of run.
- **Delta encoding:** for increasing sequences of numerical values, replace by difference between consecutive values.
- **Dictionary-based/enumeration**: for attributes taking values from a small set of options, replace by index in the set and store the dictionary in CompressorCommonData.
- **Sparse:** for attributes that rarely differ from the default value (specified in §T6), represent as coordinate position and value for the non-default values. The coordinate positions can be further delta coded within each chunk to improve compression, for example, in a 2-dimensional sparse array, the row index can be delta coded and the column index can be delta coded within each row.
- **Variable length array:** separate variable length arrays into value stream and length stream.

- **Tokenization:** for structured string attributes, split into tokens of different types and encode each token in terms of previous value (e.g., match with previous token, delta, new value, etc.).
- **No compression:** can be useful for faster selective access.
- **General purpose compression/entropy coding methods:** gzip, bzip2, 7-zip, adaptive arithmetic coding, BSC (http://libbsc.com/).

Note that this list is not exhaustive and specialized compressors for different attributes can be supported for certain applications. For example, several specialized compressors such as GTC [6] exist for genotype data in variant call supporting fast random access to rows/columns.

Some compressors above produce multiple streams, e.g., the coordinates and values for the sparse compressor. These can be further compressed using different entropy coders by specifying the appropriate parameters. For example, if

```
CompressorNameList =         [‘sparse’,’gzip’,’7-zip’]
CompressorParameterList =    [
                                  {“outStreams”: [“coordinate”, “value”]},
                                  {“inStreams” : [“coordinate”]},
                                  {“inStreams” : [“value”]}
                             ]
```

then gzip is applied to the coordinate stream and 7-zip is applied to the value stream (here we used JSON for representing the parameters). This enables the application of optimal compressors for each data stream. If the streams are not specified, the compression is applied to all the incoming streams.

Embedded compressor: In case of embedded compressors, the decompression executable is put in the compression parameters along with the digital signature as a proof of origin and authenticity to protect against malicious software. For interoperability across different platforms, a standardized virtual machine bytecode should be used for the decompression executable.

2.4 Table

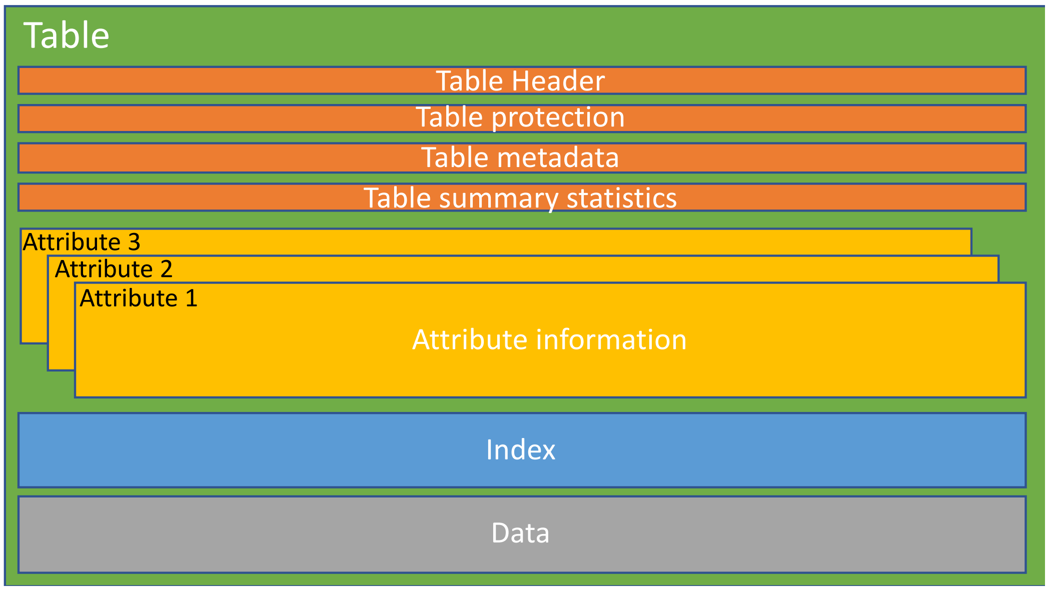


Figure 3: Illustration of table structure for the one-dimensional case.

§T5: Table

Field	Brief Description	Type
TableID	Same as in §T1	gen_info (§T3)
TableInfo	Same as in §T1	gen_info (§T3)
TableProtection	Access control policy	gen_info (§T3)
TableMetadata	Metadata/Linkage	gen_info (§T3)
SummaryStatistics	e.g., Count, Average value	List(Key-value)
nDimensions	Number of dimensions	Integer
For i in 1…nDimensions:		
Size[i]	Size of dimension i	Integer
DimensionName[i]		
DimensionMetadata[i]	Metadata/Linkage	gen_info (§T3)
If nDimensions == 2:		
SymmetryFlag	True if 2d array is symmetric	Bool
nAttributesMain		
For i in 1…nAttributesMain:		
AttributeInfoMain [i]		attr_info (§T6)
ByteOffsetMain	Byte offset of IndexMain	Integer
If nDimensions > 1:	Dimension-specific attributes	
For i in 1…nDimensions:		
nAttributesDim[i]	Number of dimension specific attributes	Integer
For j in 1…nAttributesDim[i]:		
AttributeInfoDim[i][j]		attr_info (§T6)
ByteOffsetDim[i]	Byte offset of IndexDim[i]	Integer
IndexMain		index (§T7)
DataPayloadsMain		data (§T9)
If nDimensions > 1:	Dimension-specific attributes	
For i in 1…nDimensions:	// each of these are treated as a 1-d array	
IndexDim[i]		index (§T7)
DataPayloadsDim[i]		data (§T9)

Description:

- Just like the top-level file structure, each table consists of **access control** and **metadata** (discussed in Section 2.8). The table also contains some **summary statistics** (typically averages, counts or distributions) for fast access.
- For each **dimension** within the table, we store the size, name, metadata. For the 2-dimensional case, we also store a flag denoting whether the matrix is symmetric (e.g., Hi-C data which is symmetric).
- We have the attributes for the main table – for these we store the information (Section 2.5) and also the byte offset of the data for the main table.
- This is followed by a list of dimension-specific attributes (Section 2.5). We also store the byte offset of the data for each dimension, allowing selective access to the attributes for a particular dimension.

- Finally, the table stores the index and data for the main table and each dimension (if applicable).
- Note that dimension-specific attributes are considered to be one-dimensional arrays in the following sections for chunking, indexing etc.

2.5 Attributes

§T6: Attribute information structure (attr_info)

Field	**Brief Description**	**Type**
AttributeInfoSize	To allow skipping over structure	Integer
AttributeID	Unique attribute identifier	Integer
AttributeName		String
AttributeMetadata	Metadata/Linkage/Grouping of attributes	gen_info (§T3)
AttributeType	Fundamental types (e.g., int, char, float, string) or derived types (e.g., fixed-length, variable-length array)	String
DefaultValue	For sparse encoding if most values match default	Attribute Type
SummaryStatistics	e.g., Count, Average value	List(Key-value)
CompressorID	Compressor used for this attribute	Integer
For i in 1…nDependencies:	nDependencies defined in §T4	
If nDimensions > 1:		
Dimension	If this is an attribute of the main n-dimensional table, this tells which dimension contains the dependency attribute (set to nDimensions+1 if dependency attribute is also in main n-dimensional table)	Integer
AttributeID	Attribute ID containing the dependency	Integer
CompressorCommonDataSize		Integer
CompressorCommonData	To store codebooks/statistical models for the compressor that are common to all chunks	Bytes

Description:

- For each attribute, we specify the unique identifier, name and metadata.
- The **AttributeType** can be either
  - a fundamental type like character, string (null terminated), float, double, Boolean, signed and unsigned integers with different bitwidths.
  - Derived type like variable length or fixed length arrays.
- The **DefaultValue** of the attribute allows us to use sparse encoding when most values are equal to the default.
- Each attribute can contain certain **summary statistics** (typically averages, counts or distributions) for fast access.
- The compression method used for the attribute is specified using the compressorID. In case the compressor uses side information/context during the decompression process, the corresponding dependency attributes must also be specified. In case of multidimensional arrays, the side information can either be obtained from the multidimensional array attributes or from a dimension specific attribute. For example, in a VCF file, one could use a variant specific field (which is a dimension specific attribute) as side information for compression of genotype data (which is an attribute of 2-dimensional main table).
- As previously mentioned in Section 2.3, the attribute information structure (§T6) supports storage of additional data required for decompression, which is common to all chunks, in

the variable CompressorCommonData. This can be useful for storing codebooks, dictionaries or statistical models computed from the entire data.

