fastparquet Documentation

Release 0.7.1

Continuum Analytics

Aug 03, 2021

CONTENTS

A Python interface to the Parquet file format.

CHAPTER

ONE

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, including:

1. columnar storage, only read the data of interest
2. efficient binary packing
3. choice of compression algorithms and encoding
4. split data into files, allowing for parallel processing
5. range of logical types
6. statistics stored in metadata allow for skipping unneeded chunks
7. 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 TWO

HIGHLIGHTS

The original outline plan for this project can be found upstream

Briefly, some features of interest:

1. read and write Parquet files, in single or multiple-file format
2. choice of compression per-column and various optimized encoding schemes; ability to choose row divisions and partitioning on write.
3. acceleration of both reading and writing using `cython`, competitive performance versus other frameworks
4. ability to read and write to arbitrary file-like objects, allowing interoperability with `fsspec` filesystems and others
5. can be called from dask, to enable parallel reading and writing with Parquet files, possibly distributed across a cluster.

CHAPTER THREE

CAVEATS, KNOWN ISSUES

Please see the *Release Notes*. With versions 0.6.0 and 0.7.0, a LOT of new features and enhancements were added, so please read that page carefully, this may affect you!

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.

CHAPTER

FOUR

RELATION TO OTHER PROJECTS

1. parquet-python is the original pure-Python Parquet quick-look utility which was the inspiration for fastparquet. It has continued development, but is not directed as big data vectorised loading as we are.
2. Apache Arrow and its python API define an in-memory data representation, and can read/write parquet, including conversion to pandas. It is the "other" engine available within Dask and Pandas, and gives good performance and a range of options. If you are using Arrow anyway, you probably want to use its parquet interface.
3. 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. Spark is used in some tests and some test files were produced by Spark.
4. 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 FIVE

INDEX

5.1 Installation

5.1.1 Requirements

Required:

1. numpy
2. pandas
3. cramjam
4. thrift

cramjam provides compression codecs: gzip, snappy, lz4, brotli, zstd

Optional compression codec:

1. python-lzo/lzo

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, “main” branch:

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

Please be sure to install numpy before fastparquet when using pip, as pip sometimes can fail to solve the environment. Depending on your platform, pip may pull binary wheels or attempt to rebuild fastparquet.

5.1.3 Dev requirements

To run all of the tests, you will need the following, in addition to the requirements above:

1. python=3.8
2. bson
3. lz4
4. lzo
5. pytest
6. dask
7. moto/s3fs
8. pytest-cov
9. pyspark

Some of these (e.g., pyspark) are optional and will result in skipped tests if not present.

Tests use pytest.

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`.

5.2 Release Notes

Please see the issue Future Plans for a list of features completed or planned in 2021. As of 0.7.0, only one larger item remains to be completed.

5.3 0.7.1

1. Back compile for older versions of numpy
2. Make pandas nullable types opt-out. The old behaviour (casting to float) is still available with `ParquetFile(..., pandas_nulls=False)`.
3. Fix time field regression: IsAdjustedToUTC will be False when there is no timezone
4. Micro improvements to the speed of ParquetFile creation by using simple simple string ops instead of regex and regularising filenames once at the start. Effects datasets with many files.

5.3.1 0.7.0

(July 2021)

This version institutes major, breaking changes, listed here, and incremental fixes and additions.

