
pandas-gbq Documentation

Release 0.1.0

PyData Development Team

Jun 26, 2018

Contents

1	Installation	3
1.1	Conda	3
1.2	Pip	3
1.3	Install from Source	3
1.4	Dependencies	3
2	Introduction	5
2.1	Supported Data Types	5
2.2	Integer and boolean NA handling	5
2.3	Logging	5
3	Authentication	7
3.1	Authentication with a Service Account	7
3.2	Default Authentication Methods	7
4	Reading Tables	9
5	Writing DataFrames	11
6	Creating Tables	13
7	API Reference	15
8	Contributing to pandas-gbq	19
8.1	Where to start?	20
8.2	Bug reports and enhancement requests	20
8.3	Working with the code	21
8.4	Contributing to the code base	22
8.5	Contributing your changes to <i>pandas-gbq</i>	25
9	Changelog	29
9.1	0.5.0 / 2018-06-15	29
9.2	0.4.1 / 2018-04-05	29
9.3	0.4.0 / 2018-04-03	30
9.4	0.3.1 / 2018-02-13	30
9.5	0.3.0 / 2018-01-03	30
9.6	0.2.1 / 2017-11-27	30
9.7	0.2.0 / 2017-07-24	31

9.8	0.1.6 / 2017-05-03	31
9.9	0.1.4 / 2017-03-17	31
9.10	0.1.3 / 2017-03-04	31
9.11	0.1.2 / 2017-02-23	31

10 Indices and tables **33**

The `pandas_gbq` module provides a wrapper for Google's BigQuery analytics web service to simplify retrieving results from BigQuery tables using SQL-like queries. Result sets are parsed into a pandas DataFrame with a shape and data types derived from the source table. Additionally, DataFrames can be inserted into new BigQuery tables or appended to existing tables.

Warning: To use this module, you will need a valid BigQuery account. Refer to the [BigQuery Documentation](#) for details on the service itself.

Contents:

You can install `pandas-gbq` with `conda`, `pip`, or by installing from source.

1.1 Conda

```
$ conda install pandas-gbq --channel conda-forge
```

This installs `pandas-gbq` and all common dependencies, including `pandas`.

1.2 Pip

To install the latest version of `pandas-gbq`: from the

```
$ pip install pandas-gbq -U
```

This installs `pandas-gbq` and all common dependencies, including `pandas`.

1.3 Install from Source

```
$ pip install git+https://github.com/pydata/pandas-gbq.git
```

1.4 Dependencies

This module requires following additional dependencies:

- `google-auth`: authentication and authorization for Google's API

- `google-auth-oauthlib`: integration with `oauthlib` for end-user authentication
- `google-cloud-bigquery`: Google Cloud client library for BigQuery

Note: The dependency on `google-cloud-bigquery` is new in version 0.3.0 of `pandas-gbq`. Versions less than 0.3.0 required the following dependencies:

- `httplib2`: HTTP client (no longer required)
 - `google-api-python-client`: Google's API client (no longer required, replaced by `google-cloud-bigquery`.)
 - `google-auth`: authentication and authorization for Google's API
 - `google-auth-oauthlib`: integration with `oauthlib` for end-user authentication
 - `google-auth-httplib2`: adapter to use `httplib2` HTTP client with `google-auth` (no longer required)
-

2.1 Supported Data Types

Pandas supports all these [BigQuery](#) data types: `STRING`, `INTEGER (64bit)`, `FLOAT (64 bit)`, `BOOLEAN` and `TIMESTAMP (microsecond precision)`. Data types `BYTES` and `RECORD` are not supported.

2.2 Integer and boolean NA handling

Since all columns in [BigQuery](#) queries are nullable, and [NumPy](#) lacks of NA support for integer and boolean types, this module will store `INTEGER` or `BOOLEAN` columns with at least one `NULL` value as `dtype=object`. Otherwise those columns will be stored as `dtype=int64` or `dtype=bool` respectively.

This is opposite to default pandas behaviour which will promote integer type to float in order to store NAs. [See here for how this works in pandas](#)

While this trade-off works well for most cases, it breaks down for storing values greater than 2^{53} . Such values in [BigQuery](#) can represent identifiers and unnoticed precision lost for identifier is what we want to avoid.

2.3 Logging

Because some requests take some time, this library will log its progress of longer queries. [IPython](#) & [Jupyter](#) by default attach a handler to the logger. If you're running in another process and want to see logs, or you want to see more verbose logs, you can do something like:

```
import logging
import sys
logger = logging.getLogger('pandas_gbq')
logger.setLevel(logging.DEBUG)
logger.addHandler(logging.StreamHandler(stream=sys.stdout))
```


pandas-gbq authenticates with the Google BigQuery service via OAuth 2.0.