2.6 Chunks and Indexing Structure

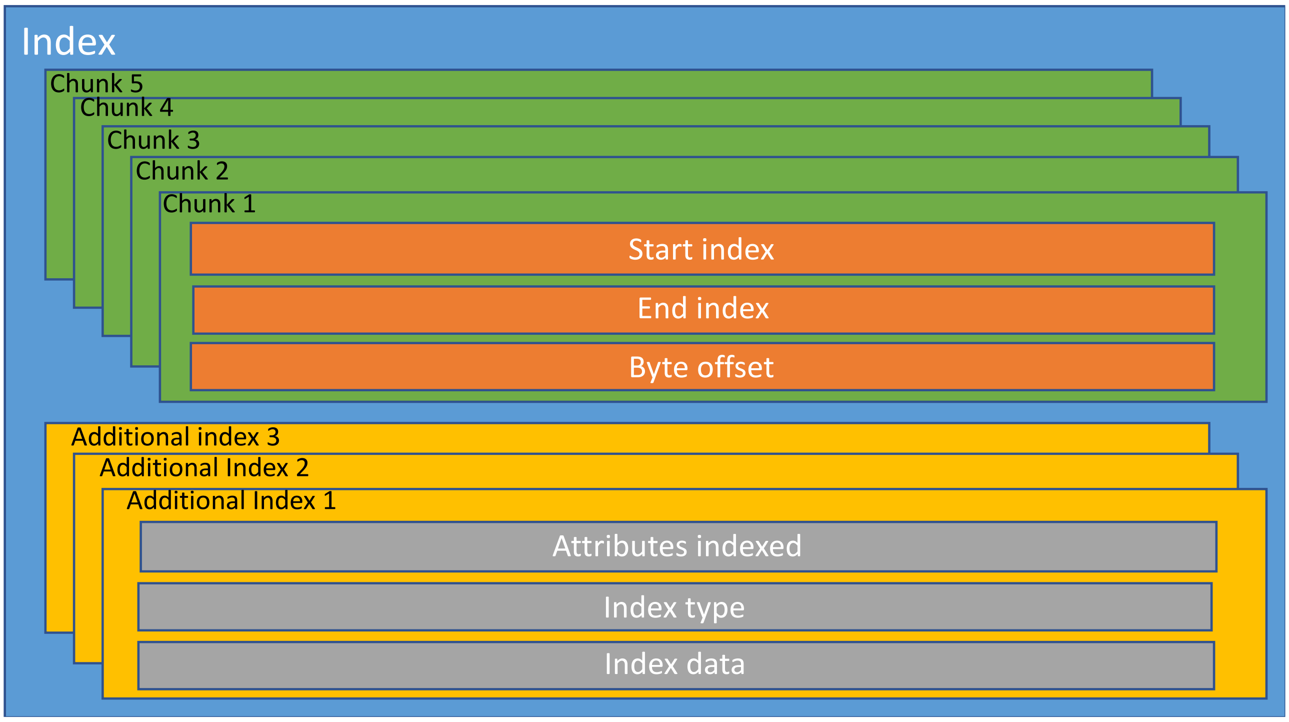


Figure 4: Illustration of index structure for the one-dimensional case when the flag AttributeDependentChunks is False.

§T7: Index structure (index)

Field	**Brief Description**	**Type**
AttributeDependentChunks	Flag denoting whether chunks sizes are dependent on the attributes or if same chunking is used for all atttributes	Bool
If not AttributeDependentChunks:		
ChunksStructure		chunks (§T8)
Else:		
For i in 1…nAttributes:		
ChunksStructure[i]		
// Additional attribute specific indexes		
nAdditionalIndexes	Number of additional indexes for faster query based on certain attributes (e.g., chromosome and position) – these return the chunk number(s) containing the desired query results	Integer
For i in 1…nAdditionalIndexes:		
AttributeIDsIndexed[i]	List of attributes indexed	List(Integer)
IndexType[i]	Index type (e.g., CSI index for chromosome and genomic position or B-	String

	tree for database type queries)	
IndexSize[i]	To allow skipping over index	Integer
IndexData[i]	Actual index data, specifics depend on IndexType[i]	Bytes

§T8: Chunks structure (chunks)

Field	**Brief Description**	**Type**
nChunks	Number of chunks	Integer
VariableSizeChunks	Flag denoting whether chunks sizes are variable or fixed (except at the boundary of each dimension)	Bool
If VariableSizeChunks:		
For j in 1…nChunks:		
For k in 1…nDimensions:		
StartIndex[j][k]	Start position of chunk along dimension k	Integer
EndIndex[j][k]	End position of chunk along dimension k	Integer
ByteOffset[j]	Byte offset of chunk j in file	Integer
Else:		
For k in 1…nDimensions:		
ChunkSize[k]	For fixed size chunks, sufficient to store size of chunk in each dimension	Integer
For j in 1…nChunks:		
ByteOffset[j]	Byte offset of chunk i in file	Integer

Description:

- Depending on whether **AttributeDependentChunks** is true, we can use the same chunking for all attributes or attribute dependent chunking.
    - Using the same chunking for all attributes requires much smaller index structure and is useful when most of the time all attributes within a chunk are queried.
    - Using attribute dependent chunking is useful when the optimal chunk size for different attributes with respect to compression and random access varies a lot. For example, if some attributes are sparse while others are dense, using the same chunk size might lead to suboptimal compression. It can also be useful when most of the time all chunks for a single attribute are queried.
    - The organization of the chunks and attributes depends on the mode of operation, as shown in Figures 7 and 8 and in §T9.
- The rectangular chunks can be **fixed size or variable size** depending on the VariableSizeChunks flag. While fixed size chunks are simpler to deal with, especially for multidimensional tables, variable size chunks can be useful when the sparsity of the data is highly varying and hence choosing a single chunk size is not optimal. In some cases, variable size chunks can allow chunks based on an attribute such as chromosome/genome position, which can allow faster random access with respect to those attributes.
- In case of variable size chunks, we store the start and end index in the table for each chunk along each dimension. In case of fixed size chunks, we just need to store the chunk size along each dimension. In both cases, we store the **byte offset of each chunk** in the file for random access. Figures 5 and 6 illustrate the chunks and the corresponding index.

- In a number of applications, random access with respect to row number or column number is not meaningful, instead **random access with respect to certain attributes** is desired. For example, random access with respect to genome position is frequently required. The proposed format supports a flexible mechanism for such applications. We can store any number of additional attribute specific indices by providing the type of the index from a standard set (e.g., B-tree for database type queries, R-trees or CSI index [7] for range queries), the attributeIDs (e.g. chromosome, position) and the actual indexing data stored in a binary format. The genomic range indexing can store the leftmost and rightmost coordinate for each chunk, allowing quick identification of the chunks overlapping the queried range. Similarly, the B-tree index can store a map from the attribute value to the chunk containing the value and the position of the value within the chunk.The lookup based on these works as follows:
    - The user specifies a query (e.g., attribute=”abcd” or attribute between 1 and 10000, etc.).
    - The attribute specific index returns the chunk number(s) that contain values that match the query condition.
    - Then these chunks are recovered using the chunk index, filtering out values that match the condition (because chunks can also contain non-matching values).
- Note that the data in different chunks are compressed independently. However, global compression data can be shared across chunks using the CompressorCommonData mechanism (§T6).
- For symmetric 2d arrays (when SymmetryFlag in §T5 is true), the chunks only need to cover the lower triangular part and the diagonal. The decompression process takes care of the upper diagonal values by filling in the corresponding lower triangular values. For all other cases, the chunks must cover the entire range of indices without overlapping.

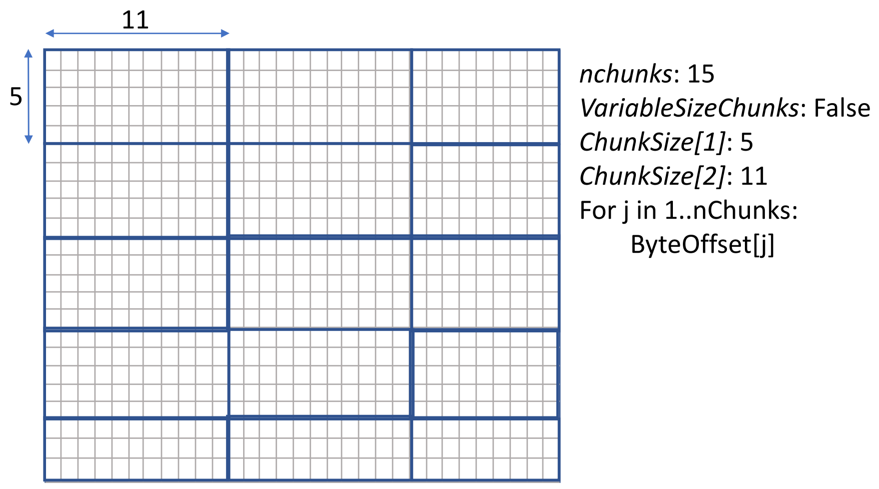
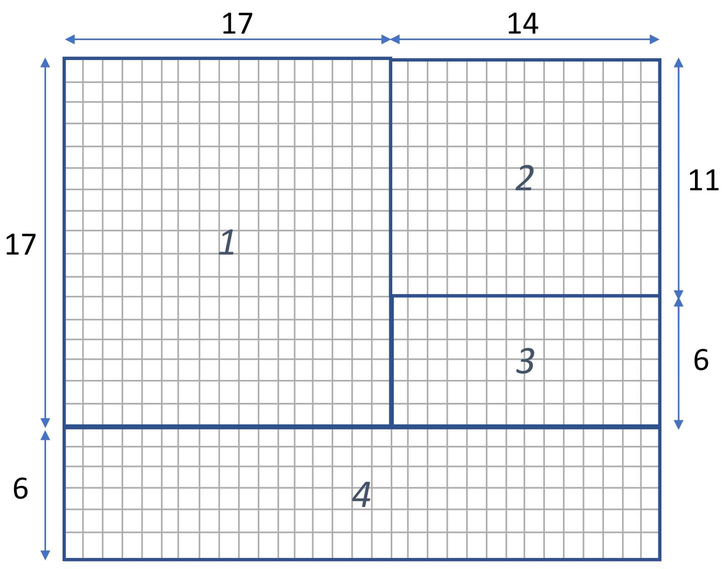


Figure 5: Illustration of fixed size chunks and the corresponding indexing data for a 2-dimensional array.



nchunks:	4
VariableSizeChunks:	True
// Chunk 1	
StartIndex[1][1]:	1
EndIndex[1][1]:	17
StartIndex[1][2]:	1
EndIndex[1][2]:	17
ByteOffset[1]	
// Chunk 2	
StartIndex[2][1]:	1
EndIndex[2][1]:	11
StartIndex[2][2]:	18
EndIndex[2][2]:	31
ByteOffset[2]	
.	
.	
.	

Figure 6: Illustration of variable size chunks and the corresponding indexing data for a 2-dimensional array.

