Zarr specs

Zarr Developers

Dec 02, 2022

CONTENTS:

A good starting point is the *Zarr core specification (version 3.0)*.

CHAPTER

ONE

CORE SPECIFICATION

Under construction.

1.1 Zarr core specification (version 3.0)

Editor's draft 25 May 2022

Specification URI:
https://purl.org/zarr/spec/core/3.0

Editors:

- Alistair Miles (@alimanfoo), Wellcome Sanger Institute
- Jonathan Striebel (@jstriebel), Scalable Minds
- Jeremy Maitin-Shepard (@jbms), Google

Corresponding ZEP:
ZEP 1 — Zarr specification version 3

Issue tracking and discussion overview:
GitHub project board

Suggest an edit for this spec:
GitHub editor

Suggest extensions or other changes as a Zarr Enhancement Proposal (ZEP):
ZEP 0 — Purpose and process

1.1.1 Abstract

This specification defines the Zarr format for N-dimensional typed arrays.

1.1.2 Status of this document

Warning: This document is a draft for review and subject to changes. It will become final when the Zarr Enhancement Proposal (ZEP) 1 is approved via the ZEP process.

1.1.3 Introduction

This specification defines a format for multidimensional array data. This type of data is common in scientific and numerical computing applications. Many domains are facing computational challenges as increasingly large volumes of data are being generated, for example, via high resolution microscopy, remote sensing imagery, genome sequencing or numerical simulation. The primary motivation for the development of Zarr has been to help address this challenge by enabling the storage of large multidimensional arrays in a way that is compatible with parallel and/or distributed computing applications.

This specification is intended to supersede the Zarr storage specification version 2 (Zarr v2). The Zarr v2 specification has been implemented in several programming languages and has been used successfully to store and analyse large scientific datasets from a variety of domains. However, as experience has been gained, it has become clear that there are several opportunities for modest but useful improvements to be made in the format, and for establishing a foundation that allows for greater interoperability, whilst also enabling a variety of more advanced and specialised features to be explored and developed.

This specification also draws heavily on the N5 API and file-system specification, which was developed in parallel to Zarr v2 and has many of the same design goals and features. This specification defines a core set of features at the intersection of both Zarr v2 and N5, and so aims to provide a common target that can be fully implemented across multiple programming environments and serve a wide range of applications.

In particular, we highlight the following areas motivating the development of this specification.

Distributed storage

The Zarr v2 specification was originally developed and implemented for use with local filesystem storage only. It then became clear that the same format could also be used with distributed storage systems, including cloud object stores such as Amazon S3, Google Cloud Storage or Azure Blob Storage. However, distributed storage systems have a number of important differences from local file systems, both in terms of the features they support and their performance characteristics. For example, cloud stores have much greater latency per request than local file systems, and this means that certain operations such as exploring a hierarchy of arrays using the Zarr v2 format can be unacceptably slow. Workarounds have been developed, such as the use of metadata consolidation, but there are opportunities for modifications to the core format that address these issues directly and work more performantly across a range of underlying storage systems with varying features and latency characteristics. For example, this specification aims to minimise the number of storage requests required when opening and exploring a hierarchy of arrays.

Interoperability

While the Zarr v2 and N5 specifications have each been implemented in multiple programming languages, there is currently not feature parity across all implementations. This is in part because the feature set includes some features that are not easily translated or supported across different programming languages. This specification aims to define a set of core features that are useful and sufficient to address a significant fraction of use cases, but are also straightforward to implement fully across different programming languages. Additional functionality can then be layered via extensions, some of which may aim for wide adoption, some of which may be more specialised and have more limited implementation.

Extensibility

The development of systems for storage of very large array-like data is a very active area of research and development, and there are many possibilities that remain to be explored. A goal of this specification is to define a format with a number of clear extension points and mechanisms, in order to provide a framework for freely building on and exploring these possibilities. We aim to make this possible, whilst also providing pathways for a graceful degradation of functionality where possible, in order to retain interoperability. We also aim to provide a framework for community-defined extensions, which can be developed and published independently without requiring centralised coordination of all specifications.

See *extensions* below.

Stability Policy

This core specification adheres to a `MAJOR.MINOR` version number format. A zarr implementation that provides the read and write API by implementing this specification can be considered compatible with all datasets following the specification with the same major version number.

Notably, this excludes extensions, codecs and stores from the compatibility of the core specification. However, versioned extensions and stores are also expected to follow this stability policy.

This means that implementations based on a `3.X` specification will be able to read and write to datasets that follow any `3.Y` specification, as long as the following conditions are met:

- only optional extensions or those supported by the implementation are used
- only codecs supported by the implementation are used
- the store must be supported by the implementation

For details, please see the *zarr_format* metadata entry.

Questions that still need to be resolved

We solicit feedback on the following area during the review period:

- Should core metadata and user attributes be stored together or separate documents? (See https://github.com/zarr-developers/zarr-specs/issues/72)
- We want to verify if the extension mechanisms fit different use cases or if they are too restrictive. See https://github.com/zarr-developers/zarr-specs/issues/89 and https://github.com/zarr-developers/zarr-specs/issues/169 for discussion on the topic.
- Node name case sensitivity: The node name is now case sensitive. This may make store implementation more complicated as some backends might not be (like some specific filesystem / object store), and we may want to recommend a standard escaping mechanism in those cases. https://github.com/zarr-developers/zarr-specs/issues/57
- Node name character set: We solicit feedback on whether store implementation should support full unicode. https://github.com/zarr-developers/zarr-specs/issues/56
- Should named dimensions be part of the core metadata spec? https://github.com/zarr-developers/zarr-specs/issues/73 https://github.com/zarr-developers/zarr-specs/pull/162

1.1.4 Document conventions

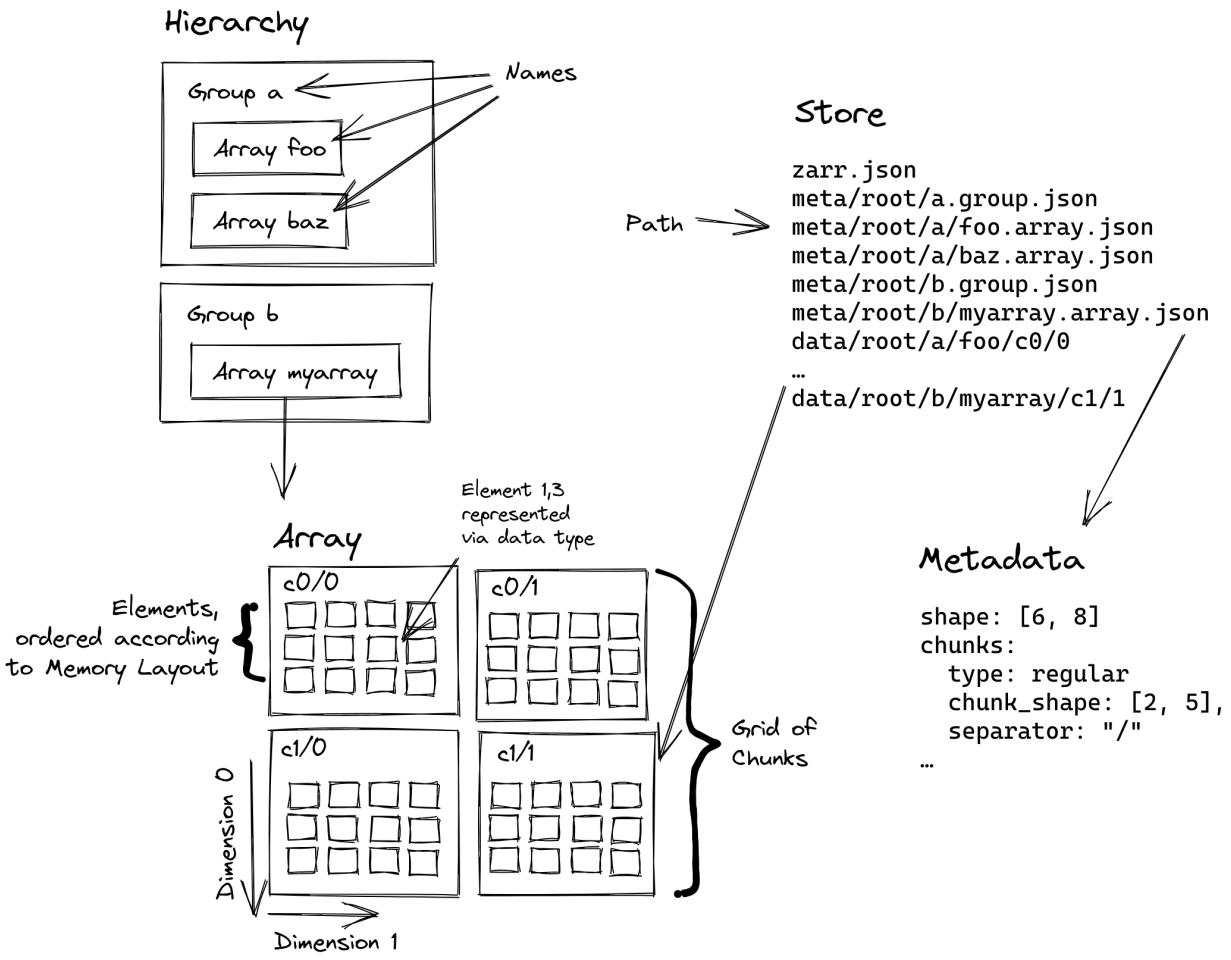
Conformance requirements are expressed with a combination of descriptive assertions and [RFC2119] terminology. The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in the normative parts of this document are to be interpreted as described in [RFC2119]. However, for readability, these words do not appear in all uppercase letters in this specification.

All of the text of this specification is normative except sections explicitly marked as non-normative, examples, and notes. Examples in this specification are introduced with the words "for example".

1.1.5 Concepts and terminology

This section introduces and defines some key terms and explains the conceptual model underpinning the Zarr format.

The following figure illustrates the first part of the terminology:



Hierarchy

A Zarr hierarchy is a tree structure, where each node in the tree is either a *group* or an *array*. Group nodes may have children but array nodes may not. All nodes in a hierarchy have a *name* and a *path*.

Group

A group is a node in a *hierarchy* that may have child nodes.

Array

An array is a node in a *hierarchy*. An array is a data structure with zero or more *dimensions* whose lengths define the *shape* of the array. An array contains zero or more data *elements*. All *elements* in an array conform to the same *data type*. An array may not have child nodes.

Name

Each node in a *hierarchy* has a name, which is a string of characters with some additional constraints defined in the section on *node names* below. Two sibling nodes cannot have the same name. The root node does not have a name and is the empty string "".

Path

Each node in a *hierarchy* has a path which uniquely identifies that node and defines its location within the *hierarchy*. The path is a string, formed by joining together the "/" character, followed by the *name* of each ancestor node separated by the "/" character, followed by the *name* of the node itself. For example, the path "/foo/bar" identifies a node named "bar", whose parent is named "foo", whose parent is the root of the hierarchy. The path "/" identifies the root node.

A path always starts with `/` and cannot end with `/`, because node names cannot contain `/`.

Dimension

An *array* has a fixed number of zero or more dimensions. Each dimension has an integer length. This specification only considers the case where the lengths of all dimensions are finite. However, *extensions* may be defined which allow a dimension to have an infinite or variable length.

Shape

The shape of an *array* is the tuple of *dimension* lengths. For example, if an *array* has 2 *dimensions*, where the length of the first *dimension* is 100 and the length of the second *dimension* is 20, then the shape of the *array* is (100, 20). A shape can be the empty tuple in the case of zero-dimension arrays (scalar)

Element

An *array* contains zero or more elements. Each element can be identified by a tuple of integer coordinates, one for each *dimension* of the *array*. If all *dimensions* of an *array* have finite length, then the number of elements in the *array* is given by the product of the *dimension* lengths.