3.1 Authentication with a Service Account

Using service account credentials is particularly useful when working on remote servers without access to user input.

Create a service account key via the [service account key creation page](#) in the Google Cloud Platform Console. Select the JSON key type and download the key file.

To use service account credentials, set the `private_key` parameter to one of:

- A file path to the JSON file.
- A string containing the JSON file contents.

See the [Getting started with authentication on Google Cloud Platform](#) guide for more information on service accounts.

3.2 Default Authentication Methods

If the `private_key` parameter is `None`, pandas-gbq tries the following authentication methods:

1. Application Default Credentials via the `google.auth.default()` function.

Note: If pandas-gbq can obtain default credentials but those credentials cannot be used to query BigQuery, pandas-gbq will also try obtaining user account credentials.

A common problem with default credentials when running on Google Compute Engine is that the VM does not have sufficient scopes to query BigQuery.

2. User account credentials.

pandas-gbq loads cached credentials from a hidden user folder on the operating system. Override the location of the cached user credentials by setting the `PANDAS_GBQ_CREDENTIALS_FILE` environment variable.

If pandas-gbq does not find cached credentials, it opens a browser window asking for you to authenticate to your BigQuery account using the product name `pandas GBQ`.

Additional information on the user credentials authentication mechanism can be found [here](#).

CHAPTER 4

Reading Tables

Suppose you want to load all data from an existing BigQuery table `test_dataset.test_table` into a DataFrame using the `read_gbq()` function.

```
# Insert your BigQuery Project ID Here
# Can be found in the Google web console
projectid = "xxxxxxxx"

data_frame = read_gbq('SELECT * FROM test_dataset.test_table', projectid)
```

You can define which column from BigQuery to use as an index in the destination DataFrame as well as a preferred column order as follows:

```
data_frame = read_gbq('SELECT * FROM test_dataset.test_table',
                      index_col='index_column_name',
                      col_order=['col1', 'col2', 'col3'], projectid)
```

You can specify the query config as parameter to use additional options of your job. For more information about query configuration parameters see [here](#).

```
configuration = {
    'query': {
        "useQueryCache": False
    }
}
data_frame = read_gbq('SELECT * FROM test_dataset.test_table',
                      configuration=configuration, projectid)
```

Note: You can find your project id in the [Google developers console](#).

Note: The `dialect` argument can be used to indicate whether to use BigQuery's 'legacy' SQL or BigQuery's 'standard' SQL (beta). The default value is 'legacy'. For more information on BigQuery's standard SQL, see

Writing DataFrames

Assume we want to write a DataFrame `df` into a BigQuery table using `to_gbq()`.

```
In [1]: import pandas as pd
```

```
In [2]: df = pd.DataFrame({'my_string': list('abc'),
...:                      'my_int64': list(range(1, 4)),
...:                      'my_float64': np.arange(4.0, 7.0),
...:                      'my_bool1': [True, False, True],
...:                      'my_bool2': [False, True, False],
...:                      'my_dates': pd.date_range('now', periods=3)})
...:
```

```
In [3]: df
```

```
Out[3]:
```

	my_string	...	my_dates
0	a	...	2018-06-26 16:31:46.380578
1	b	...	2018-06-27 16:31:46.380578
2	c	...	2018-06-28 16:31:46.380578

```
[3 rows x 6 columns]
```

```
In [4]: df.dtypes
```

```
//////////////////////////////////////
↪
my_string      object
my_int64       int64
my_float64     float64
my_bool1       bool
my_bool2       bool
my_dates       datetime64[ns]
dtype: object
```

```
to_gbq(df, 'my_dataset.my_table', projectid)
```

Note: The destination table and destination dataset will automatically be created if they do not already exist.

The `if_exists` argument can be used to dictate whether to 'fail', 'replace' or 'append' if the destination table already exists. The default value is 'fail'.

For example, assume that `if_exists` is set to 'fail'. The following snippet will raise a `TableCreationError` if the destination table already exists.

```
to_gbq(df, 'my_dataset.my_table', projectid, if_exists='fail')
```

Note: If the `if_exists` argument is set to 'append', the destination dataframe will be written to the table using the defined table schema and column types. The dataframe must contain fields (matching name and type) currently in the destination table.

