pandas 2.0 and beyond

Copy-on-Write, Arrow-backed DataFrames and more

Joris Van den Bossche (Voltron Data), Patrick Hoefler (Coiled)
PyConDE & PyData Berlin, April 17, 2023

https://github.com/jorisvandenbossche/
@jorisvdbossche/
https://github.com/phofl/

About Joris

Joris Van den Bossche

- Background: PhD bio-science engineer, air quality research
- Open source enthusiast: core developer of pandas, GeoPandas, Shapely, Apache Arrow, ...
- Currently working part-time at Voltron Data on Apache Arrow

twitter.com/jorisvdbossche github.com/jorisvandenbossche

About Patrick

Patrick Hoefler

- Background: M.Sc. in Mathematics
- Open source: pandas core dev
- Currently working at Coiled on Dask

https://github.com/phofl Website: https://phofl.github.io

pandas 2.0 and beyond

Copy-on-Write, Arrow-backed DataFrames and more

pandas 2.0 released on April 3rd!

New features:

- Index backed by all numerical NumPy dtypes
- Non-nanosecond datetime resolutions
- Consistent datetime parsing

New experimental features:

- Copy-on-Write option
- Arrow-backed DataFrames

And many more! See full release notes at https://pandas.pydata.org/docs/whatsnew/v2.0.0.html

Non-nanosecond resolution in Timestamps

pandas only supported Timestamps in nanosecond resolution up until 2.0

Non-nanosecond resolution in Timestamps

pandas only supported Timestamps in nanosecond resolution up until 2.0

→ Only dates between 1677 and 2262 can be represented

Non-nanosecond resolution in Timestamps

pandas only supported Timestamps in nanosecond resolution up until 2.0

→ Only dates between 1677 and 2262 can be represented

```
>>> pd.Timestamp("1000-01-01")
OutOfBoundsDatetime: Out of bounds nanosecond timestamp: 1000-01-01 00:00:00
```

Non-nanosecond resolution in Timestamps

pandas only supported Timestamps in nanosecond resolution up until 2.0

→ Only dates between 1677 and 2262 can be represented

```
>>> pd.Timestamp("1000-01-01")
OutOfBoundsDatetime: Out of bounds nanosecond timestamp: 1000-01-01 00:00:00
```

pandas 2.0 lifts this restriction!

Non-nanosecond resolution in Timestamps

pandas only supported Timestamps in nanosecond resolution up until 2.0

→ Only dates between 1677 and 2262 can be represented

```
>>> pd.Timestamp("1000-01-01")
OutOfBoundsDatetime: Out of bounds nanosecond timestamp: 1000-01-01 00:00:00
```

pandas 2.0 lifts this restriction!

Timestamps can be created in the following units:

- `seconds`
- `milliseconds`
- `microseconds`
- `nanoseconds`

How to enable the new resolutions

`date_range` doesn't yet support non-nano resolutions. But `as_unit` or `astype` can be used to convert between units:

How to enable the new resolutions

`date_range` doesn't yet support non-nano resolutions. But `as_unit` or `astype` can be used to convert between units:

```
>>> dr = pd.date_range("2020-01-01", periods=3, freq="D")
>>> dr
DatetimeIndex(['2020-01-01', '2020-01-02'], dtype='datetime64[ns]', freq='D')
```

How to enable the new resolutions

`date_range` doesn't yet support non-nano resolutions. But `as_unit` or `astype` can be used to convert between units:

```
>>> dr = pd.date_range("2020-01-01", periods=3, freq="D")
>>> dr
DatetimeIndex(['2020-01-01', '2020-01-02'], dtype='datetime64[ns]', freq='D')
```

```
>>> dr.as_unit("s")
DatetimeIndex(['2020-01-01', '2020-01-02'], dtype='datetime64[s]', freq='D')
```

How to enable the new resolutions

`date_range` doesn't yet support non-nano resolutions. But `as_unit` or `astype` can be used to convert between units:

```
>>> dr = pd.date_range("2020-01-01", periods=3, freq="D")
>>> dr
DatetimeIndex(['2020-01-01', '2020-01-02'], dtype='datetime64[ns]', freq='D')
```

```
>>> dr.as_unit("s")
DatetimeIndex(['2020-01-01', '2020-01-02'], dtype='datetime64[s]', freq='D')
```

```
>>> dr.astype("datetime64[s]")
DatetimeIndex(['2020-01-01', '2020-01-02'], dtype='datetime64[s]', freq='D')
```

