

---

# **parquet-writer**

***Release v0.4.0***

**Daniel Joseph Antrim**

**Nov 19, 2021**



# CONTENTS

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Storing Basic Value Types</b>	<b>5</b>
2.1	Declaring Columns of Basic Value Types . . . . .	5
2.2	Writing Columns of Basic Value Types . . . . .	6
<b>3</b>	<b>Storing Lists of Basic Value Types</b>	<b>7</b>
3.1	Declaring List Type Columns . . . . .	7
3.2	Writing List Type Columns . . . . .	7
<b>4</b>	<b>Storing Struct Type Columns</b>	<b>9</b>
4.1	Declaring Struct Type Columns . . . . .	9
4.2	Writing Struct Type Columns . . . . .	9
<b>5</b>	<b>Storing Lists of Structs</b>	<b>13</b>
5.1	Declaring Lists of Structs . . . . .	13
5.2	Writing Lists of Structs . . . . .	13
5.3	Constraints . . . . .	15
<b>6</b>	<b>Storing Structs That Have Struct Fields</b>	<b>17</b>
6.1	Declaring Structs that have Struct Fields . . . . .	17
6.2	Writing Structs with Struct Fields . . . . .	18
6.3	Constraints . . . . .	18
<b>7</b>	<b>Storing Structs That Have Struct List Fields</b>	<b>21</b>
7.1	Declaring Structs That Have Struct List Fields . . . . .	21
7.2	Writing Structs That Have Struct List Fields . . . . .	22
<b>8</b>	<b>Miscellaneous</b>	<b>23</b>
8.1	Adding File Metadata . . . . .	23
<b>9</b>	<b>Examples</b>	<b>25</b>
<b>10</b>	<b>Building parquet-writer</b>	<b>27</b>
10.1	Installation . . . . .	27
10.2	Installing Apache Arrow and Parquet . . . . .	27
<b>11</b>	<b>Easily declare and write Parquet files</b>	<b>29</b>
11.1	The Basics . . . . .	29
<b>12</b>	<b>Indices and tables</b>	<b>31</b>



`parquet-writer` is a C++ library that allows for easily writing Parquet files containing columns of (mostly) whatever type and structure you wish.



## **INTRODUCTION**

The goal of the `parquet-writer` library is to make it as easy as possible to write Parquet files containing data structures that are easily declared.

Users only need to declare the layout and data types associated with columns in their Parquet files following a simple JSON schema. With this information, the `parquet-writer` library will know the precise pathways needed to be able to write the user-provided data to the correct data columns with the correct structure without the user having to know any of the details of the C++ API of Apache Arrow or Apache Parquet.





## STORING BASIC VALUE TYPES

`parquet-writer` currently has support for storing boolean and numeric data types.

The following table describes the supported value types for data to be written to an output Parquet file, with the `parquet-writer` name that would be used in the JSON file providing the layout declaration for `parquet-writer`.

Value Types	<code>parquet-writer</code> name
Boolean	<code>bool</code>
Signed Integers	<code>int8</code> , <code>int16</code> , <code>int32</code> , <code>int64</code>
Unsigned Integers	<code>uint8</code> , <code>uint16</code> , <code>uint32</code> , <code>uint64</code>
Floating Point	<code>float</code> (32-bit precision), <code>double</code> (64-bit precision)

In addition to writing flat data columns of these basic value types, `parquet-writer` supports writing data columns that are nested data structures composed of fields whose data is comprised of these basic value types. More specifically, `parquet-writer` supports:

- 1, 2, and 3 dimensional lists of these value types
- Struct data types having any number of named fields (like a C/C++ `struct`)
- 1, 2, and 3 dimensional lists of struct data type

More information on how to declare and write Parquet files containing these nested structures is contained in later sections.