1. Reading a directory without a `_metadata` summary file now works by providing only the directory, instead of a list of constituent files. This change also makes direct of use of fsspec filesystems, if given, to be able to load the footer metadata areas of the files concurrently, if the storage backend supports it, and not directly instantiating intermediate ParquetFile instances
2. row-level filtering of the data. Whereas previously, only full row-groups could be excluded on the basis of their parquet metadata statistics (if present), filtering can now be done within row-groups too. The syntax is the same as before, allowing for multiple column expressions to be combined with AND|OR, depending on the list structure. This mechanism requires two passes: one to load the columns needed to create the boolean mask, and another to load the columns actually needed in the output. This will not be faster, and may be slower, but in some cases can save significant memory footprint, if a small fraction of rows are considered good and the columns for the filter expression are not in the output. Not currently supported for reading with DataPageV2.
3. DELTA integer encoding (read-only): experimentally working, but we only have one test file to verify against, since it is not trivial to persuade Spark to produce files encoded this way. DELTA can be extremely compact a representation for slowly varying and/or monotonically increasing integers.
4. nanosecond resolution times: the new extended "logical" types system supports nanoseconds alongside the previous millis and micros. We now emit these for the default pandas time type, and produce full parquet schema including both "converted" and "logical" type information. Note that all output has `isAdjustedToUTC=True`, i.e., these are timestamps rather than local time. The time-zone is stored in the metadata, as before, and will be successfully recreated only in fastparquet and (py)arrow. Otherwise, the times will appear to be UTC. For compatibility with Spark, you may still want to use `times="int96"` when writing.
5. DataPageV2 writing: now we support both reading and writing. For writing, can be enabled with the environment variable FASTPARQUET_DATAPAGE_V2, or module global `fastparquet.writer.DATAPAGE_VERSION` and is off by default. It will become on by default in the future. In many cases, V2 will result in better read performance, because the data and page headers are encoded separately, so data can be directly read into the output without addition allocation/copies. This feature is considered experimental, but we believe it working well for most use cases (i.e., our test suite) and should be readable by all modern parquet frameworks including arrow and spark.
6. pandas nullable types: pandas supports "masked" extension arrays for types that previously could not support NULL at all: ints and bools. Fastparquet used to cast such columns to float, so that we could represent NULLs as NaN; now we use the new(er) masked types by default. This means faster reading of such columns, as there is no conversion. If the metadata guarantees that there are no nulls, we still use the non-nullable variant *unless* the data was written with fastparquet/pyarrow, and the metadata indicates that the original datatype was nullable. We already handled writing of nullable columns.

5.3.2 0.6.0

(May 2021)

This version institutes major, breaking changes, listed here, and incremental fixes and additions.

NB: minor versions up to 0.6.3 fix build issues

1. replacement of the numba dependency with cythonized code. This also brought many performance improvements, by reducing memory copies in many places, and an overhaul of many parts of the code. Replacing numba by cython did not affect the performance of specific functions, but has made installation of fastparquet much simpler, for not needing the numba/LLVM stack, and imports faster, for not having to compile any code at runtime.

2. distribution as pip-installable wheels. Since we are cythonizing more, we want to make installation as simple as we can. So we now produce wheels.
3. using cramjam as the comp/decompression backend, instead of separate libraries for snappy, zstd, brotli… . This decreases the size and complexity of the install, guarantees the availability of codecs (cramjam is a required dependency, but with no dependencies of its own), and for the parquet read case, where we know the size of the original data, brings a handy speed-up.
4. implementation of DataPageV2: reading (see also 0.7.0 entry): this has been in the parquet spec for a long time, but only seen sporadic take-up until recently. Using standard reference files from the parquet project, we ensure correct reading of some V2-encoded files.
5. RLE_DICT: this one is more of a fix. The parquet spec renamed PLAIN_DICTIONARY, or perhaps renamed the previous definition. We now follow the new definitions for writing and support both for reading.
6. support custom key/value metadata on write and preserve this metadata on append or consolidate of many data files.

5.4 Quickstart

You may already be using fastparquet via the Dask or Pandas APIs. The options available, with `engine="fastparquet"`, are essentially the same as given here and in our *API* docs.

5.4.1 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, and a summary in `info`. 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 can get a deeper view of the parquet schema wih `print(pf.schema)`.

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. If the data originated from pandas, the categories will already be known.

