(continued from previous page)

```
3   2000-01-04
4   2000-01-05
5   2000-01-06
6   2000-01-07
7   2000-01-08
Name: index, dtype: datetime64[ns]

In [453]: store.select_column('df_dc', 'string')
Out[453]:
0    foo
1    foo
2    foo
3    foo
4    NaN
5    NaN
6    foo
7    bar
Name: string, dtype: object
```

Selecting coordinates

Sometimes you want to get the coordinates (a.k.a the index locations) of your query. This returns an `Int64Index` of the resulting locations. These coordinates can also be passed to subsequent `where` operations.

```
In [454]: df_coord = pd.DataFrame(np.random.randn(1000, 2),
   .....:                         index=pd.date_range('20000101', periods=1000))
   .....:

In [455]: store.append('df_coord', df_coord)

In [456]: c = store.select_as_coordinates('df_coord', 'index > 20020101')

In [457]: c
Out[457]:
Int64Index([732, 733, 734, 735, 736, 737, 738, 739, 740, 741,
            ...
            990, 991, 992, 993, 994, 995, 996, 997, 998, 999],
           dtype='int64', length=268)

In [458]: store.select('df_coord', where=c)
Out[458]:
                   0         1
2002-01-02 -0.165548  0.646989
2002-01-03  0.782753 -0.123409
2002-01-04 -0.391932 -0.740915
2002-01-05  1.211070 -0.668715
2002-01-06  0.341987 -0.685867
...              ...       ...
2002-09-22  1.788110 -0.405908
2002-09-23 -0.801912  0.768460
2002-09-24  0.466284 -0.457411
2002-09-25 -0.364060  0.785367
2002-09-26 -1.463093  1.187315

[268 rows x 2 columns]
```

Selecting using a where mask

Sometime your query can involve creating a list of rows to select. Usually this `mask` would be a resulting `index` from an indexing operation. This example selects the months of a datetimeindex which are 5.

```
In [459]: df_mask = pd.DataFrame(np.random.randn(1000, 2),
   .....:                        index=pd.date_range('20000101', periods=1000))
   .....:

In [460]: store.append('df_mask', df_mask)

In [461]: c = store.select_column('df_mask', 'index')

In [462]: where = c[pd.DatetimeIndex(c).month == 5].index

In [463]: store.select('df_mask', where=where)
Out[463]:
                   0         1
2000-05-01  1.735883 -2.615261
2000-05-02  0.422173  2.425154
2000-05-03  0.632453 -0.165640
2000-05-04 -1.017207 -0.005696
2000-05-05  0.299606  0.070606
...              ...       ...
2002-05-27  0.234503  1.199126
2002-05-28 -3.021833 -1.016828
2002-05-29  0.522794  0.063465
2002-05-30 -1.653736  0.031709
2002-05-31 -0.968402 -0.393583

[93 rows x 2 columns]
```

Storer object

If you want to inspect the stored object, retrieve via `get_storer`. You could use this programmatically to say get the number of rows in an object.

```
In [464]: store.get_storer('df_dc').nrows
Out[464]: 8
```

Multiple table queries

The methods `append_to_multiple` and `select_as_multiple` can perform appending/selecting from multiple tables at once. The idea is to have one table (call it the selector table) that you index most/all of the columns, and perform your queries. The other table(s) are data tables with an index matching the selector table's index. You can then perform a very fast query on the selector table, yet get lots of data back. This method is similar to having a very wide table, but enables more efficient queries.

The `append_to_multiple` method splits a given single DataFrame into multiple tables according to `d`, a dictionary that maps the table names to a list of 'columns' you want in that table. If *None* is used in place of a list, that table will have the remaining unspecified columns of the given DataFrame. The argument `selector` defines which table is the selector table (which you can make queries from). The argument `dropna` will drop rows from the input `DataFrame` to ensure tables are synchronized. This means that if a row for one of the tables being written to is entirely `np.NaN`, that row will be dropped from all tables.

If `dropna` is False, **THE USER IS RESPONSIBLE FOR SYNCHRONIZING THE TABLES**. Remember that entirely `np.Nan` rows are not written to the HDFStore, so if you choose to call `dropna=False`, some tables may have more rows than others, and therefore `select_as_multiple` may not work or it may return unexpected results.

```
In [465]: df_mt = pd.DataFrame(np.random.randn(8, 6),
   .....:                      index=pd.date_range('1/1/2000', periods=8),
   .....:                      columns=['A', 'B', 'C', 'D', 'E', 'F'])
   .....:

In [466]: df_mt['foo'] = 'bar'

In [467]: df_mt.loc[df_mt.index[1], ('A', 'B')] = np.nan

# you can also create the tables individually
In [468]: store.append_to_multiple({'df1_mt': ['A', 'B'], 'df2_mt': None},
   .....:                          df_mt, selector='df1_mt')
   .....:

In [469]: store
Out[469]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

# individual tables were created
In [470]: store.select('df1_mt')
Out[470]:
                   A         B
2000-01-01  1.251079 -0.362628
2000-01-02       NaN       NaN
2000-01-03  0.719421 -0.448886
2000-01-04  1.140998 -0.877922
2000-01-05  1.043605  1.798494
2000-01-06 -0.467812 -0.027965
2000-01-07  0.150568  0.754820
2000-01-08 -0.596306 -0.910022

In [471]: store.select('df2_mt')
Out[471]:
                   C         D         E         F  foo
2000-01-01  1.602451 -0.221229  0.712403  0.465927  bar
2000-01-02 -0.525571  0.851566 -0.681308 -0.549386  bar
2000-01-03 -0.044171  1.396628  1.041242 -1.588171  bar
2000-01-04  0.463351 -0.861042 -2.192841 -1.025263  bar
2000-01-05 -1.954845 -1.712882 -0.204377 -1.608953  bar
2000-01-06  1.601542 -0.417884 -2.757922 -0.307713  bar
2000-01-07 -1.935461  1.007668  0.079529 -1.459471  bar
2000-01-08 -1.057072 -0.864360 -1.124870  1.732966  bar

# as a multiple
In [472]: store.select_as_multiple(['df1_mt', 'df2_mt'], where=['A>0', 'B>0'],
   .....:                          selector='df1_mt')
   .....:
Out[472]:
                   A         B         C         D         E         F  foo
2000-01-05  1.043605  1.798494 -1.954845 -1.712882 -0.204377 -1.608953  bar
2000-01-07  0.150568  0.754820 -1.935461  1.007668  0.079529 -1.459471  bar
```

Delete from a table

You can delete from a table selectively by specifying a `where`. In deleting rows, it is important to understand the `PyTables` deletes rows by erasing the rows, then **moving** the following data. Thus deleting can potentially be a very expensive operation depending on the orientation of your data. To get optimal performance, it's worthwhile to have the dimension you are deleting be the first of the `indexables`.

