



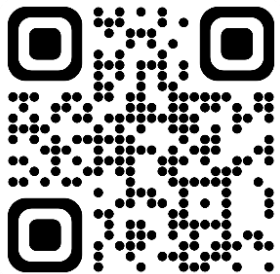
pandas Roadmap and Beyond

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

Hadi Abdi Khojasteh

hadi.abdikhojasteh@**deltatre**.com

PyData Prague, April 23rd, 2024



Presentation will be available online:
<https://github.com/hkhojasteh>



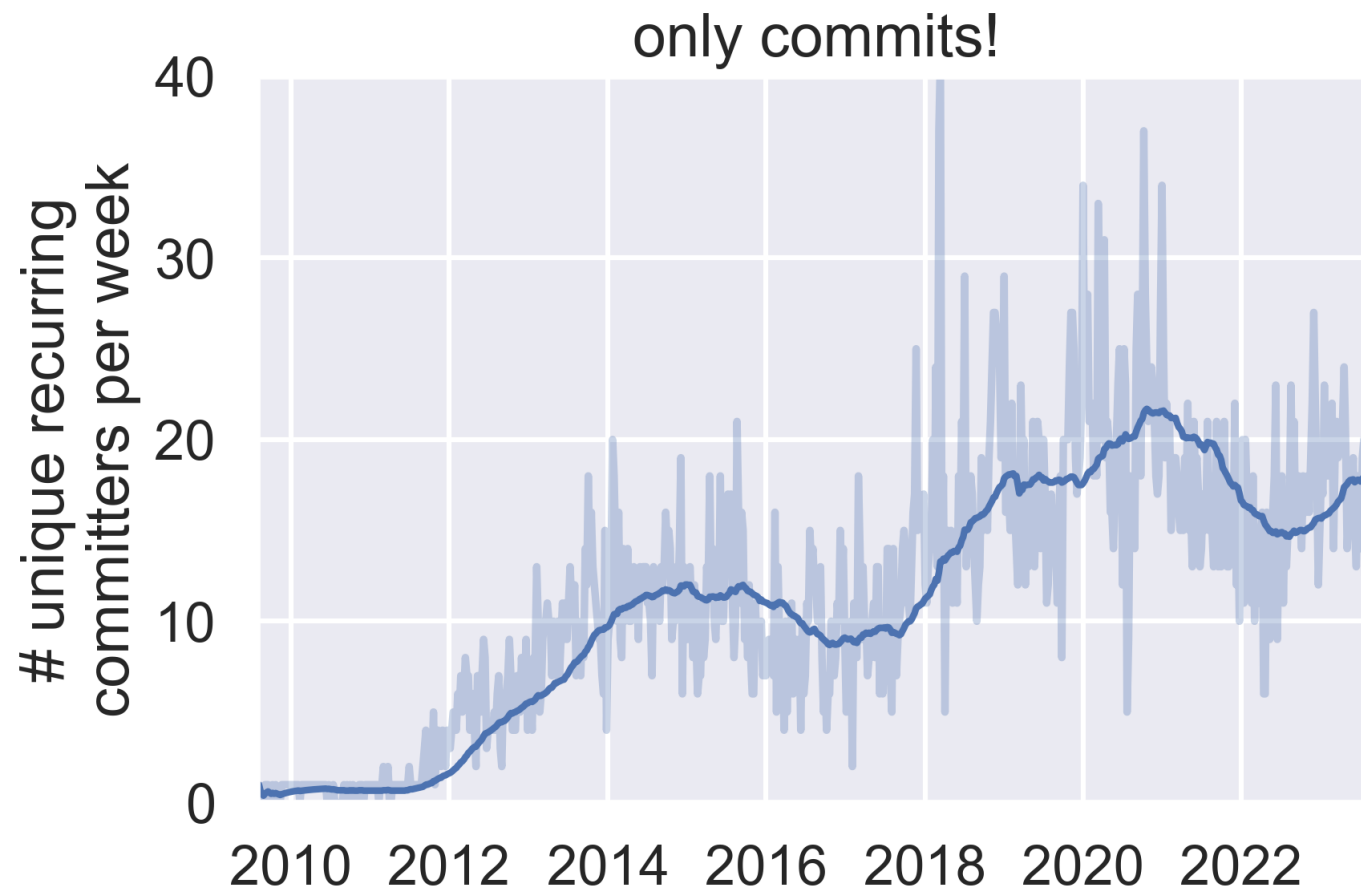
About Hadi

Hadi Abdi Khojasteh

- Background in computer science
- NumFOCUS, PyData community and  pandas core contributor
- Currently working as a Senior Machine Learning Engineer | Innovation Lab
and Technical Staff | Technology, Sports Experiences at Deltatre