Note: If an error occurs while streaming data to BigQuery, see [Troubleshooting BigQuery Errors](#).

Note: The BigQuery SQL query language has some oddities, see the [BigQuery Query Reference Documentation](#).

Note: While BigQuery uses SQL-like syntax, it has some important differences from traditional databases both in functionality, API limitations (size and quantity of queries or uploads), and how Google charges for use of the service. You should refer to [Google BigQuery documentation](#) often as the service is always evolving. BigQuery is best for analyzing large sets of data quickly, but it is not a direct replacement for a transactional database.

Creating Tables

```
In [10]: gbq.generate_bq_schema(df, default_type='STRING')
```

```
Out [10]: {'fields': [{'name': 'my_bool1', 'type': 'BOOLEAN'},  
                    {'name': 'my_bool2', 'type': 'BOOLEAN'},  
                    {'name': 'my_dates', 'type': 'TIMESTAMP'},  
                    {'name': 'my_float64', 'type': 'FLOAT'},  
                    {'name': 'my_int64', 'type': 'INTEGER'},  
                    {'name': 'my_string', 'type': 'STRING'}]}
```

Note: If you delete and re-create a BigQuery table with the same name, but different table schema, you must wait 2 minutes before streaming data into the table. As a workaround, consider creating the new table with a different name. Refer to [Google BigQuery issue 191](#).

Note: Only functions and classes which are members of the `pandas_gbq` module are considered public. Submodules and their members are considered private.

<code>read_gbq(query[, project_id, index_col, ...])</code>	Load data from Google BigQuery using google-cloud-python
<code>to_gbq(dataframe, destination_table[, ...])</code>	Write a DataFrame to a Google BigQuery table.

`pandas_gbq.read_gbq(query, project_id=None, index_col=None, col_order=None, reauth=False, private_key=None, auth_local_webserver=False, dialect='legacy', location=None, configuration=None, verbose=None)`

Load data from Google BigQuery using google-cloud-python

The main method a user calls to execute a Query in Google BigQuery and read results into a pandas DataFrame.

This method uses the Google Cloud client library to make requests to Google BigQuery, documented [here](#).

See the [How to authenticate with Google BigQuery](#) guide for authentication instructions.

Parameters

query [str] SQL-Like Query to return data values.

project_id [str, optional] Google BigQuery Account project ID. Optional when available from the environment.

index_col [str, optional] Name of result column to use for index in results DataFrame.

col_order [list(str), optional] List of BigQuery column names in the desired order for results DataFrame.

reauth [boolean, default False] Force Google BigQuery to re-authenticate the user. This is useful if multiple accounts are used.

private_key [str, optional] Service account private key in JSON format. Can be file path or

string contents. This is useful for remote server authentication (eg. Jupyter/IPython notebook on remote host).

auth_local_webserver [boolean, default False] Use the [local webserver flow](#) instead of the [console flow](#) when getting user credentials.

New in version 0.2.0.

dialect [str, default 'legacy'] SQL syntax dialect to use. Value can be one of:

'**legacy**' Use BigQuery's legacy SQL dialect. For more information see [BigQuery Legacy SQL Reference](#).

'**standard**' Use BigQuery's standard SQL, which is compliant with the SQL 2011 standard. For more information see [BigQuery Standard SQL Reference](#).

location [str, optional] Location where the query job should run. See the [BigQuery locations documentation](#) for a list of available locations. The location must match that of any datasets used in the query.

New in version 0.5.0.

configuration [dict, optional] Query config parameters for job processing. For example:

```
configuration = { 'query': { 'useQueryCache': False } }
```

For more information see [BigQuery REST API Reference](#).

verbose [None, deprecated] Deprecated in Pandas-GBQ 0.4.0. Use the [logging module to adjust verbosity instead](#).

Returns

df: DataFrame DataFrame representing results of query.

```
pandas_gbq.to_gbq(dataframe, destination_table, project_id=None, chunksize=None, reauth=False, if_exists='fail', private_key=None, auth_local_webserver=False, table_schema=None, location=None, progress_bar=True, verbose=None)
```

Write a DataFrame to a Google BigQuery table.

The main method a user calls to export pandas DataFrame contents to Google BigQuery table.

This method uses the Google Cloud client library to make requests to Google BigQuery, documented [here](#).

See the [How to authenticate with Google BigQuery](#) guide for authentication instructions.

