# fastparquet Documentation

*Release 0.2.1*

**Continuum Analytics**

**Jan 04, 2019**

# Contents

A Python interface to the Parquet file format.

# CHAPTER 1

# Introduction

The Parquet format is a common binary data store, used particularly in the Hadoop/big-data sphere. It provides several advantages relevant to big-data processing:

- columnar storage, only read the data of interest
- efficient binary packing
- choice of compression algorithms and encoding
- split data into files, allowing for parallel processing
- range of logical types
- statistics stored in metadata allow for skipping unneeded chunks
- data partitioning using the directory structure

Since it was developed as part of the Hadoop ecosystem, Parquet's reference implementation is written in Java. This package aims to provide a performant library to read and write Parquet files from Python, without any need for a Python-Java bridge. This will make the Parquet format an ideal storage mechanism for Python-based big data workflows.

The tabular nature of Parquet is a good fit for the Pandas data-frame objects, and we exclusively deal with data-frame<->Parquet.

CHAPTER 2

# Highlights

The original outline plan for this project can be found here

Briefly, some features of interest:

- read and write Parquet files, in single- or multiple-file format. The latter is commonly found in hive/Spark usage.
- choice of compression per-column and various optimized encoding schemes; ability to choose row divisions and partitioning on write.
- acceleration of both reading and writing using numba
- ability to read and write to arbitrary file-like objects, allowing interoperability with s3fs, hdfs3, adlfs and possibly others.
- can be called from dask, to enable parallel reading and writing with Parquet files, possibly distributed across a cluster.

CHAPTER 3

# Caveats, Known Issues

Not all parts of the Parquet-format have been implemented yet or tested. fastparquet is, however, capable of reading all the data files from the parquet-compatibility project. Some encoding mechanisms in Parquet are rare, and may be implemented on request - please post an issue.

Some deeply-nested columns will not be readable, e.g., lists of lists.

Not all output options will be compatible with every other Parquet framework, which each implement only a subset of the standard, see the usage notes.

A list of current issues can be found here.

CHAPTER 4

# Relation to Other Projects

- parquet-python is the original

pure-Python Parquet quick-look utility which was the inspiration for fastparquet.

- parquet-cpp is a low-level C++

implementation of the Parquet format which can be called from Python using Apache Arrow bindings. Future collaboration with parquet-cpp is possible, in the medium term, and that perhaps their low-level routines will replace some functions in fastparquet or that high-level logic in fastparquet will be migrated to C++.

- PySpark, a Python API to the Spark

engine, interfaces Python commands with a Java/Scala execution core, and thereby gives Python programmers access to the Parquet format. fastparquet has no defined relationship to PySpark, but can provide an alternative path to providing data to Spark or reading data produced by Spark without invoking a PySpark client or interacting directly with the scheduler.

- fastparquet lives within the dask ecosystem, and

although it is useful by itself, it is designed to work well with dask for parallel execution, as well as related libraries such as s3fs for pythonic access to Amazon S3.

CHAPTER 5

# Index

## 5.1 Installation

### 5.1.1 Requirements

Required:

- python 3
- numba
- numpy
- pandas
- pytest

Optional (compression algorithms; gzip is always available):

- python-snappy
- python-lzo
- brotli

### 5.1.2 Installation

Install using conda:

```
conda install -c conda-forge fastparquet
```

install from pypi:

```
pip install fastparquet
```

or install latest version from github:

```
pip install git+https://github.com/dask/fastparquet
```

For the pip methods, numba must have been previously installed (using conda, or from source).

## 5.2 Quickstart

### 5.2.1 Reading

To open and read the contents of a Parquet file:

```
from fastparquet import ParquetFile
pf = ParquetFile('myfile.parq')
df = pf.to_pandas()
```

The Pandas data-frame, `df` will contain all columns in the target file, and all row-groups concatenated together. If the data is a multi-file collection, such as generated by hadoop, the filename to supply is either the directory name, or the "_metadata" file contained therein - these are handled transparently.

One may wish to investigate the meta-data associated with the data before loading, for example, to choose which row-groups and columns to load. The properties `columns`, `count`, `dtypes` and `statistics` are available to assist with this. In addition, if the data is in a hierarchical directory-partitioned structure, then the property `cats` specifies the possible values of each partitioning field.