How to enable the new resolutions

`date_range` doesn't yet support non-nano resolutions. But `as_unit` or `astype` can be used to convert between units:

The resolution of NumPy arrays is preserved:

```
>>> arr = np.array(['2007-07-13', '2006-01-13'], dtype='datetime64[ms]')
>>> arr
array(['2007-07-13T00:00:00.000', '2006-01-13T00:00:00.000'], dtype='datetime64[ms]')

>>> pd.Series(arr)
0   2007-07-13
1   2006-01-13
dtype: datetime64[ms]
```

Some caveats

Some caveats

- Non-nanosecond support is still new and is actively developed!

Some caveats

- Non-nanosecond support is still new and is actively developed!
- Not every part of the API support non-nanosecond resolutions yet (`date_range`).

Some caveats

- Non-nanosecond support is still new and is actively developed!
- Not every part of the API support non-nanosecond resolutions yet (`date_range`).
- Comparing two arrays with differing resolutions is still relatively slow.

PDEP-4: Consistent datetime parsing

Old behaviour: when not specifying a specific format, each value was being parsed independently:

```
>>> pd.to_datetime(['12-01-2000 00:00:00', '13-01-2000 00:00:00'])
DatetimeIndex(['2000-12-01', '2000-01-13'], dtype='datetime64[ns]', freq=None)
```

PDEP-4: Consistent datetime parsing

Old behaviour: when not specifying a specific format, each value was being parsed independently:

```
>>> pd.to_datetime(['12-01-2000 00:00:00', '13-01-2000 00:00:00'])
DatetimeIndex(['2000-12-01', '2000-01-13'], dtype='datetime64[ns]', freq=None)
```

New behaviour in pandas 2.0: if no `format` is specified, the format will be guessed from the first string and applied to all values

```
>>> pd.to_datetime(['12-01-2000 00:00:00', '13-01-2000 00:00:00'])
...
# ValueError: time data "13-01-2000 00:00:00" doesn't match format "%m-%d-%Y %H:%M:%S".
# You might want to try:
#    - passing `format` if your strings have a consistent format;
#    - passing `format='ISO8601'` if your strings are all ISO8601 but not necessarily in exact
#    - passing `format='mixed'`, and the format will be inferred for each element individually.
```

Full details: https://pandas.pydata.org/pdeps/0004-consistent-to-datetime-parsing.html

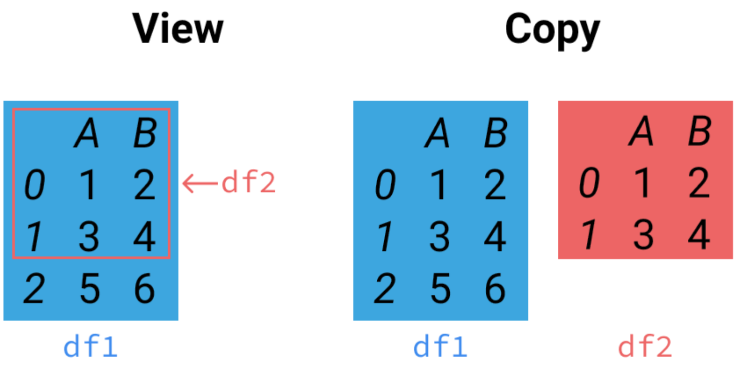
PDEP-7: Consistent copy/view semantics in pandas with Copy-on-Write