### 2.1 Declaring Columns of Basic Value Types

Declaring a column layout for storing values of the basic data types above is done using JSON as follows:

```
{
  "fields": [
    {"name": "column0", "type": "float"},
    {"name": "column1", "type": "int32"}
  ]
}
```

That is, one must specify a `fields` array containing JSON objects of the form:

```
{"name": "<string>", "type": "<value-type>"}
```

where the `name` field can be any arbitrary string. The `type` field must be one of the `parquet-writer` names for the supported basic value types appearing in the second column in the [table above](#).

Each element in the top-level `fields` array in a given JSON layout configuration will have a one-to-one correspondence with a data column appearing in the output Parquet file.

## 2.2 Writing Columns of Basic Value Types

Assuming we have the *file layout from above*, one would simply need to have variables of the corresponding C++ type to provide to the `Writer` class' `fill` function, along with the name of the column to which you want to write the data:

```
#include "parquet_writer.h"
...
float field0_data = 42.5;
int32_t field1_data = 42;
...
writer.fill("column0", field0_data);
writer.fill("column1", field1_data);
...
writer.end_row();
```

Note that the order in which the columns are filled is not important. One could also do the filling in this order:

```
writer.fill("column1", field1_data);
writer.fill("column0", field0_data);
```

## STORING LISTS OF BASIC VALUE TYPES

Storing one, two, and three dimensional lists whose elements are any of the *basic value types* is supported by `parquet-writer`. The table below describes the naming convention for the list type columns:

List Type	parquet-writer name
One dimensional list	<code>list1d</code>
Two dimensional list	<code>list2d</code>
Three dimensional list	<code>list3d</code>

### 3.1 Declaring List Type Columns

Declaring columns whose data type is a list of the basic value types is done using JSON as follows:

```
{
  "fields": [
    {"name": "column0", "type": "list1d", "contains": {"type": "float"}},
    {"name": "column1", "type": "list2d", "contains": {"type": "uint32"}},
    {"name": "column2", "type": "list3d", "contains": {"type": "double"}}
  ]
}
```

As can be seen in the above, declaring list types for output data columns requires an additional `contains` object in the JSON declaration of the column. The `contains` object defines the data type to be stored in the output, variable-lengthed list.

### 3.2 Writing List Type Columns

Writing to list type columns is done by using instances of `std::vector` containing the C++ type associated with the storage type declared in the `contains` field of the list type column.

For example, taking the layout declaration from the previous section:

```
// data for the 1D list column:
// {"name": "column0", "type": "list1d", "contains": {"type": "float"}}
std::vector<float> column0_data{1.2, 2.3, 3.4, 4.3};

// data for the 2D list column:
// {"name": "column1", "type": "list2d", "contains": {"type": "uint32"}}
```

(continues on next page)

(continued from previous page)

```
std::vector<std::vector<uint32_t>> column1_data{
    {1}, {2, 2}, {3, 3, 3}
};

// data for the 3D list column:
// {"name": "column2", "type": "list3d", "contains": {"type": "double"}}
std::vector<std::vector<std::vector<double>>> column2_data{
    { {1.1}, {2.2, 2.2}, {3.3, 3.3, 3.3} },
    { {3.1, 3.1, 3.1}, {2.2, 2.2}, {1.1} }
};

// fill using the usual "fill" method
writer.fill("column0", column0_data);
writer.fill("column1", column1_data);
writer.fill("column2", column2_data);
```

Columns of list type can be variably lengthed. That is, rows of list type columns do not all have to have the same number of contained elements (no padding is necessary). Indeed, one row of a given list column can have many elements while the next row is empty. For example:

```
// fill a row's "column0" field with length 3 1D list
std::vector<float> column0_data{1.2, 2.3, 3.4};
writer.fill("column0", column0_data);
writer.end_row();

// fill a second row's "column0" field with a length 0 1D list
column0_data.clear();
writer.fill("column0", column_data);
writer.end_row();
```

## STORING STRUCT TYPE COLUMNS

Storing complex data structures with any number of named fields of possibly different data types (think: C/C++ struct) is possible in `parquet-writer`.

### 4.1 Declaring Struct Type Columns

Declaring columns containing struct typed data is done via the `struct` type specifier.