You may specify which columns to load, which of those to keep as categoricals (if the data uses dictionary encoding), and which column to use as the pandas index. By selecting columns, we only access parts of the file, and efficiently skip columns that are not of interest.

```
df2 = pf.to_pandas(['col1', 'col2'], categories=['col1'])
# or
df2 = pf.to_pandas(['col1', 'col2'], categories={'col1': 12})
```

where the second version specifies the number of expected labels for that column.

Furthermore, row-groups can be skipped by providing a list of filters. There is no need to return the filtering column as a column in the data-frame. Note that only row-groups that have no data at all meeting the specified requirements will be skipped.

```
df3 = pf.to_pandas(['col1', 'col2'], filters=[('col3', 'in', [1, 2, 3, 4])])
```

### 5.2.2 Writing

To create a single Parquet file from a dataframe:

```
from fastparquet import write
write('outfile.parq', df)
```

The function `write` provides a number of options. The default is to produce a single output file with a row-groups up to 50M rows, with plain encoding and no compression. The performance will therefore be similar to simple binary packing such as `numpy.save` for numerical columns.

Further options that may be of interest are:

- the compression algorithms (typically "snappy", for fast, but not too space-efficient), which can vary by column

- the row-group splits to apply, which may lead to efficiencies on loading, if some row-groups can be skipped. Statistics (min/max) are calculated for each column in each row-group on the fly.
- multi-file saving can be enabled with the `file_scheme` keyword: hive-style output is a directory with a single metadata file and several data-files.
- options has_nulls, fixed_text and write_index affect efficiency see the api docs.

```
write('outfile2.parq', df, row_group_offsets=[0, 10000, 20000],
      compression='GZIP', file_scheme='hive')
```

## 5.3 Usage Notes

Some additional information to bear in mind when using fastparquet, in no particular order. Much of what follows has implications for writing parquet files that are compatible with other parquet implementations, versus performance when writing data for reading back with fastparquet.

Whilst we aim to make the package simple to use, some choices on the part of the user may effect performance and data consistency.

### 5.3.1 Categoricals

When writing a data-frame with a column of pandas type `Category`, the data will be encoded using Parquet "dictionary encoding". This stores all the possible values of the column (typically strings) separately, and the index corresponding to each value as a data set of integers. If there is a significant performance gain to be made, such as long labels, but low cardinality, users are suggested to turn their object columns into the category type:

```
df[col] = df[col].astype('category')
```

Fastparquet will automatically use metadata information to load such columns as categorical *if* the data was written by fastparquet.

To efficiently load a column as a categorical type for data from other parquet frameworks, include it in the optional keyword parameter `categories`; however it must be encoded as dictionary throughout the dataset.

```
pf = ParquetFile('input.parq')
df = pf.to_pandas(categories={'cat': 12})
```

Where we provide a hint that the column `cat` has up to 12 possible values. `categories` can also take a list, in which case up to 32767 (2**15 - 1) labels are assumed. Columns that are encoded as dictionary but not included in `categories` will be de-referenced on load which is potentially expensive.

Note that before loading, it is not possible to know whether the above condition will be met, so the `dtypes` attribute of a `ParquetFile` will show the data type appropriate for the values of column, unless the data originates with fastparquet.

### 5.3.2 Byte Arrays

Often, information in a column can be encoded in a small number of characters, perhaps a single character. Variable-length byte arrays are also slow and inefficient, however, since the length of each value needs to be stored.

Fixed-length byte arrays provide the best of both, and will probably be the most efficient storage where the values are 1-4 bytes long, especially if the cardinality is relatively high for dictionary encoding. To automatically convert string values to fixed-length when writing, use the `fixed_text` optional keyword, with a predetermined length.

```
write('out.parq', df, fixed_text={'char_code': 1})
```

Such an encoding will be the fastest to read, especially if the values are bytes type, as opposed to UTF8 strings. The values will be converted back to objects upon loading.

Fixed-length byte arrays are not supported by Spark, so files written using this may not be portable.

### 5.3.3 Short-type Integers

Types like 1-byte ints (signed or unsigned) are stored using bitpacking for optimized space and speed. Unfortunately, Spark is known not to be able to handle these types. If you want to generate files for reading by Spark, be sure to transform integer columns to a minimum of 4 bytes (numpy `int32` or `uint32`) before saving.

### 5.3.4 Nulls

In pandas, there is no internal representation difference between NULL (no value) and NaN (not a valid number) for float, time and category columns. Whether to enocde these values using parquet NULL or the "sentinel" values is a choice for the user. The parquet framework that will read the data will likely treat NULL and NaN differently (e.g., in in Spark). In the typical case of tabular data (as opposed to strict numerics), users often mean the NULL semantics, and so should write NULLs information. Furthermore, it is typical for some parquet frameworks to define all columns as optional, whether or not they are intended to hold any missing data, to allow for possible mutation of the schema when appending partitions later.