Parameters

dataframe [pandas.DataFrame] DataFrame to be written to a Google BigQuery table.

destination_table [str] Name of table to be written, in the form `dataset.tablename`.

project_id [str, optional] Google BigQuery Account project ID. Optional when available from the environment.

chunksize [int, optional] Number of rows to be inserted in each chunk from the dataframe. Set to `None` to load the whole dataframe at once.

reauth [bool, default False] Force Google BigQuery to re-authenticate the user. This is useful if multiple accounts are used.

if_exists [str, default 'fail'] Behavior when the destination table exists. Value can be one of:

'**fail**' If table exists, do nothing.

'**replace**' If table exists, drop it, recreate it, and insert data.

'**append**' If table exists, insert data. Create if does not exist.

private_key [str, optional] Service account private key in JSON format. Can be file path or string contents. This is useful for remote server authentication (eg. Jupyter/IPython notebook on remote host).

auth_local_webserver [bool, default False] Use the [local webserver flow](#) instead of the [console flow](#) when getting user credentials.

New in version 0.2.0.

table_schema [list of dicts, optional] List of BigQuery table fields to which according DataFrame columns conform to, e.g. `[{'name': 'col1', 'type': 'STRING'}, ...]`. If schema is not provided, it will be generated according to dtypes of DataFrame columns. See BigQuery API documentation on available names of a field.

New in version 0.3.1.

location [str, optional] Location where the load job should run. See the [BigQuery locations documentation](#) for a list of available locations. The location must match that of the target dataset.

New in version 0.5.0.

progress_bar [bool, default True] Use the library *tqdm* to show the progress bar for the upload, chunk by chunk.

New in version 0.5.0.

verbose [bool, deprecated] Deprecated in Pandas-GBQ 0.4.0. Use the [logging module](#) to adjust verbosity instead.

Table of contents:

- *Where to start?*
- *Bug reports and enhancement requests*
- *Working with the code*
 - *Version control, Git, and GitHub*
 - *Getting started with Git*
 - *Forking*
 - *Creating a branch*
 - *Install in Development Mode*
 - * *Conda*
 - * *Pip & virtualenv*
- *Contributing to the code base*
 - *Code standards*
 - * *Python (PEP8)*
 - * *Backwards Compatibility*
 - *Test-driven development/code writing*
 - * *Running the test suite*
 - * *Testing on multiple Python versions*
 - * *Running Google BigQuery Integration Tests*
 - *Documenting your code*

- *Contributing your changes to pandas-gbq*
 - *Committing your code*
 - *Combining commits*
 - *Pushing your changes*
 - *Review your code*
 - *Finally, make the pull request*
 - *Delete your merged branch (optional)*

8.1 Where to start?

All contributions, bug reports, bug fixes, documentation improvements, enhancements and ideas are welcome.

If you are simply looking to start working with the *pandas-gbq* codebase, navigate to the [GitHub “issues” tab](#) and start looking through interesting issues.

Or maybe through using *pandas-gbq* you have an idea of your own or are looking for something in the documentation and thinking ‘this can be improved’... you can do something about it!

Feel free to ask questions on the [mailing list](#).

8.2 Bug reports and enhancement requests

Bug reports are an important part of making *pandas-gbq* more stable. Having a complete bug report will allow others to reproduce the bug and provide insight into fixing it. Because many versions of *pandas-gbq* are supported, knowing version information will also identify improvements made since previous versions. Trying the bug-producing code out on the *master* branch is often a worthwhile exercise to confirm the bug still exists. It is also worth searching existing bug reports and pull requests to see if the issue has already been reported and/or fixed.

Bug reports must:

1. Include a short, self-contained Python snippet reproducing the problem. You can format the code nicely by using [GitHub Flavored Markdown](#)

```
>>> from pandas_gbq import gbq
>>> df = gbq.read_gbq(...)
...

```

2. Include the full version string of *pandas-gbq*.

```
>>> import pandas_gbq
>>> pandas_gbq.__version__
...

```

3. Explain why the current behavior is wrong/not desired and what you expect instead.

The issue will then show up to the *pandas-gbq* community and be open to comments/ideas from others.

8.3 Working with the code

Now that you have an issue you want to fix, enhancement to add, or documentation to improve, you need to learn how to work with GitHub and the *pandas-gbq* code base.

8.3.1 Version control, Git, and GitHub

To the new user, working with Git is one of the more daunting aspects of contributing to *pandas-gbq*. It can very quickly become overwhelming, but sticking to the guidelines below will help keep the process straightforward and mostly trouble free. As always, if you are having difficulties please feel free to ask for help.