Data type

A data type defines the set of possible values that an *array* may contain, and a default binary representation (i.e., sequence of bytes) for each possible value. For example, the 32-bit signed integer data type defines binary representations for all integers in the range 2,147,483,648 to 2,147,483,647. This specification only defines a limited set of data types, but extensions may define other data types.

Chunk

An *array* is divided into a set of chunks, where each chunk is a hyperrectangle defined by a tuple of intervals, one for each *dimension* of the *array*. The chunk shape is the tuple of interval lengths, and the chunk size (i.e., number of *elements* contained within the chunk) is the product of its interval lengths.

The chunk shape elements are non-zero when the corresponding dimensions of the arrays are of non-zero length.

Grid

The *chunks* of an *array* are organised into a grid. This specification only considers the case where all *chunks* have the same chunk shape and the chunks form a regular grid. However, extensions may define other grid types such as rectilinear grids.

Memory layout

An *array* is associated with a memory layout which defines how to construct a binary representation of a single *chunk* by organising the binary values of the *elements* within the *chunk* into a single contiguous

sequence of bytes. This specification defines two types of memory layout based on "C" (row-major) and "F" (column-major) ordering of *elements*, but extensions may define other memory layouts.

Metadata document

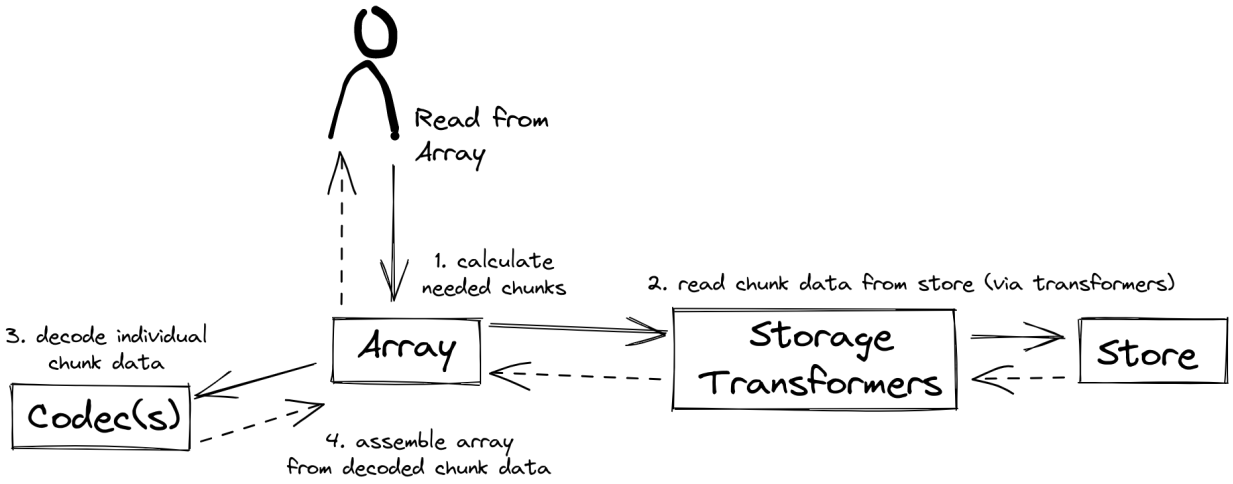
Each *array* in a *hierarchy* is represented by a metadata document, which is a machine-readable document containing essential processing information about the node. For example, an *array* metadata document will specify the number of *dimensions*, *shape*, *data type*, *grid*, *memory layout* and *codec* for that *array*.

Groups can have an optional metadata document which provides extra information about a group.

Store

The *metadata documents* and encoded *chunk* data for all nodes in a *hierarchy* are held in a store as raw bytes. To enable a variety of different store types to be used, this specification defines an *Abstract store interface* which is a common set of operations that stores may provide. For example, a directory in a file system can be a zarr store, where keys are file names, values are file contents, and files can be read, written, listed or deleted via the operating system. Equally, an S3 bucket can provide this interface, where keys are resource names, values are resource contents, and resources can be read, written or deleted via HTTP.

The following figure illustrates the codec, store and storage transformer terminology for a use case of reading from an array:



Codec

An *array* may be associated with a list of *codecs*. Each codec specifies a bidirectional transform (an *encode* transform and a *decode* transform).

Each codec has an *encoded representation* and a *decoded representation*; each of these two representations are defined to be either:

- a multi-dimensional array of some shape and data type, or
- a byte string.

Logically, a codec `c` must define three properties:

- `c.compute_encoded_representation_type(decoded_representation_type)`, a procedure that determines the encoded representation based on the decoded representation and any codec parameters. In the case of a decoded representation that is a multi-dimensional array, the shape and data type of the encoded representation must be computable based only on the shape and data type, but not the actual element values, of the encoded representation. If the `decoded_representation_type` is not supported, this algorithm must fail with an error.

- `c.encode(decoded_value)`, a procedure that computes the encoded representation, and is used when writing an array.
- `c.decode(encoded_value, decoded_representation_type)`, a procedure that computes the decoded representation, and is used when reading an array.

If more than one codec is specified for an array, each codec is applied sequentially; when encoding, the encoded output of codec `i` serves as the decoded input of codec `i+1`, and similarly when decoding, the decoded output of codec `i+1` serves as the encoded input to codec `i`.

Storage transformer

To provide performance enhancements or other optimizations, storage transformers may intercept and alter the storage keys and bytes of an *array* before they reach the underlying physical storage. Upon retrieval, the original keys and bytes are restored within the transformer. Any number of *predefined storage transformers* can be registered and stacked. In contrast to codecs, strorage transformers can act on the a complete array, rather than individual chunks. See the *storage transformers details* below.

1.1.6 Node names

The root node does not have a name and is the empty string `""`. Except for the root node, each node in a hierarchy must have a name, which is a string of characters. To ensure consistent behaviour across different storage systems, the following constraints apply to node names:

- must not be the empty string (“”)
- must use only characters in the sets `a-z`, `A-Z`, `0-9`, `-_.`
- must not be a string composed only of period characters, e.g. “.” or “..”

Node names are case sensitive, e.g., the names “foo” and “FOO” are **not** identical.

Note: The Zarr core development team recognises that restricting the set of allowed characters creates an impediment and bias against users of different languages. We are actively discussing whether the full Unicode character set could be allowed and what technical issues this would entail. If you have experience or views please comment on issue #56.

Note: The underlying store might pose additional restriction on node names, such as the following:

- 260 characters path length limit in Windows
- 1,024 bytes UTF8 object key limit for AWS S3
- Windows paths are case-insensitive by default
- MacOS paths are case-insensitive by default

1.1.7 Data types

A data type describes the set of possible binary values that an array element may take, along with some information about how the values should be interpreted.

This core specification defines a limited set of data types to represent boolean values, integers, and floating point numbers. Extensions may define additional data types. All of the data types defined here have a fixed size, in the sense that all values require the same number of bytes. However, extensions may define variable sized data types.

Note that the Zarr specification is intended to enable communication of data between a variety of computing environments. The native byte order may differ between machines used to write and read the data.

Each data type is associated with an identifier, which can be used in metadata documents to refer to the data type. For the data types defined in this specification, the identifier is a simple ASCII string. However, extensions may use any JSON value to identify a data type.

Core data types

Table 1: Data types

Identifier	Numerical type	Default binary representation
`bool`	Boolean	Single byte, with false encoded as `\\x00` and true encoded as `\\x01`.
`int8`	Integer in `[-2^7, 2^7-1]`	1 byte two's complement
`int16`	Integer in `[-2^15, 2^15-1]`	2-byte little endian two's complement
`int32`	Integer in `[-2^31, 2^31-1]`	4-byte little endian two's complement
`uint8`	Integer in `[0, 2^8-1]`	1 byte
`uint16`	Integer in `[0, 2^16-1]`	2-byte little endian
`uint32`	Integer in `[0, 2^32-1]`	4-byte little endian
`float16` (optionally supported)	IEEE 754 half-precision floating point: sign bit, 5 bits exponent, 10 bits mantissa	2-byte little endian IEEE 754 binary16
`float32`	IEEE 754 single-precision floating point: sign bit, 8 bits exponent, 23 bits mantissa	4-byte little endian IEEE 754 binary32
`float64`	IEEE 754 double-precision floating point: sign bit, 11 bits exponent, 52 bits mantissa	8-byte little endian IEEE 754 binary64
`complex64`	real and complex components are each IEEE 754 single-precision floating point	2 consecutive 4-byte little endian IEEE 754 binary32 values
`complex128`	real and complex components are each IEEE 754 double-precision floating point	2 consecutive 8-byte little endian IEEE 754 binary64 values
`r*` (Optional)	raw bits, use for extension type fallbacks	variable, given by `*`, is limited to be a multiple of 8.

Additionally to these base types, an implementation should also handle the raw/opaque pass-through type designated by the lower-case letter `r` followed by the number of bits, multiple of 8. For example, `r8`, `r16`, and `r24` should be understood as fall-back types of respectively 1, 2, and 3 byte length.

Zarr v3 is limited to type sizes that are a multiple of 8 bits but may support other type sizes in later versions of this specification.

Note: While the default binary representation is little endian, the *endian codec* may be specified to use big endian encoding instead.

Note: We are explicitly looking for more feedback and prototypes of code using the `r*`, raw bits, for various endianness and whether the spec could be made clearer.

Note: Currently only fixed size elements are supported as a core data type. There are many request for variable length element encoding. There are many ways to encode variable length and we want to keep flexibility. While we seem to agree that for random access the most likely contender is to have two arrays, one with the actual variable length data and one with fixed size (pointer + length) to the variable size data, we do not want to commit to such a structure. See https://github.com/zarr-developers/zarr-specs/issues/62.

1.1.8 Chunk grids

A chunk grid defines a set of chunks which contain the elements of an array. The chunks of a grid form a tessellation of the array space, which is a space defined by the dimensionality and shape of the array. This means that every element of the array is a member of one chunk, and there are no gaps or overlaps between chunks.

In general there are different possible types of grids. The core specification defines the regular grid type, where all chunks are hyperrectangles of the same shape. Extensions may define other grid types, such as rectilinear grids where chunks are still hyperrectangles but do not all share the same shape.

A grid type must also define rules for constructing an identifier for each chunk that is unique within the grid, which is a string of ASCII characters that can be used to construct keys to save and retrieve chunk data in a store, see also the *Storage* section.

Regular grids

A regular grid is a type of grid where an array is divided into chunks such that each chunk is a hyperrectangle of the same shape. The dimensionality of the grid is the same as the dimensionality of the array. Each chunk in the grid can be addressed by a tuple of positive integers (k, j, i, ...) corresponding to the indices of the chunk along each dimension.

The origin element of a chunk has coordinates in the array space (k * dz, j * dy, i * dx, ...) where (dz, dy, dx, ...) are the chunk sizes along each dimension. Thus the origin element of the chunk at grid index (0, 0, 0, ...) is at coordinate (0, 0, 0, ...) in the array space, i.e., the grid is aligned with the origin of the array. If the length of any array dimension is not perfectly divisible by the chunk length along the same dimension, then the grid will overhang the edge of the array space.

The shape of the chunk grid will be (ceil(z / dz), ceil(y / dy), ceil(x / dx), ...) where (z, y, x, ...) is the array shape, "/" is the division operator and "ceil" is the ceiling function. For example, if a 3 dimensional array has shape (10, 200, 3000), and has chunk shape (5, 20, 400), then the shape of the chunk grid will be (2, 10, 8), meaning that there will be 2 chunks along the first dimension, 10 along the second dimension, and 8 along the third dimension.

Table 2: Regular Grid Example

Array Shape	Chunk Shape	Chunk Grid Shape	Notes
(10, 200, 3000)	(5, 20, 400)	(2, 10, 8)	The grid does overhang the edge of the array on the 3rd dimension.

An element of an array with coordinates (c, b, a, ...) will occur within the chunk at grid index (c // dz, b // dy, a // dx, ...), where "//" is the floor division operator. The element will have coordinates (c % dz, b % dy, a % dx, ...) within that chunk, where "%" is the modulo operator. For example, if a 3 dimensional array has shape (10, 200, 3000), and

has chunk shape (5, 20, 400), then the element of the array with coordinates (7, 150, 900) is contained within the chunk at grid index (1, 7, 2) and has coordinates (2, 10, 100) within that chunk.

