RNTuple Binary Format Specification 1.0.0.0

The ROOT Team

Jakob Blomer Brian Bockelman Philippe Canal Florine Willemijn de Geus Jonas Hahnfeld Giovanna Lazzari Miotto Simon Leisibach Jerry Ling Javier Lopez-Gomez Axel Naumann Max Orok Vincenzo Eduardo Padulano Giacomo Parolini Danilo Piparo Jim Pivarski Oksana Shadura Andres Rios Tascon

February 2025

This document was created as part of the source code of the ROOT data analysis framework. Version 1.0.0.0 of the RNTuple specification was released in ROOT v6.34.00 in November 2024.

The RNTuple format is ROOT's event data format for the HL-LHC era. It is a columnar, binary data format optimized for High Energy Physics datasets. RNTuple succeeds the ROOT TTree format, which has been developed more than 25 years ago and stores more than 2 exabytes of LHC Run 1–3 data. RNTuple is a re-engineered format for higher robustness, for full exploitation of modern storage technologies such as NVMe drives and object stores, and generally for significantly better performance characteristics in data compactness, scalability, and read and write speed.

RNTuple is the designated data format for LHC data as of Run 4, with an expected overall data volume of tens of exabytes by the end of HL-LHC. The R&D on the RNTuple format has been performed in the ROOT team and supported by the CERN EP strategic R&D programme on technologies for future experiments.

Versioning Notes

The RNTuple binary format version is inspired by semantic versioning. It uses the following scheme:

`EPOCH.MAJOR.MINOR.PATCH`

Epoch: an increment of the epoch indicates backward-incompatible changes. The RNTuple pre-release has epoch 0. The first public release has epoch 1. There is currently no further epoch foreseen.

Major: an increment of the major version indicates forward-incompatible changes. A forward-incompatible change is known to break reading in previous software versions that do not support that feature. The use of new, forward-incompatible features must be indicated in the feature flag in the header (see below). For the RNTuple pre-release (epoch == 0), the major version is the release candidate number.

Minor: an increment of the minor version indicates new, optional format features. Such optional features, although unknown to previous software versions, won't prevent those software versions from properly reading the file. Old readers will safely ignore these features.

Patch: an increment of the patch version indicates clarifications or backported features from newer format versions. The backported features may correspond to a major or a minor release.

Except for the epoch, the versioning is for reporting only. Readers should use the feature flag in the header to determine whether they support reading the file.

Introduction

The RNTuple binary format describes the serialized, on-disk representation of an RNTuple data set. The data on disk is organized in **pages** (typically tens to hundreds of kilobytes in size) and several **envelopes** that contain information about the data such as header and footer. The RNTuple format specifies the binary layout of the pages and the envelopes.

Pages and envelopes are meant to be embedded in a data container such as a ROOT file or a set of objects in an object store. Envelopes can reference other envelopes and pages by means of a **locator** or an **envelope link**; for a file embedding, the locator consists of an offset and a size. The RNTuple format does *not* establish a specific order of pages and envelopes.

Every embedding must define an **anchor** that contains the format version supported by the writer, and envelope links (location, compressed and uncompressed size) of the header and footer envelopes.

ROOT File Embedding

When an RNTuple is embedded in a ROOT file, its pages and envelopes are stored in "invisible", non-indexed **RBlob** keys. The RNTuple format does *not* establish a semantic mapping from objects to keys or vice versa. For example, one key may hold a single page or a number of pages of the same cluster. The only relevant means of finding objects is the locator information, consisting of an offset and a size.

For the ROOT file embedding, the `ROOT::RNTuple` object acts as an anchor.

Anchor Schema

The anchor for a ROOT file embedding has the following schema:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Version Epoch          |         Version Major         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Version Minor          |         Version Patch         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                          Seek Header                          +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                         Nbytes Header                         +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                          Len Header                           +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                          Seek Footer                          +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                         Nbytes Footer                         +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                          Len Footer                           +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                         Max Key Size                          +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

When serialized to disk, a 64 bit checksum is appended to the anchor, calculated as the XXH3 hash of all the (serialized) fields of the anchor object.

Note that, since the anchor is serialized as a “classic” TFile key, all integers in the anchor, as well as the checksum, are encoded in big-endian, unlike the RNTuple payload which is encoded in little-endian.

The anchor may evolve in future versions only by appending new fields to the existing schema, but fields will not be removed, renamed or reordered.

`Max Key Size` represents the maximum size of an RBlob (associated to one TFile key). Payloads bigger than that size will be written as multiple RBlobs/TKeys, and the offsets of all but the first RBlob will be written at the end of the first one. This allows bypassing the inherent TKey size limit of 1 GiB.

Compression Block

RNTuple envelopes and pages are wrapped in compression blocks. In order to deserialize a page or an envelope, its compressed and uncompressed size needs to be known.