a.k.a. Getting rid of the SettingWithCopyWarning

Current situation: SettingWithCopyWarning

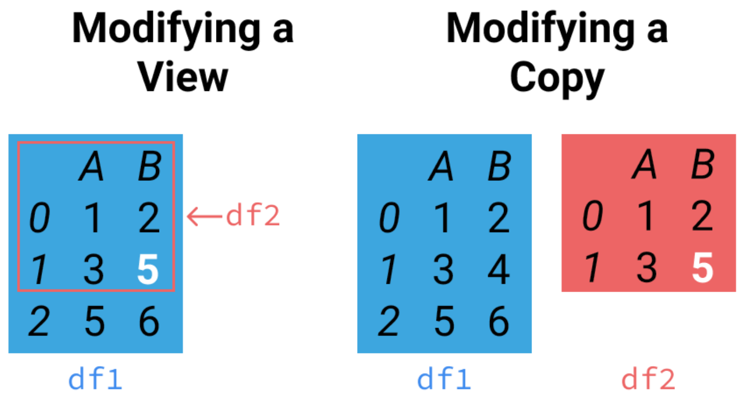
```
>>> df = pd.DataFrame({"A": [1, 2], "B": [3, 4], "C": [5, 6]})
>>> subset = df[df["A"] > 1]
>>> subset.loc[1, "C"] = 10

# SettingWithCopyWarning:
# A value is trying to be set on a copy of a slice from a DataFrame.
# Try using .loc[row_indexer,col_indexer] = value instead
#
# See the caveats in the documentation: ...
```

Current situation: copy vs view



Images from https://www.dataquest.io/blog/settingwithcopywarning/

Current situation: copy vs view



Images from https://www.dataquest.io/blog/settingwithcopywarning/

Current situation

Problems with the current copy / view semantics of pandas:

- This is confusing for many users
- You need to be aware of copy/view details of numpy
- You need defensive (and unncecessary) copying to avoid the warning

The SettingWithCopyWarning

To avoid "chained assignment (setitem)" pitfalls:

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [3, 4], "C": [5, 6]})
>>> df["C"][df["A"] > 1] = 10  # this works
>>> df[df["A"] > 1]["C"] = 10  # this doesn't work
```

The SettingWithCopyWarning

To avoid "chained assignment (setitem)" pitfalls:

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [3, 4], "C": [5, 6]})
>>> df["C"][df["A"] > 1] = 10  # this works
>>> df[df["A"] > 1]["C"] = 10  # this doesn't work
```

Rewriting the second case:

```
>>> temp = df[df["A"] > 1]
>>> temp["C"] = 10
```

The SettingWithCopyWarning

To avoid "chained assignment (setitem)" pitfalls:

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [3, 4], "C": [5, 6]})
>>> df["C"][df["A"] > 1] = 10  # this works
>>> df[df["A"] > 1]["C"] = 10  # this doesn't work
```

Rewriting the second case:

```
>>> temp = df[df["A"] > 1]
>>> temp["C"] = 10
```

Our previous example:

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [3, 4], "C": [5, 6]})
>>> subset = df[df["A"] > 1]
...
>>> subset["C"] = 10
```

The SettingWithCopyWarning

How to "solve" the warning?

The SettingWithCopyWarning

How to "solve" the warning?

I want do update `df` -> setitem in one go

```
>>> df[df["A"] > 1]["C"] = 10      # this doesn't work
>>> df.loc[df["A"] > 1, "C"] = 10  # this works
```

Or use `.assign(A=...)` method instead.

The SettingWithCopyWarning

How to "solve" the warning?

I want do update `df` -> setitem in one go

```
>>> df[df["A"] > 1]["C"] = 10      # this doesn't work
>>> df.loc[df["A"] > 1, "C"] = 10  # this works
```

Or use `.assign(A=...)` method instead.

I don't want to update `df` -> explicit (unnecessary) `copy()`

```
>>> subset = df[df["A"] > 1].copy()
...
>>> subset["C"] = 10
```

Current situation

Problems with the current copy / view semantics of pandas:

- This is confusing for many users
- You need to be aware of copy/view details of numpy
- **You need defensive (and unncecessary) copying to avoid the warning**

Can we do better?

A proposal for simplified behaviour using a single rule:

> *Any DataFrame or Series derived from another in any way (e.g. with an indexing operation) always* behaves as *a copy.*

Can we do better?

A proposal for simplified behaviour using a single rule:

> *Any DataFrame or Series derived from another in any way (e.g. with an indexing operation) always* behaves as *a copy.*

Or put differently, the implication is:

> *Mutating a DataFrame only changes the object itself, and not any other.*
>
> *If you want to change values in a DataFrame or Series, you can only do that by directly mutating the DataFrame/Series at hand.*

Can we do better?

A proposal for simplified behaviour using a single rule:

> *Any DataFrame or Series derived from another in any way (e.g. with an indexing operation) always* behaves as *a copy.*

Advantages:

- A simpler, more consistent user experience
- We can get rid of the SettingWithCopyWarning (since there is no confusion about whether we are mutating a view or a copy)
- We would no longer need defensive copying in many places in pandas, improving memory usage

Can we do better?