The logo for Deltatre, featuring the word "deltatre" in a bold, red, lowercase sans-serif font.The logo for Deltatre Innovation Lab, consisting of the text "Deltatre Innovation Lab" in white, stacked vertically on a black square background.The logo for NumFOCUS, featuring the word "NUMFOCUS" in a blue, uppercase sans-serif font, with the "O" stylized as a circle containing a bracket. Below it, the tagline "OPEN CODE = BETTER SCIENCE" is written in a smaller, blue, uppercase sans-serif font.

pandas contributor community growth



→ need to update our governance (clarifying roles, updating decision making processes)

Pandas Enhancement Proposals (PDEPs)

A PDEP is a proposal for a major change in  pandas, similar to a Python PEP or a NumPy NEP. New process introduced in process in August 2022.

PDEP-1 describes the purpose and guidelines

(currently deciding on voting mechanism for final decisions)

<https://pandas.pydata.org/pdeps/0001-purpose-and-guidelines.html>

See PDEPs on the pandas roadmap: <https://pandas.pydata.org/about/roadmap.html>

Pandas Enhancement Proposals (PDEPs)

See PDEPs on the  pandas roadmap: <https://pandas.pydata.org/about/roadmap.html>

PDEP-4 [Implemented in 2.0]	Consistent datetime parsing
PDEP-6 [Accepted for 3.0]	Ban upcasting in setitem-like operations
PDEP-7 [Open, planned for 3.0]	Consistent copy/view semantics with Copy-on-Write
PDEP-8 [Open, planned for 3.0?]	Inplace methods in pandas
PDEP-10 [Accepted for 3.0]	PyArrow as a required dependency for default string inference
PDEP-11 [Under discussion]	Change the default of dropna to False
PDEP-13 [Under discussion]	Make the Series.apply method operate Series-wise

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:

```
>>> 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.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]
```

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 exactly the same format;
# - passing `format='mixed'`, and the format will be inferred for each element individually. You might want to try:
# - passing `format='mixed'`, and the format will be inferred for each element individually. You might want to try:
```

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

View

	A	B
0	1	2
1	3	4
2	5	6

df1

←df2

Copy

	A	B
0	1	2
1	3	4
2	5	6

df1

	A	B
0	1	2
1	3	4

df2

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

Current situation: copy vs view

Modifying a View

	A	B
0	1	2
1	3	5
2	5	6

df1

←df2

Modifying a Copy

	A	B
0	1	2
1	3	4
2	5	6

df1

	A	B
0	1	2
1	3	5

df2

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

The SettingWithCopyWarning

How to "solve" the warning?

The SettingWithCopyWarning

How to "solve" the warning?

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

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

The SettingWithCopyWarning

How to "solve" the warning?

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

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

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 unnecessary) 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 or Series 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 and performance

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 and performance

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: 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 ...
```


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 and performance

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.

Avoiding copies with Copy-on-Write

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

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(10)])
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