The identifier for chunk with grid index (`k`, `j`, `i`, ...) is formed by joining together ASCII string representations of each index using a separator and prefixed with the character `c`. The default value for the separator is the slash character, `/`, but this may be configured by providing a `separator` value within the `chunk_grid` metadata object (see the section on *Array metadata* below).

For example, in a 3 dimensional array, the identifier for the chunk at grid index (1, 23, 45) is the string "c1/23/45".

Note that this specification does not consider the case where the chunk grid and the array space are not aligned at the origin vertices of the array and the chunk at grid index (0, 0, 0, ...). However, extensions may define variations on the regular grid type such that the grid indices may include negative integers, and the origin element of the array may occur at an arbitrary position within any chunk, which is required to allow arrays to be extended by an arbitrary length in a "negative" direction along any dimension.

Note: A main difference with spec v2 is that the default chunk separator changed from `.` to `/`. This helps with compatibility with N5 as well as decreases the maximum number of items in hierarchical stores like directory stores.

Note: Arrays may have 0 dimension (when for example representing scalars), in which case the coordinate of a chunk is the empty tuple, and the chunk key will consist of the string `c`.

Note: Chunks at the border of an array always have the full chunk size, even when the array only covers parts of it. For example, having an array with `"shape": [30, 30]` and `"chunk_shape": [16, 16]`, the chunk `0,1` would also contain unused values for the indices `0-16, 30-31`. When writing such chunks it is recommended to use the current fill value for elements outside the bounds of the array.

1.1.9 Chunk memory layouts

An array has a memory layout, which defines the way that the binary values of the array elements are organised within each chunk to form a contiguous sequence of bytes. This contiguous binary representation of a chunk is then the input to the array's chunk encoding pipeline, described in later sections. Typically, when reading data, an implementation will load this binary representation into a contiguous memory buffer to allow direct access to array elements without having to copy data.

The core specification defines two types of contiguous memory layout. However, extensions may define other memory layouts. Note that there may be an interdependency between memory layouts and data types, such that certain memory layouts may only be applicable to arrays with certain data types.

Row-major (C-style) memory layout

In this memory layout, the binary values of the array elements are organised into a sequence such that the last dimension of the array is the fastest changing dimension, also known as "row-major" order. This layout is only applicable to arrays with fixed size data types.

For example, for a two-dimensional array with chunk shape (*dy*, *dx*), the binary values for a given chunk are taken from chunk elements in the order (0, 0), (0, 1), (0, 2), ..., (*dy* - 1, *dx* - 3), (*dy* - 1, *dx* - 2), (*dy* - 1, *dx* - 1).

Column-major (F-style) memory layout

In this memory layout, the binary values of the array elements are organised into a sequence such that the first dimension of the array is the fastest changing dimension, also known as "column-major" order. This layout is only applicable to arrays with fixed size data types.

For example, for a two-dimensional array with chunk shape (*dy*, *dx*), the binary values for a given chunk are taken from chunk elements in the order (0, 0), (1, 0), (2, 0), ..., (*dy* - 3, *dx* - 1), (*dy* - 2, *dx* - 1), (*dy* - 1, *dx* - 1).

1.1.10 Chunk encoding

Chunks are encoded into a binary representation for storage in a *store*, using the chain of *codecs* specified by the `codecs` metadata field.

Determination of encoded representations

To encode or decode a chunk, the encoded and decoded representations for each codec in the chain must first be determined as follows:

1. The initial decoded representation, `decoded_representation[0]` is multi-dimensional array with the same data type as the zarr array, and a shape determined according to the value of `chunk_memory_layout` as follows:
    - If `chunk_memory_layout` is equal to `"C"`, the shape is equal to the chunk shape.
    - If `chunk_memory_layout` is equal to `"F"`, the shape is equal to the chunk shape, with the dimension order reversed.
    - If `chunk_memory_layout` is defined by an extension, the extension defines the shape.
2. For each codec `i`, the encoded representation is equal to the decoded representation `decoded_representation[i+1]` of the next codec, and is computed from `codecs[i].compute_encoded_representation_type(decoded_representation[i])`. If `compute_encoded_representation_type` fails because of an incompatible decoded representation, an implementation should indicate an error.

Conversion between multi-dimensional array and byte string representations

Some codecs operate directly on multi-dimensional arrays of elements, e.g. encoding a 3-d array as a multi-channel jpeg image. Other codecs operate at the byte level, e.g. gzip compression. If a codec that operates at the byte level receives as input an array that is not a 1-dimensional uint8 array, it may convert the input array to a byte string by concatenating the default binary representations of each element in lexicographical order (C order). Similarly, if a codec that expects a multi-dimensional array as input instead receives a byte string, it may decode each element in lexicographical order according to the default binary representation of each element.

Encoding procedure

Based on the computed `decoded_representations` list, a chunk is encoded using the following procedure:

1. The chunk array `A` (with a shape equal to the chunk shape, and data type equal to the zarr array data type) is logically transformed into the initial *encoded chunk* `EC[0]` of the type specified by `decoded_representation[0]` according to the `chunk_memory_layout` as follows:
    - If `chunk_memory_layout` is equal to `"C"`, `EC[0]` equals `A` (no transformation).
    - If `chunk_memory_layout` is equal to `"F"`, the dimension order is reversed.

- If `chunk_memory_layout` is defined by an extension, the extension defines the transformation to perform.

2. For each codec `codecs[i]` in `codecs`, `EC[i+1] := codecs[i].encode(EC[i])`.
3. The final encoded chunk representation `EC_final` is always a byte string. If `EC[codecs.length]` is a byte string, then `EC_final := EC[codecs.length]`. Otherwise, `EC_final` is *converted* from `EC[codecs.length]`.
4. `EC_final` is written to the *store*.

Decoding procedure

Based on the computed `decoded_representations` list, a chunk is encoded using the following procedure:

1. The encoded chunk representation `EC_final` is read from the *store*.
2. If `codecs[codecs.length]` is a byte string, `EC[codecs.length] := EC_final`. Otherwise, `EC[codecs.length]` is *converted* from `EC_final`.
3. For each codec `codecs[i]` in `codecs`, iterating in reverse order, `EC[i] := codecs[i].decode(EC[i+1], decoded_representation[i])`.
4. The chunk array `A` is computed from `EC[0]` according to the `chunk_memory_layout` as follows:
   - If `chunk_memory_layout` is equal to `"C"`, `A` equals `EC[0]` (no transformation).
   - If `chunk_memory_layout` is equal to `"F"`, the dimension order is reversed.
   - If `chunk_memory_layout` is defined by an extension, the extension defines the transformation to perform.

Specifying codecs

To allow for flexibility to define and implement new codecs, this specification does not define any codecs, nor restrict the set of codecs that may be used. Each codec must be defined via a separate specification. In order to refer to codecs in array metadata documents, each codec must have a unique identifier, which is a URI that dereferences to a human-readable specification of the codec. A codec specification must declare the codec identifier, and describe (or cite documents that describe) the encoding and decoding algorithms and the format of the encoded data.

A codec may have configuration parameters which modify the behaviour of the codec in some way. For example, a compression codec may have a compression level parameter, which is an integer that affects the resulting compression ratio of the data. Configuration parameters must be declared in the codec specification, including a definition of how configuration parameters are represented as JSON.

The Zarr core development team maintains a repository of codec specifications, which are hosted alongside this specification in the zarr-specs GitHub repository, and which are published on the zarr-specs documentation Web site. For ease of discovery, it is recommended that codec specifications are contributed to the zarr-specs GitHub repository. However, codec specifications may be maintained by any group or organisation and published in any location on the Web. For further details of the process for contributing a codec specification to the zarr-specs GitHub repository, see the Zarr community process specification.

Further details of how codecs are configured for an array are given in the section below on *Array metadata*.

1.1.11 Metadata

This section defines the structure of metadata documents for Zarr hierarchies, which consists of three types of metadata documents: an entry point metadata document (`zarr.json`), array metadata documents, and group metadata documents. Each type of metadata document is described in the following subsections.

Metadata documents are defined here using the JSON type system defined in [RFC8259]. In this section, the terms "value", "number", "string" and "object" are used to denote the types as defined in [RFC8259]. The term "array" is also used as defined in [RFC8259], except where qualified as "Zarr array". Following [RFC8259], this section also describes an object as a set of name/value pairs. This section also defines how metadata documents are encoded for storage.

Only the top level metadata document `zarr.json` is guaranteed to be of JSON type, and can be used to define other formats for array-level and group-level metadata documents. In the case where non-JSON metadata documents are used in a Zarr hierarchy, the following sections on group and array level metadata are non-normative, but other metadata formats are expected to define some equivalence relations with the JSON documents.

Entry point metadata

Each Zarr hierarchy must have an entry point metadata document, which provides essential information regarding the format version being used, the encoding being used for group and array metadata, and any extensions that affect the layout or interpretation of data in the store.

The entry point metadata document must contain a single object containing the following names:

`zarr_format`

A string containing the URI of the Zarr core specification that defines the metadata format. For Zarr hierarchies conforming to this specification, the value must be the string "https://purl.org/zarr/spec/core/3.0".

Implementations of this specification may assume that the final path segment of this URI ("3.0") represents the core specification version number, where "3" is the major version number and "0" is the minor version number. Implementations of this specification may also assume that future versions of this specification that retain the same major versioning number ("3") will be backwards-compatible, in the sense that any new features added to the specification can be safely ignored. In other words, if the major version number is "3", implementations of this specification may read and interpret metadata as defined in this specification, ignoring any name/value pairs where the name is not defined here. See also the *stability policy*.

Note that this value is given as a URI rather than as a simple version number string to help with discovery of this specification.

`metadata_encoding`

Specifies the encoding of group and array metadata. To use JSON encoding, which is the only encoding allowed by this core specification, the `metadata_encoding` value must be the following object:

```
{
    "type": "json",
    "metadata_key_suffix": ".json"
}
```

The `metadata_encoding` value is an extension point and may be defined by an extension. In this case the value must be an object containing the required names `extension`, `type` and `metadata_key_suffix`.

`extension` must be a URI that identifies the extension and dereferences to a human-readable representation of the specification. `type` is a string defined by the extension. The `metadata_key_suffix` is a string containing a suffix to add to the array and group metadata keys when saving into the store.

Note: The metadata key suffix is used to allow non hierarchy browsing and editing by non-zarr-aware tools.

`extensions`

An array containing zero or more objects, each of which identifies an extension and provides any additional extension configuration metadata. Each object must contain the name `extension` whose value is a URI that identifies a Zarr extension and dereferences to a human readable representation of the extension specification. Each object must also contain the name `must_understand` whose value is either the literal `true` or `false`. Each object may also contain the name `configuration` whose value is defined by the extension.

If an implementation of this specification encounters an extension that it does not recognize, but the value of `must_understand` is `false`, then the extension may be ignored and processing may continue. If the extension is not recognized and the value of `must_understand` is `true` then processing must terminate and an appropriate error raised.

For example, below is an entry point metadata document, specifying that JSON is being used for encoding of group and array metadata:

```
{
    "zarr_format": "https://purl.org/zarr/spec/core/3.0",
    "metadata_encoding": {
        "type": "json",
        "metadata_key_suffix": ".json"
    },
    "extensions": []
}
```

For example, below is an entry point metadata document as above, but also specifying that an extension is being used which may be ignored if not understood:

```
{
    "zarr_format": "https://purl.org/zarr/spec/core/3.0",
    "metadata_encoding": {
        "type": "json",
        "metadata_key_suffix": ".json"
    },
    "extensions": [
        {
            "extension": "http://example.org/zarr/extension/foo",
            "must_understand": false,
            "configuration": {
                "foo": "bar"
            }
        }
    ]
}
```

Array metadata

Each Zarr array in a hierarchy must have an array metadata document. This document must contain a single object with the following mandatory names:

shape

An array of integers providing the length of each dimension of the Zarr array. For example, a value `[10, 20]` indicates a two-dimensional Zarr array, where the first dimension has length 10 and the second dimension has length 20.

data_type

The data type of the Zarr array. If the data type is defined in this specification, then the value must be the data type identifier provided as a string. For example, `"<f8"` for little-endian 64-bit floating point number.