A proposal for simplified behaviour using a single rule:

> *Any DataFrame or Series derived from another in any way (e.g. with an indexing operation) always* behaves as *a copy.*

Advantages:

- **A simpler, more consistent user experience**
- We can get rid of the SettingWithCopyWarning (since there is no confusion about whether we are mutating a view or a copy)
- We would no longer need defensive copying in many places in pandas, improving memory usage

Previous example modifying a subset

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [3, 4], "C": [5, 6]})
>>> subset = df[df["A"] > 1]
>>> subset.loc[1, "C"] = 10
```

Did `df` change as well?

Previous example modifying a subset

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [3, 4], "C": [5, 6]})
>>> subset = df[df["A"] > 1]
>>> subset.loc[1, "C"] = 10
```

Did `df` change as well?

No, `subset` is a different object, so mutating `subset` does not change `df`.

And the answer is the same regardless how `subset` was created (selecting rows or columns, with a slice, mask, or list indexer, ..)

Can we do better?

A proposal for simplified behaviour using a single rule:

> *Any DataFrame or Series derived from another in any way (e.g. with an indexing operation) always* behaves as *a copy.*

Advantages:

- A simpler, more consistent user experience
- **We can get rid of the SettingWithCopyWarning** (since there is no confusion about whether we are mutating a view or a copy)
- We would no longer need defensive copying in many places in pandas, improving memory usage

The SettingWithCopyWarning

With current pandas (trying to update `df`):

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [3, 4], "C": [5, 6]})
# two examples of chained assignment
>>> df["C"][df["A"] > 1] = 10  # this works
>>> df[df["A"] > 1]["C"] = 10  # this doesn't work
```

The SettingWithCopyWarning

With current pandas (trying to update `df`):

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [3, 4], "C": [5, 6]})
# two examples of chained assignment
>>> df["C"][df["A"] > 1] = 10  # this works
>>> df[df["A"] > 1]["C"] = 10  # this doesn't work
```

With new behaviour: both examples don't work

```
>>> df["C"][df["A"] > 1] = 10  # this doesn't work
>>> df[df["A"] > 1]["C"] = 10  # this doesn't work
```

The SettingWithCopyWarning

With current pandas (trying to update `df`):

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [3, 4], "C": [5, 6]})
# two examples of chained assignment
>>> df["C"][df["A"] > 1] = 10  # this works
>>> df[df["A"] > 1]["C"] = 10  # this doesn't work
```

With new behaviour: both examples don't work

```
>>> df["C"][df["A"] > 1] = 10  # this doesn't work
>>> df[df["A"] > 1]["C"] = 10  # this doesn't work
```

```
>>> temp = df["C"]
>>> temp[temp["A"] > 1] = 10
```

Chained assignment will never work!

The SettingWithCopyWarning

With current pandas (trying to update `df`):

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [3, 4], "C": [5, 6]})
# two examples of chained assignment
>>> df["C"][df["A"] > 1] = 10  # this works
>>> df[df["A"] > 1]["C"] = 10  # this doesn't work
```

With new behaviour: chained assignment will never work -> we don't need the general warning.

But to help the transition, we can specifically warn about chained assignment not working:

```
>>> df["C"][df["A"] > 1] = 10

# ChainedAssignmentError: A value is trying to be set on a copy of a DataFrame
# or Series through chained assignment.
# When using the Copy-on-Write mode, such chained assignment never works ...
```

The SettingWithCopyWarning

With current pandas (not wanting to update `df`):

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [3, 4], "C": [5, 6]})
>>> subset = df[df["A"] > 1].copy()
...
>>> subset["C"] = 10
```

With new behaviour: additional `copy()` is no longer needed to avoid the warning.

Can we do better?

A proposal for simplified behaviour using a single rule:

> *Any DataFrame or Series derived from another in any way (e.g. with an indexing operation) always* behaves as *a copy.*

Advantages:

- A simpler, more consistent user experience
- We can get rid of the SettingWithCopyWarning (since there is no confusion about whether we are mutating a view or a copy)
- **We would no longer need defensive copying in many places in pandas, improving memory usage**

Avoiding copies with Copy-on-Write

Guarantee == "behaves as a copy"

The usage of view vs copy can become an internal implementation detail

➔ We can avoid copies by default using Copy-on-Write!

Avoiding copies with Copy-on-Write

Guarantee == "behaves as a copy"

The usage of view vs copy can become an internal implementation detail

➔ We can avoid copies by default using Copy-on-Write!

Small benchmark: create DataFrame of 2 million rows by 30 columns (mix of float, integer and string columns)