The code is hosted on [GitHub](#). To contribute you will need to sign up for a [free GitHub account](#). We use [Git](#) for version control to allow many people to work together on the project.

Some great resources for learning Git:

- the [GitHub help pages](#).
- the [NumPy's documentation](#).
- [Matthew Brett's Pydagogue](#).

8.3.2 Getting started with Git

[GitHub has instructions](#) for installing git, setting up your SSH key, and configuring git. All these steps need to be completed before you can work seamlessly between your local repository and GitHub.

8.3.3 Forking

You will need your own fork to work on the code. Go to the [pandas-gbq project page](#) and hit the `Fork` button. You will want to clone your fork to your machine:

```
git clone git@github.com:your-user-name/pandas-gbq.git pandas-gbq-yourname
cd pandas-gbq-yourname
git remote add upstream git://github.com/pydata/pandas-gbq.git
```

This creates the directory *pandas-gbq-yourname* and connects your repository to the upstream (main project) *pandas-gbq* repository.

The testing suite will run automatically on Travis-CI once your pull request is submitted. However, if you wish to run the test suite on a branch prior to submitting the pull request, then Travis-CI needs to be hooked up to your GitHub repository. Instructions for doing so are [here](#).

8.3.4 Creating a branch

You want your master branch to reflect only production-ready code, so create a feature branch for making your changes. For example:

```
git branch shiny-new-feature
git checkout shiny-new-feature
```

The above can be simplified to:

```
git checkout -b shiny-new-feature
```

This changes your working directory to the shiny-new-feature branch. Keep any changes in this branch specific to one bug or feature so it is clear what the branch brings to *pandas-gbq*. You can have many shiny-new-features and switch in between them using the git checkout command.

To update this branch, you need to retrieve the changes from the master branch:

```
git fetch upstream
git rebase upstream/master
```

This will replay your commits on top of the latest pandas-gbq git master. If this leads to merge conflicts, you must resolve these before submitting your pull request. If you have uncommitted changes, you will need to *stash* them prior to updating. This will effectively store your changes and they can be reapplied after updating.

8.3.5 Install in Development Mode

It's helpful to install pandas-gbq in development mode so that you can use the library without reinstalling the package after every change.

Conda

Create a new conda environment and install the necessary dependencies

```
$ conda create -n my-env --channel conda-forge \
  pandas \
  google-auth-oauthlib \
  google-api-python-client \
  google-auth-httplib2
$ source activate my-env
```

Install pandas-gbq in development mode

```
$ python setup.py develop
```

Pip & virtualenv

Skip this section if you already followed the conda instructions.

Create a new [virtual environment](#).

```
$ virtualenv env
$ source env/bin/activate
```

You can install pandas-gbq and its dependencies in [development mode](#) via pip.

```
$ pip install -e .
```

8.4 Contributing to the code base

Code Base:

- *Code standards*
 - *Python (PEP8)*
 - *Backwards Compatibility*
- *Test-driven development/code writing*
 - *Running the test suite*
 - *Testing on multiple Python versions*
 - *Running Google BigQuery Integration Tests*
- *Documenting your code*

8.4.1 Code standards

Writing good code is not just about what you write. It is also about *how* you write it. During testing on Travis-CI, several tools will be run to check your code for stylistic errors. Generating any warnings will cause the test to fail. Thus, good style is a requirement for submitting code to *pandas-gbq*.

In addition, because a lot of people use our library, it is important that we do not make sudden changes to the code that could have the potential to break a lot of user code as a result, that is, we need it to be as *backwards compatible* as possible to avoid mass breakages.

Python (PEP8)

pandas-gbq uses the [PEP8](#) standard. There are several tools to ensure you abide by this standard. Here are *some* of the more common PEP8 issues:

- we restrict line-length to 79 characters to promote readability
- passing arguments should have spaces after commas, e.g. `foo(arg1, arg2, kw1='bar')`

Travis-CI will run the [flake8](#) tool and report any stylistic errors in your code. Therefore, it is helpful before submitting code to run the check yourself on the diff:

```
git diff master | flake8 --diff
```

Backwards Compatibility

Please try to maintain backward compatibility. If you think breakage is required, clearly state why as part of the pull request. Also, be careful when changing method signatures and add deprecation warnings where needed.

8.4.2 Test-driven development/code writing

pandas-gbq is serious about testing and strongly encourages contributors to embrace [test-driven development \(TDD\)](#). This development process “relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test.” So, before actually writing any code, you should write your tests. Often the test can be taken from the original GitHub issue. However, it is always worth considering additional use cases and writing corresponding tests.