2.7 Data Payloads

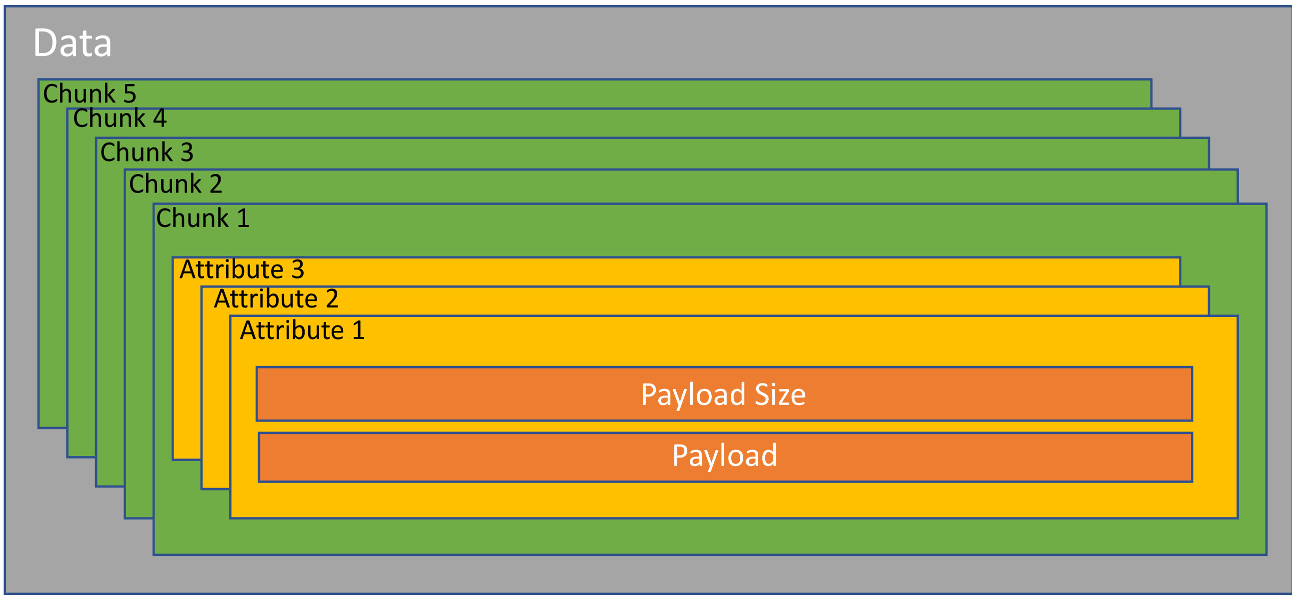


Figure 7: Illustration of data payload structure for the one-dimensional case when the flag AttributeDependentChunks is False.

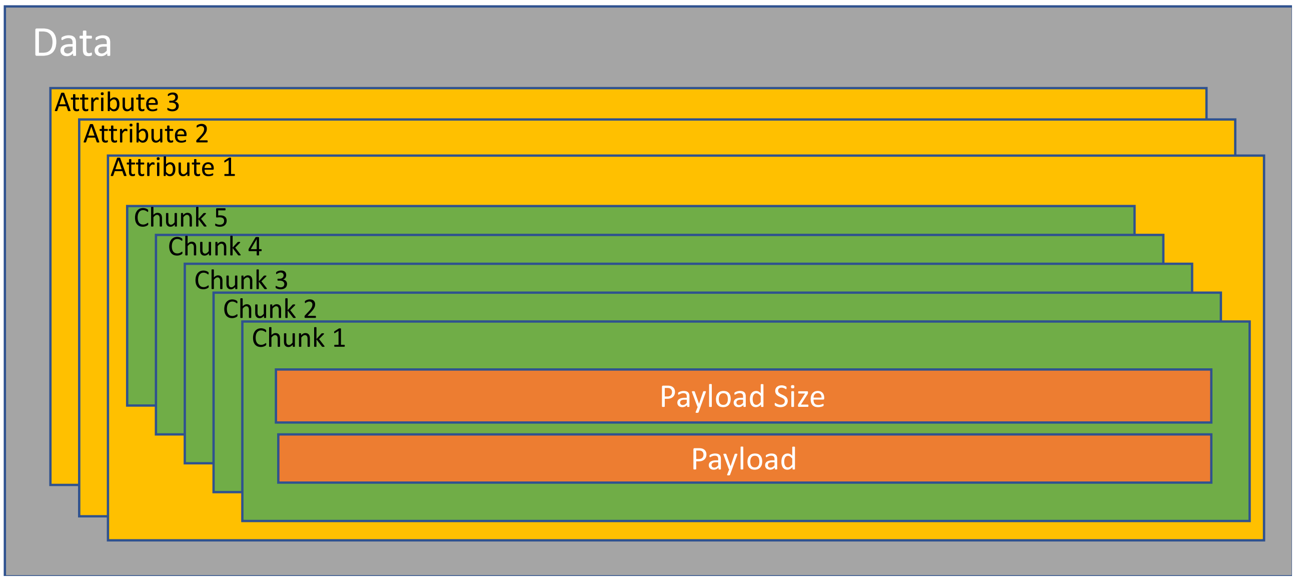


Figure 8: Illustration of data payload structure for the one-dimensional case when the flag AttributeDependentChunks is True.

§T9: Data Payloads

Field	Brief Description	Type
If not AttributeDependentChunks:		
For i in 1…nChunks:		
For j in 1…nAttributes:		
Payload Size[i][j]	To allow skipping over certain attributes	Integer
Payload[i][j]	Compressed payload	Bytes
Else:		
For j in 1…nAttributes:		
For i in 1…nChunks:		
Payload Size[j][i]		Integer
Payload[i][j]	Compressed payload	Bytes

Figures 7 and 8 illustrate two modes of storing the data based on the flag **AttributeDependentChunks**. The pros and cons of these modes are discussed in Section 2.6.

2.8 Linkages, Interoperability with MPEG-G and Access Control

2.8.1 Organization within MPEG-G file

While we describe the format as an independent file format here, it can also be used as part of an MPEG-G file by storing it in a dataset. Note that an MPEG-G file can store the data for an entire study, with each dataset group typically corresponding to an individual. Each MPEG-G dataset group is further divided into datasets corresponding to different sequencing runs.

For storing the data corresponding to a **single individual**, the different annotation files can be incorporated as distinct datasets as shown below, each dataset containing a single annotation file or sequencing data.

```
Dataset group (single individual) -->
                                        Dataset 1 (sequencing data)
                                        Dataset 2 (sequencing data)
                                        Dataset 3 (variant call data)
                                        Dataset 4 (gene expression data)
                                        …
```

For collecting annotation data from a **larger study**, we can organize as follows:

```
Dataset group (large study) -->
                                        Dataset 1 (variant call data)      -->
                                                                             Annotation file (sample 1)
                                                                             Annotation file (sample 2)
                                                                             …
                                        Dataset 2 (gene expression data) -->
                                                                             Annotation file (sample 1)
                                                                             Annotation file (sample 2)
                                                                             …
                                        …
```

Note that the different annotation files can be **merged together** for improved compression and analysis performance.

```
Dataset group (large study) -->
                                        Dataset 1 (variant call data)      -->
                                                                            Annotation file (all samples)
                                        Dataset 2 (gene expression data) -->
                                                                            Annotation file (all samples)
                                        …
```

The existing dataset header structure needs to be augmented with additional fields to support the data type (sequencing/variant/gene expression/…), the number of annotation files contained in the dataset, and the byte offset of each of these files.

When a compressor is shared across annotation files or across datasets, it's parameters can be stored at the dataset level or dataset group level, respectively. The annotation file in that case contains a compressor structure with compressor name "POINTER" and the compression parameter storing the location, e.g., `{"DatasetGroupId": 1, "DatasetId": 2, "CompressorId": 5}` denotes that the compressor is as specified in the 5th compressor in dataset group 1, dataset 2.

2.8.2 Linkages

The format provides a mechanism to store linkages between different types of annotation data and the corresponding sequencing data.

2.8.2.1 Metadata-based linkage

The dataset groups or datasets storing the sequencing data or the related annotation data can be specified in the FileMetadata or TableMetadata using a standard URI (uniform resource identifier) notation as described in MPEG-G part 3 [5] or using JSON. For example, to provide linkage to a sequencing dataset, the following JSON can be used in the FileMetadata:

```
"Linkages": [{
                "DataType"      :     "Sequencing",
                "DatasetGroup"  :     5,
                "Dataset"       :     2
            }]
```

While the example shows only a single linkage, one can have multiple linkages.

One can also have Table level linkages. There can be two types:

- **By index** – in this case, the nth row (column) in one table corresponds to the nth row (column) in another table. This can be useful to avoid repetition when multiple annotation files/tables share the same rows/columns (e.g., multiple VCFs that are not yet merged and consist of the same variants). Similarly, this is useful when the information about the samples is stored in a single table, and both VCF and gene expression tables link to this.
- **By value** – in this case, a specific attribute is linked by matching value to an attribute in another table. For example, the gene expression data might consist of only the gene names while the detailed information about the genes is available in another file. An example use case for such a linkage might be a query requesting gene expression data for all genes in the MHC (major histocompatibility complex), which corresponds to autoimmune diseases and specifies a range of coordinates in chromosome 6 for humans. To address this query, the gene names for the coordinate range can be obtained from the gene information file based on a genomic coordinate index and then these names can be queried in the gene expression file to the get the required data.

Examples:
Linking rows (dimension 1) with rows of another table (table no. 3 in same annotation file):

```
"Linkages": [{
                "Type"                      :     "byIndex",
                "DimensionInCurrentTable"   :     1,
                "Table"                     :     3,
                "DimensionInLinkedTable"    :     1
            }]
```

Linking columns (dimension 2) with rows of another table by value of attribute. (attribute 2 in dimension 2 of current table linked to attribute 5 in dimension 1 of table 3 in dataset 4, file 2).

```
"Linkages": [{
                "Type"                      :     "byValue",
                "DimensionInCurrentTable"   :     2,
                "AttributeInCurrentTable"   :     4,
                "Dataset"                   :     4,
                "AnnotationFile"            :     2,
                "Table"                     :     3,
                "DimensionInLinkedTable"    :     1,
                "AttributeInLinkedTable"    :     5,
            }]
```

Since the metadata structure supports arbitrary information storage, the framework can be extended even further to link more than 2 tables by using a standardized format (e.g., table 3 can translate the gene ids used in table 1 to the gene names in table 2). Also note that while the examples shown above use a specific JSON based format for linkages, one can also use other formats like XML.

2.8.2.2 Attribute-based linkage

The metadata-based linkage is useful for high level linkages, but in some cases, we need linkage for each row/column. For example, in a VCF file with multiple samples, the sequencing data corresponding to particular samples can be linked by adding attributes SequencingDatasetGroup and SequencingDataset to the column attributes. Such linkage attributes should have "LinkageAttributeFlag" set to True in the metadata to allow the decompressor to distinguish linkage attributes from normal attributes.