```
import pandas as pd
import numpy as np

N = 2_000_000
int_df = pd.DataFrame(np.random.randint(1, 100, (N, 10)), columns=[f"col_{i}" for i in range(1
float_df = pd.DataFrame(np.random.random((N, 10)), columns=[f"col_{i}" for i in range(10, 20)]
str_df = pd.DataFrame("a", index=range(N), columns=[f"col_{i}" for i in range(20, 30)])

df = pd.concat([int_df, float_df, str_df], axis=1)
```

Avoiding copies with Copy-on-Write

Guarantee == "behaves as a copy"

The usage of view vs copy can become an internal implementation detail

➔ We can avoid copies by default using Copy-on-Write!

```
%%timeit
(df.rename(columns={"col_1": "new_index"})
   .assign(sum_val=df["col_1"] + df["col_2"])
   .drop(columns=["col_10", "col_20"])
   .astype({"col_5": "int32"})
   .reset_index()
   .set_index("new_index")
)
```

```
2.45 s ± 293 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Avoiding copies with Copy-on-Write

Guarantee == "behaves as a copy"

The usage of view vs copy can become an internal implementation detail

➔ We can avoid copies by default using Copy-on-Write!

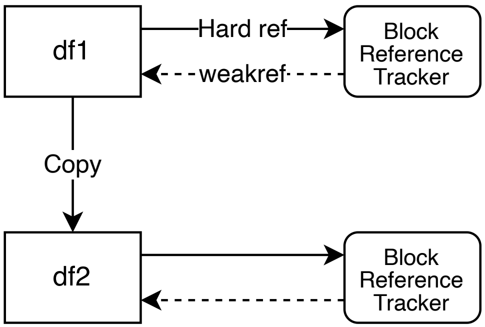
```
%%timeit
(df.rename(columns={"col_1": "new_index"})
   .assign(sum_val=df["col_1"] + df["col_2"])
   .drop(columns=["col_10", "col_20"])
   .astype({"col_5": "int32"})
   .reset_index()
   .set_index("new_index")
)
```

```
2.45 s ± 293 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

# with Copy-on-Write enabled
13.7 ms ± 286 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Hiding copy/view details with Copy-on-Write

When an operations makes an actual copy of the data
➔ each DataFrame references its own data

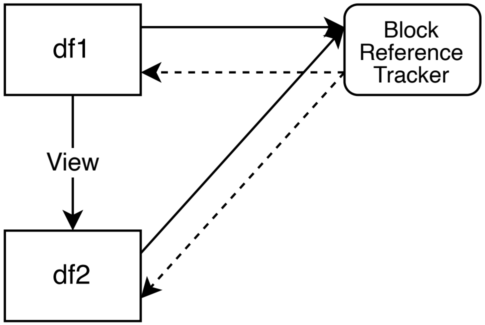
```
df2 = df1.copy()
```



Hiding copy/view details with Copy-on-Write

When an operations can use a view for the result
➔ both reference the same data

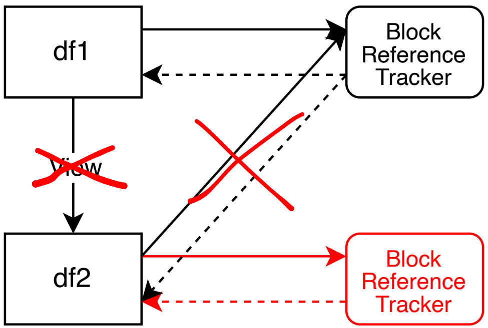
```
df2 = df1.reset_index()
```



Hiding copy/view details with Copy-on-Write

Modifying a view or its parent (`df1` or `df2`) will trigger a copy (a "copy on write")
➔ each DataFrame again owns its own memory.

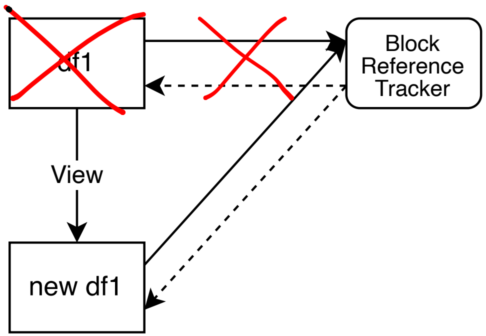
```
df2 = df1.reset_index()
df2.loc[1, "C"] = 10
```



Hiding copy/view details with Copy-on-Write

Do you want to avoid this copy when modifying `df2`, and no longer need `df1`? You can for example reassign to the same variable such that the original `df1` goes out of scope.