Data is ordered (on the disk) in terms of the `indexables`. Here's a simple use case. You store panel-type data, with dates in the `major_axis` and ids in the `minor_axis`. The data is then interleaved like this:

- **date_1**
  - id_1
  - id_2
  - .
  - id_n
- **date_2**
  - id_1
  - .
  - id_n

It should be clear that a delete operation on the `major_axis` will be fairly quick, as one chunk is removed, then the following data moved. On the other hand a delete operation on the `minor_axis` will be very expensive. In this case it would almost certainly be faster to rewrite the table using a `where` that selects all but the missing data.

> **Warning:** Please note that HDF5 **DOES NOT RECLAIM SPACE** in the h5 files automatically. Thus, repeatedly deleting (or removing nodes) and adding again, **WILL TEND TO INCREASE THE FILE SIZE**.
>
> To *repack and clean* the file, use *ptrepack*.

Notes & caveats

Compression

`PyTables` allows the stored data to be compressed. This applies to all kinds of stores, not just tables. Two parameters are used to control compression: `complevel` and `complib`.

`complevel` specifies if and how hard data is to be compressed. `complevel=0` and `complevel=None` disables compression and `0<complevel<10` enables compression.

`complib` specifies which compression library to use. If nothing is specified the default library `zlib` is used. A compression library usually optimizes for either good compression rates or speed and the results will depend on the type of data. Which type of compression to choose depends on your specific needs and data. The list of supported compression libraries:

- zlib: The default compression library. A classic in terms of compression, achieves good compression rates but is somewhat slow.
- lzo: Fast compression and decompression.
- bzip2: Good compression rates.
- blosc: Fast compression and decompression.

Support for alternative blosc compressors:

– blosc:blosclz This is the default compressor for `blosc`
– blosc:lz4: A compact, very popular and fast compressor.
– blosc:lz4hc: A tweaked version of LZ4, produces better compression ratios at the expense of speed.
– blosc:snappy: A popular compressor used in many places.
– blosc:zlib: A classic; somewhat slower than the previous ones, but achieving better compression ratios.
– blosc:zstd: An extremely well balanced codec; it provides the best compression ratios among the others above, and at reasonably fast speed.

If `complib` is defined as something other than the listed libraries a `ValueError` exception is issued.

Note: If the library specified with the `complib` option is missing on your platform, compression defaults to `zlib` without further ado.

Enable compression for all objects within the file:

```
store_compressed = pd.HDFStore('store_compressed.h5', complevel=9,
                               complib='blosc:blosclz')
```

Or on-the-fly compression (this only applies to tables) in stores where compression is not enabled:

```
store.append('df', df, complib='zlib', complevel=5)
```

ptrepack

`PyTables` offers better write performance when tables are compressed after they are written, as opposed to turning on compression at the very beginning. You can use the supplied `PyTables` utility `ptrepack`. In addition, `ptrepack` can change compression levels after the fact.

```
ptrepack --chunkshape=auto --propindexes --complevel=9 --complib=blosc in.h5 out.h5
```

Furthermore `ptrepack in.h5 out.h5` will *repack* the file to allow you to reuse previously deleted space. Alternatively, one can simply remove the file and write again, or use the `copy` method.

Caveats

Warning: `HDFStore` is **not-threadsafe for writing**. The underlying `PyTables` only supports concurrent reads (via threading or processes). If you need reading and writing *at the same time*, you need to serialize these operations in a single thread in a single process. You will corrupt your data otherwise. See the (GH2397) for more information.

- If you use locks to manage write access between multiple processes, you may want to use `fsync()` before releasing write locks. For convenience you can use `store.flush(fsync=True)` to do this for you.
- Once a `table` is created columns (DataFrame) are fixed; only exactly the same columns can be appended

- Be aware that timezones (e.g., `pytz.timezone('US/Eastern')`) are not necessarily equal across timezone versions. So if data is localized to a specific timezone in the HDFStore using one version of a timezone library and that data is updated with another version, the data will be converted to UTC since these timezones are not considered equal. Either use the same version of timezone library or use `tz_convert` with the updated timezone definition.

> **Warning:** `PyTables` will show a `NaturalNameWarning` if a column name cannot be used as an attribute selector. *Natural* identifiers contain only letters, numbers, and underscores, and may not begin with a number. Other identifiers cannot be used in a `where` clause and are generally a bad idea.

DataTypes

`HDFStore` will map an object dtype to the `PyTables` underlying dtype. This means the following types are known to work:

Type	Represents missing values
floating : `float64, float32, float16`	`np.nan`
integer : `int64, int32, int8, uint64,uint32, uint8`	
boolean	
`datetime64[ns]`	`NaT`
`timedelta64[ns]`	`NaT`
categorical : see the section below	
object : `strings`	`np.nan`

`unicode` columns are not supported, and **WILL FAIL**.

Categorical data

You can write data that contains `category` dtypes to a `HDFStore`. Queries work the same as if it was an object array. However, the `category` dtyped data is stored in a more efficient manner.

```
In [473]: dfcat = pd.DataFrame({'A': pd.Series(list('aabbcdba')).astype('category'),
   .....:                       'B': np.random.randn(8)})
   .....:

In [474]: dfcat
Out[474]:
   A         B
0  a  0.477849
1  a  0.283128
2  b -2.045700
3  b -0.338206
4  c -0.423113
5  d  2.314361
6  b -0.033100
7  a -0.965461

In [475]: dfcat.dtypes
Out[475]:
A    category
B     float64
```

(continues on next page)

(continued from previous page)

```
dtype: object

In [476]: cstore = pd.HDFStore('cats.h5', mode='w')

In [477]: cstore.append('dfcat', dfcat, format='table', data_columns=['A'])

In [478]: result = cstore.select('dfcat', where="A in ['b', 'c']")

In [479]: result
Out[479]:
   A         B
2  b -2.045700
3  b -0.338206
4  c -0.423113
6  b -0.033100

In [480]: result.dtypes
Out[480]:
A    category
B     float64
dtype: object
```

String columns

min_itemsize

The underlying implementation of `HDFStore` uses a fixed column width (itemsize) for string columns. A string column itemsize is calculated as the maximum of the length of data (for that column) that is passed to the `HDFStore`, **in the first append**. Subsequent appends, may introduce a string for a column **larger** than the column can hold, an Exception will be raised (otherwise you could have a silent truncation of these columns, leading to loss of information). In the future we may relax this and allow a user-specified truncation to occur.