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)

```
%%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

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)

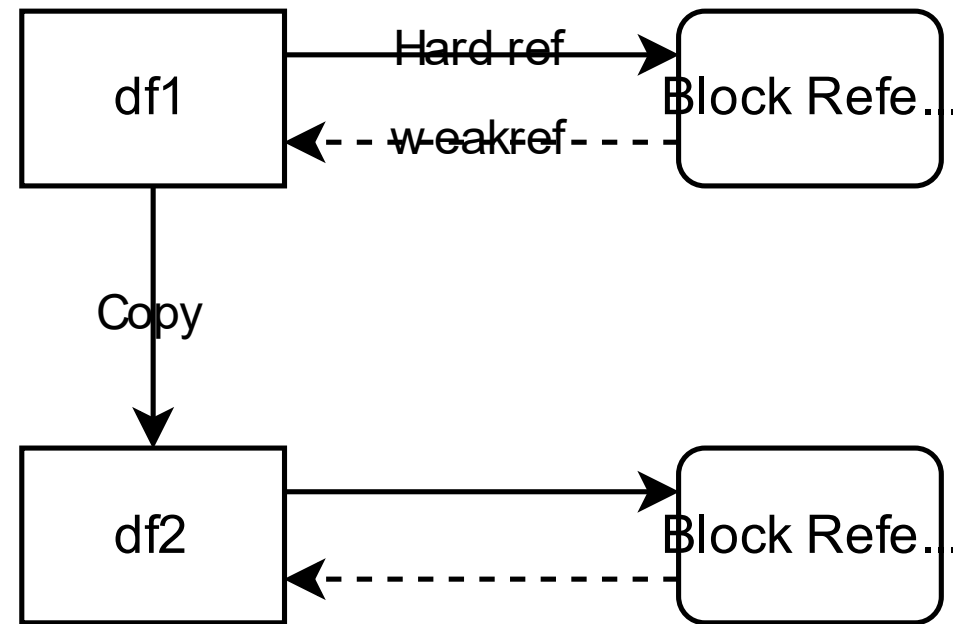
```
%%timeit
(df.rename(columns={"col_1": "new_index"})
 .assign(sum_val=df["col_1"] + df["col_2"])
 .drop(columns=["col_10", "col_20"])
 .astype({"col_5": "int32"})
 .reset_index()
 .set_index("new_index")
 )
2.45 s ± 293 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
# with Copy-on-Write enabled
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()
```

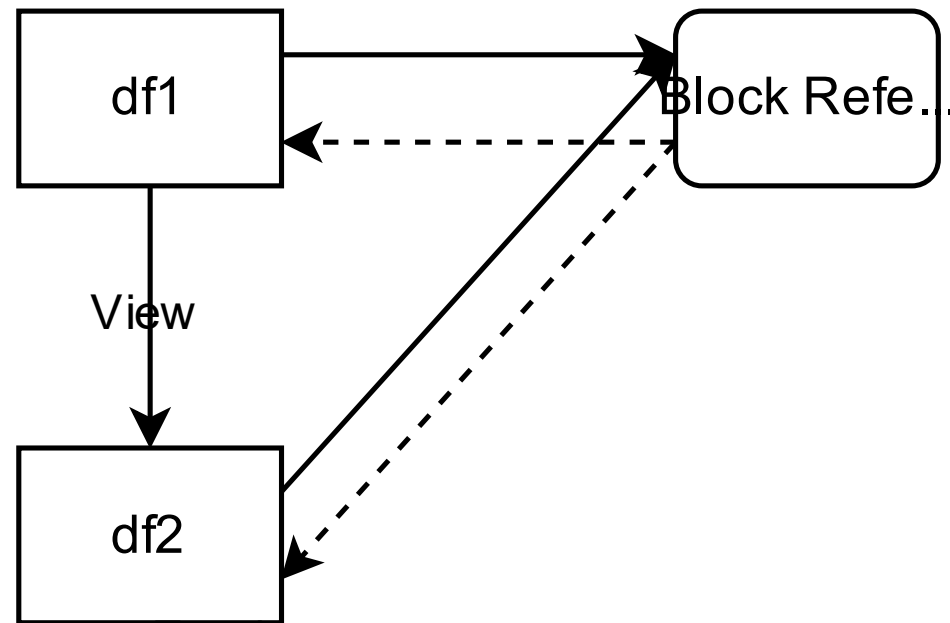


Hiding copy/view details with Copy-on-Write

When an operations can use a view for the result

→ both reference the same data