In some cases, there is a need to map between annotation datasets according to genomic region. In most cases, this should be achieved by separately indexing each of the datasets. Thus, to find the sequencing data corresponding to a region in the VCF file, one can look up the master index table of the sequencing data and find the appropriate access unit(s). Using separate indexing for different datasets allows the choice of optimal chunk sizes and other parameters for each of the datasets. Furthermore, in some cases direct linking of a variant to an AU might not be possible due to different AU classes. Similarly, in VCF files with multiple samples, the variant maps to the access units across several datasets and storing this information can take up significant storage.

If relevant, one can also store the AUId or byteoffset in the sequencing data as a row attribute in the VCF file, allowing quick lookup of the access unit corresponding to the current variant.

We can also map a gene to a list of variants by using a list-type attribute to the genes.

2.8.3 Access control

The access control policy can be specified at both the file level and the table level, typically using a standard format such as XACML.

Certain users might have access to all the data, while others might have access only to coarse resolution data (recall that different resolutions are stored in different tables). This type of policy should be specified at the file level.

On the other hand, policies specific to the attributes within a table should be specified at the table level. This can include access to only a subset of attributes, or access only to certain chunks based on the value of some attribute. Another type of policy could allow access to the metadata and information but not to the actual data.

2.9 Decompression Process

We next describe the query types supported and the corresponding decompression methods. These are not mutually exclusive and aspects of these can be combined together, e.g, decompressing both the metadata and certain attributes or decompressing selected attributes from selected chunks. Also note that the access control policy might restrict some of these queries. Standardized APIs similar to MPEG-G part 3 [5] can be used to support these.

Metadata/information queries

Only metadata and information about the tables (e.g., resolution level), compressors, attributes and/or chunks requested.

1. The top-level information in §T1 can be directly accessed at the beginning of the file.
2. The table-specific metadata/attribute details can be accessed by using the ByteOffset of the table specified in §T1.

Complete data decompression

Decompression of the entire data, including all tables and attributes.

1. First the top-level metadata and table information is read
2. Then the compression parameters are loaded
3. For each table:
    a. The table information, the dimensions and the attributes are read.
    b. The index is read to determine the positions of the chunks along each dimension
    c. The data payloads for each chunk and each attribute are decompressed (this process can be **parallelized**). If the attribute is compressed using another attribute as a dependency/context, then we first decompress the other attribute. If the attribute uses CompressorCommonData (§T6), that is loaded before decompressing any chunks.
    d. For 2-d symmetric arrays (see SymmetryFlag in §T5), we decompress only the diagonal and lower triangular matrix, filling in the upper triangular part using symmetry.

Decompression of only one table

Similar to "Complete data decompression" except that ByteOffset of the requested table (§T1) is used to jump to the table and only that table is decompressed.

Query for selected attributes of a table

Similar to "Decompression of only one table" except that

- Only the information about the requested attributes is read (skipping over other attribites using AttributeInfoSize variable in §T6).
- Only the requested attributes are decompressed by skipping over other attributes using Payload Size[i][j] in §T9. When attribute dependent chunks are used, all the chunks for a given attribute are stored together, and this process becomes straightforward (Figure 8).

Query only selected range of indices in the array

Similar to "Decompression of only one table" except that

1. The index is loaded and depending on the type of chunking (fixed size/variable size), the chunks overlapping with the requested range are determined.
2. The ByteOffset in §T8 is used to jump to the payload for the chunks determined above. The process is more efficient when attribute dependent chunks are not used and all the attributes for a given chunk are stored together.
3. The requested chunks are decompressed and only the overlapping indices are returned. Note that if the compressor of some attribute allows efficient random access within a chunk, we utilize this to further boost the decompression speed. Some cases where this might happen include sparse arrays or specialized compressors for genotypes such as GTC [6].

Query based on value/range of certain attributes

Similar to "Query only selected range of indices in the array" except that

1. If an additional attribute specific index (§T7) is available for the attributes in question, it is used to determine the relevant chunk(s).
2. If such an index is not available, we decompress the attributes in question for all the chunks and determine the relevant chunks. Note that even when an additional index is not used, we are still able to speed up the process since we only decompress some attributes for all the chunks. The rest of the attributes need to be decompressed only for the relevant chunks.

2.10 Folder Structure and Editing

The file format described above offers several advantages and is convenient for transmission, long-term storage and fast querying. However, in case the data is kept on a single machine and needs to be edited frequently, it is more suitable to store it in a directory/folder hierarchy using a file manager. The folder hierarchy allows easy manipulation of parts of the data by modifying only the files corresponding to a single chunk and attribute, rather than needing to overwrite the entire file. When the editing is completed and the data needs to be transmitted, it can be converted back to the single file format, which recomputes the index based on the data payload sizes and packs the folder hierarchy back into one file. The conversion from the file format to the folder hierarchy and back is straightforward, with each table becoming a folder, and within each table, each chunk becomes a folder (assuming AttributeDependentChunks is False). In the folder hierarchy, the index only needs to store the attribute-specific indexes since the chunks are already stored in distinct folders. A simple example is shown in Figure 9.

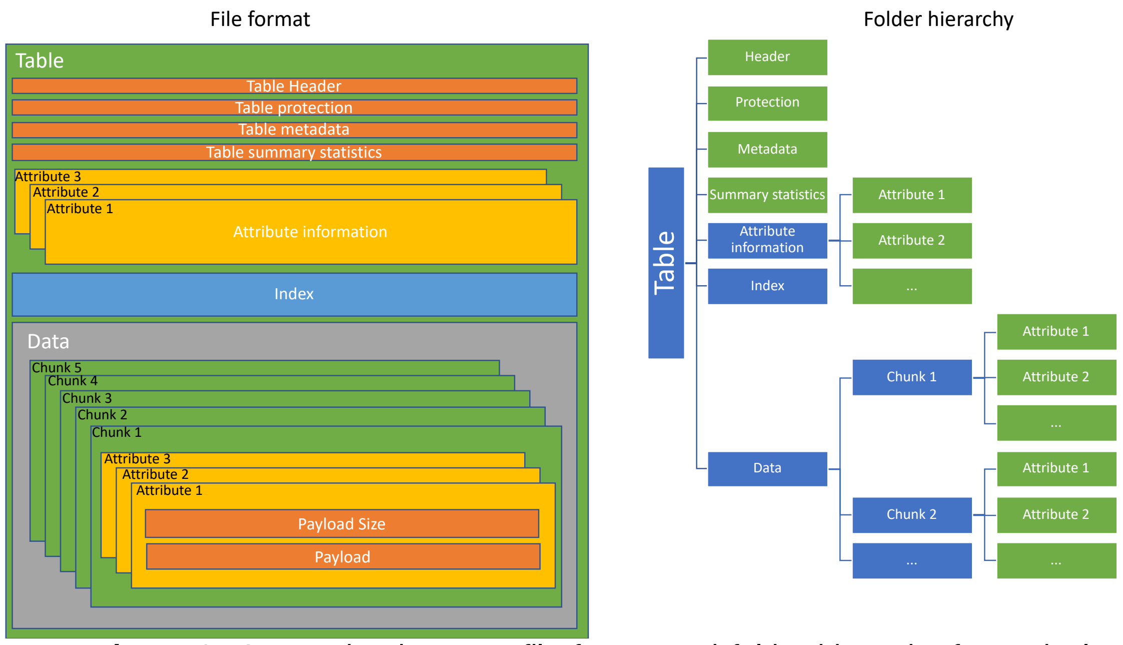


Figure 9: Conversion between file format and folder hierarchy for a single table. In the folder hierarchy, the green boxes are files while the blue boxes are folders.

2.11 Examples

To illustrate how this format can be used for storing a variety of annotation data while providing the relevant functionalities, we discuss two examples in this section.

2.11.1 Variant Call Data (VCF)

```
##fileformat=VCFv4.0
##fileDate=20090805
##source=myImputationProgramV3.1
##reference=1000GenomesPilot-NCBI36
##phasing=partial
##INFO=<ID=NS,Number=1,Type=Integer,Description="Number of Samples With Data">
##INFO=<ID=DP,Number=1,Type=Integer,Description="Total Depth">
##INFO=<ID=AF,Number=.,Type=Float,Description="Allele Frequency">
##INFO=<ID=AA,Number=1,Type=String,Description="Ancestral Allele">
##INFO=<ID=DB,Number=0,Type=Flag,Description="dbSNP membership, build 129">
##INFO=<ID=H2,Number=0,Type=Flag,Description="HapMap2 membership">
##FILTER=<ID=q10,Description="Quality below 10">
##FILTER=<ID=s50,Description="Less than 50% of samples have data">
##FORMAT=<ID=GT,Number=1,Type=String,Description="Genotype">
##FORMAT=<ID=GQ,Number=1,Type=Integer,Description="Genotype Quality">
##FORMAT=<ID=DP,Number=1,Type=Integer,Description="Read Depth">
##FORMAT=<ID=HQ,Number=2,Type=Integer,Description="Haplotype Quality">
#CHROM POS     ID        REF ALT    QUAL FILTER INFO                              FORMAT
NA00001        NA00002        NA00003
20     14370   rs6054257 G      A       29   PASS   NS=3;DP=14;AF=0.5;DB;H2           GT:GQ:DP:HQ
0|0:48:1:51,51 1|0:48:8:51,51 1/1:43:5:.,.
20     17330   .         T      A       3    q10    NS=3;DP=11;AF=0.017               GT:GQ:DP:HQ
0|0:49:3:58,50 0|1:3:5:65,3   0/0:41:3
20     1110696 rs6040355 A      G,T     67   PASS   NS=2;DP=10;AF=0.333,0.667;AA=T;DB GT:GQ:DP:HQ
1|2:21:6:23,27 2|1:2:0:18,2   2/2:35:4
20     1230237 .         T      .       47   PASS   NS=3;DP=13;AA=T                   GT:GQ:DP:HQ
0|0:54:7:56,60 0|0:48:4:51,51 0/0:61:2
20     1234567 microsat1 GTCT   G,GTACT 50   PASS   NS=3;DP=9;AA=G                    GT:GQ:DP
0/1:35:4       0/2:17:2       1/1:40:3
```

Table 1: A Simple VCF File Example (from IGSR)

Table 1 above shows a section of a VCF file, only 5 variants and 3 samples are displayed. We next describe how this can be translated to the proposed file format while preserving the data and providing additional functionalities:

Metadata

The comment lines (starting with ##) can be retained as part of the FileMetadata. If this is stored as part of an MPEG-G file with sequencing data, the metadata also contains the corresponding dataset groups that contain the sequencing data corresponding to this variant call data.