For example, a struct-typed column with three named fields `field0`, `field1`, and `field2` with data types `int32`, `float`, and `list1d[float]`, respectively, is done as follows:

```
{
  "fields": [
    {
      "name": "struct_column", "type": "struct",
      "fields": [
        {"name": "field0", "type": "int32"},
        {"name": "field1", "type": "float"},
        {"name": "field2", "type": "list1d", "contains": {"type": "float"}}
      ]
    }
  ]
}
```

As can be seen, columns of type `struct` are declared with an additional `fields` array that contains an array of objects of the usual `{"name": ..., "type": ...}` form. The additional `fields` array describes each of the named fields of the data structure to be stored in the output Parquet file.

### 4.2 Writing Struct Type Columns

There are two convenience types that are used for writing data to columns with type `struct`:

1. `parquetwriter::field_map_t`
2. `parquetwriter::field_buffer_t`

The `field_map_t` type is an alias for `std::map<std::string, parquetwriter::value_t>`, where `parquetwriter::value_t` refers to an instance of any of the *basic value types*. The `field_map_t` type allows users to fill struct type columns without worrying about the order of the struct's fields as declared in the JSON layout.

The `field_buffer_t` type is an alias for `std::vector<parquetwriter::value_t>`.

**Warning:** When using the `field_buffer_t` type to write to struct type columns, the user must provide each of the struct's field data in the order that the named fields appear in the JSON layout for the struct.

### 4.2.1 Using `field_map_t`

An example of filling the three-field struct `my_struct` declared in the *previous section* would be as follows:

```
namespace pw = parquetwriter;

// generate the data for each of the struct's fields
int32_t field0_data{42};
float field1_data{42.42};
std::vector<float> field2_data{42.0, 42.1, 42.2};

// create the mapping between column name and data value to be stored
pw::field_map_t my_struct_data{
    {"field0", field0_data},
    {"field1", field1_data},
    {"field2", field2_data}
};

// call "fill" as usual
writer.fill("my_struct", my_struct_data);
```

Note that since the `field_map_t` convenience type is an alias of `std::map`, the ordering of the column names (the keys of the `std::map`) does not matter. The following instantiation of the `field_map_t` would lead to the same output written to file as the above:

```
pw::field_map_t my_struct_data{
    {"field2", field2_data},
    {"field1", field1_data},
    {"field0", field0_data}
};
```

---

**Note:** When using the `field_map_t` approach to write to a struct type column, the call to `fill` leads to an internal check against the loaded layout for the specific struct-type column and constructs an intermediate `field_buffer_t` with the data values in the order matching that of the loaded layout.

---

### 4.2.2 Using field\_buffer\_t

The alternative approach using `field_buffer_t` to write the struct `my_struct` from *above* would be as follows:

```
namespace pw = parquetwriter;

// generate the data for each of the struct's fields
int32_t field0_data{42};
float field1_data{42.42};
std::vector<float> field2_data{42.0, 42.1, 42.2};

// create the data buffer for the given instance of "my_struct"
pw::field_buffer_t my_struct_data{field0_data, field1_data, field2_data};

// call "fill" as usual
writer.fill("my_struct", my_struct_data);
```

Since `field_buffer_t` is an alias of `std::vector`, you can also do:

```
pw::field_buffer_t my_struct_data;
my_struct_data.push_back(field0_data);
my_struct_data.push_back(field1_data);
my_struct_data.push_back(field2_data);
```

As mentioned above (and as the name implies) the data provided to an instance of `field_buffer_t` must be provided in the order matching that of the fields in the user-provided layout for the Parquet file.

For example, consider the layout for the following struct-type column:

```
{
  "fields": [
    {
      "name": "another_struct", "type": "struct",
      "fields": [
        {"name": "another_field0", "type": "float"},
        {"name": "another_field1", "type": "float"}
      ]
    }
  ]
}
```

The above layout specifies a single struct-type column named `another_struct`, with two named fields `another_field0` and `another_field1`. Both of these fields are of type `float`. In using the `field_buffer_t` approach to writing to the struct, users must be careful to provide the data in the correct order. Otherwise inconsistencies in the stored data will emerge.