Pass `min_itemsize` on the first table creation to a-priori specify the minimum length of a particular string column. `min_itemsize` can be an integer, or a dict mapping a column name to an integer. You can pass `values` as a key to allow all *indexables* or *data_columns* to have this min_itemsize.

Passing a `min_itemsize` dict will cause all passed columns to be created as *data_columns* automatically.

Note: If you are not passing any `data_columns`, then the `min_itemsize` will be the maximum of the length of any string passed

```
In [481]: dfs = pd.DataFrame({'A': 'foo', 'B': 'bar'}, index=list(range(5)))

In [482]: dfs
Out[482]:
     A    B
0  foo  bar
1  foo  bar
2  foo  bar
3  foo  bar
4  foo  bar

# A and B have a size of 30
```

(continues on next page)

(continued from previous page)

```
In [483]: store.append('dfs', dfs, min_itemsize=30)

In [484]: store.get_storer('dfs').table
Out[484]:
/dfs/table (Table(5,)) ''
  description := {
  "index": Int64Col(shape=(), dflt=0, pos=0),
  "values_block_0": StringCol(itemsize=30, shape=(2,), dflt=b'', pos=1)}
  byteorder := 'little'
  chunkshape := (963,)
  autoindex := True
  colindexes := {
    "index": Index(6, medium, shuffle, zlib(1)).is_csi=False}

# A is created as a data_column with a size of 30
# B is size is calculated
In [485]: store.append('dfs2', dfs, min_itemsize={'A': 30})

In [486]: store.get_storer('dfs2').table
Out[486]:
/dfs2/table (Table(5,)) ''
  description := {
  "index": Int64Col(shape=(), dflt=0, pos=0),
  "values_block_0": StringCol(itemsize=3, shape=(1,), dflt=b'', pos=1),
  "A": StringCol(itemsize=30, shape=(), dflt=b'', pos=2)}
  byteorder := 'little'
  chunkshape := (1598,)
  autoindex := True
  colindexes := {
    "index": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "A": Index(6, medium, shuffle, zlib(1)).is_csi=False}
```

nan_rep

String columns will serialize a `np.nan` (a missing value) with the `nan_rep` string representation. This defaults to the string value `nan`. You could inadvertently turn an actual `nan` value into a missing value.

```
In [487]: dfss = pd.DataFrame({'A': ['foo', 'bar', 'nan']})

In [488]: dfss
Out[488]:
     A
0  foo
1  bar
2  nan

In [489]: store.append('dfss', dfss)

In [490]: store.select('dfss')
Out[490]:
     A
0  foo
1  bar
2  NaN

# here you need to specify a different nan rep
In [491]: store.append('dfss2', dfss, nan_rep='_nan_')
```

(continues on next page)

(continued from previous page)

```
In [492]: store.select('dfss2')
Out[492]:
     A
0  foo
1  bar
2  nan
```

External compatibility

`HDFStore` writes `table` format objects in specific formats suitable for producing loss-less round trips to pandas objects. For external compatibility, `HDFStore` can read native `PyTables` format tables.

It is possible to write an `HDFStore` object that can easily be imported into `R` using the `rhdf5` library (Package website). Create a table format store like this:

```
In [493]: df_for_r = pd.DataFrame({"first": np.random.rand(100),
   .....:                          "second": np.random.rand(100),
   .....:                          "class": np.random.randint(0, 2, (100, ))},
   .....:                         index=range(100))
   .....:

In [494]: df_for_r.head()
Out[494]:
      first    second  class
0  0.864919  0.852910      0
1  0.030579  0.412962      1
2  0.015226  0.978410      0
3  0.498512  0.686761      0
4  0.232163  0.328185      1

In [495]: store_export = pd.HDFStore('export.h5')

In [496]: store_export.append('df_for_r', df_for_r, data_columns=df_dc.columns)

In [497]: store_export
Out[497]:
<class 'pandas.io.pytables.HDFStore'>
File path: export.h5
```

In R this file can be read into a `data.frame` object using the `rhdf5` library. The following example function reads the corresponding column names and data values from the values and assembles them into a `data.frame`:

```
# Load values and column names for all datasets from corresponding nodes and
# insert them into one data.frame object.

library(rhdf5)

loadhdf5data <- function(h5File) {

listing <- h5ls(h5File)
# Find all data nodes, values are stored in *_values and corresponding column
# titles in *_items
data_nodes <- grep("_values", listing$name)
name_nodes <- grep("_items", listing$name)
```

(continues on next page)

(continued from previous page)

```
data_paths = paste(listing$group[data_nodes], listing$name[data_nodes], sep = "/")
name_paths = paste(listing$group[name_nodes], listing$name[name_nodes], sep = "/")
columns = list()
for (idx in seq(data_paths)) {
  # NOTE: matrices returned by h5read have to be transposed to obtain
  # required Fortran order!
  data <- data.frame(t(h5read(h5File, data_paths[idx])))
  names <- t(h5read(h5File, name_paths[idx]))
  entry <- data.frame(data)
  colnames(entry) <- names
  columns <- append(columns, entry)
}

data <- data.frame(columns)

return(data)
}
```

Now you can import the `DataFrame` into R:

```
> data = loadhdf5data("transfer.hdf5")
> head(data)
         first    second class
1 0.4170220047 0.3266449     0
2 0.7203244934 0.5270581     0
3 0.0001143748 0.8859421     1
4 0.3023325726 0.3572698     1
5 0.1467558908 0.9085352     1
6 0.0923385948 0.6233601     1
```

Note: The R function lists the entire HDF5 file's contents and assembles the `data.frame` object from all matching nodes, so use this only as a starting point if you have stored multiple `DataFrame` objects to a single HDF5 file.