The `data_type` value is an extension point and may be defined by an extension. If the data type is defined by an extension, then the value must be an object containing the names `extension`, `type` and (optionally) `fallback`. The `extension` is required and its value must be a URI that identifies the extension and dereferences to a human-readable representation of the specification. The `type` is required and its value is defined by the extension. The `fallback` is optional and, if provided, its value must be one or a list of the data type identifiers defined in this specification or an extension. If an implementation does not recognise the extension or specific data type, but a `fallback` is present, then the implementation may proceed using the first known `fallback` value as the data type. For fixed-sized data types, if there is no more specific fallback available, a raw number of bytes using the raw type (`r*`) should be given.

The default list of fallbacks to put into the metadata should by defined by the data type extension, but it may be overridden by the user. *Note for implementations*: Silently using a fallback without explicit approval might cause problems for users, please consider options to (de-)activate fallback behavior and/or appropriate warnings.

chunk_grid

The chunk grid of the Zarr array. If the chunk grid is a regular chunk grid as defined in this specification, then the value must be an object with the names `type`, `chunk_shape` and `separator`. The value of `type` must be the string `"regular"`, and the value of `chunk_shape` must be an array of integers providing the lengths of the chunk along each dimension of the array. `separator` must be either `"/"` or `"."`. For example, `{"type": "regular", "chunk_shape": [2, 5], "separator":"/"}` means a regular grid where the chunks have length 2 along the first dimension and length 5 along the second dimension.

The `chunk_grid` value is an extension point and may be defined by an extension. If the chunk grid type is defined by an extension, then the value must be an object containing the names `extension` and `type`. The `extension` is required and the value must be a URI that identifies the extension and dereferences to a human-readable representation of the specification. The `type` is required and the value is defined by the extension.

chunk_memory_layout

The internal memory layout of the chunks. Use the value "C" to indicate **`C contiguous memory layout`_** or "F" to indicate **`F contiguous memory layout`_** as defined in this specification.

The `chunk_memory_layout` value is an extension point and may be defined by an extension. If the chunk memory layout type is defined by an extension, then the value must be an object containing the names `extension` and `type`. The `extension` is required and the value must be a URI that identifies the extension and dereferences to a human-readable representation of the specification. The `type` is required and the value is defined by the extension.

fill_value

Provides an element value to use for uninitialised portions of the Zarr array.

If the data type of the Zarr array is Boolean then the value must be the literal `false` or `true`. If the data type is one of the integer data types defined in this specification, then the value must be a number with no fraction or exponent part and must be within the range of the data type.

For any data type, the `fill_value` is required. The literal `null` is not permitted. The fill value needs to be defined so that the data is independent of implementation details. Internally implementations may provide a default `fill_value`, but that must be converted to a fixed value in the stored metadata.

If the `data_type` of an array is defined in a `data_type` extension, then said extension is responsible for interpreting the value of `fill_value` and return a suitable type that can be used.

For core data types for which fill values are not permitted in JSON or for which decimal representation could be lossy, a string representing of the binary (starting with `0b`) or hexadecimal value (starting with `0x`) is accepted. This string must include all leading or trailing zeroes necessary to match the given type size. The string values `"NaN"`, `"+Infinity"` and `"-Infinity"` are also understood for floating point data types.

extensions

See the top level metadata extension section for the time being.

attributes

The value must be an object. The object may contain any key/value pairs, where the key must be a string and the value can be an arbitrary JSON literal. Intended to allow storage of arbitrary user metadata

---

Note: The question of whether core metadata and user attributes should be stored together or in separate documents is a topic of ongoing discussion. (See https://github.com/zarr-developers/zarr-specs/issues/72.)

The following members are optional:

codecs

Specifies a list of codecs to be used for encoding and decoding chunks. The value must be an array of objects, each object containing a member with `type` whose value is a URI that identifies a codec and dereferences to a human-readable representation of the codec specification. The codec object may also contain a `configuration` object which consists of the parameter names and values as defined by the corresponding codec specification. An absent `codecs` member is equivalent to specifying an empty list of codecs.

storage_transformers

Specifies a stack of *storage transformers*. Each value in the list must be an object containing the names `extension` and `type`. The `extension` is required and the value must be a URI that identifies the extension and dereferences to a human-readable representation of the specification. The `type` is required and the value is defined by the extension. The object may also contain a `configuration` object which consists of the parameter names and values as defined by the corresponding storage transformer specification. When the `storage_transformers` name is absent no storage transformer is used, same for an empty list.

The array metadata object must not contain any other names. Those are reserved for future versions of this specification. An implementation must fail to open zarr hierarchies, groups or arrays with unknown metadata fields.

For example, the array metadata JSON document below defines a two-dimensional array of 64-bit little-endian floating point numbers, with 10000 rows and 1000 columns, divided into a regular chunk grid where each chunk has 1000 rows and 100 columns, and thus there will be 100 chunks in total arranged into a 10 by 10 grid. Within each chunk the binary values are laid out in C contiguous order. Each chunk is compressed using gzip compression prior to storage:

```
{
    "shape": [10000, 1000],
    "data_type": "<f8",
    "chunk_grid": {
        "type": "regular",
        "chunk_shape": [1000, 100],
        "separator" : "/"
    },
    "chunk_memory_layout": "C",
    "codecs": [{
        "type": "https://purl.org/zarr/spec/codecs/gzip/1.0",
        "configuration": {
            "level": 1
        }
    }],
    "fill_value": "NaN",
    "extensions": [],
    "attributes": {
        "foo": 42,
        "bar": "apples",
        "baz": [1, 2, 3, 4]
    }
}
```

The following example illustrates an array with the same shape and chunking as above, but using an extension data type:

```
{
    "shape": [10000, 1000],
    "data_type": {
        "extension": "https://purl.org/zarr/spec/extensions/data-types/datetime/1.0",
        "type": "<M8[ns]",
        "fallback": "<i8"
    },
    "chunk_grid": {
        "type": "regular",
        "chunk_shape": [1000, 100],
        "separator" : "/"
    },
    "chunk_memory_layout": "C",
    "codecs": [{
        "type": "https://purl.org/zarr/spec/codecs/gzip/1.0",
        "configuration": {
            "level": 1
        }
    }],
    "fill_value": null,
    "extensions": [],
    "attributes": {}
}
```

Note: comparison with spec v2, `dtype` has been renamed to `data_type`, `chunks` has been renamed to `chunk_grid`, `order` has been renamed to `chunk_memory_layout`, the separate `filters` and `compressor` fields been combined into the single `codecs` field, `zarr_format` has been removed,

Group metadata

A Zarr group metadata object must contain the `attributes` name as defined above in the *Array metadata* section. All other names are reserved for future versions of this specification. See also the section on *extensions* below.

For example, the JSON document below defines an explicit group:

```
{
    "attributes": {
        "spam": "ham",
        "eggs": 42,
    }
}
```

Note: Groups cannot have extensions attached to them as of spec v3.0. Allowing groups to have extensions would force any implementation to sequentially traverse the store hierarchy in order to check for extensions, which would defeat the purpose of a flat namespace and concurrent access.

For the time being groups can only have attributes.

Note: A group does not need a metadata document to exist. (See implicit groups.)

Metadata encoding

The entry point metadata document must be encoded as JSON. The array (`*.array` s) and group metadata documents (`*.group` s) must be encoded as per the type defined in the `metadata_encoding` field in the entry point metadata document (described below).

1.1.12 Stores

A Zarr store is a system that can be used to store and retrieve data from a Zarr hierarchy. For a store to be compatible with this specification, it must support a set of operations defined in the *Abstract store interface* subsection. The store interface can be implemented using a variety of underlying storage technologies, described in the subsection on *Store implementations*.

Abstract store interface

The store interface is intended to be simple to implement using a variety of different underlying storage technologies. It is defined in a general way here, but it should be straightforward to translate into a software interface in any given programming language. The goal is that an implementation of this specification could be modular and allow for different store implementations to be used.

The store interface defines a set of operations involving *keys* and *values*. In the context of this interface, a *key* is any string containing only characters in the ranges `a-z`, `A-Z`, `0-9`, or in the set `/.-_`, where the final character is **not** a `/` character. A *value* is a sequence of bytes.

It is assumed that the store holds (*key*, *value*) pairs, with only one such pair for any given *key*. I.e., a store is a mapping from keys to values. It is also assumed that keys are case sensitive, i.e., the keys "foo" and "FOO" are different.

To read and write partial values, a *range* specifies two integers *range_start* and *range_length*, that specify a part of the value starting at byte *range_start* (inclusive) and having a length of *range_length* bytes. *range_length* may be none, indicating all available data until the end of the referenced value. For example *range* `[0, none]` specifies the full value. Stores that do not support partial access can still fulfill partial requests by first extracting the full value and then returning a subset of bytes.

The store interface also defines some operations involving *prefixes*. In the context of this interface, a prefix is a string containing only characters that are valid for use in *keys* and ending with a trailing `/` character.

The store operations are grouped into three sets of capabilities: **readable**, **writeable** and **listable**. It is not necessary for a store implementation to support all of these capabilities.

A **readable store** supports the following operations:

`get` - Retrieve the *value* associated with a given *key*.

> Parameters: *key*
> Output: *value*

`get_partial_values` - Retrieve possibly partial *values* from given *key_ranges*.

> Parameters: *key_ranges*: ordered set of *key*, *range* pairs,
> a *key* may occur multiple times with different *ranges*
> Output: list of *values*, in the order of the *key_ranges*, may contain none
> for missing keys

A **writeable store** supports the following operations:

`set` - Store a (*key*, *value*) pair.

Parameters: *key*, *value*
Output: none

`set_partial_values` - Store *values* at a given *key*, starting at byte *range_start*.

Parameters: *key_start_values*: set of *key*,
range_start, *value* triples, a *key* may occur multiple
times with different *range_starts*, *range_starts* with
length of the respective *value* must not specify overlapping
ranges for the same *key*
Output: none

`erase` - Erase the given key/value pair from the store.

Parameters: *key*
Output: none

`erase_values` - Erase the given key/value pairs from the store.

Parameters: *keys*: set of *keys*
Output: none

`erase_prefix` - Erase all keys with the given prefix from the store:

Parameter: *prefix*
Output: none

Note: Some KV stores do allow creation and update of keys, but not deletion. For example, Zip archives do not allow removal of content without recreating the full archive.

Inability to delete can affect ability to rename keys as well, as a rename is often a sequence or atomic combination of a deletion and a creation.

A **listable store** supports any one or more of the following operations:

`list` - Retrieve all *keys* in the store.

Parameters: none
Output: set of *keys*

`list_prefix` - Retrieve all keys with a given prefix.

Parameters: *prefix*
Output: set of *keys* with the given *prefix*,

For example, if a store contains the keys "a/b", "a/c/d" and "e/f/g", then `list_prefix("a/")` would return "a/b" and "a/c/d".

Note: the behavior of `list_prefix` is undefined if `prefix` does not end with a trailing slash `/` and the store can assume there is at least one key that starts with `prefix`.

`list_dir` - Retrieve all keys and prefixes with a given prefix and which do not contain the character "/" after the given prefix.

Parameters: *prefix*

Output: set of *keys* and set of *prefixes*

For example, if a store contains the keys "a/b", "a/c", "a/d/e", "a/f/g", then `list_dir("a/")` would return keys "a/b" and "a/c" and prefixes "a/d/" and "a/f/". `list_dir("b/")` would return the empty set.

Note that because keys are case sensitive, it is assumed that the operations `set("foo", a)` and `set("FOO", b)` will result in two separate (key, value) pairs being stored. Subsequently `get("foo")` will return *a* and `get("FOO")` will return *b*.