Traceability

When this is stored as part of an MPEG-G file with sequencing data, the traceability contains the commands used for generating the variant calls starting from the raw sequencing data along with the URIs of the tools used and their versions. This can be used to validate the file in a reproducible manner.

Tables

Since variant data is typically stored in a single resolution, we store it in a single table with nDimensions = 2.

Dimensional attributes

For the first dimension (variants), there are several dimensional attributes such as CHROM, POS, ID, REF, ALT, QUAL, FILTER, and the INFO fields. The INFO field is broken into multiple attributes such as NS, DP, AF, etc. as described in the comments. The types of these attributes

are also mentioned in the comment fields. The attribute metadata can be used for grouping these together (e.g., NS, DP, AF belong to the group INFO). The default value depends on the attribute, e.g., it can be set to "PASS" for the FILTER attribute.
For the second dimension (samples), the sample name (e.g., NA00001) is the only attribute present in the original VCF file. Further attributes can be added to support linkages to the sequencing data, e.g., the datasetGroup and dataset containing the sequencing data corresponding to this sample.
More dimensional attributes can be added to support fast access to certain quantities such as counts or average quantities corresponding to a particular variant.
The description of the INFO attributes in the comments can be stored as part of the AttributeMetadata.

2-d table attributes

These are the attributes described in the FORMAT fields such as GT, GQ, DP, etc. each of which is a 2-dimensional array. The types of these attributes are again described in the comments. In cases where most variants are not expressed, the default value for the GT attribute can be set to 0/0.
The description of the attributes in the comments can be stored as part of the AttributeMetadata.

Compressors

The compressors for the attributes should be chosen based on the type and characteristics of the attribute. For example, CHROM can be compressed using an enumeration-based scheme followed by gzip, POS can be compressed using delta coding followed by gzip, etc. The sample names (NA00001 etc.) can be efficiently compressed with a tokenization-based string compressor. Some of the INFO fields are present for only a small number of variants, these can be encoded with a sparse representation. Similarly, the genotypes (GT) can be encoded with a sparse representation or with a specialized compressor for genotypes (e.g. GTC [6]).
The length of certain variable length attributes can depend on other attributes – e.g., the AF (allele frequency) attribute length is equal to the number of alternate alleles. In such cases, nDependencies for the compressor can be set to 1 and this dependency can be exploited to boost the compression.

Chunking and Indexing

The chunking for the main 2d array can be performed depending on the access patterns. If most accesses are for variants in a particular region, then each chunk should include all samples and a small number of variants (i.e., horizontal chunks). Whereas if most accesses are for all variants for a particular sample, the chunk should include all variants and a small number of samples (i.e., vertical chunks). If both types of queries are quite common, then it is better to use rectangular chunks including a small number of variants and samples. By increasing the size of chunks, random access performance can be traded off against compression ratio.

For random access based on the genomic region, an additional index can be used as shown in the table below (based on CSI indexing [7]).

AttributeIDsIndexed	CHROM, POS
IndexType	CSI
IndexSize	Size of index
IndexData	CSI index structure

Rather than specifying the actual file position as done in CSI, this will instead return the list of chunkIDs that overlap with the genomic region in question. The positions of these chunks in the file can then be determined from the default index structure. If indel variants are prevalent, the CSI indexing should be performed based on both START and END position of the variant.

More attributes can be indexed to allow fast random-access queries. E.g., the FILTER attribute can be indexed to allow faster filtering of variants based on whether FILTER=PASS or not.

Protection

The access control policy can take various forms depending on the use case. Certain users might have access to all the data, while others might have access only to variants within certain genomic regions (specified by CHROM and POS). Similarly, one can restrict access to only certain samples. Note that this requires that the chunks be chosen accordingly. The access control can also be imposed at the attribute level, e.g., allowing access to the INFO fields but not to the individual sample data.

2.11.2 Genome Functional Annotation Data (BED)

```
browser position chr7:127471196-127495720
browser hide all
track name="ItemRGBDemo" description="Item RGB demonstration" visibility=2 itemRgb="On"
chr7    127471196  127472363  Pos1  0  +  127471196  127472363  255,0,0
chr7    127472363  127473530  Pos2  0  +  127472363  127473530  255,0,0
chr7    127473530  127474697  Pos3  0  +  127473530  127474697  255,0,0
chr7    127474697  127475864  Pos4  0  +  127474697  127475864  255,0,0
chr7    127475864  127477031  Neg1  0  -  127475864  127477031  0,0,255
chr7    127477031  127478198  Neg2  0  -  127477031  127478198  0,0,255
chr7    127478198  127479365  Neg3  0  -  127478198  127479365  0,0,255
chr7    127479365  127480532  Pos5  0  +  127479365  127480532  255,0,0
chr7    127480532  127481699  Neg4  0  -  127480532  127481699  0,0,255
```

Table 2: A Simple BED File Example (from UCSC Genome Browser FAQ)

Table 2 above shows a section of a BED file, with some annotation data. We next describe how this can be translated to the proposed file format while preserving the data and providing additional functionalities:

Metadata

The comment lines (first three lines) can be retained as part of the FileMetadata. If this is stored as part of an MPEG-G file with sequencing data, the metadata also contains the corresponding dataset groups that contain the sequencing data corresponding to this annotation data.

Tables

For displaying the data at different scales and resolutions, we store multiple tables with precomputed values for different resolutions. The TableInfo field stores the details about the resolution in a predefined format, hence allowing the user to query the list of available resolutions without needing to read the whole file. The ByteOffset variable for each table allows direct access to the desired resolution. Each table has a single dimension.

Attributes

In this case, each column becomes an attribute: chrom (string), chromStart (integer), chromEnd (integer), name (string), score (integer), strand (character), thickStart (integer), thickEnd (integer), itemRGB (8-bit integer array of length 3).

Compressors

The compressors for the attributes should be chosen based on the type and characteristics of the attribute. For example, chrom can be compressed using an enumeration-based scheme followed by gzip, chromStart and chromEnd can be compressed using delta coding followed by gzip, etc. The values of thickStart and thickEnd are likely to be close to chromStart and chromEnd, suggesting that we can improve the compression by using them as side information.
Note that in the example shown the value of chromStart matches the value of chromEnd on the previous row. One way to exploit this would be to consider chromStart, chromEnd as a single attribute of type "integer array of length 2", but this should be done only if the visualization tools understand this alternate representation.

Chunking and Indexing

For random access based on the genomic region, an additional index can be used as shown in the table below (based on CSI indexing [7]).

AttributeIDsIndexed	chrom, chromStart, chromEnd
IndexType	CSI
IndexSize	Size of index
IndexData	The CSI index structure

Rather than specifying the actual file position as done in CSI, this will instead return the list of chunkIDs that overlap with the genomic region in question. The positions of these chunks in the file can then be determined from the default index structure.

Protection

The access control policy can take various forms depending on the use case. Certain users might have access to all the data, while others might have access only to coarse resolution data (recall that different resolutions are stored in different tables). Similarly, one can restrict access to only certain genomic regions. Note that this requires that the chunks be chosen accordingly.

3. Implementation

Here we discuss the current implementation status for the accompanying file format description, including the set of features in the format not implemented as of now. We also discuss results on some of the MPEG-G annotation test data sets. The GTF compression is based on ideas from GPress (https://github.com/qm2/gpress) which is noted at the appropriate places.

3.1 Storage Format

Note that the following major features are **not supported by the current implementation but are supported by the proposal**:

1. Multiple tables (e.g., for multiple resolutions)
2. Variable length chunks (currently fixed length chunks are used along each axis which automatically induces rectangular chunking for the main array in case of 2-d datasets)
3. Attribute-dependent chunks
4. In the case of compression of one attribute based on other attributes – currently only a single dependency attribute is allowed, i.e., compression of one attribute conditioned on

two or more other attributes is not implemented. Also, compression of an attribute in a 2-d array (e.g., VCF genotype) conditioned on a dimension-specific attribute (e.g., an INFO field) is not implemented.

5. Linkage to MPEG-G parts 1-5
6. Embedding decompressor code/executable for a specific attribute compressor within the compressed file
7. Including compressor global data (e.g., codebooks, dictionaries, trained models) that can be shared across chunks
8. Other high-level features – protection/traceability

Below is the currently implemented file format.

Top-level

Name	**Description**	**Type**
TableName		string
TableMetadata	Stores headers (comment lines) for VCF/matrix market file	string
TableType	VCF/GTF/scRNA_expression	string
nDim	1 or 2	uint8
DimSize[i] for i in nDim	Size along each dimension	uint32
DimName[i] for i in nDim	Name of each dimension	string
DimMetadata[i] for i in nDim	Metadata of each dimension	string
// nArrays = 1 if nDim = 1, =nDim+1 otherwise	For a 2-dimensional table, we have a main array and 2 dimension-specific (row & column) arrays	
DimNattrs[i] for i in nArrays	Number of attributes in each array	uint32

For i in nArrays:
    For j in DimNattrs[i]:
        AttrParams[i][j]
	Attribute parameters, discussed below	See table on AttrParams

ChunkSize[i] for i in nDim	Chunk size along each dimension	uint32
numChunks[i] for i in nArrays	Number of chunks for each array (the 2-d main array has rectangular chunks organized in row-major fashion)	uint32
DimByteOffset[i] for i in nArrays	Byte offset for each array (i.e., different dimension-specific attributes and the main array)	uint32
For i in nArrays:	// go over the dimensional attributes and the main attributes	
numAdditionalIndexes[i]	Number of attribute-specific indexes	uint8
For j in numAdditionalIndexes[i]:	Attribute-specific indexes	
AdditionalIndexType[i][j]	0 (chrom_pos), 1 (levelDB)	uint8
AdditionalIndexData[i][j]	Binary data depending on index type	Bytes
For j in numChunks[i]:	Main index	
ChunkByteOffset[i][j]	This is for the random access to a specific chunk in the array	uint64
For j in numChunks[i]:	Payload data	
For k in DimNattrs[i]:		
PayloadSize[i][j][k]	Size of compressed payload for array i, chunk j, attribute k	uint64
Payload[i][j][k]	Compressed payload for array i, chunk j, attribute k	Bytes