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

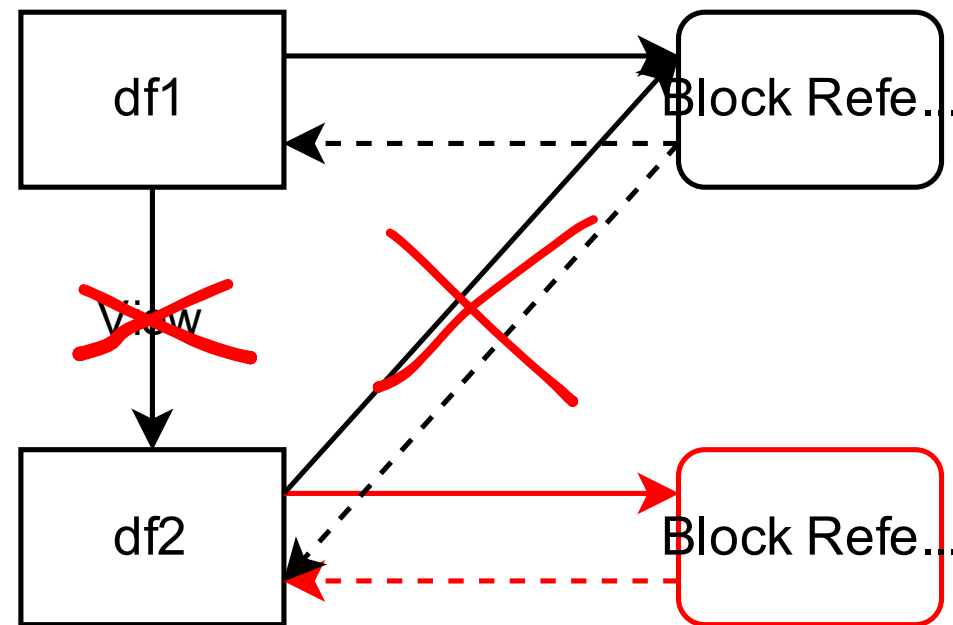


Hiding copy/view details with Copy-on-Write

Modifying a view or its parent (`df1` or `df2`) will trigger a copy (a "copy on write")

→ each DataFrame again owns its own memory.

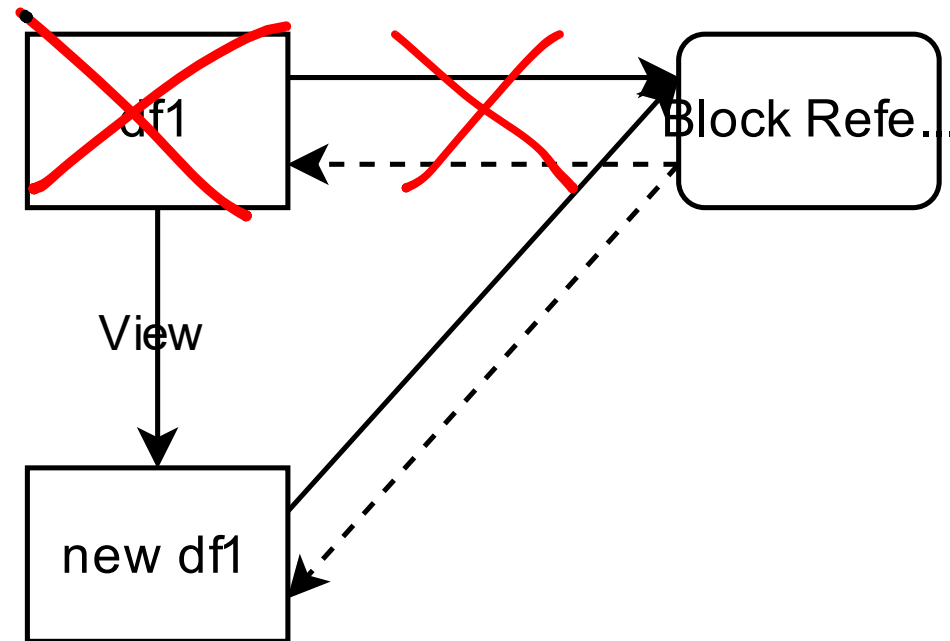
```
df2 = df1.reset_index()  
df2.loc[1, "C"] = 10
```



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/>

PDEP-10: PyArrow as a required dependency for default string inference implementation

Arrow-backed data types

Using PyArrow arrays to store the data of a DataFrame.



meets

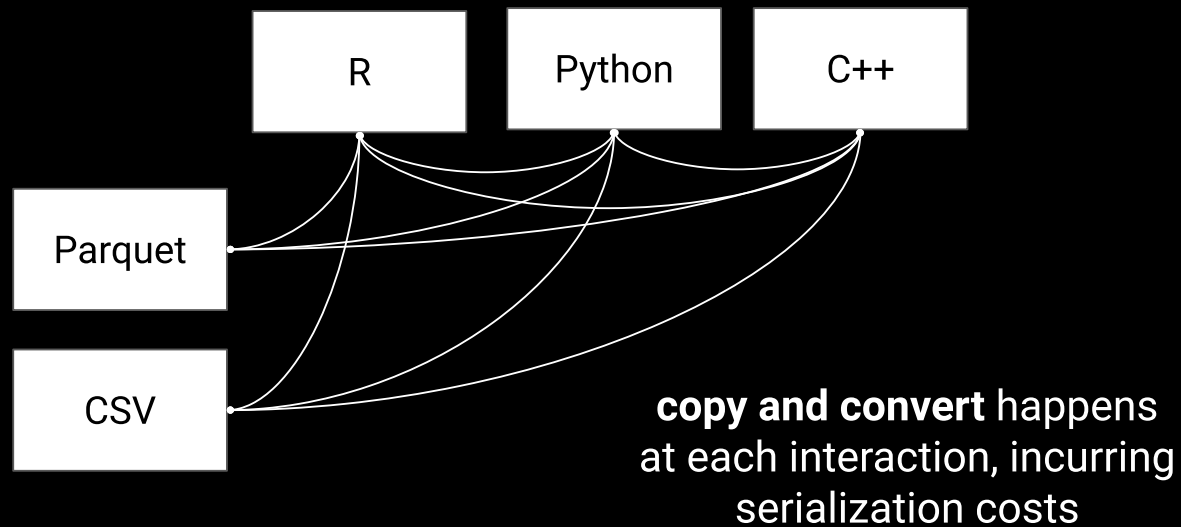