Adding tests is one of the most common requests after code is pushed to *pandas-gbq*. Therefore, it is worth getting in the habit of writing tests ahead of time so this is never an issue.

Like many packages, *pandas-gbq* uses `pytest`.

Running the test suite

The tests can then be run directly inside your Git clone (without having to install *pandas-gbq*) by typing:

```
pytest tests/unit
pytest tests/system.py
```

The tests suite is exhaustive and takes around 20 minutes to run. Often it is worth running only a subset of tests first around your changes before running the entire suite.

The easiest way to do this is with:

```
pytest tests/path/to/test.py -k regex_matching_test_name
```

Or with one of the following constructs:

```
pytest tests/[test-module].py
pytest tests/[test-module].py::[TestClass]
pytest tests/[test-module].py::[TestClass]::[test_method]
```

For more, see the `pytest` documentation.

Testing on multiple Python versions

pandas-gbq uses `nox` to automate testing in multiple Python environments. First, install `nox`.

```
$ pip install --upgrade nox-automation
```

To run tests in all versions of Python, run `nox` from the repository's root directory.

Running Google BigQuery Integration Tests

You will need to create a Google BigQuery private key in JSON format in order to run Google BigQuery integration tests on your local machine and on Travis-CI. The first step is to create a [service account](#).

To run the integration tests locally, set the following environment variables before running `pytest`:

1. `GBQ_PROJECT_ID` with the value being the ID of your BigQuery project.
2. `GBQ_GOOGLE_APPLICATION_CREDENTIALS` with the value being the *path* to the JSON key that you downloaded for your service account.

Integration tests are skipped in pull requests because the credentials that are required for running Google BigQuery integration tests are `encrypted` on Travis-CI and are only accessible from the `pydata/pandas-gbq` repository. The credentials won't be available on forks of *pandas-gbq*. Here are the steps to run `gbq` integration tests on a forked repository:

1. Go to [Travis CI](#) and sign in with your GitHub account.
2. Click on the + icon next to the `My Repositories` list and enable Travis builds for your fork.
3. Click on the gear icon to edit your travis build, and add two environment variables:

- `GBQ_PROJECT_ID` with the value being the ID of your BigQuery project.
- `SERVICE_ACCOUNT_KEY` with the value being the *contents* of the JSON key that you downloaded for your service account. Use single quotes around your JSON key to ensure that it is treated as a string.

For both environment variables, keep the “Display value in build log” option `DISABLED`. These variables contain sensitive data and you do not want their contents being exposed in build logs.

4. Your branch should be tested automatically once it is pushed. You can check the status by visiting your Travis branches page which exists at the following location: <https://travis-ci.org/your-user-name/pandas-gbq/branches>. Click on a build job for your branch.

8.4.3 Documenting your code

Changes should be reflected in the release notes located in `doc/source/changelog.rst`. This file contains an ongoing change log. Add an entry to this file to document your fix, enhancement or (unavoidable) breaking change. Make sure to include the GitHub issue number when adding your entry (using “`GH#1234`” where *1234* is the issue/pull request number).

If your code is an enhancement, it is most likely necessary to add usage examples to the existing documentation. Further, to let users know when this feature was added, the `versionadded` directive is used. The sphinx syntax for that is:

```
.. versionadded:: 0.1.3
```

This will put the text *New in version 0.1.3* wherever you put the sphinx directive. This should also be put in the docstring when adding a new function or method.

8.5 Contributing your changes to *pandas-gbq*

8.5.1 Committing your code

Keep style fixes to a separate commit to make your pull request more readable.

Once you’ve made changes, you can see them by typing:

```
git status
```

If you have created a new file, it is not being tracked by git. Add it by typing:

```
git add path/to/file-to-be-added.py
```

Doing ‘git status’ again should give something like:

```
# On branch shiny-new-feature
#
#       modified:   /relative/path/to/file-you-added.py
#
```

Finally, commit your changes to your local repository with an explanatory message. *pandas-gbq* uses a convention for commit message prefixes and layout. Here are some common prefixes along with general guidelines for when to use them:

- `ENH`: Enhancement, new functionality
- `BUG`: Bug fix

- DOC: Additions/updates to documentation
- TST: Additions/updates to tests
- BLD: Updates to the build process/scripts
- PERF: Performance improvement
- CLN: Code cleanup

The following defines how a commit message should be structured. Please reference the relevant GitHub issues in your commit message using GH1234 or #1234. Either style is fine, but the former is generally preferred:

- a subject line with < 80 chars.
- One blank line.
- Optionally, a commit message body.