Performance

- `tables` format come with a writing performance penalty as compared to `fixed` stores. The benefit is the ability to append/delete and query (potentially very large amounts of data). Write times are generally longer as compared with regular stores. Query times can be quite fast, especially on an indexed axis.
- You can pass `chunksize=<int>` to `append`, specifying the write chunksize (default is 50000). This will significantly lower your memory usage on writing.
- You can pass `expectedrows=<int>` to the first `append`, to set the TOTAL number of rows that `PyTables` will expect. This will optimize read/write performance.
- Duplicate rows can be written to tables, but are filtered out in selection (with the last items being selected; thus a table is unique on major, minor pairs)
- A `PerformanceWarning` will be raised if you are attempting to store types that will be pickled by PyTables (rather than stored as endemic types). See Here for more information and some solutions.

2.1.11 Feather

Feather provides binary columnar serialization for data frames. It is designed to make reading and writing data frames efficient, and to make sharing data across data analysis languages easy.

Feather is designed to faithfully serialize and de-serialize DataFrames, supporting all of the pandas dtypes, including extension dtypes such as categorical and datetime with tz.

Several caveats.

- This is a newer library, and the format, though stable, is not guaranteed to be backward compatible to the earlier versions.
- The format will NOT write an `Index`, or `MultiIndex` for the `DataFrame` and will raise an error if a non-default one is provided. You can `.reset_index()` to store the index or `.reset_index(drop=True)` to ignore it.
- Duplicate column names and non-string columns names are not supported
- Non supported types include `Period` and actual Python object types. These will raise a helpful error message on an attempt at serialization.

See the Full Documentation.

```
In [498]: df = pd.DataFrame({'a': list('abc'),
   .....:                    'b': list(range(1, 4)),
   .....:                    'c': np.arange(3, 6).astype('u1'),
   .....:                    'd': np.arange(4.0, 7.0, dtype='float64'),
   .....:                    'e': [True, False, True],
   .....:                    'f': pd.Categorical(list('abc')),
   .....:                    'g': pd.date_range('20130101', periods=3),
   .....:                    'h': pd.date_range('20130101', periods=3, tz='US/Eastern
↪'),
   .....:                    'i': pd.date_range('20130101', periods=3, freq='ns')})
   .....:

In [499]: df
Out[499]: 
   a  b  c    d      e  f          g                         h                    ␣
↪     i
0  a  1  3  4.0   True  a 2013-01-01 2013-01-01 00:00:00-05:00 2013-01-01 00:00:00.
↪000000000
1  b  2  4  5.0  False  b 2013-01-02 2013-01-02 00:00:00-05:00 2013-01-01 00:00:00.
↪000000001
2  c  3  5  6.0   True  c 2013-01-03 2013-01-03 00:00:00-05:00 2013-01-01 00:00:00.
↪000000002

In [500]: df.dtypes
Out[500]: 
a                        object
b                         int64
c                         uint8
d                       float64
e                          bool
f                      category
g                datetime64[ns]
h    datetime64[ns, US/Eastern]
i                datetime64[ns]
dtype: object
```

Write to a feather file.

```
In [501]: df.to_feather('example.feather')
```

Read from a feather file.

```
In [502]: result = pd.read_feather('example.feather')

In [503]: result
Out[503]:
   a  b  c    d      e  f          g                         h                      ␣
↪     i
0  a  1  3  4.0   True  a 2013-01-01 2013-01-01 00:00:00-05:00 2013-01-01 00:00:00.
↪000000000
1  b  2  4  5.0  False  b 2013-01-02 2013-01-02 00:00:00-05:00 2013-01-01 00:00:00.
↪000000001
2  c  3  5  6.0   True  c 2013-01-03 2013-01-03 00:00:00-05:00 2013-01-01 00:00:00.
↪000000002

# we preserve dtypes
In [504]: result.dtypes
Out[504]:
a                        object
b                         int64
c                         uint8
d                       float64
e                          bool
f                      category
g                datetime64[ns]
h    datetime64[ns, US/Eastern]
i                datetime64[ns]
dtype: object
```

2.1.12 Parquet

New in version 0.21.0.

Apache Parquet provides a partitioned binary columnar serialization for data frames. It is designed to make reading and writing data frames efficient, and to make sharing data across data analysis languages easy. Parquet can use a variety of compression techniques to shrink the file size as much as possible while still maintaining good read performance.

Parquet is designed to faithfully serialize and de-serialize `DataFrame` s, supporting all of the pandas dtypes, including extension dtypes such as datetime with tz.

Several caveats.

- Duplicate column names and non-string columns names are not supported.
- The `pyarrow` engine always writes the index to the output, but `fastparquet` only writes non-default indexes. This extra column can cause problems for non-Pandas consumers that are not expecting it. You can force including or omitting indexes with the `index` argument, regardless of the underlying engine.
- Index level names, if specified, must be strings.
- In the `pyarrow` engine, categorical dtypes for non-string types can be serialized to parquet, but will de-serialize as their primitive dtype.
- The `pyarrow` engine preserves the `ordered` flag of categorical dtypes with string types. `fastparquet` does not preserve the `ordered` flag.

- Non supported types include `Interval` and actual Python object types. These will raise a helpful error message on an attempt at serialization. `Period` type is supported with pyarrow >= 0.16.0.
- The `pyarrow` engine preserves extension data types such as the nullable integer and string data type (requiring pyarrow >= 0.16.0, and requiring the extension type to implement the needed protocols, see the *extension types documentation*).

You can specify an `engine` to direct the serialization. This can be one of `pyarrow`, or `fastparquet`, or `auto`. If the engine is NOT specified, then the `pd.options.io.parquet.engine` option is checked; if this is also `auto`, then `pyarrow` is tried, and falling back to `fastparquet`.

See the documentation for pyarrow and fastparquet.