Apache Arrow defines a language-independent columnar memory format for flat and hierarchical data, organized for efficient analytic operations on modern hardware like CPUs and GPUs.

Apache Arrow is a specification defining a common, language-agnostic in-memory representation for columnar data

+ A multi-language toolbox for accelerated data interchange and in-memory processing

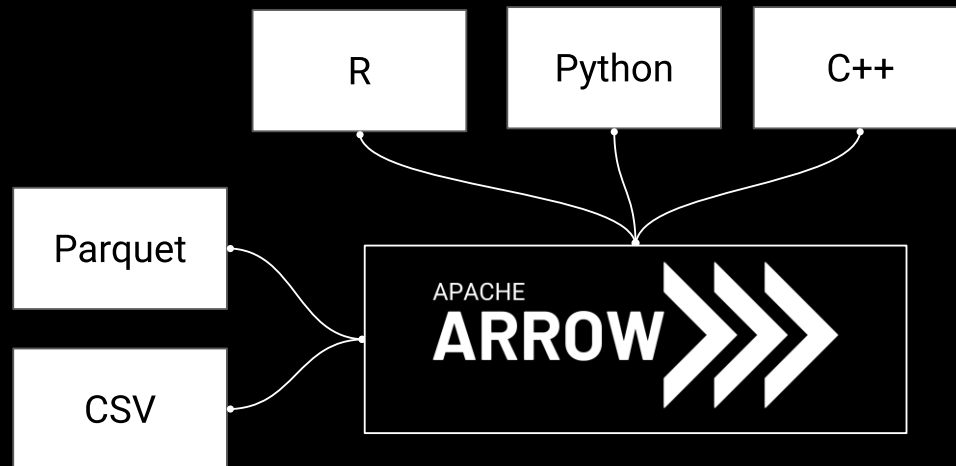
Arrow Data Interchange



Apache Arrow is a specification defining a common, language-agnostic in-memory representation for columnar data

+ A multi-language toolbox for accelerated data interchange and in-memory processing

Arrow Data Interchange



Arrow provides a standard in-memory format and reduces serialization costs when connecting systems

Avoid object dtype for strings by default

Currently:

```
>>> pd.Series(["a", "b", "c"])  
0 a  
1 b  
2 c  
dtype: object
```


Avoid object dtype for strings by default

Currently:

```
>>> pd.Series(["a", "b", "c"])
0 a
1 b
2 c
dtype: object
```

Future behaviour:

```
>>> pd.options.future.infer_string = True
>>> pd.Series(["a", "b", "c"])
0 a
1 b
2 c
dtype: string[pyarrow]
```

Planned for 3.0 by default, but you can already opt-in starting with pandas 2.1

PyArrow-backed string dtype

PyArrow offers fast and efficient in-memory string operations.

This provides:

- significantly improved performance compared to NumPy's object dtype with Python `str` operations
- smaller memory footprint

PyArrow-backed string dtype

PyArrow offers fast and efficient in-memory string operations.

This provides:

- significantly improved performance compared to NumPy's object dtype with Python `str` operations
- smaller memory footprint
- compatible with existing object-dtype-based string methods (all `.str.` string accessor methods keep working)

Let's look at some performance/memory comparisons

```
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

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'
```

PyArrow-backed string dtype by default in 3.0

- PyArrow will become a **required** dependency of  pandas starting with pandas 3.0
- The PyArrow-backed string dtype will be used by default (no more object dtype for strings!)

How do I try this now?

Enable it in pandas 2.1:

```
pd.options.future.infer_string = True
```

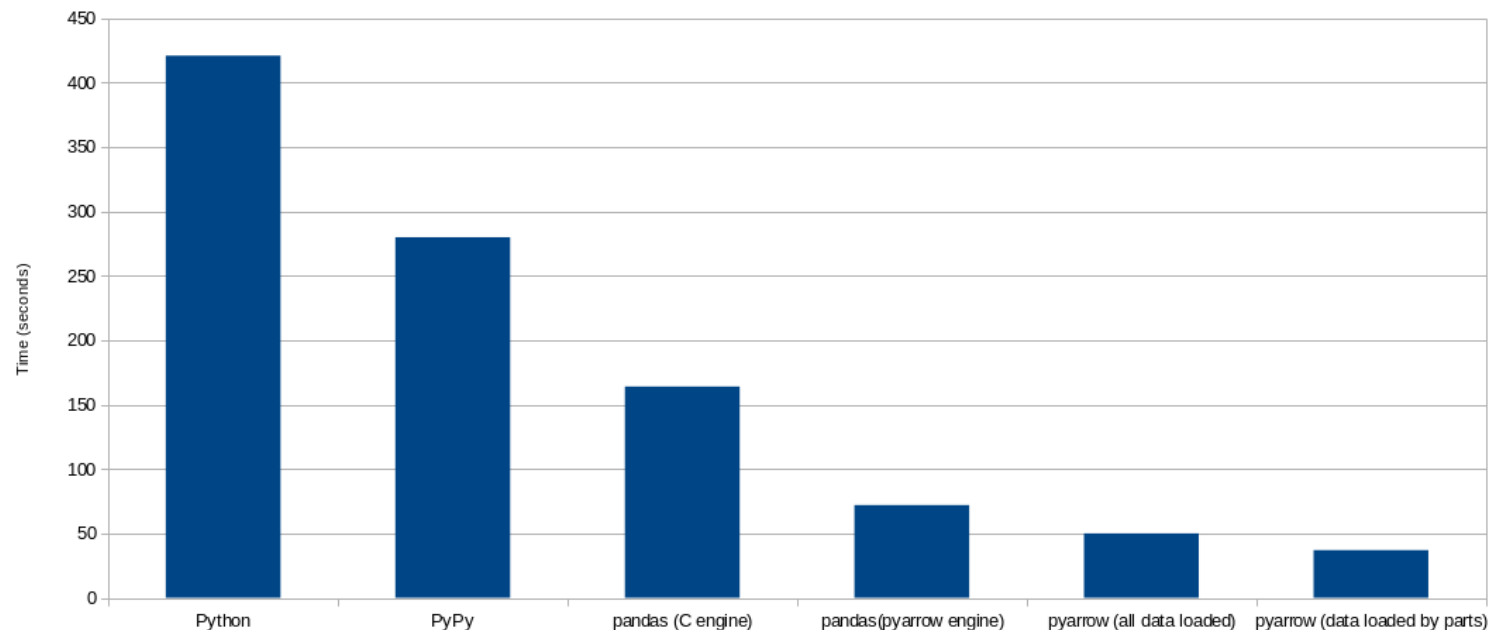
- PDEP: <https://pandas.pydata.org/pdeps/0010-required-pyarrow-dependency.html>
- Blogpost of Patrick Höfler about improvements in pandas and dask:
<https://towardsdatascience.com/utilizing-pyarrow-to-improve-pandas-and-dask-workflows-2891d3d96d2b>

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.

→ Huge [performance improvements](#) and uses multithreading with Zero-copy



Arrow-backed data types

PyArrow offers support for a wide variety of dtypes (not well supported by pandas right now):

- string, bytes, decimal, date, explicit null-datatype, nested data and [many more](#).

Arrow-backed data types

PyArrow offers support for a wide variety of dtypes (not well supported by pandas right now):

- string, bytes, decimal, date, explicit null-datatype, nested data and [many more](#).

Experimental `ArrowDtype` for using any Arrow type in a column

`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]")
```

These columns use the PyArrow memory layout and compute functionality.