AttrParams

Name	Description	Type
AttrName	Name of attribute	string
AttrMetadata	Metadata, e.g., comment line corresponding to attribute (INFO/FORMAT field) in VCF or “REQUIRED” in case of compulsory attributes.	string
AttrType	Attribute type (described below)	uint8
DefaultValue	Default value for attribute represented as a string	string
MissingValue	Missing value for attribute represented as a string (e.g., “.”). These are present in the decompressed payload	string
// Compression parameters		
deltaFlag	Whether delta coding is to be applied	Bool
CompressorName	BSC/GZip	string
dependencyFlag	Whether this attribute is compressed dependent on another attribute	Bool
If dependencyFlag:		
DependencyAttributeId	Attribute id for the dependency attribute	uint32
DependencyTransform	Reorder/GTF_start_end/GTF_strand (discussed below)	uint8
sparseFlag	Whether sparse coding is to be applied	Bool
If sparseFlag:		
nDimSparse	Dimensionality of the array (needed to appropriately interpret the coordinate and value streams) (this is redundant as this information can be obtained from the top-level structure)	uint8

3.2 Attribute Types

Several attribute types are currently supported:

1. Fundamental data types: 8/16/32/64 bit signed/unsigned integers, float/double, char, bool (1 byte). These are represented in binary in the decompressed payload.
2. Derived types:
    a. String: represented as a 0 terminated char stream in the decompressed payload. (we tried other representations such as separation into length and value streams but those gave worse results when BSC was applied)
    b. Start/end: for GTF files, we use a pair of uint32 to represent the start and end values in the decompressed payload. This is helpful for applying the conditional compression from GPress (GTF start end transform) where the start and end fields are jointly transformed based on the feature column.

3.3 Compression Modes and Parameters

The attribute compression details are discussed below.

3.3.1 Delta Coding

Delta coding can be used on any integer data type and is applied to the value stream before any sparse coding/conditional compression transformation. The integer bitwidth is kept the same after delta coding.

3.3.2 Sparse Coding

For sparse coding, the coordinate (uint32) and value streams are separated. For 1-d arrays, the coordinates are delta coded. For 2-d arrays, the row coordinates are delta coded and the column coordinates are delta coded within each row. Finally, the coordinate and value streams are concatenated with the number of values written at the start. The coordinates are represented as uint32.

3.3.3 Compressors

The stream for a given chunk and attribute is compressed using BSC/GZip after all transforms are applied. Gzip is used at level 9 (best compression) and BSC is used with flag (-b64 -e2). These parameters are currently hardcoded in the implementation and are not part of the file format, only the compressor name is stored for each attribute. BSC is used by default.

3.3.4 Conditional Compression

The file format supports conditional compression of one attribute based on another. We have not implemented context-based arithmetic coding, which is a classic example for this. Currently, we only allow a single dependency and make sure that there are no directed cycles in the dependency graph.

3.3.4.1 Reorder Transform

Here we reorder the values of one attribute based on the values of another attribute, as shown in the example below.

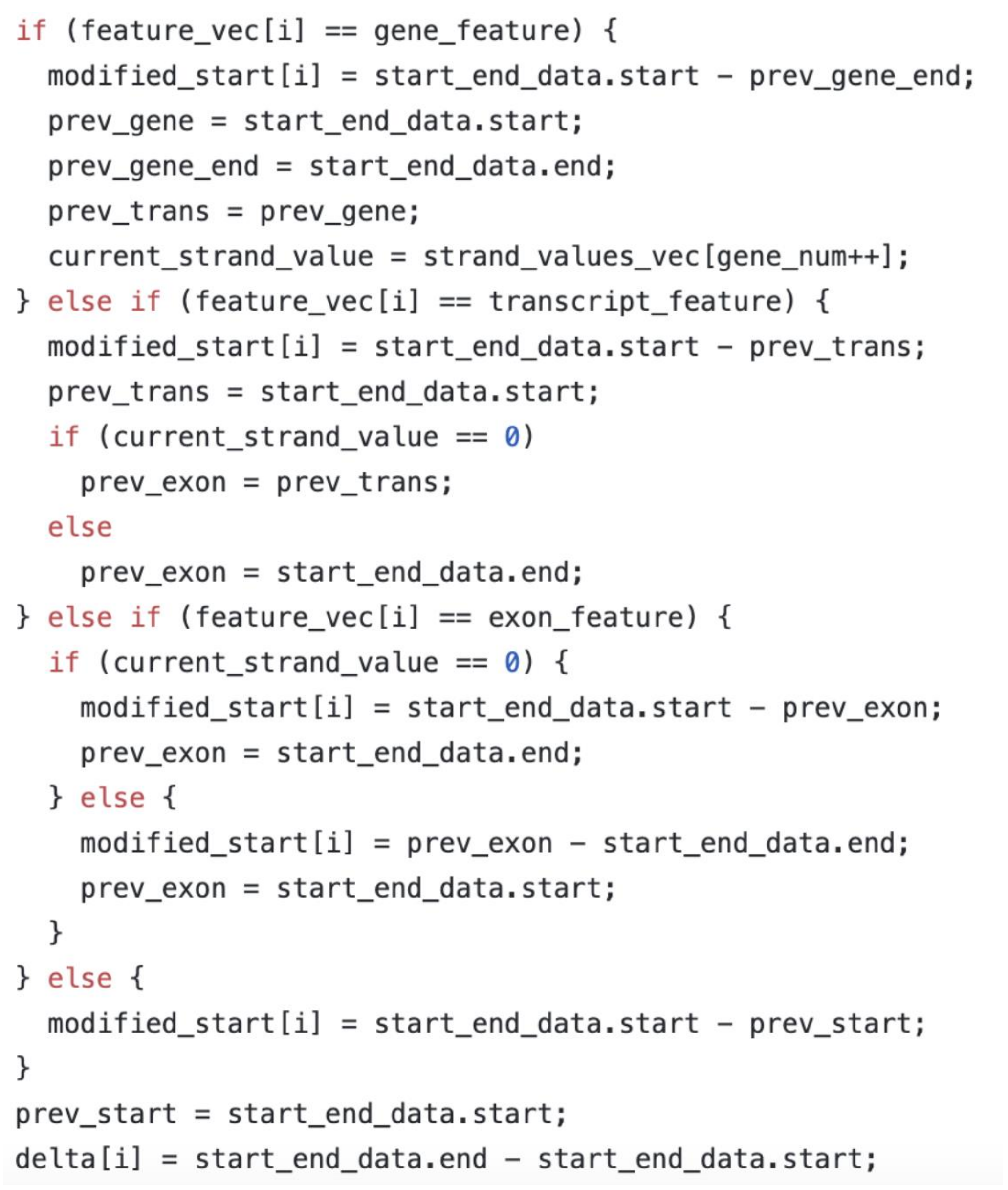
- Attribute 1
- 0, 1, 2, 2, 1, 0, 1, 1, 2, 1
- Attribute 2
- a, b, c, d, e, f, g, h, i, j
- Attribute 2 reordered according to attribute 1 values
- a, f, b, e, g, h, j, c, d, i

This allows BSC/GZip to exploit the dependency across attributes by bringing similar values together. In information-theoretic terms, this can achieve the conditional entropy of one attribute conditioned on the other (asymptotically). This is suitable when the dependency attribute takes on relatively small number of unique values, in particular this might not be suitable for continuous valued or integral data which has ordinal structure.

We use this for VCF genotype likelihood and dosage values (conditioned on genotype) and in GTF for compressing the frame (conditioned on feature as done in Gpress).

3.3.4.2 GTF Start-End Transform

This is based on GPress and involves compression of the start & end attributes in the GTF file based on the feature column (that can take value gene/transcript/exon etc.). The idea is to delta code the end wrt the start. The start itself can be modified based on start or end of the previous feature/transcript/exon. The precise algorithm used is shown below. Note that Gpress also uses the strand value, but in our case, we figure out the strand value based on the start and end values for consecutive exons and store this in the stream (this has very small contribution to size). This is because conditional compression based on two attributes (feature+strand) is currently not implemented.



```
if (feature_vec[i] == gene_feature) {
  modified_start[i] = start_end_data.start - prev_gene_end;
  prev_gene = start_end_data.start;
  prev_gene_end = start_end_data.end;
  prev_trans = prev_gene;
  current_strand_value = strand_values_vec[gene_num++];
} else if (feature_vec[i] == transcript_feature) {
  modified_start[i] = start_end_data.start - prev_trans;
  prev_trans = start_end_data.start;
  if (current_strand_value == 0)
    prev_exon = prev_trans;
  else
    prev_exon = start_end_data.end;
} else if (feature_vec[i] == exon_feature) {
  if (current_strand_value == 0) {
    modified_start[i] = start_end_data.start - prev_exon;
    prev_exon = start_end_data.end;
  } else {
    modified_start[i] = prev_exon - start_end_data.end;
    prev_exon = start_end_data.start;
  }
} else {
  modified_start[i] = start_end_data.start - prev_start;
}
prev_start = start_end_data.start;
delta[i] = start_end_data.end - start_end_data.start;
```

3.3.4.3 GTF Strand Transform

The strand value is compressed conditioned on the feature column (based on GPress). Basically, only the strand value for the gene needs to be stored (also the strand value for the first feature in the chunk if it is not a gene).

3.4 Additional Indexes

Additional attribute-specific indexes are used to perform random access based on the value or range of a given attribute. Currently the specific attributes being indexed are hardcoded for each file type (VCF: chrom/pos, GTF: chrom/pos, gene id, scRNA_expression: gene id), ideally this information should be made available in the file format itself.

3.4.1 ChromPos Index

This consists of a list of chromosome names (strings) and the leftmost and rightmost chromosome, position pair in each chunk. This can be used to rapidly identify chunks overlapping with a given genomic range.

3.4.2 LevelDB Index

LevelDB (https://github.com/google/leveldb) is a generic disk-based key-value. The key and value are byte arrays. This can be used for creating a gene index for GTF or gene expression data mapping the gene id to the chunk containing the gene as well as position within the chunk. LevelDB creates multiple files in a folder which are tarred, compressed with BSC and stored in the compressed file along with the compressed size.