Since there is some cost associated with reading and writing NULLs information, fastparquet provides the `has_nulls` keyword when writing to specify how to handle NULLs. In the case that a column has no NULLs, including NULLs information will not produce a great performance hit on reading, and only a slight extra time upon writing, while determining that there are zero NULL values.

The following cases are allowed for `has_nulls`:

- True: all columns become optional, and NaNs are always stored as NULL. This is the best option for compatibility. This is the default.
- False: all columns become required, and any NaNs are stored as NaN; if there are any fields which cannot store such sentinel values (e.g,. string), but do contain None, there will be an error.
- 'infer': only object columns will become optional, since float, time, and category columns can store sentinel values, and pandas int columns cannot contain any NaNs. This is the best-performing option if the data will only be read by fastparquet.
- list of strings: the named columns will be optional, others required (no NULLs)

This value can be stored in float and time fields, and will be read back such that the original data is recovered. They are not, however, the same thing as missing values, and if querying the resultant files using other frameworks, this should be born in mind. With `has_nulls=None` (the default) on writing, float, time and category fields will not write separate NULLs information, and the metadata will give num_nulls=0.

### 5.3.5 Data Types

There is fairly good correspondence between pandas data-types and Parquet simple and logical data types. The types documentation gives details of the implementation spec.

A couple of caveats should be noted:

- fastparquet will not write any Decimal columns, only float, and when reading such columns, the output will also be float, with potential machine-precision errors;

- only UTF8 encoding for text is automatically handled, although arbitrary byte strings can be written as raw bytes type;
- the time types have microsecond accuracy, whereas pandas time types normally are nanosecond accuracy;
- all times are stored as UTC, and timezone information will be lost;
- complex numbers must have their real and imaginary parts stored as two separate float columns.

### 5.3.6 Spark Timestamps

Fastparquet can read and write int96-style timestamps, as typically found in Apache Spark and Map-Reduce output.

Currently, int96-style timestamps are the only known use of the int96 type without an explicit schema-level converted type assignment. They will be automatically converted to times upon loading.

Similarly on writing, the `times` keyword controls the encoding of timestamp columns: "int64" is the default and faster option, producing parquet standard compliant data, but "int96" is required to write data which is compatible with Spark.

### 5.3.7 Reading Nested Schema

Fastparquet can read nested schemas. The principal mechamism is *flattening*, whereby parquet schema struct columns become top-level columns. For instance, if a schema looks like

```
root
| - visitor: OPTIONAL
  | - ip: BYTE_ARRAY, UTF8, OPTIONAL
    - network_id: BYTE_ARRAY, UTF8, OPTIONAL
```

then the `ParquetFile` will include entries "visitor.ip" and "visitor.network_id" in its `columns`, and these will become ordinary Pandas columns.

Fastparquet also handles some parquet LIST and MAP types. For instance, the schema may include

```
| - tags: LIST, OPTIONAL
    - list: REPEATED
       - element: BYTE_ARRAY, UTF8, OPTIONAL
```

In this case, `columns` would include an entry "tags", which evaluates to an object column containing lists of strings. Reading such columns will be relatively slow. If the 'element' type is anything other than a primitive type, i.e., a struct, map or list, than fastparquet will not be able to read it, and the resulting column will either not be contained in the output, or contain only `None` values.

### 5.3.8 Partitions and row-groups

The Parquet format allows for partitioning the data by the values of some (low-cardinality) columns and by row sequence number. Both of these can be in operation at the same time, and, in situations where only certain sections of the data need to be loaded, can produce great performance benefits in combination with load filters.

Splitting on both row-groups and partitions can potentially result in many data-files and large metadata. It should be used sparingly, when partial selecting of the data is anticipated.

**Row groups**

The keyword parameter `row_group_offsets` allows control of the row sequence-wise splits in the data. For example, with the default value, each row group will contain 50 million rows. The exact index of the start of each

row-group can also be specified, which may be appropriate in the presence of a monotonic index: such as a time index might lead to the desire to have all the row-group boundaries coincide with year boundaries in the data.

**Partitions**

In the presence of some low-cardinality columns, it may be advantageous to split data data on the values of those columns. This is done by writing a directory structure with *key=value* names. Multiple partition columns can be chosen, leading to a multi-level directory tree.

