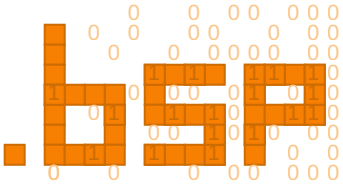

Binary Sparse Format Specification

Commit Snapshot, 29 August 2026

This version:
https://Binsparse.github.io/versions/0.1.0/

Issue Tracking:
GitHub

Editors:
Benjamin Brock (Intel)
Tim Davis (Texas A&M)
Jim Kitchen (Anaconda)
Erik Welch (NVIDIA)
Isaac Virshup (Helmholtz Munich)
Willow Ahrens (MIT)

Abstract

A cross-platform binary storage format for sparse data, particularly sparse matrices.

Table of Contents

Specification version: 0.1.0

§ 1. Introduction

The Binsparse Specification (name tentative) provides a cross-platform format for efficiently storing data, particularly sparse multidimensional arrays, in binary format. Binsparse is designed to be both a standalone and embeddable format. It consists of two parts:

- a JSON *descriptor* detailing the structure of the chosen binary storage format
- one or more *binary arrays*, each stored under a predefined dataset name based on the format

Both the descriptor and binary arrays are stored in a supported *binary container*.

§ 2. Definitions

Binsparse is intended to support multidimensional *sparse arrays*, meaning arrays in which not every location has a value. We refer to each location in the array with a value as a *stored value*. Stored values have associated with them a *scalar value*, which is the value stored in that location in the array, and one or more *indices*, which describe the location where the stored value is located in the array. Some or all of these indices may be stored explicitly, or they may be implicitly derived, depending on storage format. When stored explicitly, indices are 0-based positive integers.

§ 3. Binsparse JSON Descriptors

Binsparse descriptors are key-value metadata that describe the binary format of sparse data. The key-value data is namespaced as "binsparse" to avoid any conflict with other metadata in the container. The required entries in the "binsparse" entry are listed below. Optional attributes may be defined to hold additional metadata and must be stored outside of the "binsparse" namespace.

EXAMPLE 1 ¶

Example of a JSON descriptor for a compressed-sparse column (CSC) matrix with 10 rows and 12 columns, containing 20 float32 values, along with user-defined attributes.

```
{
  "binsparse": {
    "version": "0.1.0",
    "format": "CSC",
    "shape": [10, 12],
    "number_of_stored_values": 20,
    "data_types": {
      "pointers_to_1": "uint64",
      "indices_1": "uint64",
      "values": "float32"
    }
  },
  "original_source": "https://url/of/original/file.mtx",
  "author": "John Doe"
}
```

§ 3.1. Version

Version indicates the version of the Binsparse specification used here. It MUST be a valid Semantic Versioning 2.0.0 version of the form `major.minor.patch`, optionally followed by a pre-release identifier or build metadata.

Given a version number `major.minor.patch`, the major version MUST be incremented for incompatible changes, the minor version MUST be incremented for backwards-compatible additions, and the patch version MUST be incremented for backwards-compatible fixes (usually changes in wording that serve to clarify but not change the format). Major version zero (`0.y.z`) is for initial development and does not provide a backwards-compatibility guarantee.

§ 3.2. Shape

The `shape` key must be present and shall define the shape of the sparse tensor. It shall contain a JSON array of integers, with index `i` containing the size of the `i`'th dimension. For matrices, index `0` shall contain the number of rows, and index `1` shall contain the number of columns. For vectors, index `0` shall contain the vector's dimension.

> NOTE: a matrix has shape [`number_of_rows`, `number_of_columns`] regardless of whether the format orientation is row-wise or column-wise.

§ 3.3. Number of Stored Values

The `number_of_stored_values` key must be present and shall define the number of explicit values that are stored as explicit entries in the sparse tensor format.