For example, the below would not be caught as an invalid column write since the types of the provided data match those specified in the layout but the intended meaning of the data is lost since the data for `another_field1` will be written to the column for `another_field0` and vice versa:

```
float another_field0_data{42.42};
float another_field1_data{84.84};

// incorrect order!
pw::field_buffer_t another_struct_data{another_field1_data, another_field0_data};
```

Instead of the correct ordering:

```
// correct order!  
pw::field_buffer_t another_struct_data{another_field0_data, another_field1_data};
```



## STORING LISTS OF STRUCTS

Storing lists containing elements that are of type `struct` is supported.

### 5.1 Declaring Lists of Structs

Declaring columns that contain lists whose elements are of type `struct` is done by composing the *list type* and *struct type* declarations.

For example, the following declares a one-dimensional list containing struct-type elements that have three named fields:

```
{
  "fields": [
    {
      "name": "structlist", "type": "list1d",
      "contains": { "type": "struct",
        "fields": [
          {"name": "field0", "type": "float"},
          {"name": "field1", "type": "int32"},
          {"name": "field2", "type": "list1d", "contains": {"type": "float"}}
        ]
      }
    }
  ]
}
```

To declare two- or three-dimensional lists, one would simply swap the `type` field for the `structlist` column from `list1d` to either `list2d` or `list3d`.

### 5.2 Writing Lists of Structs

Writing to columns that contain lists of struct-type elements is done by building up instances of `std::vector` containing elements of either *field\_map\_t* or *field\_buffer\_t*.

For example, writing a one-dimensional list containing the three-field struct elements described above would be done as follows:

```
namespace pw = parquetwriter;

// 1D vector of struct elements
```

(continues on next page)

(continued from previous page)

```

std::vector<pw::field_map_t> structlist_data;

// fill the 1D vector with struct data elements
for(...) {
    // generate struct field data
    float field0_data{42.42};
    int32_t field1_data{42};
    std::vector<float> field2_data{42.0, 42.1, 42.2};

    // create the struct element
    pw::field_map_t struct_data{
        {"field0", field0_data},
        {"field1", field1_data},
        {"field2", field2_data}
    };

    // append to the struct list
    structlist_data.push_back(struct_data);
}

// call "fill" as usual
writer.fill("structlist", structlist_data);

```

The two-dimensional case:

```

namespace pw = parquetwriter;

// 2D vector of struct elements
std::vector<std::vector<pw::field_map_t>> structlist_data;

// fill the 2D vector with struct data elements
for(...) {
    std::vector<pw::field_map_t> inner_structlist_data;
    for(...) {
        pw::field_map_t struct_data{
            {"field0", field0_data},
            {"field1", field1_data},
            {"field2", field2_data}
        };
        inner_structlist_data.push_back(struct_data);
    }
    structlist_data.push_back(inner_structlist_data);
}

// call "fill" as usual
writer.fill("structlist", structlist_data);

```

And the three-dimensional case:

```

namespace pw = parquetwriter;

// 3D vector of struct elements

```

(continues on next page)

(continued from previous page)

```

std::vector<std::vector<std::vector<pw::field_map_t>>> structlist_data;

// fill the 3D vector with struct data elements
for(...) {
    std::vector<std::vector<pw::field_map_t>> inner_structlist_data;
    for(...) {
        std::vector<pw::field_map_t> inner_inner_structlist_data;
        for(...) {
            pw::field_map_t struct_data{
                {"field0", field0_data},
                {"field1", field1_data},
                {"field2", field2_data}
            };
            inner_inner_structlist_data.push_back(struct_data);
        }
        inner_structlist_data.push_back(inner_inner_structlist_data);
    }
    structlist_data.push_back(inner_structlist_data);
}

// call "fill" as usual
writer.fill("structlist", structlist_data);

```

## 5.3 Constraints

**Warning:** The struct type elements contained in lists of struct cannot themselves contain fields that are of type struct.