If the compressed size == uncompressed size, the data is stored unmodified in uncompressed form. Otherwise, data is represented as a series of compressed chunks. Each chunk is prepended with the following 9 bytes header.

```
Byte
0      1      2      3      4      5      6      7      8      9
+------+------+------+------+------+------+------+------+------+
|     Algorithm      | Compressed size    | Uncompressed size |
+------+------+------+------+------+------+------+------+------+
|                         <COMPRESSED DATA>                    |
|                               ...                            |
```

Algorithm: Identifies the compression algorithm used to compress the data. This can take one of the following values

Algorithm	Meaning
‘Z’ ‘L’ ‘\x08’	zlib
‘C’ ‘S’ ‘\x08’	Old Jean-loup Gailly’s deflation algorithm
‘X’ ‘Z’ ‘\x00’	LZMA
‘L’ ‘4’	LZ4; third byte encodes major version number
‘Z’ ‘S’ ‘\x01’	Zstd

Compressed size: An unsigned, little-endian integer that indicates the compressed size of the data that follows the header.

Uncompressed size: An unsigned, little-endian integer that indicates the uncompressed size of the data that follows. The maximum representable value is $(2^{24}) - 1$, i. e. 16777215, and thus each compressed chunk can represent up to 16 MiB of uncompressed data. If the original data is larger than this value, more compressed chunks will follow.

Basic Types

Data stored in envelopes is encoded using the following type system. Note that this type system is independent (and different) from the regular ROOT serialization.

Integer: Integers are encoded in two's complement, little-endian format. They can be signed or unsigned and have lengths up to 64 bit.

String: A string is stored as a 32 bit unsigned integer indicating the length of the string followed by the characters. Strings are ASCII encoded; every character is a signed 8 bit integer.

Compression settings: A 32 bit integer containing both a compression algorithm and the compression level. The compression settings are encoded according to this formula: $\text{settings} = \text{algorithm} * 100 + \text{level}$. The level is between 1 and 9 and is extrapolated to the spectrum of levels of the corresponding algorithm.

Feature Flags

Feature flags are 64 bit integers where every bit represents a certain forward-incompatible feature that is used in the binary format of the RNTuple at hand (see "Versioning Notes"). The most significant bit is used to indicate that one or more flags are active with a bit number higher than 62. That means that readers need to continue reading feature flags as long as their signed integer value is negative.

Readers should gracefully abort reading when they encounter unknown bits set.

At the moment, there are no feature flag bits defined.

Frames

RNTuple envelopes can store records and lists of basic types and other records by means of **frames**.

A frame has the following format

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                              Size                           +-+
|                                                             |T|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               Number of Items (for list frames)               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          FRAME PAYLOAD                        |
|                              ...                              |
```

Size: The absolute value gives the (uncompressed) size in bytes of the frame and the payload.

T(ype): Can be either 0 for a **record frame** or 1 for a **list frame**. The type should be interpreted as the sign bit of the size, i. e. negative sizes indicate list frames.

Number of items: Only used for list frames to indicate the length of the list in the frame payload.

File format readers should use the size provided in the frame to seek to the data that follows a frame instead of summing up the sizes of the elements in the frame. This approach ensures that frames can be extended in future file format versions without breaking the deserialization of older readers.

Locators and Envelope Links

A locator is a generalized way to specify a certain byte range on the storage medium. For disk-based storage, the locator is just byte offset and byte size. For other storage systems, the locator contains enough information to retrieve the referenced block, e.g. in object stores, the locator can specify a certain object ID. The locator has the following format

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             Size                            |T|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                            Offset                             +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Size: If `T` is zero, the number of bytes to read, i. e. the compressed size of the referenced block. Otherwise, the 16 least-significant bits, i. e. bits 0:15, specify the size of the locator itself (see below).

T(ype): Zero for a simple on-disk or in-file locator, 1 otherwise. Can be interpreted as the sign bit of the size, i. e. negative sizes indicate non-standard locators. In this case, the locator should be interpreted like a frame, i. e. size indicates the *size of the locator itself*.

Offset: For on-disk / in-file locators, the 64 bit byte offset of the referenced byte range counted from the start of the file.

For non-standard locators, i. e. `T` == 1, the locator format is as follows

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Size              |   Reserved    |     Type    |T|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         LOCATOR PAYLOAD                       |
|                              ...                              |
```

In this case, the last 8 bits of the size should be interpreted as a locator type. To determine the locator type, the absolute value of the 8 bit integer should be taken. The type can take one of the following values

Type	Meaning	Payload format
0x01	Large locator	64 bit size followed by 64 bit offset

Each locator type follows a given format for the payload (see "Well-Known Payload Formats" below). The range 0x02 - 0x7f is reserved for future use.

Reserved is an 8 bit field that can be used by the storage backend corresponding to the type in order to store additional information about the locator.

An envelope link consists of a 64 bit unsigned integer that specifies the uncompressed size of the envelope followed by a locator.

Well-Known Payload Formats

This section describes the well-known payload formats used in non-standard locators. Note that locators having a different value for *Type* may share a given payload format (see the table above).

Large

Like the standard on-disk locator but with a 64 bit size.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                         Content size                          +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                        Content offset                         +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Content size: the number of bytes to read, i. e. the compressed size of the referenced block.