```
df1 = df1.reset_index()
```



How do I try this?

Enable it in pandas 2.0:

```
import pandas as pd
pd.options.mode.copy_on_write = True
```

- We encourage you to try it out!
- We expect it to become the default behaviour in pandas 3.0
- Blogposts:
  - https://jorisvandenbossche.github.io/blog/2022/04/07/pandas-copy-views/
  - https://medium.com/towards-data-science/a-solution-for-inconsistencies-in-indexing-operations-in-pandas-b76e10719744
- Full proposal: https://github.com/pandas-dev/pandas/pull/51463/

Feedback welcome!

Arrow-backed DataFrames

Arrow-backed DataFrames

Using PyArrow arrays to store the data of a DataFrame.

pandas meets APACHE ARROW

Apache Arrow *defines a language-independent columnar memory format for flat and hierarchical data, organized for efficient analytic operations on modern hardware like CPUs and GPUs.*

→ Check out Joris [talk about Arrow] on Wednesday!

ArrowDtype

`pd.ArrowDtype` or `f"{dtype}[pyarrow]"` creates Arrow-backed columns

ArrowDtype

`pd.ArrowDtype` or `f"{dtype}[pyarrow]"` creates Arrow-backed columns

```
>>> import pyarrow as pa
>>> pd.Series([1, 2, 3], dtype=pd.ArrowDtype(pa.int64()))
0    1
1    2
2    3
dtype: int64[pyarrow]

>>> pd.Series([1, 2, 3], dtype="int64[pyarrow]")
0    1
1    2
2    3
dtype: int64[pyarrow]
```

ArrowDtype

`pd.ArrowDtype` or `f"{dtype}[pyarrow]"` creates Arrow-backed columns

```
>>> import pyarrow as pa
>>> pd.Series([1, 2, 3], dtype=pd.ArrowDtype(pa.int64()))
0    1
1    2
2    3
dtype: int64[pyarrow]

>>> pd.Series([1, 2, 3], dtype="int64[pyarrow]")
0    1
1    2
2    3
dtype: int64[pyarrow]
```

These columns use the PyArrow memory layout and compute functionality.

Dispatching to pyarrow.compute

The ExtensionArray interface of pandas dispatches to [compute functions of PyArrow](#).

```
In [3]: ser = pd.Series(np.random.randint(1, 100, (5_000_000, )))
```

Dispatching to pyarrow.compute

The ExtensionArray interface of pandas dispatches to [compute functions of PyArrow](#).

```
In [3]: ser = pd.Series(np.random.randint(1, 100, (5_000_000, )))
```

```
In [4]: %timeit ser.unique()
10.6 ms ± 31.8 μs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Dispatching to pyarrow.compute

The ExtensionArray interface of pandas dispatches to [compute functions of PyArrow](#).

```
In [3]: ser = pd.Series(np.random.randint(1, 100, (5_000_000, )))
```

```
In [4]: %timeit ser.unique()
10.6 ms ± 31.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
In [5]: ser_arrow = ser.astype(pd.ArrowDtype(pa.int64()))

In [6]: %timeit ser_arrow.unique()
6.71 ms ± 6.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Dispatching to pyarrow.compute

The ExtensionArray interface of pandas dispatches to [compute functions of PyArrow](#).

```
In [3]: ser = pd.Series(np.random.randint(1, 100, (5_000_000, )))
```

```
In [4]: %timeit ser.unique()
10.6 ms ± 31.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
In [5]: ser_arrow = ser.astype(pd.ArrowDtype(pa.int64()))

In [6]: %timeit ser_arrow.unique()
6.71 ms ± 6.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

→ PyArrow can provide a significant performance improvement

Not every method of pandas supports the compute functionality of PyArrow yet.