It is recommended that the implementation of the `get_partial_values`, `set_partial_values` and `erase_values` methods is made optional, providing fallbacks for them by default. However, it is recommended to supply those operations where possible for efficiency. Also, the `get`, `set` and `erase` can easily be mapped onto their *partial_values* counterparts. Therefore, it is also recommended to supply fallbacks for those if the *partial_values* operations can be implemented. An entity containing those fallbacks could be named `StoreWithPartialAccess`.

Store implementations

(This subsection is not normative.)

A store implementation maps the abstract operations of the store interface onto concrete operations on some underlying storage system. This specification does not constrain or make any assumptions about the nature of the underlying storage system. Thus it is possible to implement the store interface in a variety of different ways.

For example, a store implementation might use a conventional file system as the underlying storage system, mapping keys onto file paths and values onto file contents. The `get` operation could then be implemented by reading a file, the `set` operation implemented by writing a file, and the `list_dir` operation implemented by listing a directory.

For example, a store implementation might use a key-value database such as BerkeleyDB or LMDB as the underlying storage system. In this case the implementation of `get` and `set` operations would be whatever native operations are provided by the database for getting and setting key/value pairs. Such a store implementation might natively support the `list` operation but might not support `list_prefix` or `list_dir`, although these could be implemented via `list` with post-processing of the returned keys.

For example, a store implementation might use a cloud object storage service such as Amazon S3, Azure Blob Storage, or Google Cloud Storage as the underlying storage system, mapping keys to object names and values to object contents. The store interface operations would then be implemented via concrete operations of the service's REST API, i.e., via HTTP requests. E.g., the `get` operation could be implemented via an HTTP GET request to an object URL, the `set` operation could be implemented via an HTTP PUT request to an object URL, and the list operations could be implemented via an HTTP GET request to a bucket URL (i.e., listing a bucket).

The examples above are meant to be illustrative only, and other implementations are possible. This specification does not attempt to standardise any store implementations, however where a store implementation is expected to be widely used then it is recommended to create a store implementation spec and contribute it to the zarr-specs GitHub repository. For an example of a store implementation spec, see the *File system store (version 1.0)* specification.

1.1.13 Storage

This section describes how to translate high level operations to create, erase or modify Zarr hierarchies, groups or arrays, into low level operations on the key/value store interface defined above.

In this section a "hierarchy path" is a logical path which identifies a group or array node within a Zarr hierarchy, and a "storage key" is a key used to store and retrieve data via the store interface. There is a further distinction between "metadata keys" which are storage keys used to store metadata documents, and "chunk keys" which are storage keys used to store encoded chunks.

Note that any non-root hierarchy path will have ancestor paths that identify ancestor nodes in the hierarchy. For example, the path "/foo/bar" has ancestor paths "/foo" and "/".

Storage keys

The entry point metadata document is stored under the key `zarr.json`.

For a group at a non-root hierarchy path *P*, the metadata key for the group metadata document is formed by concatenating "meta", *P*, ".group", and the metadata key suffix (which defaults to ".json").

For example, for a group at hierarchy path `/foo/bar`, the corresponding metadata key is "meta/foo/bar.group.json".

For an array at a non-root hierarchy path *P*, the metadata key for the array metadata document is formed by concatenating "meta", *P*, ".array", and the metadata key suffix.

The data key for array chunks is formed by concatenating "data", *P*, "/", and the chunk identifier as defined by the chunk grid layout.

To get the path `P` from a non-root metadata key, remove the trailing ".array.json" or ".group.json" and the "meta" prefix.

For example, for an array at hierarchy path "/foo/baz", the corresponding metadata key is "meta/foo/baz.array.json". If the array has two dimensions and a regular chunk grid, the data key for the chunk with grid coordinates (0, 0) is "data/foo/baz/c0/0".

If the root node is a group, the metadata key is "meta/group.json". If the root node is an array, the metadata key is "meta/array.json", and the data keys are formed by concatenating "data/" and the chunk identifier.

Table 3: Metadata Storage Key example

Type	Path "P"	Key for Metadata at path *P*
Entry-Point metadata (zarr.json)	*n/a*	*zarr.json*
Array (Root)	*/*	*meta/array.json*
Group (Root)	*/*	*meta/group.json*
Group	*/foo*	*meta/foo.group.json*
Array	*/foo*	*meta/foo.array.json*
Group	*/foo/bar*	*meta/foo/bar.group.json*
Array	*/foo/baz*	*meta/foo/baz.array.json*

Table 4: Data Storage Key example

Path *P* of array	Chunk grid indices	Data key
/foo/baz	*(1, 0)*	*data/foo/baz/c1/0*

Operations

Todo: The following section descripes possible operations of an implementation as a guide-line. Those descriptions are not yet finalized.

Let *P* be an arbitrary hierarchy path.

Let `array_meta_key(P)` be the array metadata key for *P*. Let `group_meta_key(P)` be the group metadata key for *P*.

Let `data_key(P, j, i ...)` be the data key for *P* for the chunk with grid coordinates (*j*, *i*, ...).

Let "+" be the string concatenation operator.

Note: Store and implementation can assume that a client will not try to create both an *array* and *group* at the same path, and thus may skip check of existence of a group/array of the same name.

Create a group

To create an explicit group at hierarchy path *P*, perform `set(group_meta_key(P), value)`, where *value* is the serialization of a valid group metadata document.

If *P* is a non-root path then it is **not** necessary to create or check for the existence of metadata documents for groups at any of the ancestor paths of *P*. Creating a group at path *P* implies the existence of groups at all ancestor paths of *P*.

Create an array

To create an array at hierarchy path *P*, perform `set(array_meta_key(P), value)`, where *value* is the serialisation of a valid array metadata document.

If *P* is a non-root path then it is **not** necessary to create or check for the existence of metadata documents for groups at any of the ancestor paths of *P*. Creating an array at path *P* implies the existence of groups at all ancestor paths of *P*.

Store chunk data in an array

To store chunk data in an array at path *P* and chunk coordinate (j, i, ...), perform `set(data_key(P, j, i, ...), value)`, where *value* is the serialisation of the corresponding chunk, encoded according to the information in the array metadata stored under the key `array_meta_key(P)`.

Retrieve chunk data in an array

To retrieve chunk data in an array at path *P* and chunk coordinate (i, j, ...), perform `get(data_key(P, j, i, ...), value)`. The returned value is the serialisation of the corresponding chunk, encoded according to the array metadata stored at `array_meta_key(P)`.

Discover children of a group

To discover the children of a group at hierarchy path *P*, perform `list_dir("meta" + P + "/")`. Any returned key ending in ".array.json" indicates an array. Any returned key ending in ".group.json" indicates a group. Any returned prefix indicates a child group implied by some descendant.

For example, if a group is created at path "/foo/bar" and an array is created at path "/foo/baz/qux", then the store will contain the keys "meta/foo/bar.group.json" and "meta/foo/bar/baz/qux.array.json". Groups at paths "/", "/foo" and "/foo/baz" have not been explicitly created but are implied by their descendants. To list the children of the group at path "/foo", perform `list_dir("meta/foo/")`, which will return the key "meta/foo/bar.group.json" and the prefix "meta/foo/baz/". From this it can be inferred that child groups "/foo/bar" and "/foo/baz" are present.

If a store does not support any of the list operations then discovery of group children is not possible, and the contents of the hierarchy must be communicated by some other means, such as via an extension, or via some out of band communication.

Discover all nodes in a hierarchy

To discover all nodes in a hierarchy, one can call `list_prefix("meta/")`. All keys represent either explicit group or arrays. All intermediate prefixes ending in a `/` are implicit groups.

Erase a group or array

To erase an array at path *P*:

- erase the metadata document for the array, `erase(array_meta_key(P))`

- erase all data keys which prefix have path pointing to this array, `erase_prefix("data" + P + "/")`

To erase an implicit group at path *P*:

- erase all nodes under this group - it should be sufficient to perform `erase_prefix("meta" + P + "/")` and `erase_prefix("data" + P + "/")`.

To erase an explicit group at path *P*:

- erase the metadata document for the group, `erase(group_meta_key(P))`
- erase all nodes under this group - it should be sufficient to perform `erase_prefix("meta" + P + "/")` and `erase_prefix("data" + P + "/")`.

Determine if a node exists

To determine if a node exists at path `P`, try in the following order `get(array_meta_key(P))` (success implies an array at `P`); `get(group_meta_key(P))` (success implies an explicit group at `P`); `list_dir("meta" + P + "/")` (non-empty result set implies an implicit group at `P`).

Note: For listable store, `list_dir(parent(P))` can be an alternative.

1.1.14 Storage transformers

A Zarr storage transformer allows to change the zarr-compatible data before storing it. The stored transformed data is restored to its original state whenever data is requested by the Array. Storage transformers can be configured per array via the *storage_transformers* name in the *array metadata*. Storage transformers which do not change the storage layout (e.g. for caching) may be specified at runtime without adding them to the array metadata.

A storage transformer serves the same *abstract store interface* as the *store*. However, it should not persistently store any information necessary to restore the original data, but instead propagates this to the next storage transformer or the final store. From the perspective of an array or a previous stage transformer both store and storage transformer follow the same protocol and can be interchanged regarding the protocol. The behaviour can still be different, e.g. requests may be cached or the form of the underlying data can change.

Storage transformers may be stacked to combine different functionalities:

A fixed set of storage providers is recommended for implementation with this specification:

Predefined storage transformers

- *Sharding storage transformer (version 1.0)* (pending, part of ZEP 2)

1.1.15 Extensions

Many types of extensions can exist and they can be grouped as following:

extension	metadata	is extension required
generic	`extensions` in *entry point metadata*	`must_understand`
metadata encoding	`metadata_encoding` in *entry point metadata*	always
array	`extensions` in *Array metadata*	`must_understand`
data type	*data_type*	no `fallback`
chunk grid	*chunk_grid*	always
chunk memory layout	*chunk_memory_layout*	always
storage transformer	*storage_transformers*	always

There are no group extensions in Zarr v3.0.

See https://github.com/zarr-developers/zarr-specs/issues/49 for a list of potential extensions

1.1.16 Implementation Notes

This section is non-normative and presents notes from implementers about cases that need to be carefully considered but do not strictly fall into the spec.

Explicit vs implicit group

While this zarr spec v3 defines implicit and explicit groups, implementations may decide to create an explicit group for all implicit groups they encounter; in particular when using a hierarchical storage.

Erasure of an implicit group may automatically erase any empty parent. For example on a S3 store where the namespace is flat, erasure of the last key with a prefix will erase all the implicit group in the prefix.

Care must thus be taken when erasing an array or a group if the parent needs to be converted into an explicit group.

1.1.17 Comparison with Zarr v2

This section is informative.

Below is a summary of the key differences between this specification (v3) and Zarr v2.

- In v3 each hierarchy has an explicit root, and must be opened at the root. In v2 there was no explicit root and a hierarchy could be opened at its original root or at any sub-group.
- In v3 the storage keys have been redesigned to separate the space of keys used for metadata and data, by using different prefixes. This is intended to allow for more performant listing and querying of metadata documents on high latency stores. There are also differences including a change to the default separator used to construct chunk keys, and the addition of a key suffix for metadata keys.
- v3 has explicit support for extensions via defined extension points and mechanisms.
- v3 allows for greater flexibility in how groups and arrays are created. In particular, v3 supports implicit groups, which are groups that do not have a metadata document but whose existence is implied by descendant nodes. This change enables multiple arrays to be created in parallel without generating race conditions for the metadata when creating parent groups.
- The set of data types specified in v3 is less than in v2. Additional data types will be defined via extensions.

1.1.18 References

1.1.19 Change log

All notable and possibly implementation-affecting changes to this specification are documented in this section, grouped by the specification status and ordered by time.