Content offset: the 64 bit byte offset of the referenced byte range counted from the start of the file.

Envelopes

An Envelope is a data block containing information that describes the RNTuple data. The following envelope types exist

Type	ID	Contents
reserved	0x00	unused and reserved
Header	0x01	RNTuple schema: field and column types
Footer	0x02	Description of clusters
Page list	0x03	Location of data pages

Envelopes have the following format

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Envelope Type ID       |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+        Envelope Length        +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
                        ENVELOPE PAYLOAD
                              ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                           XxHash-3                            +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Envelope type ID: As specified in the table above, encoded in the least significant 16 bits of the first 64 bit integer.

Envelope length: Uncompressed size of the envelope, encoded in the 48 most significant bits of the first 64 bit integer.

XxHash-3: Checksum of the envelope and the payload bytes together.

Note that the compressed size (and also the length) of envelopes is given by the RNTuple anchor (header, footer) or by a locator that references the envelope.

Header Envelope

The header consists of the following elements:

- Feature flag
- String: name of the ntuple
- String: description of the ntuple
- String: identifier of the library or program that writes the data
- List frame: list of field record frames
- List frame: list of column record frames
- List frame: list of alias column record frames
- List frame: list of extra type information

The last four list frames containing information about fields and columns are collectively referred to as **schema description**.

Field Description

Every field record frame of the list of fields has the following contents

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Field Version                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Type Version                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Parent Field ID                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Structural Role        |             Flags             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The block of integers is followed by a list of strings:

- String: field name
- String: type name
- String: type alias

- String: field description

The field version and type version are used for schema evolution.
The structural role of the field can have one of the following values:

Value	Structural role
0x00	Leaf field in the schema tree
0x01	The field is the parent of a collection (e. g., a vector)
0x02	The field is the parent of a record (e. g., a struct)
0x03	The field is the parent of a variant
0x04	The field stores objects serialized with the ROOT streamer

The "flags" field can have any of the following bits set:

Bit	Meaning
0x01	Repetitive field, i. e. for every entry n copies of the field are stored
0x02	Projected field
0x04	Has ROOT type checksum as reported by TClass

If `flag==0x01` (*repetitive field*) is set, the field represents a fixed-size array. For fixed-size arrays, another (sub) field with `Parent Field ID` equal to the ID of this field is expected to be found, representing the array content. The field backing `std::bitmap<N>` is a single repetitive field. (See "Mapping of C++ Types to Fields and Columns").

If `flag==0x02` (*projected field*) is set, the field has been created as a virtual field from another, non-projected source field. If a projected field has attached columns, these columns are alias columns to physical columns attached to the source field. The following restrictions apply on field projections:

- The source field and the target field must have the same structural role, except for an `RNTupleCardinality` field, which must have a collection field as a source.
- For streamer fields and leaf fields, the type name of the source field and the projected field must be identical.
- Projections involving variants or fixed-size arrays are unsupported.
- Projected fields must be on the same schema path of collection fields as the source field. For instance, one can project a vector of structs with floats to individual vectors of floats but cannot project a vector of a vector of floats to a vector of floats.

If `flag==0x04` (*type checksum*) is set, the field metadata contain the checksum of the ROOT streamer info. This checksum is only used for I/O rules in order to find types that are identified by checksum.

Depending on the flags, the following optional values follow:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+               Array Size (if flag 0x01 is set)                +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Source Field ID (if flag 0x02 is set)             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            ROOT Type Checksum (if flag 0x04 is set)           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The order of fields matters: every field gets an implicit field ID which is equal the zero-based index of the field in the serialized list; subfields are ordered from smaller IDs to larger IDs. Top-level fields have their own field ID set as parent ID.

Column Description

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Type             |        Bits on Storage        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            Field ID                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Flags             |      Representation Index     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+            First element index (if flag 0x01 is set)          +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                Min value (if flag 0x02 is set)                +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                Max value (if flag 0x02 is set)                +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The order of columns matter: every column gets an implicit column ID which is equal to the zero-based index of the column in the serialized list. Multiple columns attached to the same field should be attached from smaller to larger IDs.

A field can have multiple alternative column representations. The representation index distinguishes the different representations. For any given cluster, only one of the representations is the primary representation. All the other, secondary representations are **suppressed** in the cluster. All column representations of a cluster need to have the same number of columns, and the number of elements in each of the corresponding columns must be the same. The page list (see "Page List Envelope") indicates suppressed columns through a negative element index. Columns need to be stored in order from smaller to larger representation indexes. The representation index is consecutive starting at zero.