Arrow-backed data types

PyArrow offers support for a wide variety of dtypes (not well supported by pandas right now):

- string, bytes, decimal, date, explicit null-datatype, nested data and [many more](#).

Experimental **ArrowDtype** for using any Arrow type in a column

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

These columns use the PyArrow memory layout and compute functionality.

→ opt-in using new **dtype_backend="pyarrow"** keyword in IO methods, or **df.convert_dtypes(dtype_backend="pyarrow")** to convert afterwards

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)
```

```
n [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)
```

```
n [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.

PDEP-6: Ban upcasting in setitem-like operations

```
df = pd.DataFrame({"a": [1, 2, 3]})  
df.loc[1, "a"] = 3.5  
df.loc[1, "a"] = "3"
```

PDEP-6: Ban upcasting in setitem-like operations

```
df = pd.DataFrame({"a": [1, 2, 3]})  
df.loc[1, "a"] = 3.5  
df.loc[1, "a"] = "3"
```

When assigning a value into a Series that would cause the dtype to change, it is not clear what the user wants to happen. Different users may want different things.

→ pandas shouldn't guess!

Summary: Assignments into an existing Series that would change the dtype will now raise.

Full proposal: <https://github.com/pandas-dev/pandas/pull/50402>

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>

PDEP-11: Change the default of dropna to False

Motivating Example: Compute the average time for groups of 3 runners.

```
df = pd.read_excel("runner_times.xlsx")
df.value_counts("group")
result = df.groupby("group")[["seconds"]].mean()
result.to_excel("group_means.xlsx")
```

- Some methods (`stack`, `pivot_table`) do not behave well with `dropna=False`
 - You can try the new implementation of `stack` in version 2.1!
- What should we do with `mode`?
 - If `mode` is counting values, then it should have `dropna=False`.
 - If `mode` is a stats method like `mean` and `median`, then it should have `skipna=True`

Full proposal: <https://github.com/pandas-dev/pandas/pull/53094>

pandas 2.2.0 released on January 19th!

Upcoming changes in pandas 3.0 which bring two bigger changes to the default behavior of pandas:

- Copy-on-Write
- Dedicated string data type (backed by Arrow) by default

New enhancements on

- pandas Apache Arrow ADBC Driver support in `to_sql` and `read_sql`; significant performance improvements
- Create a Series based on one or more conditions; `Series.case_when()`
- `Series.struct` and `Series.list` accessor for PyArrow structured data

And many more! See full release notes at <https://pandas.pydata.org/docs/dev/whatsnew/v2.2.0.html>

We recommend that all users upgrade to this version.

Thanks to all contributors!