3.5 Notes on Specific File Types Currently Tested

Here we describe the default configurations that were tested for three of the file types. It is possible to change the compression parameters (e.g., disable delta coding, change BSC to Gzip, add some dependency across attributes) by changing the JSON configuration during compression.

3.5.1 VCF

For a one-dimensional (i.e., with no samples) VCF, the first six columns become separate attributes and the seventh column (INFO) is split into multiple attributes. CHROM is stored as a 8-bit unsigned integer (chromosome name is stored as part of the ChromPos index), POS is stored as a 32-bit unsigned integer and is delta coded. The INFO fields are stored as bool when they are flags and as strings otherwise. The decompressed file might have a different ordering of the INFO fields and in some cases, fields missing in the original file might be displayed in the decompressed file with the value ".". Thus, the decompressed file doesn't match the original VCF byte by byte.

For two-dimensional (i.e., with samples) VCF, the FORMAT field is stored as a row attribute in addition to the attributes mentioned in the previous paragraph. The SAMPLE name becomes a column attribute and the actual genotype data is split by colons (":") and stored as a 2-d array of multiple attributes based on the FORMAT field. The implementation also supports not splitting the genotype fields as it might give slightly better compression in some cases.

Chunking is done for both rows and columns. Random access by genomic position range is performed using the ChromPos index while random access by sample name is performed by first decompressing all sample names (column attributes), identifying the relevant column number and decompressing the relevant chunks.

3.5.2 GTF (Based on GPress)

Here the columns become different attributes: chromosome is as done in VCF, start and end are stored as a single attribute (as discussed before), strand is stored as a bool and the rest are stored as string attributes. Note that this is just a 1-dimensional array. We use reorder transform for frame, GTF start end transform and GTF strand transform (all dependent on feature column). Two indexes are used – chromPos and LevelDB. The LevelDB index maps each gene id to the start and end chunk and line with that gene id (where the end is delta coded wrt the start). This allows us to access quickly identify the chunks and the lines within the chunk containing a specific gene id (i.e., a gene and all its children transcripts, exons, etc.).

3.5.3 scRNA Expression (Matrix Market or TSV) (Partially Based on GPress)

This consists of three files: a matrix file with the expression values (stored as sparse 2-d integer attributes with genes as rows and barcodes as columns), features.tsv file (stored as row attributes – first attribute is the gene id, there might be more associated attributes, stored as string attributes), barcodes.tsv file (single column attribute stored as a string attributes). A single large column chunk is used since random access by barcode is not commonly used, while the rows are divided into multiple chunks.

A LevelDB index is used, mapping the gene id to the chunk containing the gene id, the position of the gene id in the chunk along the vertical axis and the position in the sparse 2-d array. When random access by gene id is used, the whole barcode list is decompressed, and then only the barcodes expressing the gene are written to the decompressed barcodes.tsv file. The corresponding expression values are written to the .mtx file and the information associated with the gene id is written to the features.tsv. We observed that the decompression of the barcodes.tsv file takes up significant fraction of the decompression time and hence added a flag in the decompression configuration to disable the barcode decompression when a specific gene id is being decompressed.

Features of GPress not yet implemented:

- GFF3 file compression
- Bulk RNA seq expression compression
- Linking of GTF/GFF3 with gene expression
- Random access based on transcript ids/exon ids

3.5.4 Other File Types Not Yet Implemented/Tested

The following file types have not been tested and their parser not yet implemented (note that the proposal does support these types).

- Mapping statistics
- Quantitative tracks (wig)
- Hi-C
- Bulk RNAseq
- Parser for scRNA expression files represented as HDF5/Loom

4. Performance Evaluation

All experiments were run on an Ubuntu 18.04 server with 2.2 GHz Intel Xeon processor. All tools were run with a single thread (Gzip with default settings, BSC with “-b64 -2 -t1T” flag which is the flag used in the end stage for the proposed compressor). The BSC version used here is as available at https://github.com/shubhamchandak94/libbsc. The JSON configuration files and the commands used for compression/decompression are mentioned below for each specific experiment. The linux executable, the compressed bitstreams for the main experiments and the JSON configuration files are provided with this proposal. The chunk size used was 10,000 (x 100) for 1 (2)-d VCF files, 10,000 for GTF files, 1000 for scRNA_expression files. The compressed bit streams, executable and JSON configuration files are available here.

4.1 VCF

4.1.1 Variants Only (No Samples)

4.1.1.1 Datasets

Dataset no.	Link
1	ftp://ftp.ensembl.org/pub/release-95/variation/vcf/homo_sapiens/homo_sapiens_somatic.vcf.gz
2	ftp://ftp.ensembl.org/pub/release-95/variation/vcf/homo_sapiens/homo_sapiens_structural_variations.vcf.gz

Dataset no.	Uncompressed file size (bytes)	Number of variants
1	347,839,686	4,417,937
2	3,689,444,771	28,953,093

4.1.1.2 Main Compression Results

Dataset no.	Size (bytes)				Compression Time			Decompression Time		
	Original	Gzip	BSC	Proposed	Gzip	BSC	Proposed	Gzip	BSC	Proposed
1	347,839,686	33,813,985	28,799,010	18,593,682	7s	42s	22s	2s	17s	15s
2	3,689,444,771	209,297,354	165,315,184	131,286,050	46s	4m25s	2m42s	16s	2m8s	1m59s

We see close to 37% better compression over Gzip and around 20% better compression than BSC. Most of the improvement is due to compression of columns independently as separate attributes and due to delta coding of POS. The compression/decompression times are better than BSC but worse than Gzip.

4.1.1.3 Random Access Results

For dataset no. 2, compared to ~2m for decompression of whole file, the decompression of chrom 22, position 20M-30M takes less than 2s. The chunk size used was 10,000.

4.1.1.4 Commands Used for Proposed Compressor

Compression:
```
./linux_executable -c -i vcf_file.vcf.gz -o compressed_file.bin -p
VCF -g -j vcf_1d_compression.json
```

Decompression (whole file):
```
./linux_executable -d -i compressed_file.bin -o decompressed.vcf
```

Decompression (genomic range – update json file as needed):
```
./linux_executable -d -i compressed_file.bin -o decompressed.vcf -
j vcf_decompression_range.json
```

Note that as discussed above in the VCF section, the decompressed file doesn't match the original VCF file byte by byte due to reordering of INFO fields.

4.1.2 Variants with Sample Genotypes (1000 Genome Project)

4.1.2.1 Datasets

Dataset no.	Link
1	ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase1/analysis_results/integrated_call_sets/ALL.chr1.integrated_phase1_v3.20101123.snps_indels_svs.genotypes.vcf.gz
2	ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase1/analysis_results/integrated_call_sets/ALL.chr22.integrated_phase1_v3.20101123.snps_indels_svs.genotypes.vcf.gz

Dataset no.	Uncompressed file size (bytes)	Number of variants	Number of samples
1	93,086,828,627	3,007,196	1,092
2	15,304,146,564	494,328	1,092

4.1.2.2 Main Compression Results

Dataset no.	Size (bytes)				Compression Time			Decompression Time		
	Original	Gzip	BSC	Proposed	Gzip	BSC	Proposed	Gzip	BSC	Proposed
1	93,086,828,627	10,781,170,344	4,337,628,848	4,254,124,733	49m	3h36m	2h12m	12m	2h24m	1h36m
2	15,304,146,564	1,796,657,847	728,642,384	717,980,220	8m	36m	23m	2m	24m	15m

4.1.2.3 Random Access Results

For dataset no. 2, compared to ~15m for decompression of whole file, the decompression of chrom 22, position 20M-30M takes 22s. Decompression of a single sample takes around 1m40s. The chunk size used was 10,000 x 100.

4.1.2.4 Impact of Conditional Compression of GL and DS Fields

The table below shows the impact of using the conditional reorder transform for the DS and GL attributes wrt the GT attribute (for dataset 2). We see that the sizes for these are reduced but the GL still takes up most of the total space. Note that theoretically, we expect the maximum improvement due to this transform on each of DS and GL to be bounded by the entropy of the GT. That is, the improvement in this example cannot be more than 19.4 MB for each of GL and DS (under certain ideality assumptions). A specialized compressor/lossy compressor for GL can lead to huge savings in this regard.

Compressor	**Mode**	**Total size (MB)**	**GT+DS+GL (MB)**	**GT (MB)**	**DS (MB)**	**GL (MB)**
CDTC	Without conditional compression	749.7	742.3	19.4	40.9	682
CDTC	With conditional compression (default)	718.0	710.6	19.4	24.2	667

4.1.2.5 Commands Used for Proposed Compressor

Compression (default: use conditional compression of GL and DS based on GT):
```
./linux_executable -c -i vcf_file.vcf.gz -o compressed_file.bin -p
VCF -g -j vcf_2d_compression_default.json
```

Compression (don't use conditional compression of GL and DS based on GT):
```
./linux_executable -c -i vcf_file.vcf.gz -o compressed_file.bin -p
VCF -g -j vcf_2d_compression_no_conditional.json
```

Decompression (whole file):
```
./linux_executable -d -i compressed_file.bin -o decompressed.vcf
```

Decompression (genomic range – update json file as needed):
```
./linux_executable -d -i compressed_file.bin -o decompressed.vcf -
j vcf_decompression_range.json
```

Decompression (sample name – update json file as needed):
```
./linux_executable -d -i compressed_file.bin -o decompressed.vcf -
j vcf_decompression_sample.json
```

Note that as discussed above in the VCF section, the decompressed file doesn't match the original VCF file byte by byte due to reordering of INFO fields.