The column type and bits on storage integers can have one of the following values

Type	Bits	Name	Contents
0x00	1	Bit	Boolean value
0x01	8	Byte	An uninterpreted byte, e. g. part of a blob
0x02	8	Char	ASCII character
0x03	8	Int8	Two's complement, 1-byte signed integer
0x04	8	UInt8	1-byte unsigned integer
0x05	16	Int16	Two's complement, little-endian 2-byte signed integer
0x06	16	UInt16	Little-endian 2-byte unsigned integer
0x07	32	Int32	Two's complement, little-endian 4-byte signed integer
0x08	32	UInt32	Little-endian 4-byte unsigned integer
0x09	64	Int64	Two's complement, little-endian 8-byte signed integer
0x0A	64	UInt64	Little-endian 8-byte unsigned integer
0x0B	16	Real16	IEEE-754 half precision float
0x0C	32	Real32	IEEE-754 single precision float
0x0D	64	Real64	IEEE-754 double precision float
0x0E	32	Index32	Parent columns of (nested) collections, counting is relative to the cluster
0x0F	64	Index64	Parent columns of (nested) collections, counting is relative to the cluster
0x10	96	Switch	Tuple of a kIndex64 value followed by a 32 bits dispatch tag to a column ID
0x11	16	SplitInt16	Like Int16 but in split + zigzag encoding
0x12	16	SplitUInt16	Like UInt16 but in split encoding
0x13	64	SplitInt32	Like Int32 but in split + zigzag encoding
0x14	32	SplitUInt32	Like UInt32 but in split encoding
0x15	64	SplitInt64	Like Int64 but in split + zigzag encoding
0x16	64	SplitUInt64	Like UInt64 but in split encoding

Type	Bits	Name	Contents
0x17	16	SplitReal16	Like Real16 but in split encoding
0x18	32	SplitReal32	Like Real32 but in split encoding
0x19	64	SplitReal64	Like Real64 but in split encoding
0x1A	32	SplitIndex32	Like Index32 but pages are stored in split + delta encoding
0x1B	64	SplitIndex64	Like Index64 but pages are stored in split + delta encoding
0x1C	10-31	Real32Trunc	IEEE-754 single precision float with truncated mantissa
0x1D	1-32	Real32Quant	Real value contained in a specified range with an underlying quantized integer representation

The "split encoding" columns apply a byte transformation encoding to all pages of that column and in addition, depending on the column type, delta or zigzag encoding:

Split (only): Rearranges the bytes of elements: All the first bytes first, then all the second bytes, etc.

Delta + split: The first element is stored unmodified, all other elements store the delta to the previous element. Followed by split encoding.

Zigzag + split: Used on signed integers only; it maps x to $2x$ if x is positive and to $-(2x+1)$ if x is negative. Followed by split encoding.

Note: these encodings always happen within each page, thus decoding should be done page-wise, not cluster-wise.

The `Real32Trunc` type column is a variable-sized floating point column with lower precision than `Real32` and `SplitReal32`. It is an IEEE-754 single precision float with some of the mantissa's least significant bits truncated.

The `Real32Quant` type column is a variable-sized real column that is internally represented as an integer within a specified range of values. For this column type, flag 0x02 (column with range) is always set (see paragraphs below).

Future versions of the file format may introduce additional column types without changing the minimum version of the header or introducing a feature flag. Old readers need to ignore these columns and fields constructed from such columns. Old readers can, however, figure out the number of elements stored in such unknown columns.

The "flags" field can have one of the following bits set

Bit	Meaning
0x01	Deferred column: index of first element in the column is not zero

Bit	Meaning
0x02	Column with a range of possible values

If flag 0x01 (deferred column) is set, the index of the first element in this column is not zero, which happens if the column is added at a later point during write. In this case, an additional 64 bit integer containing the first element index follows the representation index field. Compliant implementations should yield synthetic data pages made up of 0x00 bytes when trying to read back elements in the range $[0, \text{firstElementIndex} - 1]$. This results in zero-initialized values in the aforementioned range for fields of any supported C++ type, including `std::variant<Ts...>` and collections such as `std::vector<T>`. The leading zero pages of deferred columns are *not* part of the page list, i.e. they have no page locator. In practice, deferred columns only appear in the schema extension record frame (see “Footer Envelope”).

If flag 0x02 (column with range) is set, the column metadata contains the inclusive range of valid values for this column (used e. g. for quantized real values). The range is represented as a min and a max value, specified as IEEE 754 little-endian double precision floats.

If the index of the first element is negative (sign bit set), the column is deferred *and* suppressed. In this case, no (synthetic) pages exist up to and including the cluster of the first element index. See “Page List Envelope” for further information about suppressed columns.

Alias Columns

An alias column has the following format

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Physical Column ID                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            Field ID                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Alias columns do not have associated data pages. Instead, their data comes from another column referred to below as **physical column**. The first 32 bit integer references the physical column ID. The second 32 bit integer references the associated **projected field**. A projected field is a field using alias columns to present available data by an alternative C++ type. Alias columns have no prescribed column ID of their own, since alias columns are not referenced. In the footer and page list envelopes, only physical column IDs must be referenced. However, columns should be attached to projected fields in their serialization order (first header, then footer).

Extra Type Information

Certain field types may come with additional information required, e. g., for schema evolution. The type information record frame has the following contents followed by a string containing the type name.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Content Identifier                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Type Version                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The combination of type version, type name, and content identifier should be unique in the list. However, not every type needs to provide additional type information.