PyArrow dtypes

PyArrow offers support for a wide variety of dtypes.

PyArrow dtypes

PyArrow offers support for a wide variety of dtypes.

Benefits

- Support of missing value indicators in every datatype

```
>>> pd.Series([1, None], dtype="int64[pyarrow]")
0       1
1    <NA>
dtype: int64[pyarrow]
```

PyArrow dtypes

PyArrow offers support for a wide variety of dtypes.

Benefits

- Support of missing value indicators in every datatype

```
>>> pd.Series([1, None], dtype="int64[pyarrow]")
0       1
1    <NA>
dtype: int64[pyarrow]
```

- An efficient string-datatype implementation

PyArrow dtypes

PyArrow offers support for a wide variety of dtypes.

Benefits

- Support of missing value indicators in every datatype

```
>>> pd.Series([1, None], dtype="int64[pyarrow]")
0       1
1    <NA>
dtype: int64[pyarrow]
```

- An efficient string-datatype implementation
- Bytes, decimal, explicit null-datatype, nested data and many more.

PyArrow string dtype

PyArrow offers fast and efficient in-memory string operations.

pandas implements PyArrow-based string operations through `"string[pyarrow]"` or `pd.ArrowDtype(pa.string())`.

PyArrow string dtype

PyArrow offers fast and efficient in-memory string operations.

pandas implements PyArrow-based string operations through `"string[pyarrow]"` or `pd.ArrowDtype(pa.string())`.

These implementations provide

- significantly improved performance compared to NumPy's object dtype
- smaller memory footprint

Let's look at some performance/memory comparisons

```python
import string
import random

import pandas as pd


def random_string() -> str:
    return "".join(random.choices(string.printable, k=random.randint(10, 100)))


ser_object = pd.Series([random_string() for _ in range(1_000_000)])
ser_string = ser_object.astype("string[pyarrow]")
```

Let's look at some performance/memory comparisons

Let's look at some performance/memory comparisons

str.length

```
In[1]: %timeit ser_object.str.len()
118 ms ± 260 μs per loop (mean ± std. dev. of 7 runs, 10 loops each)

In[2]: %timeit ser_string.str.len()
24.2 ms ± 187 μs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Let's look at some performance/memory comparisons

str.length

```
In[1]: %timeit ser_object.str.len()
118 ms ± 260 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

In[2]: %timeit ser_string.str.len()
24.2 ms ± 187 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

str.startswith

```
In[3]: %timeit ser_object.str.startswith("a")
136 ms ± 300 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

In[4]: %timeit ser_string.str.startswith("a")
11 ms ± 19.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Let's look at some performance/memory comparisons

memory footprint

```
In [5]: "{:.2f} MiB".format(ser_object.memory_usage(deep=True) / 1024**2)
Out[5]: '106.82 MiB'

In [6]: "{:.2f} MiB".format(ser_string.memory_usage(deep=True) / 1024**2)
Out[6]: '56.28 MiB'
```

Opting into Arrow-backed DataFrames

Most I/O methods gained a new `dtype_backend` keyword to convert the input data to Arrow datatypes during reading.

Opting into Arrow-backed DataFrames

Most I/O methods gained a new `dtype_backend` keyword to convert the input data to Arrow datatypes during reading.

```
>>> data = """a,b,c\n1,1.5,x\n,2.5,y"""

>>> df = pd.read_csv(StringIO(data), dtype_backend="pyarrow")
>>> df
      a    b  c
0     1  1.5  x
1  <NA>  2.5  y

>>> df.dtypes
a     int64[pyarrow]
b    double[pyarrow]
c    string[pyarrow]
dtype: object
```

Opting into Arrow-backed DataFrames

Most I/O methods gained a new `dtype_backend` keyword to convert the input data to Arrow datatypes during reading.