4.2 GTF

4.2.1 Datasets

Dataset no.	Link
1	ftp://ftp.ensembl.org/pub/release-95/gtf/homo_sapiens/Homo_sapiens.GRCh38.95.chr.gtf.gz
2	ftp://ftp.ensembl.org/pub/release-95/gtf/homo_sapiens/Homo_sapiens.GRCh38.95.gtf.gz

Dataset no.	Uncompressed file size (bytes)	Number of lines	Number of genes
1	1,162,883,375	2,736,850	58,676
2	1,163,163,881	2,737,564	58,735

4.2.2 Main Compression Results

Dataset no.	Size (bytes)				Compression Time			Decompression Time		
	Original	Gzip	BSC	Proposed	Gzip	BSC	Proposed	Gzip	BSC	Proposed
1	1,162,883,375	43,656,910	24,855,264	18,536,141	15s	1m8s	31s	5s	29s	18s
2	1,163,163,881	43,668,708	24,863,150	18,541,903	18s	1m9s	31s	5s	28s	18s

We see close to 60% better compression over Gzip and around 25% better compression than BSC. Most of the improvement is due to compression of columns independently as separate attributes, while a small contribution is made by the conditional compression ideas from Gpress (see below). The compression/decompression times are better than BSC but worse than Gzip.

4.2.3 Random Access Results

For dataset 2, compared to 18s for decompression of the entire file, decompression of range chr22:20M-30M took less than 1s, and decompression of a single gene took less than 1s. The chunk size used was 10,000.

4.2.4 Impact of Conditional Compression Based on Feature Column (Ideas from GPress)

Here we look at the results without applying the conditional compression ideas from GPress (for the start/end, strand and frame columns) on the dataset no. 2. In the table below, we see that applying the conditional compression leads to around 5% improvement overall, but the improvement on the specific columns can be as high as 50%. Note that the last column "attribute" takes up most of the space and hence a specialized compressor for this can significantly improve the overall compression. Finally, note that the leveldb index takes up a very small size, partly because the index is also kept compressed with BSC.

Component	**Size in bytes** Without conditional compression	**Size in bytes** With conditional compression (default)
Chrom pos index	3384	3384
LevelDB gene index	349746	349852
Chunk index	2192	2192
seqname	16046	16046
source	227424	227424
feature	367876	367876
Start_end	6695274	5742458
score	16986	16986
strand	58652	22524
frame	263454	149328
attribute	11634274	11634274
TOTAL	**19644947**	**18541903**

4.2.5 Commands used for proposed compressor

Compression (default: use conditional compression of start/end, strand, frame based on feature):
```
./linux_executable -c -i gtf_file.gtf.gz -o compressed_file.bin -p
GTF -g -j gtf_compression_default.json
```

Compression (don’t use conditional compression of start/end, strand, frame based on feature):
```
./linux_executable -c -i gtf_file.gtf.gz -o compressed_file.bin -p
GTF -g -j gtf_compression_no_conditional.json
```

Decompression (whole file):
```
./linux_executable -d -i compressed_file.bin -o decompressed.gtf
```

Decompression (genomic range – update json file as needed):
```
./linux_executable -d -i compressed_file.bin -o decompressed.gtf -
j gtf_decompression_range.json
```

Decompression (gene id – update json file as needed):
```
./linux_executable -d -i compressed_file.bin -o decompressed.gtf -
j gtf_decompression_gene.json
```

4.3 scRNA_expression

4.3.1 Datasets

Dataset no.	Link	Comments
1	http://cf.10xgenomics.com/samples/cell-exp/3.0.0/heart_10k_v3/heart_10k_v3_filtered_feature_bc_matrix.tar.gz	scRNA-seq: 10k heart cells from an E18 mouse
2	http://cf.10xgenomics.com/samples/cell-exp/3.0.0/heart_10k_v3/heart_10k_v3_raw_feature_bc_matrix.tar.gz	scRNA-seq: 10k heart cells from an E18 mouse
3	http://cf.10xgenomics.com/samples/cell-exp/3.0.0/malt_10k_protein_v3/malt_10k_protein_v3_filtered_feature_bc_matrix.tar.gz	scRNA-seq: 10k Cells from a MALT Tumor
4	http://cf.10xgenomics.com/samples/cell-exp/3.0.0/malt_10k_protein_v3/malt_10k_protein_v3_raw_feature_bc_matrix.tar.gz	scRNA-seq: 10k Cells from a MALT Tumor
5	http://cf.10xgenomics.com/samples/cell-exp/3.0.0/neuron_10k_v3/neuron_10k_v3_filtered_feature_bc_matrix.tar.gz	scRNA-seq: 10k brain cells from an E18 mouse
6	http://cf.10xgenomics.com/samples/cell-exp/3.0.0/neuron_10k_v3/neuron_10k_v3_raw_feature_bc_matrix.tar.gz	scRNA-seq: 10k brain cells from an E18 mouse
7	http://cf.10xgenomics.com/samples/cell-exp/3.0.0/pbmc_10k_v3/pbmc_10k_v3_filtered_feature_bc_matrix.tar.gz	scRNA-seq: 10k PBMCs from a healthy donor
8	http://cf.10xgenomics.com/samples/cell-exp/3.0.0/pbmc_10k_v3/pbmc_10k_v3_raw_feature_bc_matrix.tar.gz	scRNA-seq: 10k PBMCs from a healthy donor

Dataset no.	Uncompressed file size (bytes)	Number of genes	Number of barcodes	Number of entries in sparse matrix
1	240,678,371	31,053	7,713	19,049,671
2	542,853,620	31,053	6,794,880	26,541,357
3	137,630,766	33,555	8,412	10,794,402
4	364,846,709	33,555	6,794,880	14,985,831
5	402,747,405	31,053	11,843	31,522,268
6	757,325,088	31,053	6,794,880	40,438,578
7	318,717,693	33,538	11,769	24,825,783
8	630,371,244	33,538	6,794,880	32,136,028

4.3.2 Main Compression Results

Dataset no.	Size (bytes)				Compression Time			Decompression Time		
	Original	Gzip	BSC	Proposed	Gzip	BSC	Proposed	Gzip	BSC	Proposed
1	240,678,371	58,913,493	63,451,206	14,587,061	11s	39s	19s	2s	25s	14s
2	542,853,620	108,442,884	108,331,318	53,941,730	25s	1m15s	53s	4s	47s	37s
3	137,630,766	35,582,444	36,602,020	9,622,149	7s	23s	11s	2s	15s	9s
4	364,846,709	72,131,344	68,688,884	38,162,432	18s	50s	37s	3s	30s	25s
5	402,747,405	95,748,877	102,746,848	21,798,784	17s	1m4s	32s	3s	43s	24s
6	757,325,088	152,000,818	153,468,180	67,004,508	34s	1m45s	1m6s	5s	1m3s	50s
7	318,717,693	77,266,573	81,889,186	18,787,268	14s	51s	24s	3s	33s	18s
8	630,371,244	127,244,744	126,217,478	60,783,568	28s	1m27s	56s	5s	53s	40s

We see close to 75% better compression over BSC/GZip on the "filtered" datasets (1, 3, 5, 7). The improvement on the "raw" datasets is closer to 50%. This is because the main improvement in the proposed approach is on the sparse matrix which is a bigger contributor in the filtered datasets (see below). The compression/decompression times are better than BSC but worse than Gzip.

4.3.3 Random Access Results

For dataset 8, compared to 40s for decompression of whole file, decompression of a single gene takes 12s. Most of this time is taken up for decompression of barcodes (since all barcodes are compressed in a single chunk, we need to decompress all the barcodes and then output only the ones that express the given gene). If the barcodes are not decompressed, the time for decompression of a single gene reduces to less than 2s. The chunk size used here was 1000 genes.

4.3.4 Breakdown into Individual Components

We see below the breakdown of the size into individual components for Gzip, BSC and the proposed compressor for dataset no. 6 which is a "raw" dataset. Note that "raw" datasets have significantly larger barcode files than the "filtered" datasets. We see that the proposed approach provides the most benefits for the sparse matrix due to the separation of coordinate and value streams and the delta coding of the sparse coordinates. The index takes up a relatively small fraction.

Compressor	Barcode list	Feature/gene info	Sparse matrix	Index	Total
Uncompressed	129.1 MB	1.32 MB	626.9 MB		757.3 MB
Gzip	19.36 MB	0.25 MB	132.4 MB		152.0 MB
BSC	15.24 MB	0.17 MB	138.1 MB		153.5 MB
Proposed	15.24 MB	0.19 MB	51.33 MB	0.24 MB	67.00 MB

4.3.5 Commands Used for Proposed Compressor

Compression:

```
./linux_executable -c -i matrix.mtx.gz features.tsv.gz
barcodes.tsv.gz -o compressed_file.bin -p scRNA_expression -g -j
scRNA_expression_compression_default.json
```

Decompression (whole file):

```
./linux_executable -d -i compressed_file.bin -o
decompressed_matrix.mtx decompressed_features.tsv
decompressed_barcodes.tsv
```

Decompression (gene id – update json file as needed):

```
./linux_executable -d -i compressed_file.bin -o
decompressed_matrix.mtx decompressed_features.tsv
decompressed_barcodes.tsv -j
scRNA_expression_decompression_gene.json
```

Decompression (gene id – don't decompress barcodes):

```
./linux_executable -d -i compressed_file.bin -o
decompressed_matrix.mtx decompressed_features.tsv
decompressed_barcodes.tsv -j
scRNA_expression_decompression_gene_no_barcodes.json
```

Note that the decompressed matrix.mtx file is only guaranteed to be same as original up to reordering since the original mtx file might not be sorted according to a specific criterion (by row/column).

4.4 Conclusions

We observe that the proposed file format offers improved compression and fast random access across a variety of file types, while offering a high degree of customizability and ability to incorporate additional specialized compressors.

References

1. “The Variant Call Format Specification”, http://samtools.github.io/hts-specs/VCFv4.3.pdf
2. Kent, W. James, et al. "BigWig and BigBed: enabling browsing of large distributed datasets." *Bioinformatics* 26.17 (2010): 2204-2207.
3. Abdennur, Nezar, and Leonid Mirny. "Cooler: scalable storage for Hi-C data and other genomically-labeled arrays." *BioRxiv* (2019): 557660.
4. Zheng, Xiuwen, et al. "SeqArray—a storage-efficient high-performance data format for WGS variant calls." *Bioinformatics* 33.15 (2017): 2251-2257.
5. Alberti, Claudio, et al. "An introduction to MPEG-G, the new ISO standard for genomic information representation." *bioRxiv*(2018): 426353.
6. Danek, Agnieszka, and Sebastian Deorowicz. "GTC: how to maintain huge genotype collections in a compressed form." *Bioinformatics* 34.11 (2018): 1834-1840.
7. “CSI index”, http://samtools.github.io/hts-specs/CSIv2.pdf