The following kinds of content are supported:

Content identifier	Meaning of content
0x00	Serialized ROOT streamer info; see notes

The serialized ROOT streamer info is not bound to a specific type. It is the combined streamer information from all fields serialized by the ROOT streamer. Writers set the version to zero and use an empty type name. Readers should ignore the type-specific information. The format of the content is a ROOT streamed `TList` of `TStreamerInfo` objects.

Footer Envelope

The footer envelope has the following structure:

- Feature flags
- Header checksum (XxHash-3 64 bit)
- Schema extension record frame
- List frame of cluster group record frames

The header checksum can be used to cross-check that header and footer belong together. The meaning of the feature flags is the same as for the header. The header flags do not need to be repeated. Readers should combine (logical `or` of the bits) the feature flags from header and footer for the full set of flags.

Schema Extension Record Frame

The schema extension record frame contains an additional schema description that is incremental with respect to the schema contained in the header (see

“Header Envelope”). Specifically, it is a record frame with the following four fields (identical to the last four fields in the header envelope):

- List frame: list of field record frames
- List frame: list of column record frames
- List frame: list of alias column record frames
- List frame: list of extra type information

In general, a schema extension is optional, and thus this record frame might be empty. The interpretation of the information contained therein should be identical as if it was found directly at the end of the header. This is necessary when fields have been added during writing.

Note that the field IDs and physical column IDs given by the serialization order should continue from the largest IDs found in the header.

Note that is it possible to extend existing fields by additional column representations. This means that columns of the extension header may point to fields of the regular header.

Cluster Group Record Frame

The cluster group record frame references the page list envelopes for groups of clusters. A cluster group record frame has the following contents followed by a page list envelope link.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                      Minimum Entry Number                     +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                          Entry Span                           +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Number of clusters                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

To compute the minimum entry number, take first entry number from all clusters in the cluster group, and take the minimum among these numbers. The entry span is the number of entries that are covered by this cluster group. The entry range allows for finding the right page list for random access requests to entries. The number of clusters information allows for using consistent cluster IDs even if cluster groups are accessed non-sequentially.

Page List Envelope

The page list envelope contains cluster summaries and page locations. It has the following structure

- Header checksum (XxHash-3 64 bit)
- List frame of cluster summary record frames
- Nested list frame of page locations

Cluster Summary Record Frame

The cluster summary record frame contains the entry range of a cluster:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                      First Entry Number                       +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      Number of Entries                        |
+                                               +-+-+-+-+-+-+-+-+
|                                               |     Flags     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The order of the cluster summaries defines the cluster IDs, starting from the first cluster ID of the cluster group that corresponds to the page list.

Flag 0x01 is reserved for a future specification version that will support sharded clusters. The future use of sharded clusters will break forward compatibility and thus introduce a corresponding feature flag. For now, readers should abort when this flag is set. Other flags should be ignored.

Page Locations

The page locations are stored in a nested list frame as follows. A top-most list frame where every item corresponds to a cluster. The order of items corresponds to the cluster IDs as defined by the cluster groups and cluster summaries.

Every item of the top-most list frame consists of an outer list frame where every item corresponds to a column. Every item of the outer list frame is an inner list frame whose items correspond to the pages of the column in the cluster. The inner list is followed by a 64 bit signed integer element offset and, unless the column is suppressed, the 32 bit compression settings. See “Suppressed Columns” for additional details. Note that the size of the inner list frame includes the element offset and compression settings. The order of the outer items must match the order of columns in the header and the extension header (small to large).

The order of the inner items must match the order of pages or elements, respectively. Every inner item (that describes a page) has the following structure followed by a locator for the page.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      Number of Elements                     |C|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Note that locators for byte ranges in a file may reference identical byte ranges, but they must not reference arbitrarily overlapping byte ranges.

C(hecksum): If set, an XxHash-3 64 bit checksum of the compressed page data is stored just after the page. This bit should be interpreted as the sign bit of the number of elements, i. e. negative values indicate pages with checksums. Note that the page size stored in the locator does *not* include the checksum.

Note that we do not need to store the uncompressed size of the page because the uncompressed size is given by the number of elements in the page and the element size. We do need, however, the per-column and per-cluster element offset in order to read a certain entry range without inspecting the meta-data of all the previous clusters.

The hierarchical structure of the frames in the page list envelope is as follows:

```
# this is `List frame of cluster group record frames`
# mentioned above
- Top-most cluster list frame (one item for each cluster
|    in this RNTuple)
|
|---- Cluster 1 column list frame (outer list frame,
|      |   one item for each column in this RNTuple)
|      |
|      |---- Column 1 page list frame (inner list frame,
|      |      |   one item for each page in this column)
|      |      |
|      |      |---- Page 1 description (inner item)
|      |      |---- Page 2 description (inner item)
|      |      | ...
|      |---- Column 1 element offset (Int64),
|      |         negative if the column is suppressed
|      |---- Column 1 compression settings (UInt32),
|      |         available only if the column is not suppressed
|      |
|      |---- Column 2 page list frame
|      | ...
|
|---- Cluster 2 column list frame
| ...
```

In order to save space, the page descriptions (inner items) are *not* in a record frame. If at a later point more information per page is needed, the page list envelope can be extended by additional list and record frames.