Note: These engines are very similar and should read/write nearly identical parquet format files. Currently `pyarrow` does not support timedelta data, `fastparquet>=0.1.4` supports timezone aware datetimes. These libraries differ by having different underlying dependencies (`fastparquet` by using `numba`, while `pyarrow` uses a c-library).

```
In [505]: df = pd.DataFrame({'a': list('abc'),
   .....:                    'b': list(range(1, 4)),
   .....:                    'c': np.arange(3, 6).astype('u1'),
   .....:                    'd': np.arange(4.0, 7.0, dtype='float64'),
   .....:                    'e': [True, False, True],
   .....:                    'f': pd.date_range('20130101', periods=3),
   .....:                    'g': pd.date_range('20130101', periods=3, tz='US/Eastern
↪'),
   .....:                    'h': pd.Categorical(list('abc')),
   .....:                    'i': pd.Categorical(list('abc'), ordered=True)})
   .....:

In [506]: df
Out[506]:
   a  b  c    d      e          f                         g  h  i
0  a  1  3  4.0   True 2013-01-01 2013-01-01 00:00:00-05:00  a  a
1  b  2  4  5.0  False 2013-01-02 2013-01-02 00:00:00-05:00  b  b
2  c  3  5  6.0   True 2013-01-03 2013-01-03 00:00:00-05:00  c  c

In [507]: df.dtypes
Out[507]:
a                        object
b                         int64
c                         uint8
d                       float64
e                          bool
f                datetime64[ns]
g    datetime64[ns, US/Eastern]
h                      category
i                      category
dtype: object
```

Write to a parquet file.

```
In [508]: df.to_parquet('example_pa.parquet', engine='pyarrow')

In [509]: df.to_parquet('example_fp.parquet', engine='fastparquet')
```

Read from a parquet file.

```
In [510]: result = pd.read_parquet('example_fp.parquet', engine='fastparquet')

In [511]: result = pd.read_parquet('example_pa.parquet', engine='pyarrow')

In [512]: result.dtypes
Out[512]:
a                        object
b                         int64
c                         uint8
d                       float64
e                          bool
f                datetime64[ns]
g    datetime64[ns, US/Eastern]
h                      category
i                      category
dtype: object
```

Read only certain columns of a parquet file.

```
In [513]: result = pd.read_parquet('example_fp.parquet',
   .....:                          engine='fastparquet', columns=['a', 'b'])
   .....:

In [514]: result = pd.read_parquet('example_pa.parquet',
   .....:                          engine='pyarrow', columns=['a', 'b'])
   .....:

In [515]: result.dtypes
Out[515]:
a    object
b     int64
dtype: object
```

Handling indexes

Serializing a `DataFrame` to parquet may include the implicit index as one or more columns in the output file. Thus, this code:

```
In [516]: df = pd.DataFrame({'a': [1, 2], 'b': [3, 4]})

In [517]: df.to_parquet('test.parquet', engine='pyarrow')
```

creates a parquet file with *three* columns if you use `pyarrow` for serialization: `a`, `b`, and `__index_level_0__`. If you're using `fastparquet`, the index may or may not be written to the file.

This unexpected extra column causes some databases like Amazon Redshift to reject the file, because that column doesn't exist in the target table.

If you want to omit a dataframe's indexes when writing, pass `index=False` to *`to_parquet()`*:

```
In [518]: df.to_parquet('test.parquet', index=False)
```

This creates a parquet file with just the two expected columns, `a` and `b`. If your `DataFrame` has a custom index, you won't get it back when you load this file into a `DataFrame`.

Passing `index=True` will *always* write the index, even if that's not the underlying engine's default behavior.

Partitioning Parquet files

New in version 0.24.0.

Parquet supports partitioning of data based on the values of one or more columns.

```
In [519]: df = pd.DataFrame({'a': [0, 0, 1, 1], 'b': [0, 1, 0, 1]})

In [520]: df.to_parquet(path='test', engine='pyarrow',
   .....:               partition_cols=['a'], compression=None)
   .....:
```

The *path* specifies the parent directory to which data will be saved. The *partition_cols* are the column names by which the dataset will be partitioned. Columns are partitioned in the order they are given. The partition splits are determined by the unique values in the partition columns. The above example creates a partitioned dataset that may look like:

```
test
├── a=0
│   ├── 0bac803e32dc42ae83fddfd029cbdebc.parquet
│   └──  ...
└── a=1
    ├── e6ab24a4f45147b49b54a662f0c412a3.parquet
    └── ...
```

2.1.13 ORC

New in version 1.0.0.

Similar to the *parquet* format, the ORC Format is a binary columnar serialization for data frames. It is designed to make reading data frames efficient. Pandas provides *only* a reader for the ORC format, `read_orc()`. This requires the pyarrow library.

2.1.14 SQL queries

The `pandas.io.sql` module provides a collection of query wrappers to both facilitate data retrieval and to reduce dependency on DB-specific API. Database abstraction is provided by SQLAlchemy if installed. In addition you will need a driver library for your database. Examples of such drivers are psycopg2 for PostgreSQL or pymysql for MySQL. For SQLite this is included in Python's standard library by default. You can find an overview of supported drivers for each SQL dialect in the SQLAlchemy docs.

If SQLAlchemy is not installed, a fallback is only provided for sqlite (and for mysql for backwards compatibility, but this is deprecated and will be removed in a future version). This mode requires a Python database adapter which respect the Python DB-API.

See also some *cookbook examples* for some advanced strategies.

The key functions are:

`read_sql_table`(table_name, con[, schema, ...])	Read SQL database table into a DataFrame.
`read_sql_query`(sql, con[, index_col, ...])	Read SQL query into a DataFrame.
`read_sql`(sql, con[, index_col, ...])	Read SQL query or database table into a DataFrame.
`DataFrame.to_sql`(self, name, con[, schema, ...])	Write records stored in a DataFrame to a SQL database.

Note: The function `read_sql()` is a convenience wrapper around `read_sql_table()` and *read_sql_query()* (and for backward compatibility) and will delegate to specific function depending on the provided input (database table name or sql query). Table names do not need to be quoted if they have special characters.

In the following example, we use the SQlite SQL database engine. You can use a temporary SQLite database where data are stored in "memory".