For simplicity, any list type data column whose elements are of type struct, cannot contain struct type elements that have fields that are themselves of type struct.

For example, the following Parquet file layout declaration is not allowed:

```

{
  "fields": [
    {
      "name": "structlist",
      "type": "listId",
      "contains": {
        "type": "struct",
        "fields": [
          {"name": "field0", "type": "float"},
          {
            "name": "inner_struct", "type": "struct",
            "fields": [{"name": "inner_field0", "type": "float"}]
          }
        ]
      }
    }
  ]
}

```

(continues on next page)

(continued from previous page)

```
]
}
```

---

**Note:** The above `list1d` type column is not allowed since its `struct` typed elements are declared as having an internal `struct` typed column (the field named `inner_struct`).

---

## STORING STRUCTS THAT HAVE STRUCT FIELDS

Storing struct-type columns that contain fields that are themselves of type `struct` is supported.

### 6.1 Declaring Structs that have Struct Fields

Specifying a struct-type column that contains a named field that is itself of type `struct` (with its own additional set of named fields) is done as follows:

```
{
  "fields": [
    {
      "name": "outer_struct", "type": "struct",
      "fields": [
        {"name": "outer_field0", "type": "float"},
        {
          "name": "inner_struct", "type": "struct",
          "fields": [
            {"name": "inner_field0", "type": "float"},
            {"name": "inner_field1", "type": "int32"},
            {"name": "inner_field2", "type": "list1d", "contains": {"type": "float"}}
          ]
        }
      ]
    }
  ]
}
```

The above describes a struct-type column named `outer_struct` which has two named fields `outer_field0` and `inner_struct`.

The named field `outer_field0` is a field having a basic value type `float`.

The named field `inner_struct` is a field of type `struct` that has three named fields `inner_field0`, `inner_field1`, and `inner_field2` of type `float`, `int32`, and `list1d[float]`, respectively.

## 6.2 Writing Structs with Struct Fields

Writing to struct-type columns having fields that are of type struct is done as follows (assuming the layout declaration from the previous section):

```
namespace pw = parquetwriter;

// data for the non-struct fields of the struct "outer_struct"
float outer_field0_data{42.0};
pw::field_map_t outer_struct_data{
    {"outer_field0", outer_field0_data}
};

// data for the non-struct fields of the struct "inner_struct"
float inner_field0_data{42.0};
int32_t inner_field1_data{42};
std::vector<float> inner_field2_data{42.0, 42.1, 42.2};
pw::field_map_t inner_struct{
    {"inner_field0", inner_field0_data},
    {"inner_field1", inner_field1_data},
    {"inner_field2", inner_field2_data}
};

// call "fill" for each struct
writer.fill("outer_struct", outer_struct_data);
writer.fill("outer_struct.inner_struct", inner_struct_data);
```

As can be seen, for each level of nesting of struct-typed columns/fields, one provides a `field_map_t` (or `field_buffer_t`) instance containing the data for all fields that are not of type struct.

Internal named fields that are of type struct are written to using the dot (.) notation in the call to `fill`, with the convention `<outer_struct_name>.<inner_struct_name>` as seen in the above: `writer.fill("outer_struct.inner_struct", ...)`.

## 6.3 Constraints

**Warning:** A column of type struct cannot itself contain named fields of type struct that have fields of type struct.

For simplicity, any named field of type struct of a struct-type column is not itself allowed to have a field of type struct.

For example, the following Parquet file layout declaration is not allowed:

```
{
  "fields": [
    {
      "name": "struct0", "type": "struct",
      "fields": [
        {"name": "field0", "type": "float"},
        {"name": "struct1", "type": "struct",
```

(continues on next page)

(continued from previous page)

```
    "fields": [  
      {"name": "inner_field0", "type": "float"},  
      {"name": "struct2", "type": "struct",  
        "fields": [  
          {"name": "inner_inner_field0", "type": "float"}  
        ]  
      }  
    ]  
  }  
]  
}
```

---

**Note:** The above is not allowed since the inner struct `struct1` contains a struct-typed field (the field named `struct2`).