Suppressed Columns

If the element offset in the inner list frame is negative (sign bit set), the column is suppressed. Writers should write the lowest `int64_t` value, readers should check for a negative value. Suppressed columns always have an empty list of pages. Suppressed columns omit the compression settings in the inner list frame.

Suppressed columns belong to a secondary column representation (see "Column Description") that is inactive in the current cluster. The number of columns and the absolute values of the element offsets of primary and secondary representations are identical. When reading a field of a certain entry, this assertion allows for searching the corresponding cluster and column element indexes using any of the column representations. It also means that readers need to get the element index offset and the number of elements of suppressed columns from the corresponding columns of the primary column representation.

In every cluster, every field has exactly one primary column representation. All other representations must be suppressed. Note that the primary column representation can change from cluster to cluster.

Mapping of C++ Types to Fields and Columns

This section is a comprehensive list of the C++ types with RNTuple I/O support. Within the supported type system complex types can be freely composed, e. g. `std::vector<MyEvent>` or `std::vector<std::vector<float>>`.

Fundamental Types

The following fundamental types are stored as `leaf` fields with a single column each. Fundamental C++ types can potentially be stored in multiple possible column types. The possible combinations are marked as `W` in the following table. Additionally, some types allow for reading from certain column types but not to write into them. Such cases are marked as `R` in the table.

Possibly available `const` and `volatile` qualifiers of the C++ types are ignored for serialization. The default column for serialization is denoted with

Column Type \ C++ Type	`bool`	`std::byte`	`char`	`int8_t`	`uint8_t`	`int16_t`	`uint16_t`	`int32_t`	`uint32_t`	`int64_t`	`uint64_t`	`float`	`double`
Bit	W*		R	R	R	R	R	R	R	R	R		
Byte		W*											
Char	R		W*	R	R	R	R	R	R	R	R		
Int8	R		R	W*	R	R	R	R	R	R	R		
UInt8	R		R	R	W*	R	R	R	R	R	R		
(Split)Int16	R		R	R	R	W*	R	R	R	R	R		
(Split)UInt16	R		R	R	R	R	W*	R	R	R	R		
(Split)Int32	R		R	R	R	R	R	W*	R	R	R		
(Split)UInt32	R		R	R	R	R	R	R	W*	R	R		
(Split)Int64	R		R	R	R	R	R	R	R	W*	R		
(Split)UInt64	R		R	R	R	R	R	R	R	R	W*		
Real16												W	W
(Split)Real32												W*	W
(Split)Real64												R	W*
Real32Trunc												W	W
Real32Quant												W	W

an asterisk. If the ntuple is stored uncompressed, the default changes from split encoding to non-split encoding where applicable.

Low-precision Floating Points

The ROOT type `Double32_t` is stored on disk as a `double` field with a `SplitReal32` column representation. The field's type alias is set to `Double32_t`.

Stdlib Types and Collections

Generally, collections have a parent column of type `(Split)Index32` or `(Split)Index64`. The parent column stores the offsets of the next collection entries relative to the cluster. For instance, a `std::vector<float>` with the values `{1.0}`, `{}`, `{1.0, 2.0}` for the first 3 entries results in an index column `[1, 1, 3]` and a value column `[1.0, 1.0, 2.0]`.

std::string

A string is stored as a single field with two columns. The first (principle) column is of type `(Split)Index[64|32]`. The second column is of type `Char`.

std::vector<T> and ROOT::RVec<T>

STL vector and ROOT's RVec have identical on-disk representations. They are stored as two fields:

- Collection parent field whose principal column is of type `(Split)Index[64|32]`.
- Child field of type `T`, which must by a type with RNTuple I/O support.

The name of the child field is `_0`.

For RVecs, ROOT will always store the fully qualified type name `ROOT::VecOps::RVec<T>`. Implementations should also be able to parse the shorter alias `ROOT::Vec<T>`.

std::array<T, N> and array type of the form T[N]

Fixed-sized arrays are stored as two fields:

- A repetitive field of type `std::array<T, N>` with no attached columns. The array size `N` is stored in the field meta-data.
- Child field of type `T` named `_0`, which must be a type with RNTuple I/O support.

Note that `T` can itself be an array type, which implies support for multidimensional C-style arrays.

std::variant<T1, T2, . . . , Tn>

Variants are stored in $n+1$ fields:

- Variant parent field with one column of type `Switch`; the dispatch tag points to the active subfield number.
- Child fields of types `T1`, . . . , `Tn`; their names are `_0`, `_1`, . . .