> NOTE: For sparse tensors with all values the same (ISO), `number_of_stored_values` still refers to the number of explicit entries in the sparse tensor format whose indices are stored, regardless of the fact that the individual scalar values themselves are not explicitly stored.

§ 3.4. Fill

The `fill` key may be present. If the `fill` key is present, it shall have a boolean value. If the value is true, it signifies the presence of a `fill_value` array, whose single element defines the value at indices not specified by the sparse tensor structure.

§ 3.5. Format

The `format` key must be present and shall describe the binary storage format of dense arrays used to represent the sparse array. The format defined by the `format` key determines the named binary arrays that shall exist in the binary storage container.

§ 3.5.1. Pre-defined Formats

The following is a list of all pre-defined formats and the arrays that shall be present in the binary container. `number_of_elements` refers to the number of stored values, `number_of_rows` refers to the number of rows, and `number_of_columns` refers to the number of columns.

§ *3.5.1.1. DVEC*

Dense Vector format

values
: Array of size `number_of_elements` containing stored values.

The element of the vector located at index `i` has scalar value `values[i]`.

§ 3.5.1.2. DMATR

Row-Major Dense Matrix format

values
: Array of size `number_of_elements` containing stored values.

The element of the vector located at index `i, j` has scalar value `values[i * number_of_columns + j]`.

§ 3.5.1.3. DMATC

Column-Major Dense Matrix format

values
: Array of size `number_of_elements` containing stored values.

The element of the vector located at index `i, j` has scalar value `values[i + j * number_of_rows]`.

§ 3.5.1.4. DMAT

DMAT format is an alias for § 3.5.1.2 DMATR format.

§ 3.5.1.5. CVEC

Compressed Sparse Vector format

indices_0
: Array of size `number_of_elements` containing indices.

values
: Array of size `number_of_elements` containing stored values.

The element of the vector located at index `indices_0[i]` has scalar value `values[i]`. Elements shall be sorted by index and must not be duplicated.

§ 3.5.1.6. CSR

Compressed-Sparse Row format

pointers_to_1
: Array of size `number_of_rows + 1` containing start and end positions by row.

indices_1
: Array of size `number_of_elements` containing 0-based column indices.

values
: Array of size `number_of_elements` containing stored values.

The column indices of the stored values located in row `i` are located in the range `[pointers_to_1[i], pointers_to_1[i+1])` in the `indices_1` array. The scalar values for each of those stored values is stored in the corresponding index in the `values` array.

Within a row, elements shall be sorted by column index and must not be duplicated.

§ 3.5.1.7. CSC

Compressed-Sparse Column format

pointers_to_1
: Array of size `number_of_columns + 1` containing start and end positions by column.

indices_1
: Array of size `number_of_elements` containing 0-based row indices.

values
: Array of size `number_of_elements` containing stored values.

The rows indices of the stored values located in column `j` are located in the range `[pointers_to_1[j], pointers_to_1[j+1])` in the `indices_1` array. The scalar values for each of those stored values is stored in the corresponding index in the `values` array.

Within a column, elements shall be sorted by row index and must not be duplicated.

§ *3.5.1.8. DCSR*

Doubly Compressed-Sparse Row format

indices_0
: Array of size `number_of_nonempty_rows` containing 0-based row indices corresponding to positions within `pointers_to_1`.

pointers_to_1
: Array of size `number_of_nonempty_rows + 1` containing start and end positions.

indices_1
: Array of size `number_of_elements` containing 0-based column indices.

values
: Array of size `number_of_elements` containing stored values.

DCSR is similar to CSR, except that rows which are entirely empty are not stored. `pointers_to_1` contains no repeated values. Because the position within `pointers_to_1` no longer dictates the corresponding row index, `indices_0` provides the row index.

Rows shall be sorted and must not be duplicated. Within each row, elements shall be sorted by column index and must not be duplicated.