Consider the following directory tree from this Spark example:

**table/**

**gender=male/**

**country=US/** data.parquet

**country=CN/** data.parquet

**gender=female/**

**country=US/** data.parquet

**country=CN/** data.parquet

Here the two partitioned fields are *gender* and *country*, each of which have two possible values, resulting in four datafiles. The corresponding columns are not stored in the data-files, but inferred on load, so space is saved, and if selecting based on these values, potentially some of the data need not be loaded at all.

If there were two row groups and the same partitions as above, each leaf directory would contain (up to) two files, for a total of eight. If a row-group happens to contain no data for one of the field value combinations, that data file is omitted.

### 5.3.9 Iteration

For data-sets too big to fit conveniently into memory, it is possible to iterate through the row-groups in a similar way to reading by chunks from CSV with pandas.

```
pf = ParquetFile('myfile.parq')
for df in pf.iter_row_groups():
    print(df.shape)
    # process sub-data-frame df
```

Thus only one row-group is in memory at a time. The same set of options are available as in `to_pandas` allowing, for instance, reading only specific columns, loading to categoricals or to ignore some row-groups using filtering.

To get the first row-group only, one would go:

```
first = next(iter(pf.iter_row_groups()))
```

### 5.3.10 Connection to Dask

Dask usage is still in development. Expect the features to lag behind those in fastparquet, and sometimes to become incompatible, if a change has been made in the one but not the other.

Dask provides a pandas-like dataframe interface to larger-than-memory and distributed datasets, as part of a general parallel computation engine. In this context, it allows the parallel loading and processing of the component pieces of a Parquet dataset across the cored of a CPU and/or the nodes of a distributed cluster.

Dask will provide two simple end-user functions:

- `dask.dataframe.read_parquet` with keyword options similar to `ParquetFile.to_pandas`. The URL parameter, however, can point to various filesystems, such as S3 or HDFS. Loading is *lazy*, only happening on demand.
- `dask.dataframe.DataFrame.to_parquet` with keyword options similar to `fastparquet.write`. One row-group/file will be generated for each division of the dataframe, or, if using partitioning, up to one row-group/file per division per partition combination.

## 5.4 API

| | |
|---|---|
| *ParquetFile*(fn[, verify, open_with, root, sep]) | The metadata of a parquet file or collection |
| *ParquetFile.to_pandas*([columns, categories, ...]) | Read data from parquet into a Pandas dataframe. |
| *ParquetFile.iter_row_groups*([columns, ...]) | Read data from parquet into a Pandas dataframe. |
| `ParquetFile.info` | Some metadata details |
| *write*(filename, data[, row_group_offsets, ...]) | Write Pandas DataFrame to filename as Parquet Format |

**class** fastparquet.**ParquetFile**(*fn*, *verify=False*, *open_with=<function default_open>*, *root=False*, *sep=None*)

The metadata of a parquet file or collection

Reads the metadata (row-groups and schema definition) and provides methods to extract the data from the files.

Note that when reading parquet files partitioned using directories (i.e. using the hive/drill scheme), an attempt is made to coerce the partition values to a number, datetime or timedelta. Fastparquet cannot read a hive/drill parquet file with partition names which coerce to the same value, such as "0.7" and ".7".

**Parameters**

**fn: path/URL string or list of paths** Location of the data. If a directory, will attempt to read a file "_metadata" within that directory. If a list of paths, will assume that they make up a single parquet data set. This parameter can also be any file-like object, in which case this must be a single-file dataset.

**verify: bool [False]** test file start/end byte markers

**open_with: function** With the signature *func(path, mode)*, returns a context which evaluated to a file open for reading. Defaults to the built-in *open*.

**root: str** If passing a list of files, the top directory of the data-set may be ambiguous for partitioning where the upmost field has only one value. Use this to specify the data'set root directory, if required.

**Attributes**

**cats: dict** Columns derived from hive/drill directory information, with known values for each column.

**categories: list** Columns marked as categorical in the extra metadata (meaning the data must have come from pandas).

**columns: list of str** The data columns available

**count: int** Total number of rows

**dtypes: dict** Expected output types for each column

**file_scheme: str** 'simple': all row groups are within the same file; 'hive': all row groups are in other files; 'mixed': row groups in this file and others too; 'empty': no row groups at all.