To connect with SQLAlchemy you use the `create_engine()` function to create an engine object from database URI. You only need to create the engine once per database you are connecting to. For more information on `create_engine()` and the URI formatting, see the examples below and the SQLAlchemy documentation

```
In [521]: from sqlalchemy import create_engine

# Create your engine.
In [522]: engine = create_engine('sqlite:///:memory:')
```

If you want to manage your own connections you can pass one of those instead:

```
with engine.connect() as conn, conn.begin():
    data = pd.read_sql_table('data', conn)
```

Writing DataFrames

Assuming the following data is in a `DataFrame` `data`, we can insert it into the database using *to_sql()*.

id	Date	Col_1	Col_2	Col_3
26	2012-10-18	X	25.7	True
42	2012-10-19	Y	-12.4	False
63	2012-10-20	Z	5.73	True

```
In [523]: data
Out[523]:
   id        Date Col_1  Col_2  Col_3
0  26 2010-10-18     X  27.50   True
1  42 2010-10-19     Y -12.50  False
2  63 2010-10-20     Z   5.73   True

In [524]: data.to_sql('data', engine)
```

With some databases, writing large DataFrames can result in errors due to packet size limitations being exceeded. This can be avoided by setting the `chunksize` parameter when calling `to_sql`. For example, the following writes `data` to the database in batches of 1000 rows at a time:

```
In [525]: data.to_sql('data_chunked', engine, chunksize=1000)
```

SQL data types

`to_sql()` will try to map your data to an appropriate SQL data type based on the dtype of the data. When you have columns of dtype `object`, pandas will try to infer the data type.

You can always override the default type by specifying the desired SQL type of any of the columns by using the `dtype` argument. This argument needs a dictionary mapping column names to SQLAlchemy types (or strings for the sqlite3 fallback mode). For example, specifying to use the sqlalchemy `String` type instead of the default `Text` type for string columns:

```
In [526]: from sqlalchemy.types import String

In [527]: data.to_sql('data_dtype', engine, dtype={'Col_1': String})
```

Note: Due to the limited support for timedelta's in the different database flavors, columns with type `timedelta64` will be written as integer values as nanoseconds to the database and a warning will be raised.

Note: Columns of `category` dtype will be converted to the dense representation as you would get with `np.asarray(categorical)` (e.g. for string categories this gives an array of strings). Because of this, reading the database table back in does **not** generate a categorical.

Datetime data types

Using SQLAlchemy, `to_sql()` is capable of writing datetime data that is timezone naive or timezone aware. However, the resulting data stored in the database ultimately depends on the supported data type for datetime data of the database system being used.

The following table lists supported data types for datetime data for some common databases. Other database dialects may have different data types for datetime data.

Database	SQL Datetime Types	Timezone Support
SQLite	`TEXT`	No
MySQL	`TIMESTAMP` or `DATETIME`	No
PostgreSQL	`TIMESTAMP` or `TIMESTAMP WITH TIME ZONE`	Yes

When writing timezone aware data to databases that do not support timezones, the data will be written as timezone naive timestamps that are in local time with respect to the timezone.

`read_sql_table()` is also capable of reading datetime data that is timezone aware or naive. When reading `TIMESTAMP WITH TIME ZONE` types, pandas will convert the data to UTC.

Insertion method

New in version 0.24.0.

The parameter `method` controls the SQL insertion clause used. Possible values are:

- `None`: Uses standard SQL `INSERT` clause (one per row).
- `'multi'`: Pass multiple values in a single `INSERT` clause. It uses a *special* SQL syntax not supported by all backends. This usually provides better performance for analytic databases like *Presto* and *Redshift*, but has worse performance for traditional SQL backend if the table contains many columns. For more information check the SQLAlchemy documention.
- callable with signature `(pd_table, conn, keys, data_iter)`: This can be used to implement a more performant insertion method based on specific backend dialect features.

Example of a callable using PostgreSQL COPY clause:

```
# Alternative to_sql() *method* for DBs that support COPY FROM
import csv
from io import StringIO

def psql_insert_copy(table, conn, keys, data_iter):
    """
    Execute SQL statement inserting data

    Parameters
    ----------
    table : pandas.io.sql.SQLTable
    conn : sqlalchemy.engine.Engine or sqlalchemy.engine.Connection
    keys : list of str
        Column names
    data_iter : Iterable that iterates the values to be inserted
    """
    # gets a DBAPI connection that can provide a cursor
    dbapi_conn = conn.connection
    with dbapi_conn.cursor() as cur:
        s_buf = StringIO()
        writer = csv.writer(s_buf)
        writer.writerows(data_iter)
        s_buf.seek(0)

        columns = ', '.join('"{}"'.format(k) for k in keys)
        if table.schema:
            table_name = '{}.{}'.format(table.schema, table.name)
        else:
            table_name = table.name

        sql = 'COPY {} ({}) FROM STDIN WITH CSV'.format(
            table_name, columns)
        cur.copy_expert(sql=sql, file=s_buf)
```

Reading tables

read_sql_table() will read a database table given the table name and optionally a subset of columns to read.

Note: In order to use *read_sql_table()*, you **must** have the SQLAlchemy optional dependency installed.

```
In [528]: pd.read_sql_table('data', engine)
Out[528]:
   index  id        Date Col_1  Col_2  Col_3
0      0  26 2010-10-18     X  27.50   True
1      1  42 2010-10-19     Y -12.50  False
2      2  63 2010-10-20     Z   5.73   True
```

Note: Note that pandas infers column dtypes from query outputs, and not by looking up data types in the physical database schema. For example, assume `userid` is an integer column in a table. Then, intuitively, `select userid ...` will return integer-valued series, while `select cast(userid as text) ...` will return object-valued (str) series. Accordingly, if the query output is empty, then all resulting columns will be returned as object-valued (since they are most general). If you foresee that your query will sometimes generate an empty result, you may want to explicitly typecast afterwards to ensure dtype integrity.