§ *3.5.1.9. DCSC*

Doubly Compressed-Sparse Column format

indices_0
: Array of size `number_of_nonempty_columns` containing 0-based column indices corresponding to positions within `pointers_to_1`.

pointers_to_1
: Array of size `number_of_nonempty_columns + 1` containing start and end positions.

indices_1
: Array of size `number_of_elements` containing 0-based row indices.

values
: Array of size `number_of_elements` containing stored values.

DCSC is similar to CSC, except that columns which are entirely empty are not stored. `pointers_to_1` contains no repeated values. Because the position within `pointers_to_1` no longer dictates the corresponding column index, `indices_0` provides the column index.

Columns shall be sorted and not duplicated. Within each column, elements shall be sorted by row index and must not be duplicated.

3.5.1.10. COOR

Row-wise Coordinate format

indices_0
: Array of size `number_of_elements` containing 0-based row indices.

indices_1
: Array of size `number_of_elements` containing 0-based column indices.

values
: Array of size `number_of_elements` containing stored values.

Pairs of (row index, column index) shall be sorted first by row and then by column. Pairs must not be duplicated.

3.5.1.11. COOC

Column-wise Coordinate format

indices_0
: Array of size `number_of_elements` containing 0-based column indices.

indices_1
: Array of size `number_of_elements` containing 0-based row indices.

values
: Array of size `number_of_elements` containing stored values.

Pairs of (column index, row index) shall be sorted first by column and then by row. Pairs must not be duplicated.

3.5.1.12. COO

Coordinate format is an alias for § 3.5.1.10 COOR format.

3.5.2. Custom Formats

The contents of this section are optional for all parsers, but enable customizable sparse formats for matrices and tensors. If the parser does not implement tensor formats, it should error when the "custom" key is present.

Binsparse describes custom multidimensional formats hierarchically. We can understand these formats as arrays of arrays, where the parent array and child arrays might use different formats. For example, we could have a dense outer array which contains sparse inner arrays, so the first index would be dense and the second index would be sparse. To achieve efficient storage, all arrays in the same level are stored contiguously in a specialized datastructure called a level.

A level is a collection of zero or more arrays which all have the same format. The elements of arrays in a level may be subarrays in a sublevel. The global array we wish to store is represented by a level that holds a single root array.

For example, the simplest level is the element format, which represents a collection of scalars. We can represent a collection of dense vectors with a dense level format. Each vector in the collection would be composed from contiguous scalars in an element level (analogously to the numpy.stack operator). We can represent a collection of sparse vectors using a sparse level. The sparse level format represents sparse vectors by listing the locations of nonzeros, and storing only the nonzero scalars inside an element level.

In addition to storing scalars, dense and sparse levels may themselves store multidimensional arrays. This leads to multiple ways to store sparse matrices and tensors. For example, a dense vector of sparse vectors is equivalent to the CSR matrix format, and a sparse vector of sparse vectors is equivalent to the hypersparse DCSR matrix format.

When defining a custom format, the outermost `level` key is defined as the root level descriptor (a level which will only hold one array). If a level holds many different arrays, we refer to the `p`th array as the array in position `p`.

Levels are row-major by default (adding an outer level adds a row dimension). The format descriptor may optionally define a `transpose` key, equal to a list of the described dimensions in the order they should appear. If the tensor we wish to represent is `A` and the tensor described by the format descriptor is `B`, then `A[i_1, ..., i_n] = B[i_(transpose[1]), ..., i_(transpose[n])]`. `transpose` must be a permutation.

If the `custom` key is present, it holds a dictionary for custom formats. The root level is stored under the `level` key. Each level mush have a `level_desc` attribute which describe the storage format of the level.

The level descriptors are dictionaries defined as follows:

§ *3.5.2.1. Element*

If the level descriptor is "element", the level represents zero or more scalars.

values
: Array of size `number_of_positions` whose `p`th element holds the value of the scalar at position `p`.