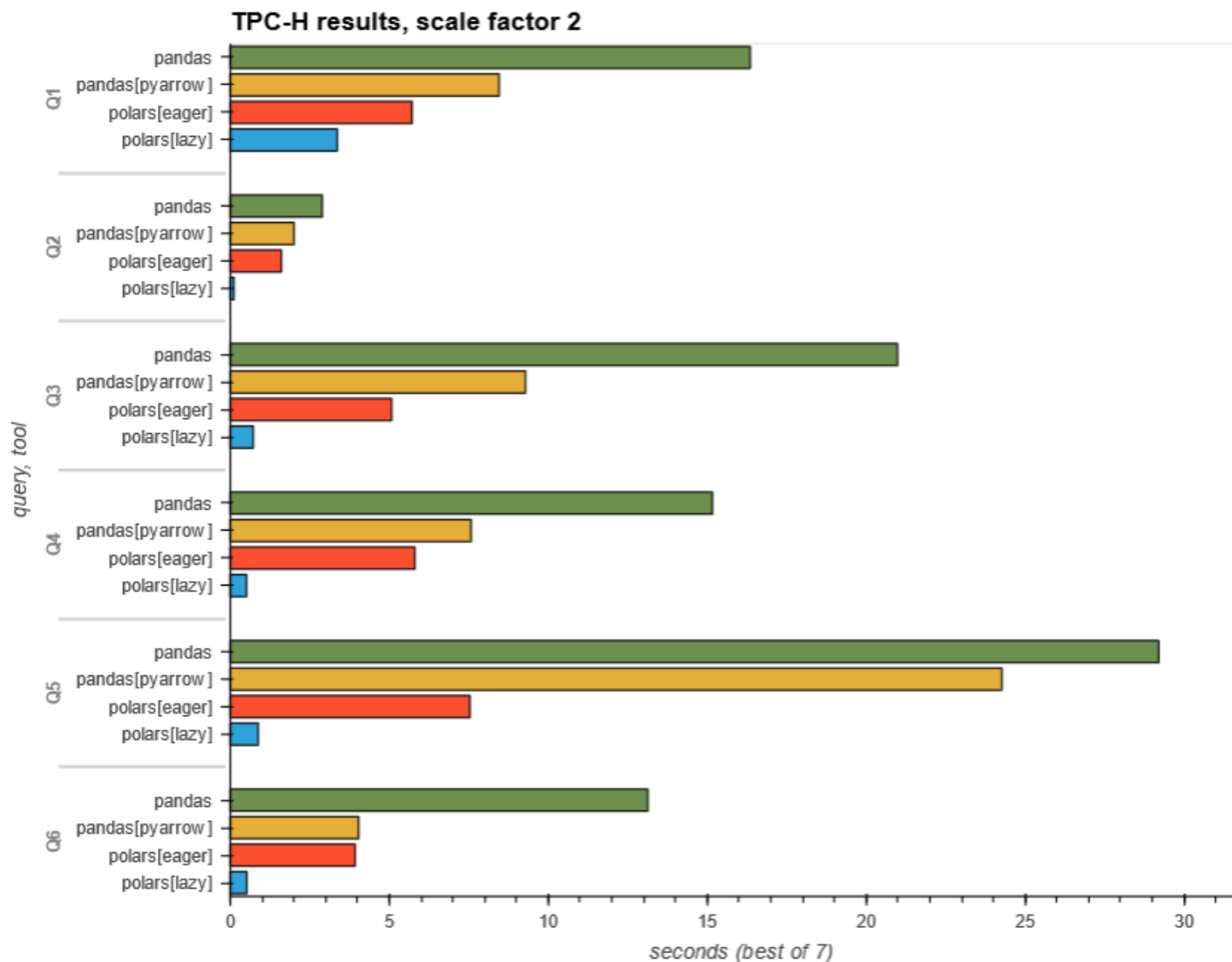
Pandas is a community project,
and everything we talked about is the result of this community of contributors

A total of 162 people contributed to the 2.2.0 (January 19, 2024) release!
(and that's only counting commits on the main repo)

You can become part of this community as well!

<https://pandas.pydata.org/docs/development/contributing.html>

pandas vs Polars / query optimization and multi threading



pandas + Polars / Interoperability example

```
loaded_pandas_data = pandas.read_sas(fname)

polars_data = polars.from_pandas(loaded_pandas_data)
# perform operations with pandas polars

to_export_pandas_data = polars.to_pandas(use_pyarrow_extension_array=True)
to_export_pandas_data.to_latex()
```


Pay attention to warnings!


571





Found this on [github...](#)

```
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)

import pandas
```

Share Improve this answer Follow

 [smci](#)
smci's 33.4k ● 21 ● 116 ● 148

 [bdiamante](#)
16.8k ● 6 ● 41 ● 47

edited Feb 20, 2020 at 8:37

answered Apr 3, 2013 at 3:19

61 nb: put the `warnings...ignore` before the `import pandas...` to cause the `FutureWarning` to be ignored. – [michael](#) Dec 8, 2017 at 13:31

29 Hey I'm getting the future warning despite adding these lines. – [Eswar](#) Oct 6, 2020 at 17:19

It's working. Vote up. – [Zbyszek](#) Feb 14 at 14:16

Thank you for your attention :)



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



Find me in   

Presentation will be available online:
<https://github.com/hkhojasteh>