You can also specify the name of the column as the `DataFrame` index, and specify a subset of columns to be read.

```
In [529]: pd.read_sql_table('data', engine, index_col='id')
Out[529]:
    index        Date Col_1  Col_2  Col_3
id
26      0 2010-10-18     X  27.50   True
42      1 2010-10-19     Y -12.50  False
63      2 2010-10-20     Z   5.73   True

In [530]: pd.read_sql_table('data', engine, columns=['Col_1', 'Col_2'])
Out[530]:
  Col_1  Col_2
0     X  27.50
1     Y -12.50
2     Z   5.73
```

And you can explicitly force columns to be parsed as dates:

```
In [531]: pd.read_sql_table('data', engine, parse_dates=['Date'])
Out[531]:
   index  id        Date Col_1  Col_2  Col_3
0      0  26 2010-10-18     X  27.50   True
1      1  42 2010-10-19     Y -12.50  False
2      2  63 2010-10-20     Z   5.73   True
```

If needed you can explicitly specify a format string, or a dict of arguments to pass to *pandas.to_datetime()*:

```
pd.read_sql_table('data', engine, parse_dates={'Date': '%Y-%m-%d'})
pd.read_sql_table('data', engine,
                  parse_dates={'Date': {'format': '%Y-%m-%d %H:%M:%S'}})
```

You can check if a table exists using `has_table()`

Schema support

Reading from and writing to different schema's is supported through the `schema` keyword in the *read_sql_table()* and *to_sql()* functions. Note however that this depends on the database flavor (sqlite does not have schema's). For example:

```
df.to_sql('table', engine, schema='other_schema')
pd.read_sql_table('table', engine, schema='other_schema')
```

Querying

You can query using raw SQL in the *read_sql_query()* function. In this case you must use the SQL variant appropriate for your database. When using SQLAlchemy, you can also pass SQLAlchemy Expression language constructs, which are database-agnostic.

```
In [532]: pd.read_sql_query('SELECT * FROM data', engine)
Out[532]:
   index  id                        Date Col_1  Col_2  Col_3
0      0  26  2010-10-18 00:00:00.000000     X  27.50      1
1      1  42  2010-10-19 00:00:00.000000     Y -12.50      0
2      2  63  2010-10-20 00:00:00.000000     Z   5.73      1
```

Of course, you can specify a more "complex" query.

```
In [533]: pd.read_sql_query("SELECT id, Col_1, Col_2 FROM data WHERE id = 42;",␣
↪engine)
Out[533]:
   id Col_1  Col_2
0  42     Y  -12.5
```

The *read_sql_query()* function supports a `chunksize` argument. Specifying this will return an iterator through chunks of the query result:

```
In [534]: df = pd.DataFrame(np.random.randn(20, 3), columns=list('abc'))

In [535]: df.to_sql('data_chunks', engine, index=False)
```

```
In [536]: for chunk in pd.read_sql_query("SELECT * FROM data_chunks",
   .....:                                engine, chunksize=5):
   .....:     print(chunk)
   .....:
          a         b         c
0  0.092961 -0.674003  1.104153
1 -0.092732 -0.156246 -0.585167
2 -0.358119 -0.862331 -1.672907
3  0.550313 -1.507513 -0.617232
4  0.650576  1.033221  0.492464
          a         b         c
0 -1.627786 -0.692062  1.039548
1 -1.802313 -0.890905 -0.881794
2  0.630492  0.016739  0.014500
3 -0.438358  0.647275 -0.052075
4  0.673137  1.227539  0.203534
          a         b         c
0  0.861658  0.867852 -0.465016
```

(continues on next page)

(continued from previous page)

```
1  1.547012 -0.947189 -1.241043
2  0.070470  0.901320  0.937577
3  0.295770  1.420548 -0.005283
4 -1.518598 -0.730065  0.226497
          a         b         c
0 -2.061465  0.632115  0.853619
1  2.719155  0.139018  0.214557
2 -1.538924 -0.366973 -0.748801
3 -0.478137 -1.559153 -3.097759
4 -2.320335 -0.221090  0.119763
```

You can also run a plain query without creating a `DataFrame` with `execute()`. This is useful for queries that don't return values, such as INSERT. This is functionally equivalent to calling `execute` on the SQLAlchemy engine or db connection object. Again, you must use the SQL syntax variant appropriate for your database.

```
from pandas.io import sql
sql.execute('SELECT * FROM table_name', engine)
sql.execute('INSERT INTO table_name VALUES(?, ?, ?)', engine,
            params=[('id', 1, 12.2, True)])
```

Engine connection examples

To connect with SQLAlchemy you use the `create_engine()` function to create an engine object from database URI. You only need to create the engine once per database you are connecting to.

```
from sqlalchemy import create_engine

engine = create_engine('postgresql://scott:tiger@localhost:5432/mydatabase')

engine = create_engine('mysql+mysqldb://scott:tiger@localhost/foo')

engine = create_engine('oracle://scott:tiger@127.0.0.1:1521/sidname')

engine = create_engine('mssql+pyodbc://mydsn')

# sqlite://<nohostname>/<path>
# where <path> is relative:
engine = create_engine('sqlite:///foo.db')

# or absolute, starting with a slash:
engine = create_engine('sqlite:////absolute/path/to/foo.db')
```

For more information see the examples the SQLAlchemy documentation

Advanced SQLAlchemy queries

You can use SQLAlchemy constructs to describe your query.

Use `sqlalchemy.text()` to specify query parameters in a backend-neutral way

```
In [537]: import sqlalchemy as sa

In [538]: pd.read_sql(sa.text('SELECT * FROM data where Col_1=:col1'),
   .....:             engine, params={'col1': 'X'})
   .....:
Out[538]:
   index  id                        Date Col_1  Col_2  Col_3
0      0  26  2010-10-18 00:00:00.000000     X   27.5      1
```

If you have an SQLAlchemy description of your database you can express where conditions using SQLAlchemy expressions

```
In [539]: metadata = sa.MetaData()

In [540]: data_table = sa.Table('data', metadata,
   .....:                       sa.Column('index', sa.Integer),
   .....:                       sa.Column('Date', sa.DateTime),
   .....:                       sa.Column('Col_1', sa.String),
   .....:                       sa.Column('Col_2', sa.Float),
   .....:                       sa.Column('Col_3', sa.Boolean),
   .....:                       )
   .....:

In [541]: pd.read_sql(sa.select([data_table]).where(data_table.c.Col_3 is True),
↪engine)
Out[541]:
Empty DataFrame
Columns: [index, Date, Col_1, Col_2, Col_3]
Index: []
```

You can combine SQLAlchemy expressions with parameters passed to *read_sql()* using `sqlalchemy.bindparam()`

```
In [542]: import datetime as dt

In [543]: expr = sa.select([data_table]).where(data_table.c.Date > sa.bindparam('date
↪'))

In [544]: pd.read_sql(expr, engine, params={'date': dt.datetime(2010, 10, 18)})
Out[544]:
   index       Date Col_1  Col_2  Col_3
0      1 2010-10-19     Y -12.50  False
1      2 2010-10-20     Z   5.73   True
```

Sqlite fallback

The use of sqlite is supported without using SQLAlchemy. This mode requires a Python database adapter which respect the Python DB-API.