§ *3.5.2.2. Dense*

If the level descriptor is "dense", the `level` key must be present. The `rank` key must be present, and set to an integer `r` greater than or equal to 1. The dense level represents zero or more `r`-dimensional dense arrays whose elements are themselves arrays specified by `level`. For example, a dense level of rank 2 represents a collection of dense matrices of subarrays.

Assuming that the level describes arrays of shape `I_0, ..., I_(N - 1)`, the array at position `p` in a dense level of rank `r` is an array whose slice

```
A[i_0, ..., i_(r - 1), :, ..., :]
```

is described by the row-major position

```
q = (((((p * I_0) + i_0) * I_1) + i_1) * I_2 + i_2) * ... + i_(r - 1)
```

of the sublevel.

§ *3.5.2.3. Sparse*

If the level descriptor is "sparse", the `level` key must be present. The `rank` key must be present, and set to an integer `r` greater than or equal to `1`. The sparse level represents zero or more `r`-dimensional sparse arrays whose non-implicit elements are themselves arrays specified by `level`. For example, a sparse level of rank 1 represents a collection of sparse vectors of subarrays.

Assume that this level represents `n`-dimensional subarrays and the root array is `N`-dimensional. The sparse level implies the following binary arrays are present:

pointers_to_(N - n)
: Array of size `number_of_positions + 1` whose 1st element is equal to `0` and whose `p + 1`th element is equal to the sum of `pointers_to_(N - n)[p]` and the number of explicitly represented slices in the `p`th position.

indices_(N - n), ..., indices(N - n + r - 1)
: There are `r` such arrays. When `A[i_0, ..., i_(r - 1), :, ..., :]` is explicitly represented by the subarray in position `q`, `indices_(N-n+s)[q] = i_s`. The arrays must be ordered such that the tuples `(indices_(N-n)[q], ..., indices_(N-n+r-1)[q])` are unique and appear in lexicographic order for all `q` in each range `pointers_to_(N-n)[p] <= q < pointers_to_(N-n)[p + 1]`. This array must contain no other elements.

Special note: If the sparse level is the root level, the `pointers` array should be ommitted, as its first value will be `0` and its last value will be the length of any of the `indices` arrays in this level.

§ 3.5.3. Equivalent Formats

The following formats are equivalent. Parsers which support custom formats should also write `format` aliases when appropriate.

§ *3.5.3.1. DVEC*

```
"custom": {
  "level": {
    "level_desc": "dense",
    "rank": 1,
    "level": {
      "level_desc": "element",
    }
  }
}
```

§ *3.5.3.2. DMATR*

```
"custom": {
  "level": {
    "level_desc": "dense",
    "rank": 1,
    "level": {
      "level_desc": "dense",
      "rank": 1,
      "level": {
        "level_desc": "element",
      }
    }
  }
}
```

§ *3.5.3.3. DMATC*

```
"custom": {
  "transpose": [1, 0],
  "level": {
    "level_desc": "dense",
```

```
    "rank": 1,
    "level": {
      "level_desc": "dense",
      "rank": 1,
      "level": {
        "level_desc": "element",
      }
    }
  }
}
```

3.5.3.4. CVEC

```
"custom": {
  "level": {
    "level_desc": "sparse",
    "rank": 1,
    "level": {
      "level_desc": "element",
    }
  }
}
```

3.5.3.5. CSR

```
"custom": {
  "level": {
    "level_desc": "dense",
    "rank": 1,
    "level": {
      "level_desc": "sparse",
      "rank": 1,
      "level": {
        "level_desc": "element",
      }
    }
  }
}
```

3.5.3.6. CSC

```
"custom": {
  "transpose": [1, 0],
  "level": {
    "level_desc": "dense",
    "rank": 1,
    "level": {
      "level_desc": "sparse",
      "rank": 1,
      "level": {
        "level_desc": "element",
      }
```

```
    }
  }
}
```

3.5.3.7. DCSR

```
"custom": {
  "level": {
    "level_desc": "sparse",
    "rank": 1,
    "level": {
      "level_desc": "sparse",
      "rank": 1,
      "level": {
        "level_desc": "element",
      }
    }
  }
}
```