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])])
```

(new in *0.7.0*: row-level filtering)

5.4.2 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:

1. the compression algorithms (typically "snappy", for fast, but not too space-efficient), which can vary by column
2. 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.
3. multi-file saving can be enabled with the `file_scheme="hive"|"drill"`: directory-tree-partitioned output with a single metadata file and several data-files, one or more per leaf directory. The values used for partitioning are encoded into the paths of the data files.
4. append to existing data sets with `append=True`, adding new row-groups. For the specific case of "hive"-partitioned data and one file per partition, `append="overwrite"` is also allowed, which replaces partitions of the data where new data are given, but leaves other existing partitions untouched.

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

5.5 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. Please also read the *Release Notes* for newer or experimental features.

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

5.5.1 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/pyarrow.

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, *with the same labels in every part*.

```
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. For data not written by fastparquet/pyarrow, columns that are encoded as dictionary but not included in `categories` will be de-referenced on load, which is potentially expensive.

5.5.2 Byte Arrays

Fixed-length byte arrays provide a modest speed boost for binary data (bytestrings) whose lengths really are all the same or nearly so. 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})
```

This is not recommended for strings, since UTF8 encoding/decoding must be done anyway, and converting to fixed will probably just waste time.

5.5.3 Nulls

In pandas, there is no internal representation difference between NULL (no value) and NaN/NaT (not a valid number) for float and time columns. Other parquet frameworks (e.g., in in Spark) might likely treat NULL and NaN differently. 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`:

1. True: all columns become optional, and NaNs are always stored as NULL. This is the best option for compatibility. This is the default.
2. 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.
3. 'infer': only object columns will become optional, since float, time, and category columns can store sentinel values, and ordinary pandas int columns cannot contain any NaNs. This is the best-performing option if the data will only be read by fastparquet. Pandas nullable columns will be stored as optional, whether or not they contain nulls.
4. list of strings: the named columns will be optional, others required (no NULLs)

5.5.4 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:

1. 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;
2. only UTF8 encoding for text is automatically handled, although arbitrary byte strings can be written as raw bytes type;
3. all times are stored as UTC, but the timezone is stored in the metadata, so will be recreated if loaded into pandas

5.5.5 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. We do not generate a hierarchical column index.

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.

5.5.6 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.

5.5.7 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()))
```

You can also grab the first N rows of the first row-group with *fastparquet.ParquetFile.head()*, or select from among a data-set's row-groups using slice notation `pf_subset = pf[2:8]`.

5.5.8 Dask/Pandas

Dask and Pandas fully support calling `fastparquet` directly, with the function `read_parquet` and method `to_parquet`, specifying `engine="fastparquet"`. Please see their relevant docstrings. Remote filesystems are supported by using a URL with a "protocol://" specifier and any `storage_options` to be passed to the file system implementation.

5.6 API

`ParquetFile`(fn[, verify, open_with, root, . . .])	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`([filters])	Iterate a dataset by row-groups
`ParquetFile.info`	Dataset summary
`ParquetFile.head`(nrows, **kwargs)	Get the first nrows of data
`ParquetFile.count`([filters, row_filter])	Total number of rows
`ParquetFile.__getitem__`(item)	Select among the row-groups using integer/slicing
`write`(filename, data[, row_group_offsets, . . .])	Write Pandas DataFrame to filename as Parquet Format.

`class fastparquet.`**`ParquetFile`**(*fn*, *verify=False*, *open_with=<built-in function open>*, *root=False*, *sep=None*, *fs=None*, *pandas_nulls=True*)

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 dataset root directory, if required.

fs: fsspec-compatible filesystem You can use this instead of open_with (otherwise, it will be inferred)

pandas_nulls: bool (True) If True, columns that are int or bool in parquet, but have nulls, will become pandas nullale types (Uint, Int, boolean). If False (the only behaviour prior to v0.7.0), both kinds will be cast to float, and nulls will be NaN. Pandas nullable types were introduces in v1.0.0, but were still marked as experimental in v1.3.0.