---





## STORING STRUCTS THAT HAVE STRUCT LIST FIELDS

Storing struct-typed columns that have named fields that are lists containing elements of type `struct` is supported.

### 7.1 Declaring Structs That Have Struct List Fields

Declaring a struct-typed column with a field that is a list of elements of type `struct` is done as follows:

```
{
  "fields": [
    {"name": "my_struct", "type": "struct",
     "fields": [
       {"name": "field0", "type": "float"},
       {"name": "structlist", "type": "list1d",
        "contains": {
          "type": "struct",
          "fields": [
            {"name": "foo", "type": "float"},
            {"name": "bar", "type": "int32"}
          ]
        }
      ]
    }
  ]
}
```

The above describes a struct-typed column named `my_struct` with two named fields `field0` and `structlist`.

The field `field0` holds the basic value type `float`. The field `structlist` is a one-dimensional list of struct-type elements each having two named fields `foo` and `bar`.

The above pattern works for two- and three-dimensional lists of struct-typed elements simply by swapping out the `list1d` type for `list2d` or `list3d` where appropriate.

**Warning:** The *struct list constraints* still hold for the case when the struct list is associated with a named field of a struct typed column.

## 7.2 Writing Structs That Have Struct List Fields

Writing to struct-type columns that contain fields that are lists of struct-type elements is done similarly to the case of writing to struct-type columns containing struct-typed fields by using the dot (.) notation for nested struct-types.

For example, assuming the layout declared in the previous section:

```
namespace pw = parquetwriter;

// data for the non-struct fields of the struct "my_struct"
float field0_data{42.0};
pw::field_map_t my_struct_data{
    {"field0", field0_data}
};

// data for the struct-list field named "structlist"
std::vector<pw::field_map_t> structlist_data;
for(...) {
    // generate struct field data
    float foo_data{42.42};
    int32_t bar_data{42};

    // create the struct element
    pw::field_map_t struct_data{
        {"foo", foo_data},
        {"bar", bar_data}
    };
    structlist_data.push_back(struct_data);
}

// call "fill" using dot notation for nested struct types
writer.fill("my_struct", my_struct_data);
writer.fill("my_struct.structlist", structlist_data);
```

## MISCELLANEOUS

### 8.1 Adding File Metadata

Arbitrary metadata can be stored in JSON format and added to the output Parquet file using the `parquet-writer` library.

This is done by first creating a JSON object containing whatever arbitrary information you wish and providing it to your `parquetwriter::Writer` instance. Suppose we have the file `metadata.json` containing the following JSON:

```
{
  "dataset_name": "example_dataset",
  "foo": "bar",
  "creation_date": "2021/10/11",
  "bar": {
    "faz": "baz"
  }
}
```

We would pass this to our writer instance as follows:

```
std::ifstream metadata_file("metadata.json");
writer.set_metadata(metadata_file);
```

The above stores the contained JSON to the Parquet file as an instance of `key:value` pairs.

The example Python script `dump-metadata.py` (requires `pyarrow`) that extracts the metadata stored by `parquet-writer` shows how to extract the metadata and can be run as follows:

```
$ python examples/python/dump-metadata.py <file>
```

where `<file>` is a Parquet file written by `parquet-writer`.

Running `dump-metadata.py` on a file with the metadata from above would look like:

```
$ python examples/python/dump-metadata.py example_file.parquet
{
  "dataset_name": "example_dataset",
  "foo": "bar",
  "creation_date": "2021/10/11",
  "bar": {
    "faz": "baz"
  }
}
```



## **EXAMPLES**

Concrete examples for how to write any of the supported data types to a Parquet file are found in the [examples/cpp](#) directory.

The examples get built during the build of the `parquet-writer` library.



## BUILDING PARQUET-WRITER

### 10.1 Installation

Below are steps to build the `parquet-writer` library using CMake on various architectures.