3.5.3.8. DCSC

```
"custom": {
  "transpose": [1, 0],
  "level": {
    "level_desc": "sparse",
    "rank": 1,
    "level": {
      "level_desc": "sparse",
      "rank": 1,
      "level": {
        "level_desc": "element",
      }
    }
  }
}
```

3.5.3.9. COOR

```
"custom": {
  "level": {
    "level_desc": "sparse",
    "rank": 2,
    "level": {
      "level_desc": "element",
    }
  }
}
```

§ 3.5.3.10. COOC

Column-wise Coordinate format

```
"custom": {
  "transpose": [1, 0],
  "level": {
    "level_desc": "sparse",
    "rank": 2,
    "level": {
      "level_desc": "element",
    }
  }
}
```

§ 3.6. Data Types

The `data_types` key must be present and shall define the data types of all required arrays based on the § 3.5 Format. The data type declares the type of both the on-disk array as well as the in-memory array.

For a given § 3.5 Format, all named binary arrays for that format shall have a corresponding name in `data_types`.

The following strings shall be used to describe data types:

"uint8"
: unsigned 8-bit integer

"uint16"
: unsigned 16-bit integer

"uint32"
: unsigned 32-bit integer

"uint64"
: unsigned 64-bit integer

"int8"
: signed 8-bit integer

"int16"
: signed 16-bit integer

"int32"
: signed 32-bit integer

"int64"
: signed 64-bit integer

"float32"
: IEEE binary32 floating point number

"float64"
: IEEE binary64 floating point number

"bint8"
: An unsigned 8-bit integer, to be reinterpreted as a Boolean number, however that is represented in the host language. The value 0 shall map to false and the value 1 shall map to true. When parsing, implementations may choose to interpret values other than 0 or 1 as true, or throw an error.

§ 3.7. Value Modifiers

When the value array is meant to be reinterpreted before reading, a special bracket syntax is provided to indicate modifications to the underlying element level.

§ 3.7.1. Complex Values (complex)

When a value array is composed of alternating real and imaginary components of complex numbers, the type is written as `complex[<type>]`. For example, a value array of complex `float64` would have a datatype of `complex[float64]` The real component of the `i`th element in the modified array shall be stored at position `2i` in the original array, and the imaginary component of the `i`th element in the modified array shall be at position `2i + 1` in the underlying array. The `complex` value modifier may only be used with the types `float32` and `float64`.

§ 3.7.2. All Values the Same (ISO)

When all values of a sparse array are the same identical value, the type is written as `iso[<type>]`. This indicates that the array will store only a single element which is common to all stored indices. All elements in the modified array shall be stored at position 0 of the underlying array.

EXAMPLE 2 ¶

Example of a CSR Matrix whose values are all 7.

	0	**1**	**2**	**3**	**4**
0	.	.	.	7	.
1	.	7	.	.	7
2	.	.	.	.	.
3	.	7	7	.	.
4	.	.	.	7	.

```
{
  "version": "0.1.0",
  "format": "CSR",
  "shape": [5, 5],
  "number_of_stored_values": 6,
  "data_types": {
    "pointers_to_1": "uint64",
    "indices_1": "uint64",
    "values": "iso[int8]"
  }
}
```

- `pointers_to_1` = [0, 1, 3, 3, 5, 6]
- `indices_1` = [3, 1, 4, 1, 2, 3]
- `values` = [7]

NOTE: Structure-only matrices (allowed in matrix market format) can be stored using this technique with a value of 1. This adds only a small amount of overhead while describing essentially the same matrix.

3.8. Structure

The `structure` key, if present, denotes a special matrix structure in which only one triangle of the matrix is stored and the structure and values in the other triangle are inferred.

3.8.1. Pre-Defined Structures

The follow pre-defined values can be supplied for the `structure` key to indicate the structure of the matrix.