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

fs: fsspec-compatible filesystem You can use this instead of open_with (otherwise, it will be inferred)

Methods

`count`([filters, row_filter])	Total number of rows
`head`(nrows, **kwargs)	Get the first nrows of data
`iter_row_groups`([filters])	Iterate a dataset by row-groups
`read_row_group_file`(rg, columns, categories)	Open file for reading, and process it as a row-group
`to_pandas`([columns, categories, filters, ...])	Read data from parquet into a Pandas dataframe.

check_categories	
pre_allocate	
row_group_filename	

property columns
Column names

count(*filters=None*, *row_filter=False*)
Total number of rows

filters and row_filters have the same meaning as in to_pandas. Unless both are given, this method will not need to decode any data

head(*nrows*, ***kwargs*)
Get the first nrows of data

This will load the whole of the first valid row-group for the given columns. If it has fewer rows than requested, we will not fetch more data.

kwargs can include things like columns, filters, etc., with the same semantics as to_pandas()

returns: dataframe

property info
Dataset summary

iter_row_groups(*filters=None*, ***kwargs*)
Iterate a dataset by row-groups

If filters is given, omits row-groups that fail the filer (saving execution time)

Returns

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

`read_row_group_file`(*rg*, *columns*, *categories*, *index=None*, *assign=None*, *partition_meta=None*, *row_filter=False*, *infile=None*)

Open file for reading, and process it as a row-group

assign is None if this method is called directly (not from to_pandas), in which case we return the resultant dataframe

row_filter can be:

- False (don't do row filtering)
- a list of filters (do filtering here for this one row-group; only makes sense if assign=None
- bool array with a size equal to the number of rows in this group and the length of the assign arrays

`to_pandas`(*columns=None*, *categories=None*, *filters=[]*, *index=None*, *row_filter=False*)

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 list of tuples or list of tuples To filter out data. Filter syntax: [[(column, op, val), …],…] where op is [==, =, >, >=, <, <=, !=, in, not in] The innermost tuples are transposed into a set of filters applied through an *AND* operation. The outer list combines these sets of filters through an *OR* operation. A single list of tuples can also be used, meaning that no *OR* operation between set of filters is to be conducted.

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.

row_filter: bool Whether filters are applied to whole row-groups (False, default) or row-wise (True, experimental). The latter requires two passes of any row group that may contain valid rows, but can be much more memory-efficient, especially if the filter columns are not required in the output.

Returns

Pandas data-frame

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

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 list of tuples or list of tuples To filter out data. Filter syntax: [[(column, op, val), . . .],. . .] where op is [==, =, >, >=, <, <=, !=, in, not in] The innermost tuples are transposed into a set of filters applied through an *AND* operation. The outer list combines these sets of filters through an *OR* operation. A single list of tuples can also be used, meaning that no *OR* operation between set of filters is to be conducted.

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.

row_filter: bool Whether filters are applied to whole row-groups (False, default) or row-wise (True, experimental). The latter requires two passes of any row group that may contain valid rows, but can be much more memory-efficient, especially if the filter columns are not required in the output.

Returns

Pandas data-frame

`ParquetFile.`**`iter_row_groups`**(*filters=None*, ***kwargs*)

Iterate a dataset by row-groups

If filters is given, omits row-groups that fail the filer (saving execution time)

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=<built-in function open>*, *mkdirs=<function default_mkdirs>*, *has_nulls=True*, *write_index=None*, *partition_on=[]*, *fixed_text=None*, *append=False*, *object_encoding='infer'*, *times='int64'*, *custom_metadata=None*)

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": {
            "mode": "high_compression",
            "compression": 9
         }
    },
    col2: {
        "type": "SNAPPY",
```

(continues on next page)

(continued from previous page)

```
        "args": None
    }
    "_default": {
        "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'|'drill' If simple: all goes in a single file If hive or drill: 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) or 'overwrite' 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.

If 'overwrite', existing partitions will be replaced in-place, where the given data has any rows within a given partition. To enable this, these other parameters have to be set to specific values, or will raise ValueError:

- `row_group_offsets=0`
- `file_scheme='hive'`
- `partition_on` has to be used, set to at least a column name

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|decimal, 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. The decimal.Decimal type is a valid choice, but will result in float encoding with possible loss of accuracy.

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.

custom_metadata: dict key-value metadata to write

Examples

```
>>> fastparquet.write('myfile.parquet', df)
```

5.7 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)

1. genindex 1. modindex 1. search

INDEX