The dispatch tag ranges from 1 to n. A value of 0 indicates that the variant is in the invalid state, i. e., it does not hold any of the valid alternatives. Variants must not have more than 125 subfields. This follows common compiler implementation limits.

std::pair<T1, T2>

A pair is stored using an empty parent field with two subfields, one of type `T1` and one of type `T2`. `T1` and `T2` must be types with RNTuple I/O support. The child fields are named `_0` and `_1`.

std::tuple<T1, T2, . . . , Tn>

A tuple is stored using an empty parent field with n subfields of type `T1`, `T2`, . . . , `Tn`. All types must have RNTuple I/O support. The child fields are named `_0`, `_1`, . . .

std::bitset<N>

A bitset is stored as a repetitive leaf field with an attached `Bit` column. The bitset size `N` is stored as repetition parameter in the field meta-data. Within the repetition blocks, bits are stored in little-endian order, i. e. the least significant bits come first.

std::unique_ptr<T>, std::optional<T>

A unique pointer and an optional type have the same on disk representation. They are represented as a collection of `T`s of zero or one elements. The collection parent field has a principal column of type `(Split)Index[64|32]`. It has a single subfield named `_0` for `T`, where `T` must have RNTuple I/O support. Note that RNTuple does not support polymorphism, so the type `T` is expected to be `T` and not a child class of `T`.

std::set<T>, std::unordered_set<T>, std::multiset<T>, std::unordered_multiset<T>

While STL (unordered) (multi)sets by definition are associative containers (i. e., elements are referenced by their keys, which in the case for sets are equal to the values), on disk they are represented as sequential collections. This means that they have the same on-disk representation as `std::vector<T>`, using two fields:

- Collection parent field whose principal column is of type `(Split)Index[64|32]`.
- Child field of type `T`, which must be a type with RNTuple I/O support. The name of the child field is `_0`.

std::map<K, V>, std::unordered_map<K, V>, std::multimap<K, V>, std::unordered_multimap<K, V>

An (unordered) (multi)map is stored using a collection parent field, whose principal column is of type `(Split)Index[64|32]` and a child field of type `std::pair<K, V>` named `_0`.

std::atomic<T>

Atomic types are stored as a leaf field with a single subfield named `_0`. The parent field has no attached columns. The subfield corresponds to the inner type `T`.

User-defined enums

User-defined enums are stored as a leaf field with a single subfield named `_0`. The parent field has no attached columns. The subfield corresponds to the integer type that underlies the enum. Unscoped and scoped enums are supported as long as the enum has a dictionary.

User-defined classes

User-defined classes might behave either as a record or as a collection of elements of a given type. The behavior depends on whether the class has an associated collection proxy.

Regular class / struct

User defined C++ classes are supported with the following limitations:

- The class must have a dictionary.
- All persistent members and base classes must be themselves types with RNTuple I/O support.
- Transient members must be marked, e. g. by a `//!` comment.
- The class must not be in the `std` namespace.
- The class must be empty or splittable (e. g., the class must not provide a custom streamer).

- There is no support for polymorphism, i. e. a field of class `A` cannot store class `B` that derives from `A`.
- Virtual inheritance is unsupported.

User classes are stored as a record parent field with no attached columns. Direct base classes and persistent members are stored as subfields with their respective types. The field name of member subfields is identical to the C++ field name. The field name of base class subfields are numbered and preceded by a colon (`:`), i. e. `:_0`, `:_1`, ...

Classes with an associated collection proxy

User classes that specify a collection proxy behave as collections of a given value type.

The on-disk representation of non-associative collections is identical to a `std::vector<T>`, using two fields:

- Collection parent field whose principal column is of type `(Split)Index[64|32]`.
- Child field of type `T`, which must be a type with RNTuple I/O support.

The on-disk representation of associative collections is identical to a `std::map<K, V>`, using two fields:

- Collection parent field whose principal column is of type `(Split)Index[64|32]`.
- Child field of type `std::pair<K, V>`, where `K` and `V` must be types with RNTuple I/O support.

N.B., proxy-based associative collections are supported in the RNTuple binary format, but currently are not implemented in ROOT's RNTuple reader and writer. This will be added in the future.

ROOT::RNTupleCardinality

A field whose type is `ROOT::RNTupleCardinality<SizeT>` is associated to a single column of type `(Split)Index[32|64]`. This field presents the offsets in the index column as lengths that correspond to the cardinality of the pointed-to collection. It is meant to be used as a projected field and only for reading the size of a collection.

The value for the i-th element is computed by subtracting the $(i-1)$-th value from the i-th value in the index column. If $i == 0$, i. e. it falls on the start of a cluster, the $(i-1)$-th value in the index column is assumed to be 0, e. g. given the index column values `[1, 1, 3]`, the values yielded by `RNTupleCardinality` shall be `[1, 0, 2]`.

The `SizeT` template parameter defines the in-memory integer type of the collection size. The valid types are `std::uint32_t` and `std::uint64_t`.

ROOT streamed types

A field with the structural role 0x04 ("streamer") represents an object serialized by the ROOT streamer into a single `Byte` column. It can have any type supported by `TClass` (even types that are not available in the native RNTuple type system). The first (principal) column is of type `(Split)Index[32|64]`. The second column is of type `Byte`. In effect, the column representation is identical to a collection of `std::byte`.

Untyped collections and records

Untyped collections and records are fields with a collection or record role and an empty type name. Only top-level fields as well as direct subfields of untyped fields may be untyped. Except for the empty type name, untyped collections have the same on-disk representation as `std::vector` and untyped records have the same on-disk representation as a user-defined class.