3.8.1.1. symmetric_lower

The `symmetric_lower` value indicates that the matrix has a symmetric structure with only the lower triangle stored. For all matrix entries with row and column indices `i,j` and value `v`, `i >= j`. If `i != j`, the entry implies the presence of an entry at row and column index `j,i` with value `v`.

3.8.1.2. symmetric_upper

The `symmetric_upper` value indicates that the matrix has a symmetric structure with only the upper triangle stored. For all matrix entries with row and column indices `i,j` and value `v`, `i <= j`. If `i != j`, the entry implies the presence of an entry at row and column index `j,i` with value `v`.

3.8.1.3. hermitian_lower

The `hermitian_lower` value indicates that the matrix has a Hermitian structure with only the lower triangle stored. For all matrix entries with row and column indices `i,j` and value `v`, `i >= j`. If `i != j`, the entry implies the presence of an entry at row and column index `j,i` with a value equal to the complex conjugate of `v`. The matrix's value type must be complex.

3.8.1.4. hermitian_upper

The `hermitian_upper` value indicates that the matrix has a Hermitian structure with only the upper triangle stored. For all matrix entries with row and column indices `i,j` and value `v`, `i <= j`. If `i != j`, the entry implies the presence of an entry at row and column index `j,i` with a value equal to the complex conjugate of `v`. The matrix's value type must be complex.

3.8.1.5. skew_symmetric_lower

The `skew_symmetric_lower` value indicates that the matrix has a skew-symmetric structure with only the lower triangle stored. For all matrix entries with row and column indices `i,j` and value `v`, `i >= j`. If `i != j`, the entry implies the presence of an entry at row and column index `j,i` with value `-v`.

3.8.1.6. skew_symmetric_upper

The `symmetric_upper` value indicates that the matrix has a skew-symmetric structure with only the upper triangle stored. For all matrix entries with row and column indices `i,j` and value `v`, `i <= j`. If `i != j`, the entry implies the presence of an entry at row and column index `j,i` with value `-v`.

EXAMPLE 3

Example of a symmetric CSR matrix.

	0	**1**	**2**	**3**	**4**
0	1	.	.	.	.
1	2	9	.	.	.
2	7	.	2	.	.
3	.	2	.	3	.
4	.	.	3	.	7

```
{
  "version": "0.1.0",
  "format": "CSR",
  "shape": [5, 5],
  "number_of_stored_values": 9,
  "structure": "symmetric_lower",
  "data_types": {
    "pointers_to_1": "uint64",
    "indices_1": "uint64",
    "values": "int8"
  }
}
```

- `pointers_to_1` = [0, 1, 3, 5, 7, 9]
- `indices_1` = [0, 0, 1, 0, 2, 1, 3, 2, 4]
- `values` = [1, 2, 9, 7, 2, 2, 3, 3, 7]

NOTE: `number_of_stored_values` reflects the number of entries explicitly stored in the sparse tensor format. This means that for symmetric, Hermittion, and skew-symmetric matrices, `number_of_stored_values` reflects the number of values that are stored, not the number of logical values in both matrix triangles. If the optional attribute `number_of_diagonal_elements` is provided, the number of logical values in both triangles can be computed in constant time.

4. Binary Containers

Binary containers must store binary arrays in a standardized, cross-platform manner, using the corresponding dataset names previously defined.

4.1. Supported Binary Containers

Currently supported binary containers include HDF5 and NetCDF (but should include more).

Conformance

Conformance requirements are expressed with a combination of descriptive assertions and RFC 2119 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 RFC 2119. 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. [RFC2119]

Examples in this specification are introduced with the words “for example” or are set apart from the normative text with `class="example"`, like this:

> EXAMPLE 4 ¶
> This is an example of an informative example.

Informative notes begin with the word “Note” and are set apart from the normative text with `class="note"`, like this:

> Note, this is an informative note.

§ References

§ Normative References

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