Draft Changes

- Removed the 255 character limit for paths. PR #175
- Removed the `/root` prefix for paths. PR #175
  - `meta/root.array.json` is now `meta/array.json`
  - `meta/root/foo/bar.group.json` is now `meta/foo/bar.group.json`
- Moved the `metadata_key_suffix` entrypoint metadata key into `metadata_encoding`, which now just specifies *"json"* via the *type* key and is an extension point. PR #171
- Changed data type names and changed endianness to be handled by a codec. PR #155
- Replaced the `compressor` field in the array metadata with a `codecs` field that can specify a list of codecs. PR #153
- Required `fill_value` in the array metadata to be defined. PR #145
- Added array storage transformers which can be configured per array via the storage_transformers name in the array metadata. PR #134
- The changelog is incomplete before 2022, please refer to the commits on GitHub.

@@tag@@

Links: view spec; view source

@@TODO summary of changes since previous tag.

CHAPTER

TWO

EXTENSIONS

Under construction.

2.1 Array Extensions

Under construction.

2.2 Data Types

Under construction.

2.2.1 Complex number data types (version 1.0)

Editor's draft 11 June 2019

Specification URI:
https://purl.org/zarr/spec/extensions/data-types/complex/1.0

Issue tracking:
GitHub issues

Suggest an edit for this spec:
GitHub editor

Abstract

This specification is a Zarr extension defining data types for complex numbers.

Status of this document

Warning: This document is a **Work in Progress**. It may be updated, replaced or obsoleted by other documents at any time. It is inappapropriate to cite this document as other than work in progress.

Comments, questions or contributions to this document are very welcome. Comments and questions should be raised via GitHub issues. When raising an issue, please add the label "complex-dtypes-v1.0".

This document was produced by the Zarr core development team.

Document conventions

Conformance requirements are expressed with a combination of descriptive assertions and [RFC2119] terminology. The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in the normative parts of this document are to be interpreted as described in [RFC2119]. However, for readability, these words do not appear in all uppercase letters in this specification.

All of the text of this specification is normative except sections explicitly marked as non-normative, examples, and notes. Examples in this specification are introduced with the words "for example".

Extension data types

@@TODO define complex data types

Data type identifiers

@@TODO define identifiers for complex data types

References

Change log

@@TODO

2.2.2 Datetime data types (version 1.0)

Editor's draft 11 June 2019

Specification URI:
https://purl.org/zarr/spec/extensions/data-types/datetime/1.0

Issue tracking:
GitHub issues

Suggest an edit for this spec:
GitHub editor

Abstract

This specification is a Zarr protocol extension defining data types for datetimes (moment in time) and timedeltas (difference between two datetimes).

Status of this document

Warning: This document is a **Work in Progress**. It may be updated, replaced or obsoleted by other documents at any time. It is inappapropriate to cite this document as other than work in progress.

Comments, questions or contributions to this document are very welcome. Comments and questions should be raised via GitHub issues. When raising an issue, please add the label "datetime-dtypes-v1.0".

This document was produced by the Zarr core development team.

Document conventions

Conformance requirements are expressed with a combination of descriptive assertions and [RFC2119] terminology. The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in the normative parts of this document are to be interpreted as described in [RFC2119]. However, for readability, these words do not appear in all uppercase letters in this specification.

All of the text of this specification is normative except sections explicitly marked as non-normative, examples, and notes. Examples in this specification are introduced with the words "for example".

Extension data types

Two extension data types are defined to represent datetime and timedelta values. The definitions of these data types are based on the datetime64 and timedelta64 data types as described in [NumPy], but are intended to be portable across Zarr implementations in different programming languages.

Datetime

The datetime data type represents a single moment in time. It is a logical data type based on the 64-bit integer data type, with associated *Units*. Datetimes are always stored based on POSIX time, with an epoch of 1970-01-01T00:00Z. This means the supported dates are always a symmetric interval around the epoch, called "time span" in the *Units* table below.

The length of the span is the range of a 64-bit integer times the length of the date or unit. For example, the time span for 'W' (week) is exactly 7 times longer than the time span for 'D' (day), and the time span for 'D' (day) is exactly 24 times longer than the time span for 'h' (hour).

Timedelta

The timedelta data type represents a difference between two datatimes. It is a logical data type based on the 64-bit integer data type, with associated *Units*. Note that two timedelta units ('Y', years and 'M', months) represent different amounts of time depending on when they are used. E.g., While a timedelta day unit is equivalent to 24 hours, there is no way to convert a month unit into days, because different months have different numbers of days.

Units

Code	Meaning	Time span (relative)	Time span (absolute)
Y	year	+/- 9.2e18 years	[9.2e18 BCE, 9.2e18 CE]
M	month	+/- 7.6e17 years	[7.6e17 BCE, 7.6e17 CE]
W	week	+/- 1.7e17 years	[1.7e17 BCE, 1.7e17 CE]
D	day	+/- 2.5e16 years	[2.5e16 BCE, 2.5e16 CE]
h	hour	+/- 1.0e15 years	[1.0e15 BCE, 1.0e15 CE]
m	minute	+/- 1.7e13 years	[1.7e13 BCE, 1.7e13 CE]
s	second	+/- 2.9e11 years	[2.9e11 BCE, 2.9e11 CE]
ms	millisecond	+/- 2.9e8 years	[2.9e8 BCE, 2.9e8 CE]
us	microsecond	+/- 2.9e5 years	[290301 BCE, 294241 CE]
ns	nanosecond	+/- 292 years	[1678 CE, 2262 CE]
ps	picosecond	+/- 106 days	[1969 CE, 1970 CE]
fs	femtosecond	+/- 2.6 hours	[1969 CE, 1970 CE]
as	attosecond	+/- 9.2 seconds	[1969 CE, 1970 CE]

Data type identifiers

An identifier for a datetime data type is constructed from the string pattern "[endianness]M8[[units]]" where *endianness* is either "<" (little-endian) or ">" (big-endian) and *units* is one of the unit codes defined above. The endianness and units must be given. E.g., "<M8[ns]" identifies a little-endian datatime data type with nanosecond units.

An identifier for a timedelta data type is constructed from the string pattern "[endianness]m8[[units]]" where *endianness* is either "<" (little-endian) or ">" (big-endian) and *units* is one of the unit codes defined above. The endianness and units must be given. E.g., "<m8[ns]" identifies a little-endian timedelta data type with nanosecond units.

References

Change log

@@TODO

2.2.3 String data types (version 1.0)

Editor's draft 2 March 2022

Specification URI:
https://purl.org/zarr/spec/extensions/data-types/string/1.0

Issue tracking:
GitHub issues

Suggest an edit for this spec:
GitHub editor

Abstract

This specification is a Zarr extension defining data types for strings. It is an early draft and currently just describes existing support for NumPy string types that have already worked with zarr-python, but are not part of the core v3 spec.

Status of this document

Warning: This document is a **Work in Progress**. It may be updated, replaced or obsoleted by other documents at any time. It is inappapropriate to cite this document as other than work in progress.

Comments, questions or contributions to this document are very welcome. Comments and questions should be raised via GitHub issues. When raising an issue, please add the label "string-dtypes-v1.0".

This document was produced by the Zarr core development team.

Document conventions

Conformance requirements are expressed with a combination of descriptive assertions and [RFC2119] terminology. The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in the normative parts of this document are to be interpreted as described in [RFC2119]. However, for readability, these words do not appear in all uppercase letters in this specification.

All of the text of this specification is normative except sections explicitly marked as non-normative, examples, and notes. Examples in this specification are introduced with the words "for example".

Extension data types

Two extension data types are defined to represent zero-terminated bytestrings as well as fixed-length 32-bit unicode arrays.

Fixed length byte strings (zero-terminated)

These are fixed width strings corresponding to NumPy dtypes with *kind* 'S'. For backward compatibility with Python 2's `str` these are zero-terminated bytes and correspond to *numpy.bytes_ <https://numpy.org/doc/stable/reference/arrays.scalars.html#numpy.bytes_>* (or its alias *numpy.string_ <https://numpy.org/doc/stable/reference/arrays.scalars.html#numpy.bytes_>*.)

For example `a = np.array(["a", "bcd", "efgh"], dtype="S4")` creates an array where `a.dtype.kind` is 'S' and `a.data.tobytes()` is `b'a\x00\x00\x00bcd\x00efgh'`. Note that any elements of length less than 4 characters were padded with zeros so that each array element uses 4 bytes (as indicated by `dtype="S4"`).

Fixed width unicode strings

These are fixed width strings corresponding to NumPy dtypes with *kind* 'U'. These are zero-terminated bytes and correspond to *numpy.str_ <https://numpy.org/doc/stable/reference/arrays.scalars.html#numpy.str_>* (or its alias *numpy.unicode_ <https://numpy.org/doc/stable/reference/arrays.scalars.html#numpy.unicode_>*.)

For example `a = np.array(["a", "bcd", "efgh"], dtype="U4")` creates an array where `a.dtype.kind` is 'U' and each element is a sequence of 4 characters where each character occupies 4 bytes (UTF-32).

Data Types added by this extension

Table 1: Data types

Identifier	Numerical type	Size (no. bytes)	Byte order
`Sn`	`n` character fixed-width byte string	n	None
`<Un`	`n` character fixed-width unicode string (UTF-32)	4n	little-endian
`>Un`	`n` character fixed-width unicode string (UTF-32)	4n	big-endian

References

Change log

@@TODO

2.2.4 Struct data types (version 1.0)

Editor's draft 2 March 2022

Specification URI:
https://purl.org/zarr/spec/extensions/data-types/struct/1.0

Issue tracking:
GitHub issues

Suggest an edit for this spec:
GitHub editor

Abstract

This specification is a Zarr extension defining data types for structured arrays. It is an early draft and currently just describes existing support for NumPy-style structured arrays that already have support in zarr-python, but are not part of the core Zarr v3 spec.

Status of this document

Warning: This document is a **Work in Progress**. It may be updated, replaced or obsoleted by other documents at any time. It is inappapropriate to cite this document as other than work in progress.

Comments, questions or contributions to this document are very welcome. Comments and questions should be raised via GitHub issues. When raising an issue, please add the label "struct-dtypes-v1.0".

This document was produced by the Zarr core development team.

Document conventions

Conformance requirements are expressed with a combination of descriptive assertions and [RFC2119] terminology. The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in the normative parts of this document are to be interpreted as described in [RFC2119]. However, for readability, these words do not appear in all uppercase letters in this specification.

All of the text of this specification is normative except sections explicitly marked as non-normative, examples, and notes. Examples in this specification are introduced with the words "for example".

Extension data types

NumPy allows representation of Structured Arrays where each element of the array is actually some combination of fields, each of which may have its own unique data type. NumPy's Record Arrays (`numpy.recarray`) also use this data type. The actual data is stored as an opaque sequence of bytes (i.e. a structure) as represented by (`numpy.void`) and thus the string representation of this dtype in NumPy is `'|Vn'` where `n` is some integer number of bytes. In order to be able to properly interpret data of this type if is necessary to store information on the fields

A concrete example of such an array from the NumPy docs is:

```
dogs = np.array([('Rex', 9, 81.0), ('Fido', 3, 27.0)],
                dtype=[('name', 'U10'), ('age', 'i4'), ('weight', 'f4')])
```

where here `dogs.dtype.kind` is 'V' and `dogs.dtype.str` is `'|V48'` indicating the 48 bytes are needed to store each element (4 bytes each for `age` and `weight` and 4 * 10 = 40 bytes for a 10-character UTF-32 `name`). If we were to read such a sequence of bytes from a Zarr array, we need the dtype description to know how to properly interpret this sequence of 48 bytes. The NumPy dtype object has a `descr` attribute that describes this. In this case `dogs.dtype.descr` is `[('name', '<U10'), ('age', '<i4'), ('weight', '<f4')]`.

Data Types added by this extension

Table 2: Data types

Identifier	Numerical type	Size (no. bytes)	Byte order
list of (<name>, <type>) tuples	structure with named fields, each with possibly unique data type	sum over the size of the dtypes in the identifier	None

Here <field name> is the name of the struct field and <type> is any of the scalar dtypes supported by the core Zarr v3 spec or the available extensions. In the case of NumPy's structured arrays this identifier is simply array's `.dtype.descr` attribute.