**info: dict** Combination of some of the other attributes

**key_value_metadata: list** Additional information about this data's origin, e.g., pandas description.

**row_groups: list** Thrift objects for each row group

**schema: schema.SchemaHelper** print this for a representation of the column structure

**selfmade: bool** If this file was created by fastparquet

**statistics: dict** Max/min/count of each column chunk

### Methods

| | |
|---|---|
| `filter_row_groups`(filters) | Select row groups using set of filters |
| `grab_cats`(columns[, row_group_index]) | Read dictionaries of first row_group |
| *`iter_row_groups`*([columns, categories, ...]) | Read data from parquet into a Pandas dataframe. |
| `read_row_group`(rg, columns, categories[, ...]) | Access row-group in a file and read some columns into a data-frame. |
| `read_row_group_file`(rg, columns, categories) | Open file for reading, and process it as a row-group |
| *`to_pandas`*([columns, categories, filters, index]) | Read data from parquet into a Pandas dataframe. |

| | |
|---|---|
| **check_categories** | |
| **pre_allocate** | |
| **row_group_filename** | |

`ParquetFile.`**`to_pandas`**(*columns=None*, *categories=None*, *filters=[]*, *index=None*)

Read data from parquet into a Pandas dataframe.

**Parameters**

**columns: list of names or 'None'** Column to load (see *ParquetFile.columns*). Any columns in the data not in this list will be ignored. If *None*, read all columns.

**categories: list, dict or 'None'** If a column is encoded using dictionary encoding in every row-group and its name is also in this list, it will generate a Pandas Category-type column, potentially saving memory and time. If a dict {col: int}, the value indicates the number of categories, so that the optimal data-dtype can be allocated. If `None`, will automatically set *if* the data was written from pandas.

**filters: list of tuples** To filter out (i.e., not read) some of the row-groups. (This is not row-level filtering) Filter syntax: [(column, op, val), ...], where op is [==, >, >=, <, <=, !=, in, not in]

**index: string or list of strings or False or None** Column(s) to assign to the (multi-)index. If None, index is inferred from the metadata (if this was originally pandas data); if the metadata does not exist or index is False, index is simple sequential integers.

**Returns**

**Pandas data-frame**

`ParquetFile.`**`iter_row_groups`**(*columns=None*, *categories=None*, *filters=[]*, *index=None*)

Read data from parquet into a Pandas dataframe.

**Parameters**

**columns: list of names or 'None'** Column to load (see *ParquetFile.columns*). Any columns in the data not in this list will be ignored. If *None*, read all columns.

**categories: list, dict or 'None'** If a column is encoded using dictionary encoding in every row-group and its name is also in this list, it will generate a Pandas Category-type column, potentially saving memory and time. If a dict {col: int}, the value indicates the number of categories, so that the optimal data-dtype can be allocated. If `None`, will automatically set *if* the data was written by fastparquet.

**filters: list of tuples** To filter out (i.e., not read) some of the row-groups. (This is not row-level filtering) Filter syntax: [(column, op, val), ...], where op is [==, >, >=, <, <=, !=, in, not in]

**index: string or list of strings or False or None** Column(s) to assign to the (multi-)index. If None, index is inferred from the metadata (if this was originally pandas data); if the metadata does not exist or index is False, index is simple sequential integers.

**assign: dict {cols: array}** Pre-allocated memory to write to. If None, will allocate memory here.

**Returns**

**Generator yielding one Pandas data-frame per row-group**

`fastparquet.`**`write`**(*filename*, *data*, *row_group_offsets=50000000*, *compression=None*, *file_scheme='simple'*, *open_with=<function default_open>*, *mkdirs=<function default_mkdirs>*, *has_nulls=True*, *write_index=None*, *partition_on=[]*, *fixed_text=None*, *append=False*, *object_encoding='infer'*, *times='int64'*)

Write Pandas DataFrame to filename as Parquet Format

**Parameters**

**filename: string** Parquet collection to write to, either a single file (if file_scheme is simple) or a directory containing the metadata and data-files.

**data: pandas dataframe** The table to write

**row_group_offsets: int or list of ints** If int, row-groups will be approximately this many rows, rounded down to make row groups about the same size; if a list, the explicit index values to start new row groups.