Now you can commit your changes in your local repository:

```
git commit -m
```

8.5.2 Combining commits

If you have multiple commits, you may want to combine them into one commit, often referred to as “squashing” or “rebasing”. This is a common request by package maintainers when submitting a pull request as it maintains a more compact commit history. To rebase your commits:

```
git rebase -i HEAD~#
```

Where # is the number of commits you want to combine. Then you can pick the relevant commit message and discard others.

To squash to the master branch do:

```
git rebase -i master
```

Use the *s* option on a commit to *squash*, meaning to keep the commit messages, or *f* to *fixup*, meaning to merge the commit messages.

Then you will need to push the branch (see below) forcefully to replace the current commits with the new ones:

```
git push origin shiny-new-feature -f
```

8.5.3 Pushing your changes

When you want your changes to appear publicly on your GitHub page, push your forked feature branch’s commits:

```
git push origin shiny-new-feature
```

Here *origin* is the default name given to your remote repository on GitHub. You can see the remote repositories:

```
git remote -v
```

If you added the upstream repository as described above you will see something like:

```
origin  git@github.com:yourname/pandas-gbq.git (fetch)
origin  git@github.com:yourname/pandas-gbq.git (push)
upstream    git://github.com/pydata/pandas-gbq.git (fetch)
upstream    git://github.com/pydata/pandas-gbq.git (push)
```

Now your code is on GitHub, but it is not yet a part of the *pandas-gbq* project. For that to happen, a pull request needs to be submitted on GitHub.

8.5.4 Review your code

When you're ready to ask for a code review, file a pull request. Before you do, once again make sure that you have followed all the guidelines outlined in this document regarding code style, tests, performance tests, and documentation. You should also double check your branch changes against the branch it was based on:

1. Navigate to your repository on GitHub – <https://github.com/your-user-name/pandas-gbq>
2. Click on Branches
3. Click on the Compare button for your feature branch
4. Select the base and compare branches, if necessary. This will be master and shiny-new-feature, respectively.

8.5.5 Finally, make the pull request

If everything looks good, you are ready to make a pull request. A pull request is how code from a local repository becomes available to the GitHub community and can be looked at and eventually merged into the master version. This pull request and its associated changes will eventually be committed to the master branch and available in the next release. To submit a pull request:

1. Navigate to your repository on GitHub
2. Click on the Pull Request button
3. You can then click on Commits and Files Changed to make sure everything looks okay one last time
4. Write a description of your changes in the Preview Discussion tab
5. Click Send Pull Request.

This request then goes to the repository maintainers, and they will review the code. If you need to make more changes, you can make them in your branch, push them to GitHub, and the pull request will be automatically updated. Pushing them to GitHub again is done by:

```
git push -f origin shiny-new-feature
```

This will automatically update your pull request with the latest code and restart the Travis-CI tests.

8.5.6 Delete your merged branch (optional)

Once your feature branch is accepted into upstream, you'll probably want to get rid of the branch. First, merge upstream master into your branch so git knows it is safe to delete your branch:

```
git fetch upstream
git checkout master
git merge upstream/master
```

Then you can just do:

```
git branch -d shiny-new-feature
```

Make sure you use a lower-case `-d`, or else git won't warn you if your feature branch has not actually been merged.

The branch will still exist on GitHub, so to delete it there do:

```
git push origin --delete shiny-new-feature
```


9.1 0.5.0 / 2018-06-15

- Project ID parameter is optional in `read_gbq` and `to_gbq` when it can be inferred from the environment. Note: you must still pass in a project ID when using user-based authentication. (GH#103)
- Progress bar added for `to_gbq`, through an optional library `tqdm` as dependency. (GH#162)
- Add location parameter to `read_gbq` and `to_gbq` so that `pandas-gbq` can work with datasets in the Tokyo region. (GH#177)

9.1.1 Documentation

- Add *authentication how-to guide*. (GH#183)
- Update *Contributing to pandas-gbq* guide with new paths to tests. (GH#154, GH#164)

9.1.2 Internal changes

- Tests now use `nox` to run in multiple Python environments. (GH#52)
- Renamed internal modules. (GH#154)
- Refactored auth to an internal auth module. (GH#176)
- Add unit tests for `get_credentials()`. (GH#184)

9.2 0.4.1 / 2018-04-05

- Only show `verbose` deprecation warning if Pandas version does not populate it. (GH#157)