References

Change log

@@TODO

2.3 Storage Transformers

Under construction.

2.3.1 Sharding storage transformer (version 1.0)

Editor's draft 18 02 2022

Specification URI:
https://purl.org/zarr/spec/extensions/storage-transformers/sharding/1.0

Corresponding ZEP:
ZEP 2 — Sharding storage transformer

Issue tracking:
GitHub issues <https://github.com/zarr-developers/zarr-specs/labels/storage-transformers-sharding-v1.0>-

Suggest an edit for this spec:
GitHub editor

Abstract

This specification defines an implementation of the Zarr storage transformer specification for sharding.

Sharding co-locates multiple chunks within a storage object, bundling them in shards.

Status of this document

Warning: This document is a draft for review and subject to changes. It will become final when the Zarr Enhancement Proposal (ZEP) 2 is approved via the ZEP process.

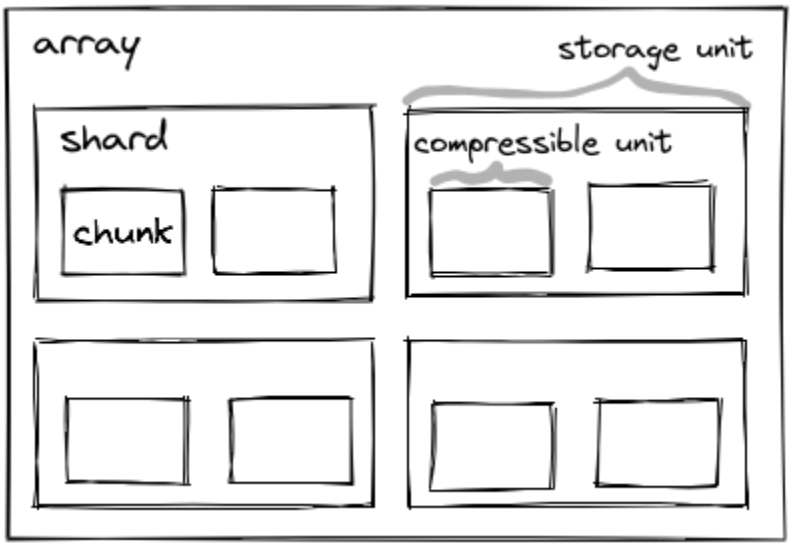
Motivation

In many cases, it becomes inefficient or impractical to store a large number of chunks in single files or objects due to the design constraints of the underlying storage. For example, the file block size and maximum inode number restrict the usage of numerous small files for typical file systems, also cloud storage such as S3, GCS, and various distributed filesystems do not efficiently handle large numbers of small files or objects.

Increasing the chunk size works only up to a certain point, as chunk sizes need to be small for read and write efficiency requirements, for example to stream data in browser-based visualization software.

Therefore, chunks may need to be smaller than the minimum size of one storage key. In those cases, it is efficient to store objects at a more coarse granularity than reading chunks.

Sharding solves this by allowing to store multiple chunks in one storage key, which is called a shard:



Document conventions

Conformance requirements are expressed with a combination of descriptive assertions and [RFC2119] terminology. The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in the normative parts of this document are to be interpreted as described in [RFC2119]. However, for readability, these words do not appear in all uppercase letters in this specification.

All of the text of this specification is normative except sections explicitly marked as non-normative, examples, and notes. Examples in this specification are introduced with the words "for example".

Configuration

Sharding can be configured per array in the *Array metadata* as follows:

```
{
  storage_transformers: [
    {
      "extension": "https://purl.org/zarr/spec/extensions/storage_transformers/sharding/
↪1.0",
      "type": "indexed",
      "configuration": {
        "chunks_per_shard": [
            2,
            2
        ]
      }
  ]
}
```

`type`

Specifies a *Binary shard format*. In this version, the only binary format is the `indexed` format.

`chunks_per_shard`

An array of integers providing the number of chunks that are combined in a shard for each dimension of the Zarr array, where each chunk may only start at a position that is divisible by `chunks_per_shard` per dimension, e.g. starting at the zero-origin. The length of the array must match the length of the array metadata `shape` entry. For example, a value `[32, 2]` indicates that 64 chunks are combined in one shard, 32 along the first dimension, and for each of those 2 along the second dimension. Some valid starting positions for a shard in the chunk-grid are therefore *[0, 0]*, *[32, 2]*, *[32, 4]*, *[64, 2]* or *[96, 18]*.

Storage transformer implementation

Key & value transformation

The storage transformer specification defines the abstract interface to be the same as the *Abstract store interface*.

The Zarr store interface is defined as a mapping of *keys* and *values*, where a *key* is a sequence of characters and a *value* is a sequence of bytes. A key-value pair is called *entry* in the following part.

This sharding transformer only adapts entries where the key starts with *data/root*, as they indicate data keys for array chunks, see *Storage keys*. All other entries are simply passed on.

Entries starting with `data/root` are grouped by their common shard, assuming storage keys from a regular chunk grid which may use a custom configured `chunk separator`: For all entries that are part of the same shard the key is changed to the shard-key and the values are combined in the *Binary shard format* as described below. The new shard-key is the chunk key divided by `chunks_per_shard` and floored per dimension. For example for `chunks_per_shard=[32, 2]`, the chunk grid position `[96, 18]` (e.g. key "data/root/foo/baz/c96/18") is transformed to the shard grid position `[3, 9]` and reassigned to the respective new key, honoring the original chunk separator (e.g. "data/root/foo/baz/c3/9"). Chunk grid positions `[96, 19]`, `[97, 18]`, ..., up to `[127, 19]` will also have the same shard grid position `[3, 9]`.

Binary shard format

The only binary format is the `indexed` format, as specified by the `type` configuration key. Other binary formats might be added in future versions.

In the indexed binary format, chunks are written successively in a shard, where unused space between them is allowed, followed by an index referencing them. The index is placed at the end of the file and has a size of 16 bytes multiplied by the number of chunks in a shard, for example `16 bytes * 64 = 1014 bytes` for `chunks_per_shard=[32, 2]`. The index holds an *offset, nbytes* pair of little-endian uint64 per chunk, the chunks-order in the index is row-major (C) order, for example for `chunks_per_shard=[2, 2]` an index would look like:

```
| chunk (0, 0)    | chunk (0, 1)    | chunk (1, 0)    | chunk (1, 1)    |
| offset | nbytes | offset | nbytes | offset | nbytes | offset | nbytes |
| uint64 | uint64 | uint64 | uint64 | uint64 | uint64 | uint64 | uint64 |
```

Empty chunks are denoted by setting both offset and nbytes to `2^64 - 1`. The index always has the full shape of all possible chunks per shard, even if they are outside of the array size.

The actual order of the chunk content is not fixed and may be chosen by the implementation as all possible write orders are valid according to this specification and therefore can be read by any other implementation. When writing partial chunks into an existing shard no specific order of the existing chunks may be expected. Some writing strategies might be

- **Fixed order**: Specify a fixed order (e.g. row-, column-major, or Morton order). When replacing existing chunks larger or equal-sized chunks may be replaced in-place, leaving unused space up to an upper limit that might

possibly be specified. Please note that for regular-sized uncompressed data all chunks have the same size and can therefore be replaced in-place. > *

- **Append-only**: Any chunk to write is appended to the existing shard, followed by an updated index. If previous chunks are updated, their storage space becomes unused, as well as the previous index. This might be useful for storage that only allows append-only updates.
- **Other formats**: Other formats that accept additional bytes at the end of the file (such as HDF) could be used for storing shards, by writing the chunks in the order the format prescribes and appending a binary index derived from the byte offsets and lengths at the end of the file.

Any configuration parameters for the write strategy must not be part of the metadata document, they need to be configured at runtime, as this is implementation specific.

API implementation

The section below defines an implementation of the *Abstract store interface* in terms of the operations of this storage transformer as a `StoreWithPartialAccess`. The term *underlying store* references either the next storage transformer in the stack or the actual store if this transformer is the last one in the stack. Any operations with keys not starting with `data/root` are simply relayed to the underlying store and not described explicitly.

- `get_partial_values(key_ranges) -> values`: For each referenced key, request the indices from the underlying store using `get_partial_values`. For each *key*, *range* pair in in *key_ranges*, check if the chunk exists by checking if the index offset and nbytes are both `2^64 - 1`. For existing keys, request the actual chunks by their ranges as read from the index using `get_partial_values`. This operation should be implemented using two `get_partial_values` operations on the underlying store, one for retrieving the indices and one for retrieving existing chunks.
- `set_partial_values(key_start_values)`: For each referenced key, check if all available chunks in a shard are referenced. In this case, a shard can be constructed according to the *Binary shard format* directly. For all other keys, request the indices from the underlying store using `get_partial_values`. All chunks that are not updated completely and exist according to the index (index offset and nbytes are both `2^64 - 1`) need to be read via `get_partial_values` from the underlying store. For simplification purposes a shard may also be read completely, combining the previous two *get* operations into one. Based on the existing chunks and value ranges that need to be updated new shards are constructed according to the *Binary shard format*. All shards that need to be updated must now be set via `set` or `set_partial_values(key_start_values)`, depending on the chosen writing strategy provided by the implementation. Specialized store implementations that allow appending to a storage object may only need to read the index to update it.
- `erase_values(keys)`: For each referenced key, check if all available chunks in a shard are referenced. In this case, the full shard is removed using `erase_values` on the underlying store. For all other keys, request the indices from the underlying store using `get_partial_values`. Update the index using an offset and nbytes of `2^64 - 1` to mark missing chunks. The updated index may be written in-place using `set_partial_values(key_start_values)`, or a larger rewrite of the shard may be done including the index update, but also removing value ranges corresponding to the erased chunks.
- `erase_prefix()`: If the prefix contains a part of the chunk-grid key, this part is translated to the referenced shard and contained chunks. For affected shards where all contained chunks are erased the prefix is rewritten to the corresponding shard key and the operation is relayed to the underlying store. For all shards where only some chunks are erased the affected chunks are removed by invoking the operation `erase_values` on this storage transformer with the respective chunk keys.
- `list()`: See `list_prefix` with the prefix `/`.
- `list_prefix(prefix)`: If the prefix contains a part of the chunk-grid key, this part is translated to the referenced shard and contained chunks. Then, `list_prefix` is called on the underlying store with the translated

prefix. For all listed shards request the indices from the underlying store using `get_partial_values`. Existing chunks, where the index offset or nbytes are not `2^64 - 1` are then listed by their original key.

- `list_dir(prefix)`: If the prefix contains a part of the chunk-grid key, this part is translated to the referenced shard and contained chunks. Then, `list_dir` is called on the underlying store with the translated prefix. For all *retrieved prefixes* (not full keys) with partial shard keys, the corresponding original prefixes covering all possible chunks in the shard are listed. For *retrieved full keys* the indices from the underlying store are requested using `get_partial_values`. Existing chunks, where the index offset or nbytes are not `2^64 - 1` are then listed by their original key.

Note: Not all listed prefixes must necessarily contain keys, as shard prefixes with partially available chunks return prefixes for all possible chunks without verifying their existence for performance reasons. Listing those prefixes is still safe as some chunks in their corresponding shard exist, but not necessarily in the requested prefix, possibly leading to empty responses. Please note that this only applies to returned prefixes, *not* for full keys referencing storage objects. Returned full keys always reflect the available chunks and are safe to request.

References

Change log

This section is a placeholder for keeping a log of the snapshots of this document that are tagged in GitHub and what changed between them.

A number of other features might be included in the core v3 specification, but are currently considered as extensions.

- Dimensions names for arrays: see https://github.com/zarr-developers/zarr-specs/issues/73
- Ability to provide soft / hard ? links, and region references.

CHAPTER

THREE

CODECS

Under construction.