You can create connections like so:

```
import sqlite3
con = sqlite3.connect(':memory:')
```

And then issue the following queries:

```
data.to_sql('data', con)
pd.read_sql_query("SELECT * FROM data", con)
```

2.1.15 Google BigQuery

Warning: Starting in 0.20.0, pandas has split off Google BigQuery support into the separate package `pandas-gbq`. You can `pip install pandas-gbq` to get it.

The `pandas-gbq` package provides functionality to read/write from Google BigQuery.

pandas integrates with this external package. if `pandas-gbq` is installed, you can use the pandas methods `pd.read_gbq` and `DataFrame.to_gbq`, which will call the respective functions from `pandas-gbq`.

Full documentation can be found here.

2.1.16 Stata format

Writing to stata format

The method `to_stata()` will write a DataFrame into a .dta file. The format version of this file is always 115 (Stata 12).

```
In [545]: df = pd.DataFrame(np.random.randn(10, 2), columns=list('AB'))

In [546]: df.to_stata('stata.dta')
```

Stata data files have limited data type support; only strings with 244 or fewer characters, `int8`, `int16`, `int32`, `float32` and `float64` can be stored in `.dta` files. Additionally, *Stata* reserves certain values to represent missing data. Exporting a non-missing value that is outside of the permitted range in Stata for a particular data type will retype the variable to the next larger size. For example, `int8` values are restricted to lie between -127 and 100 in Stata, and so variables with values above 100 will trigger a conversion to `int16`. `nan` values in floating points data types are stored as the basic missing data type (`.` in *Stata*).

Note: It is not possible to export missing data values for integer data types.

The *Stata* writer gracefully handles other data types including `int64`, `bool`, `uint8`, `uint16`, `uint32` by casting to the smallest supported type that can represent the data. For example, data with a type of `uint8` will be cast to `int8` if all values are less than 100 (the upper bound for non-missing `int8` data in *Stata*), or, if values are outside of this range, the variable is cast to `int16`.

Warning: Conversion from `int64` to `float64` may result in a loss of precision if `int64` values are larger than 2**53.

Warning: `StataWriter` and `to_stata()` only support fixed width strings containing up to 244 characters, a limitation imposed by the version 115 dta file format. Attempting to write *Stata* dta files with strings longer than 244 characters raises a `ValueError`.

Reading from Stata format

The top-level function `read_stata` will read a dta file and return either a `DataFrame` or a `StataReader` that can be used to read the file incrementally.

```
In [547]: pd.read_stata('stata.dta')
Out[547]:
   index         A         B
0      0  0.608228  1.064810
1      1 -0.780506 -2.736887
2      2  0.143539  1.170191
3      3 -1.573076  0.075792
4      4 -1.722223 -0.774650
5      5  0.803627  0.221665
6      6  0.584637  0.147264
7      7  1.057825 -0.284136
8      8  0.912395  1.552808
9      9  0.189376 -0.109830
```

Specifying a `chunksize` yields a `StataReader` instance that can be used to read `chunksize` lines from the file at a time. The `StataReader` object can be used as an iterator.

```
In [548]: reader = pd.read_stata('stata.dta', chunksize=3)

In [549]: for df in reader:
   .....:     print(df.shape)
   .....:
(3, 3)
(3, 3)
(3, 3)
(1, 3)
```

For more fine-grained control, use `iterator=True` and specify `chunksize` with each call to `read()`.

```
In [550]: reader = pd.read_stata('stata.dta', iterator=True)

In [551]: chunk1 = reader.read(5)

In [552]: chunk2 = reader.read(5)
```

Currently the `index` is retrieved as a column.

The parameter `convert_categoricals` indicates whether value labels should be read and used to create a `Categorical` variable from them. Value labels can also be retrieved by the function `value_labels`, which requires `read()` to be called before use.

The parameter `convert_missing` indicates whether missing value representations in Stata should be preserved. If `False` (the default), missing values are represented as `np.nan`. If `True`, missing values are represented using `StataMissingValue` objects, and columns containing missing values will have `object` data type.

Note: *`read_stata()`* and `StataReader` support .dta formats 113-115 (Stata 10-12), 117 (Stata 13), and 118 (Stata 14).

Note: Setting `preserve_dtypes=False` will upcast to the standard pandas data types: `int64` for all integer types and `float64` for floating point data. By default, the Stata data types are preserved when importing.

Categorical data

`Categorical` data can be exported to *Stata* data files as value labeled data. The exported data consists of the underlying category codes as integer data values and the categories as value labels. *Stata* does not have an explicit equivalent to a `Categorical` and information about *whether* the variable is ordered is lost when exporting.

Warning: *Stata* only supports string value labels, and so `str` is called on the categories when exporting data. Exporting `Categorical` variables with non-string categories produces a warning, and can result a loss of information if the `str` representations of the categories are not unique.

Labeled data can similarly be imported from *Stata* data files as `Categorical` variables using the keyword argument `convert_categoricals` (`True` by default). The keyword argument `order_categoricals` (`True` by default) determines whether imported `Categorical` variables are ordered.

Note: When importing categorical data, the values of the variables in the *Stata* data file are not preserved since `Categorical` variables always use integer data types between `-1` and `n-1` where `n` is the number of categories. If the original values in the *Stata* data file are required, these can be imported by setting `convert_categoricals=False`, which will import original data (but not the variable labels). The original values can be matched to the imported categorical data since there is a simple mapping between the original *Stata* data values and the category codes of imported Categorical variables: missing values are assigned code `-1`, and the smallest original value is assigned `0`, the second smallest is assigned `1` and so on until the largest original value is assigned the code `n-1`.

Note: *Stata* supports partially labeled series. These series have value labels for some but not all data values. Importing a partially labeled series will produce a `Categorical` with string categories for the values that are labeled and numeric categories for values with no label.