It is assumed that you have *installed Apache Arrow and Parquet* before following the procedures below.

Upon a successful build, the shared library `parquet-writer` will be located under `build/lib`.

#### 10.1.1 macOS

```
mkdir build && cd build
cmake -DARROW_PATH=$(brew --prefix apache-arrow) ..
make
```

#### 10.1.2 Debian/Ubuntu

```
mkdir build && cd build
cmake -DCMAKE_MODULE_PATH=$(find /usr/lib -type d -name arrow) ..
make
```

### 10.2 Installing Apache Arrow and Parquet

The `parquet-writer` library naturally depends on the Apache Arrow and Apache Parquet libraries. Below are reproduced the necessary steps to install the dependencies on various architectures. See the [official documentation](#) for further details.

## 10.2.1 macOS

```
brew install apache-arrow
```

## 10.2.2 Debian/Ubuntu

```
apt install -y -V lsb-release wget pkg-config
wget https://apache.jfrog.io/artifactory/arrow/$(lsb_release --id --short | tr 'A-Z' 'a-z
↪')/apache-arrow-apt-source-latest-$(lsb_release --codename --short).deb
apt-get install -y ./apache-arrow-apt-source-latest-$(lsb_release --codename --short).deb
apt-get update -y
apt-get install -y libarrow-dev=5.0.0-1 libparquet-dev=5.0.0-1
```



## EASILY DECLARE AND WRITE PARQUET FILES

The idea is for `parquet-writer` to make it simple to both specify the desired layout of a Parquet file (i.e. the number and structure of data columns) and to subsequently write your data to that file.

In summary, `parquet-writer` provides support for:

- Specifying the layout of Parquet files using JSON
- Storing numeric and boolean data types to output Parquet files
- Storing struct objects (think: C/C++ structs) having any number of arbitrarily-typed fields
- Storing 1, 2, and 3 dimensional lists of the supported data types
- A simple interface for writing the supported data types to Parquet files

### 11.1 The Basics

`parquet-writer` provides users with the `parquetwriter::Writer` class, which they provide with a JSON object specifying the desired “layout” of their Parquet files and then fill accordingly.

An example JSON layout, stored in the file `layout.json`, could be:

```
{
  "fields": [
    {"name": "foo", "type": "float"},
    {"name": "bar", "type": "uint32"},
    {"name": "baz", "type": "list1d", "contains": {"type": "float"}}
  ]
}
```

The above describes an output Parquet file containing three data columns named `foo`, `bar`, and `baz` which contain data of types `float` (32-bit precision float), `uint32` (32-bit unsigned integer), and `list[float]` (variable-lengthed 1-dimensional list of elements of type `float`), respectively.

The basics of initializing a `parquetwriter::Writer` instance with the above layout, writing some values to a single row, and storing the output is below:

```
#include "parquet_writer.h"

namespace pw = parquetwriter;
pw::Writer writer;
std::ofstream layout_file("layout.json"); // file containing JSON layout spec
writer.set_layout(layout_file);
```

(continues on next page)

(continued from previous page)

```
writer.set_dataset("my_dataset"); // must give a name to the output
writer.initialize();

// generate some data for each of the columns
float foo_data{42.0};
uint32_t bar_data{42};
std::vector<float> baz_data{42.0, 42.1, 42.2, 42.3};

// call "fill" for each of the columns, giving the associated data
writer.fill("foo", foo_data);
writer.fill("bar", bar_data);
writer.fill("baz", baz_data);

// signal the end of a row
writer.end_row();

// call "finish" when done writing to the file
writer.finish();
```

The above would generate an output file called `my_dataset.parquet`. We can use `parquet-tools` to quickly dump the contents of the Parquet file:

```
$ parquet-tools show my_dataset.parquet
+-----+
| foo  | bar  | baz                                |
+-----+
| 42.0 | 42   | [42.0, 42.1, 42.2, 42.3]         |
+-----+
```

## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`