```
>>> data = """a,b,c\n1,1.5,x\n,2.5,y"""

>>> df = pd.read_csv(StringIO(data), dtype_backend="pyarrow")
>>> df
      a    b  c
0     1  1.5  x
1  <NA>  2.5  y

>>> df.dtypes
a     int64[pyarrow]
b    double[pyarrow]
c    string[pyarrow]
dtype: object
```

`.convert_dtypes(dtype_backend="pyarrow")` can be used, if a function does not support the `dtype_backend` keyword yet.

Speeding up I/O operations with PyArrow engine

Some I/O functions gained an `engine` keyword to parse the input with PyArrow.

- `read_csv` and `read_json` can dispatch to PyArrow readers.
- `read_parquet` and `read_orc` use PyArrow natively to read the input.

Speeding up I/O operations with PyArrow engine

Some I/O functions gained an `engine` keyword to parse the input with PyArrow.

- `read_csv` and `read_json` can dispatch to PyArrow readers.
- `read_parquet` and `read_orc` use PyArrow natively to read the input.

Benefits

- Huge performance improvements
- Multithreading
- Zero-copy when using Arrow-backed DataFrames

WARNING: Arrow support is still experimental

It is still early in adopting PyArrow dtypes and PyArrow-backed DataFrames.

WARNING: Arrow support is still experimental

It is still early in adopting PyArrow dtypes and PyArrow-backed DataFrames.

This means:

- The Arrow-backend is not yet supported everywhere.
  - Can lead to performance problems
  - Potential bugs
  - `GroupBy` and `merge` are popular examples.

WARNING: Arrow support is still experimental

It is still early in adopting PyArrow dtypes and PyArrow-backed DataFrames.

This means:

- The Arrow-backend is not yet supported everywhere.
    - Can lead to performance problems
    - Potential bugs
    - `GroupBy` and `merge` are popular examples.
- Potential upstream bugs in Arrow itself that aren't addressed yet.

Roadmap for PyArrow support

Roadmap for PyArrow support

- pandas will continue working on supporting PyArrow everywhere.

  This goal is reached when PyArrow-dtypes and NumPy-dtypes are both equally well supported.

  Ensure support for new available dtypes (Decimal, bytes, ...).

Roadmap for PyArrow support

- pandas will continue working on supporting PyArrow everywhere.

  This goal is reached when PyArrow-dtypes and NumPy-dtypes are both equally well supported.

  Ensure support for new available dtypes (Decimal, bytes, ...).

- Provide a way for users to opt into PyArrow-backed DataFrames globally.

Pandas Enhancement Proposals (PDEPs)

Pandas Enhancement Proposals (PDEPs)

A PDEP is a proposal for a major change in pandas, in a similar way as a Python PEP or a NumPy NEP.

See "PDEP-1: Purpose and guidelines" for the details: https://pandas.pydata.org/pdeps/0001-purpose-and-guidelines.html

Pandas Enhancement Proposals (PDEPs)

A PDEP is a proposal for a major change in pandas, in a similar way as a Python PEP or a NumPy NEP.

See "PDEP-1: Purpose and guidelines" for the details: https://pandas.pydata.org/pdeps/0001-purpose-and-guidelines.html

Currently open PDEPs (https://pandas.pydata.org/about/roadmap.html):

- PDEP-6: Ban upcasting in setitem-like operations
- PDEP-7: Consistent copy/view semantics in pandas with Copy-on-Write
- PDEP-8: Inplace methods in pandas
- PDEP-9: Allow third-party projects to register pandas connectors with a standard API

PDEP-8: In-place methods in pandas

Proposal (still being discussed!):

- The `inplace` parameter will be deprecated and removed from any method that can never be done inplace
- The `inplace` parameter is kept only in a few methods such as `fillna()`

For example, replace

```
df.reset_index(inplace=True)
```

with

```
df = df.reset_index()
```

Full proposal: https://github.com/pandas-dev/pandas/pull/51466

Thanks to all contributors!

Pandas is a community project, and everything we talked about is the result of this community of contributors

A total of 260 people contributed to the 2.0 release! (and that's only counting commits on the main repo)

Thanks to all contributors!

Pandas is a community project, and everything we talked about is the result of this community of contributors

A total of 260 people contributed to the 2.0 release! (and that's only counting commits on the main repo)

You can become part of this community as well!

➔ "Let's contribute to pandas" workshop tomorrow afternoon (2-5pm)

Thanks to all contributors!

Pandas is a community project, and everything we talked about is the result of this community of contributors

A total of 260 people contributed to the 2.0 release! (and that's only counting commits on the main repo)

You can become part of this community as well!

➔ "Let's contribute to pandas" workshop tomorrow afternoon (2-5pm)

Those slides: https://github.com/phofl/pydata-berlin/