3.1 Blosc codec (version 1.0)

Editor's draft 26 July 2019

Specification URI:
https://purl.org/zarr/spec/codecs/blosc/1.0

Corresponding ZEP:
ZEP 1 — Zarr specification version 3

Issue tracking:
GitHub issues

Suggest an edit for this spec:
GitHub editor

3.1.1 Abstract

This specification defines an implementation of the Zarr abstract store API using a file system.

3.1.2 Status of this document

Warning: This document is a draft for review and subject to changes. It will become final when the Zarr Enhancement Proposal (ZEP) 1 is approved via the ZEP process.

3.1.3 Document conventions

Conformance requirements are expressed with a combination of descriptive assertions and [RFC2119] terminology. The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in the normative parts of this document are to be interpreted as described in [RFC2119]. However, for readability, these words do not appear in all uppercase letters in this specification.

All of the text of this specification is normative except sections explicitly marked as non-normative, examples, and notes. Examples in this specification are introduced with the words "for example".

3.1.4 Configuration parameters

cname:
A string identifying the internal compression algorithm to be used. At the time of writing, the following values are supported by the c-blosc library: "lz4", "lz4hc", "blosclz", "zstd", "snappy", "zlib".

clevel:
An integer from 0 to 9 which controls the speed and level of compression. A level of 1 is the fastest compression method and produces the least compressions, while 9 is slowest and produces the most compression. Compression is turned off completely when level is 0.

shuffle:
An integer value in the set {0, 1, 2, -1} indicating the way bytes or bits are rearranged, which can lead to faster and/or greater compression. A value of 1 indicates that byte-wise shuffling is performed prior to compression. A value of 2 indicates the bit-wise shuffling is performed prior to compression. If a value of -1 is given, then default shuffling is used: bit-wise shuffling for buffers with item size of 1 byte, byte-wise shuffling otherwise. Shuffling is turned off completely when the value is 0.

blocksize:
An integer giving the size in bytes of blocks into which a buffer is divided before compression. A value of 0 indicates that an automatic size will be used.

For example, the array metadata document below specifies that the compressor is the Blosc codec configured with a compression level of 1, byte-wise shuffling, the `lz4` compression algorithm and the default block size:

```
{
    "codecs": [{
        "type": "https://purl.org/zarr/spec/codecs/blosc/1.0",
        "configuration": {
            "cname": "lz4",
            "clevel": 1,
            "shuffle": 1,
            "blocksize": 0
        }
    }],
}
```

3.1.5 Format and algorithm

Blosc is a meta-compressor, which divides an input buffer into blocks, then applies an internal compression algorithm to each block, then packs the encoded blocks together into a single output buffer with a header. The format of the encoded buffer is defined in [BLOSC]. The reference implementation is provided by the c-blosc library.

3.1.6 References

3.1.7 Change log

No changes yet.

3.2 Endian codec (version 1.0)

Editor's draft 26 July 2019

Specification URI:
https://purl.org/zarr/spec/codecs/endian/1.0

Corresponding ZEP:
ZEP 1 — Zarr specification version 3

Issue tracking:
GitHub issues

Suggest an edit for this spec:
GitHub editor

3.2.1 Abstract

This specification defines an implementation of the Zarr abstract store API using a file system.

3.2.2 Status of this document

Warning: This document is a draft for review and subject to changes. It will become final when the Zarr Enhancement Proposal (ZEP) 1 is approved via the ZEP process.

3.2.3 Document conventions

Conformance requirements are expressed with a combination of descriptive assertions and [RFC2119] terminology. The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in the normative parts of this document are to be interpreted as described in [RFC2119]. However, for readability, these words do not appear in all uppercase letters in this specification.

All of the text of this specification is normative except sections explicitly marked as non-normative, examples, and notes. Examples in this specification are introduced with the words "for example".

3.2.4 Configuration parameters

endian:
Required. A string equal to either `"big"` or `"little"`.

3.2.5 Format and algorithm

Each element of the array is encoded using the specified endian variant of its default binary representation. Array elements are encoded in lexicographical order. For example, with `endian` specified as `big`, the `int32` data type is encoded as a 4-byte big endian two's complement integer, and the `complex128` data type is encoded as two consecutive 8-byte big endian IEEE 754 binary64 values.

Note: Since the default binary representation of all data types is little endian, specifying this codec with `endian` equal to `"little"` is equivalent to omitting this codec, because if this codec is omitted, the default binary representation of the data type, which is always little endian, is used instead.

3.2.6 References

3.2.7 Change log

No changes yet.

3.3 Gzip codec (version 1.0)

Editor's draft 26 July 2019

Specification URI:
https://purl.org/zarr/spec/codecs/gzip/1.0

Corresponding ZEP:
ZEP 1 — Zarr specification version 3

Issue tracking:
GitHub issues

Suggest an edit for this spec:
GitHub editor

3.3.1 Abstract

This specification defines an implementation of the Zarr abstract store API using a file system.

3.3.2 Status of this document

Warning: This document is a draft for review and subject to changes. It will become final when the Zarr Enhancement Proposal (ZEP) 1 is approved via the ZEP process.

3.3.3 Document conventions

Conformance requirements are expressed with a combination of descriptive assertions and [RFC2119] terminology. The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in the normative parts of this document are to be interpreted as described in [RFC2119]. However, for readability, these words do not appear in all uppercase letters in this specification.

All of the text of this specification is normative except sections explicitly marked as non-normative, examples, and notes. Examples in this specification are introduced with the words "for example".

3.3.4 Configuration parameters

level:
: An integer from 0 to 9 which controls the speed and level of compression. A level of 1 is the fastest compression method and produces the least compressions, while 9 is slowest and produces the most compression. Compression is turned off completely when level is 0.

For example, the array metadata below specifies that the compressor is the Gzip codec configured with a compression level of 1:

```
{
    "codecs": [{
        "type": "https://purl.org/zarr/spec/codec/gzip",
        "configuration": {
            "level": 1
        }
    }],
}
```

3.3.5 Format and algorithm

Encoding and decoding is performed using the algorithm defined in [RFC1951].

Encoded data should conform to the Gzip file format [RFC1952].

3.3.6 References

3.3.7 Change log

No changes yet.

CHAPTER

FOUR

STORES

Under construction.

4.1 File system store (version 1.0)

Editor's draft 26 July 2019

Specification URI:
https://purl.org/zarr/spec/stores/filesystem/1.0

Issue tracking:
GitHub issues

Suggest an edit for this spec:
GitHub editor

4.1.1 Abstract

This specification defines an implementation of the Zarr abstract store API using a file system.

4.1.2 Status of this document

Warning: This document is a **Work in Progress**. It may be updated, replaced or obsoleted by other documents at any time. It is inappapropriate to cite this document as other than work in progress.

Comments, questions or contributions to this document are very welcome. Comments and questions should be raised via GitHub issues. When raising an issue, please add the label "stores-filesystem-v1.0".

This document was produced by the Zarr core development team.

4.1.3 Notes about design decisions for the native File System Store

The original file system store is designed for simplicity and easy manipulation and transfer by external tools not aware of the store structure. In particular tools like `gsutil` can be use to transfer a local directory store to cloud base storage, hence the keys choices will be conserved.

4.1.4 Document conventions

Conformance requirements are expressed with a combination of descriptive assertions and [RFC2119] terminology. The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in the normative parts of this document are to be interpreted as described in [RFC2119]. However, for readability, these words do not appear in all uppercase letters in this specification.

All of the text of this specification is normative except sections explicitly marked as non-normative, examples, and notes. Examples in this specification are introduced with the words "for example".

4.1.5 Native storage operations

Here we consider a file system to be any system comprised of files and directories, where:

- Each file has a name (sequence of characters) and contents (sequence of bytes).
- Each directory has a name (sequence of characters) and children (set of zero or more files and/or directories).
- Each file or directory can be addressed by a path, comprised of its name and the names of all ancestor directories, which uniquely identifies it within the file system.

…and where the following native operations are supported:

- Create a file.
- Write the contents of a file.
- Read the contents of a file.
- Delete a file.
- Create a directory.
- List the children of a directory, returning the name and type (file or directory) of each child.
- Delete a directory.

4.1.6 Key translation

The Zarr store interface is defined in terms of *keys* and *values*, where a *key* is a sequence of characters and a *value* is a sequence of bytes. A file system store translates keys into file system paths. This translation assumes that the store has been defined relative to a base directory. The translation is as follows:

- Replace any forward slash characters ('/') in the key with the native directory separator for the file system.
- Join the result to the base directory path, using the native directory separator.

For example, if the file system is a POSIX file system, and the base directory path is "/data", then the key "foo/bar" is translated to the file system path "/data/foo/bar".

For example, if the file system is a Windows file system, and the base directory path is "C:\data", then the key "foo/bar" is translated to the file system path "C:\data\foo\bar".

When returning information about available keys, a file system store performs the reverse translation from file system paths to keys. This translation is as follows:

- Replace any native directory separator characters with the forward slash character.
- Strip the base directory path from the beginning of the path.

For example, if the file system is a POSIX file system, and the base directory path is "/data", then the file system path "/data/foo/bar" is translated to the key "foo/bar".

For example, if the file system is a Windows file system, and the base directory path is "C:\data", then the file system path "C:\data\foo\bar" is translated to the key "foo/bar".

4.1.7 Store API implementation

The section below defines an implementation of the Zarr *Abstract store interface* in terms of the native operations of this storage system. Below `fspath_to_key()` is a function that translates file system paths to store keys, and `key_to_fspath()` is a function that translates store keys to file system paths, as defined in the section above.

- `get(key) -> value` : Read and return the contents of the file at file system path `key_to_fspath(key)`.
- `set(key, value)` : Write `value` as the contents of the file at file system path `key_to_fspath(key)`.
- `delete(key)` : Delete the file or directory at file system path `key_to_fspath(key)`.
- `list()` : Recursively walk the file system from the base directory, returning an iterator over keys obtained by calling `fspath_to_key(fp)` for each descendant file path `fp`.
- `list_prefix(prefix)` : Obtain a file system path by calling `key_to_fspath(prefix)`. If the result is a directory path, recursively walk the file system from this directory, returning an iterator over keys obtained by calling `fspath_to_key(fp)` for each descendant file path `fp`.
- `list_dir(prefix)` : Obtain a file system path by calling `key_to_fspath(prefix)`. If the result is a director path, list the directory children. Return a set of keys obtained by calling `fspath_to_key(fp)` for each child file path `fp`, and a set of prefixes obtained by calling `fspath_to_key(dp)` for each child directory path `dp`.

4.1.8 References

4.1.9 Change log

@@TODO

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[RFC8259] T. Bray, Ed. The JavaScript Object Notation (JSON) Data Interchange Format. December 2017. Best Current Practice. URL: https://tools.ietf.org/html/rfc8259

[RFC2119] S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[RFC2119] S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[RFC2119] S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[NumPy] NumPy Datetimes and Timedeltas. NumPy version 1.16.0 documentation. URL: https://docs.scipy.org/doc/numpy-1.16.0/reference/arrays.datetime.html

[UTF-32] UTF-32 on Wikipedia. documentation. URL: https://en.wikipedia.org/wiki/UTF-32

[NumPy] NumPy Data type objects. NumPy version 1.22.0 documentation. URL: https://numpy.org/doc/1.22/reference/arrays.dtypes.html

[NumPy] NumPy Data type objects. NumPy version 1.22.0 documentation. URL: https://numpy.org/doc/1.22/reference/arrays.dtypes.html

[RFC2119] S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[RFC2119] S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[BLOSC] F. Alted. Blosc Chunk Format. URL: https://github.com/Blosc/c-blosc/blob/HEAD/README_CHUNK_FORMAT.rst

[RFC2119] S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[RFC2119] S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[RFC1951] P. Deutsch. DEFLATE Compressed Data Format Specification version 1.3. Requirement Levels. May 1996. Informational. URL: https://tools.ietf.org/html/rfc1951

[RFC1952] P. Deutsch. GZIP file format specification version 4.3. Requirement Levels. May 1996. Informational. URL: https://tools.ietf.org/html/rfc1952

[RFC2119] S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119