9.3 0.4.0 / 2018-04-03

- Fix bug in `read_gbq` when building a dataframe with integer columns on Windows. Explicitly use 64bit integers when converting from BQ types. (GH#119)
- Fix bug in `read_gbq` when querying for an array of floats (GH#123)
- Fix bug in `read_gbq` with configuration argument. Updates `read_gbq` to account for breaking change in the way `google-cloud-python` version 0.32.0+ handles query configuration API representation. (GH#152)
- Fix bug in `to_gbq` where seconds were discarded in timestamp columns. (GH#148)
- Fix bug in `to_gbq` when supplying a user-defined schema (GH#150)
- **Deprecate** the `verbose` parameter in `read_gbq` and `to_gbq`. Messages use the logging module instead of printing progress directly to standard output. (GH#12)

9.4 0.3.1 / 2018-02-13

- Fix an issue where Unicode couldn't be uploaded in Python 2 (GH#106)
- Add support for a passed schema in `:func:to_gbq` instead inferring the schema from the passed `DataFrame` with `DataFrame.dtypes` (GH#46)
- Fix an issue where a dataframe containing both integer and floating point columns could not be uploaded with `to_gbq` (GH#116)
- `to_gbq` now uses `to_csv` to avoid manually looping over rows in a dataframe (should result in faster table uploads) (GH#96)

9.5 0.3.0 / 2018-01-03

- Use the `google-cloud-bigquery` library for API calls. The `google-cloud-bigquery` package is a new dependency, and dependencies on `google-api-python-client` and `httplib2` are removed. See the [installation guide](#) for more details. (GH#93)
- Structs and arrays are now named properly (GH#23) and BigQuery functions like `array_agg` no longer run into errors during type conversion (GH#22).
- `to_gbq()` now uses a load job instead of the streaming API. Remove `StreamingInsertError` class, as it is no longer used by `to_gbq()`. (GH#7, GH#75)

9.6 0.2.1 / 2017-11-27

- `read_gbq()` now raises `QueryTimeout` if the request exceeds the `query.timeoutMs` value specified in the BigQuery configuration. (GH#76)
- Environment variable `PANDAS_GBQ_CREDENTIALS_FILE` can now be used to override the default location where the BigQuery user account credentials are stored. (GH#86)
- BigQuery user account credentials are now stored in an application-specific hidden user folder on the operating system. (GH#41)

9.7 0.2.0 / 2017-07-24

- Drop support for Python 3.4 ([GH#40](#))
- The dataframe passed to `.to_gbq(..., if_exists='append')` needs to contain only a subset of the fields in the BigQuery schema. ([GH#24](#))
- Use the `google-auth` library for authentication because `oauth2client` is deprecated. ([GH#39](#))
- `read_gbq()` now has a `auth_local_webserver` boolean argument for controlling whether to use web server or console flow when getting user credentials. Replaces `-noauth_local_webserver` command line argument. ([GH#35](#))
- `read_gbq()` now displays the BigQuery Job ID and standard price in verbose output. ([GH#70](#) and [GH#71](#))

9.8 0.1.6 / 2017-05-03

- All gbq errors will simply be subclasses of `ValueError` and no longer inherit from the deprecated `PandasError`.

9.9 0.1.4 / 2017-03-17

- `InvalidIndexColumn` will be raised instead of `InvalidColumnOrder` in `read_gbq()` when the index column specified does not exist in the BigQuery schema. ([GH#6](#))

9.10 0.1.3 / 2017-03-04

- Bug with appending to a BigQuery table where fields have modes (`NULLABLE`, `REQUIRED`, `REPEATED`) specified. These modes were compared versus the remote schema and writing a table via `to_gbq()` would previously raise. ([GH#13](#))

9.11 0.1.2 / 2017-02-23

Initial release of transfered code from `pandas`

Includes patches since the 0.19.2 release on `pandas` with the following:

- `read_gbq()` now allows query configuration preferences [pandas-GH#14742](#)
- `read_gbq()` now stores `INTEGER` columns as `dtype=object` if they contain `NULL` values. Otherwise they are stored as `int64`. This prevents precision lost for integers greater than 2^{53} . Furthermore `FLOAT` columns with values above 10^{14} are no longer casted to `int64` which also caused precision loss [pandas-GH#14064](#), and [pandas-GH#14305](#)

CHAPTER 10

Indices and tables

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

R

`read_gbq()` (in module `pandas_gbq`), 15

T

`to_gbq()` (in module `pandas_gbq`), 16