Limits

This section summarizes key design limits of RNTuple data sets. The limits refer to a single RNTuple and do not consider combinations/joins such as "friends" and "chains".

Limit	Value	Reason / Comment
Maximum volume	10 PB (theoretically more)	Assuming 10 k cluster groups of 10 k clusters of 100 MB
Maximum number of elements, entries	2^{63}	Using default `(Split)Index64`, otherwise 2^{32}
Maximum cluster & entry size	8 TB (depends on pagination)	Assuming limit of 4 B pages of 4 kB each
Maximum page size	2 B elements, 256 MB - 24 GB	#elements · element size
Maximum element size	8 kB	16 bit for number of bits per element
Maximum number of column types	64 k	16 bit for column type
Maximum envelope size	2^{48} B (~280 TB)	Envelope header encoding
Maximum frame size	2^{62} B, 4 B items (list frame)	Frame preamble encoding
Maximum field / type version	4 B	Field meta-data encoding
Maximum number of fields, columns	4 B (foreseen: <10 M)	32 bit column / field IDs, list frame limit

Limit	Value	Reason / Comment
Maximum number of cluster groups	4 B (foreseen: <10 k)	List frame limits
Maximum number of clusters per group	4 B (foreseen: <10 k)	List frame limits, cluster group summary encoding
Maximum number of pages per cluster per column	4 B	List frame limits
Maximum number of entries per cluster	2^{56}	Cluster summary encoding
Maximum string length (meta-data)	4 GB	String encoding
Maximum RBlob size	128 PiB	1GiB/8B · 1GiB (with maxKeySize == 1 GiB, offsetSize == 8 B)

Naming Specification

The name of an RNTuple as well as the name of a field cannot be represented with an empty string when persistified (e. g. when written to disk). Furthermore, the allowed character set is restricted to Unicode characters encoded as UTF-8, with the following exceptions:

- All control codes. These notably include newline (U+000A) and horizontal tab (U+0009).
- Full stop (U+002E '.')
- Space (U+0020 ' ')
- Backslash (U+005C '\')
- Slash (U+002F '/')

Defaults

This section summarizes default settings of `RNTupleWriteOptions`.

Default	Value
Approximate Zipped Cluster	128 MiB
Max Unzipped Cluster	1280 MiB
Max Unzipped Page	1 MiB

Glossary

Anchor

The anchor is a data block that represents the entry point to an RNTuple. The anchor is specific to the RNTuple container in which the RNTuple data are embedded (e. g., a ROOT file or an object store). The anchor must provide the information to load the header and the footer **envelopes**.

Cluster

A cluster is a set of **pages** that contain all the data belonging to an entry range. The data set is partitioned in clusters. A typical cluster size is tens to hundreds of megabytes.

Column

A column is a storage backed vector of a number of **elements** of a simple type. Column elements have a fixed bit-length that depends on the column type. Some column types allow setting the bit lengths within specific limits (e. g. for floats with truncated mantissa).

Envelope

An envelope is a data block with RNTuple meta-data, such as the header and the footer.

Field

A field describes a serialized C++ type. A field can have a hierarchy of subfields representing a composed C++ type (e. g., a vector of integers). A field has zero, one, or multiple **columns** attached to it. The columns contain the data related to the field but not to its subfields, which have their own columns.

Frame

A frame is a byte range with metadata information in an **envelope**. A frame starts with its size and thus can be extended in a forward-compatible way.

Locator

A locator is a generalized way to identify a byte range in the RNTuple container. For a file container, for instance, a locator consists of an offset and a size.

Page

A page is segment of a column. Columns are partitioned in pages. A page is a unit of compression. Typical page sizes are of the order of tens to hundreds of kilobytes.

Indications of Size

In this document, the `length` of something (e. g., a page) refers to its size in bytes in memory, uncompressed. The `size` of something refers to the size in bytes on disk, possibly compressed.

Notes on Backward and Forward Compatibility

Note that this section covers the backward and forward compatibility of the binary format itself. It does not discuss schema evolution of the written types.

Readers supporting a certain version of the specification should support reading files that were written according to previous versions of the same epoch.

Readers should support reading data written according to *newer* format versions of the same epoch in the following way:

- Unknown trailing information in the anchor, in envelopes, and in frames should be ignored. For instance, when reading frames, readers should continue reading after the frame-provided frame length rather than summing up the lengths of the known contents of the frame. Checksum verification, however, should still take place and must include both known and unknown contents.
- Unknown column, cluster, or field flags should be ignored.
- Unknown IDs for extra type information should be ignored.
- When a reader encounters an unknown column type or an unknown field type, field version or field structure, it should ignore the entire top-level field this column or field belongs to. It should also ignore any projected fields and alias columns whose source fields or columns are already ignored.
- When a reader encounters an unknown feature flag, it must refuse reading any further.

Writers using format features that will prevent older readers from correctly reading the data must set the corresponding feature flags.

Writers should write in the anchor which format version they support, independent of whether they use the all the features that this version provides. Only the feature flags signal which features are actually used in this particular instance.