**compression: str, dict** compression to apply to each column, e.g. `GZIP` or `SNAPPY` or a `dict` like `{"col1": "SNAPPY", "col2": None}` to specify per column compression types. In both cases, the compressor settings would be the underlying compressor defaults. To pass arguments to the underlying compressor, each `dict` entry should itself be a dictionary:

```
{
    col1: {
        "type": "LZ4",
        "args": {
            "compression_level": 6,
            "content_checksum": True
         }
    },
    col2: {
        "type": "SNAPPY",
        "args": None
    }
    "_default": {
```

(continues on next page)

(continued from previous page)

```
        "type": "GZIP",
        "args": None
    }
}
```

where `"type"` specifies the compression type to use, and `"args"` specifies a `dict` that will be turned into keyword arguments for the compressor. If the dictionary contains a "_default" entry, this will be used for any columns not explicitly specified in the dictionary.

**file_scheme: 'simple'|'hive'** If simple: all goes in a single file If hive: each row group is in a separate file, and a separate file (called "_metadata") contains the metadata.

**open_with: function** When called with a f(path, mode), returns an open file-like object

**mkdirs: function** When called with a path/URL, creates any necessary dictionaries to make that location writable, e.g., `os.makedirs`. This is not necessary if using the simple file scheme

**has_nulls: bool, 'infer' or list of strings** Whether columns can have nulls. If a list of strings, those given columns will be marked as "optional" in the metadata, and include null definition blocks on disk. Some data types (floats and times) can instead use the sentinel values NaN and NaT, which are not the same as NULL in parquet, but functionally act the same in many cases, particularly if converting back to pandas later. A value of 'infer' will assume nulls for object columns and not otherwise.

**write_index: boolean** Whether or not to write the index to a separate column. By default we write the index *if* it is not 0, 1, . . . , n.

**partition_on: list of column names** Passed to groupby in order to split data within each row-group, producing a structured directory tree. Note: as with pandas, null values will be dropped. Ignored if file_scheme is simple.

**fixed_text: {column: int length} or None** For bytes or str columns, values will be converted to fixed-length strings of the given length for the given columns before writing, potentially providing a large speed boost. The length applies to the binary representation *after* conversion for utf8, json or bson.

**append: bool (False)** If False, construct data-set from scratch; if True, add new row-group(s) to existing data-set. In the latter case, the data-set must exist, and the schema must match the input data.

**object_encoding: str or {col: type}** For object columns, this gives the data type, so that the values can be encoded to bytes. Possible values are bytes|utf8|json|bson|bool|int|int32, where bytes is assumed if not specified (i.e., no conversion). The special value 'infer' will cause the type to be guessed from the first ten non-null values.

**times: 'int64' (default), or 'int96':** In "int64" mode, datetimes are written as 8-byte integers, us resolution; in "int96" mode, they are written as 12-byte blocks, with the first 8 bytes as ns within the day, the next 4 bytes the julian day. 'int96' mode is included only for compatibility.

### Examples

```
>>> fastparquet.write('myfile.parquet', df)  # doctest: +SKIP
```

## 5.5 Backend File-systems

Fastparquet can use alternatives to the local disk for reading and writing parquet.

One example of such a backend file-system is s3fs, to connect to AWS's S3 storage. In the following, the login credentials are automatically inferred from the system (could be environment variables, or one of several possible configuration files).

```
import s3fs
from fastparquet import ParquetFile
s3 = s3fs.S3FileSystem()
myopen = s3.open
pf = ParquetFile('/mybucket/data.parquet', open_with=myopen)
df = pf.to_pandas()
```

The function `myopen` provided to the constructor must be callable with `f(path, mode)` and produce an open file context.

The resultant `pf` object is the same as would be generated locally, and only requires a relatively short read from the remote store. If '/mybucket/data.parquet' contains a sub-key called "_metadata", it will be read in preference, and the data-set is assumed to be multi-file.

Similarly, providing an open function and another to make any necessary directories (only necessary in multi-file mode), we can write to the s3 file-system:

```
write('/mybucket/output_parq', data, file_scheme='hive',
      row_group_offsets=[0, 500], open_with=myopen, mkdirs=noop)
```

(In the case of s3, no intermediate directories need to be created)

## 5.6 Developer Guide

Fastparquet is a free and open-source project. We welcome contributions in the form of bug reports, documentation, code, design proposals, and more. This page provides resources on how best to contribute.

**Bug reports**

Please file an issue on github.

**Running tests**

Aside from the requirements for using this package, the following additional packages should be present:

- pytest

Some tests also require:

- s3fs
- moto
- pyspark

**Building Docs**

The *docs/* directory contains source code for the documentation. You will need sphinx and numpydoc to successfully build. sphinx allows output in many formats, including html:

```
# in directory docs/
make html
```

This will produce a `build/html/` subdirectory, where the entry point is `index.html`.

- genindex
- modindex
- search

# Index