**Understand Polars’ Lack of Indexes**

Switch from Pandas to Polars and forget about indexes

[[](https://medium.com/@carlmkadie?source=post_page-----526ea75e413--------------------------------)](https://medium.com/@carlmkadie?source=post_page-----526ea75e413--------------------------------)

[[Towards Data Science](https://towardsdatascience.com/?source=post_page-----526ea75e413--------------------------------)](https://towardsdatascience.com/?source=post_page-----526ea75e413--------------------------------)

[Carl M. Kadie](https://medium.com/@carlmkadie?source=post_page-----526ea75e413--------------------------------)

·

Follow

Published in

[Towards Data Science](https://towardsdatascience.com/?source=post_page-----526ea75e413--------------------------------)

·

7 min read

·

Jan 7

158

5



A Polar Bear racing a Panda — Source: [https://openai.com/dall-e-2](https://openai.com/dall-e-2/)

Pandas and Polars are two dataframe libraries for Python. In a [previous article](https://medium.com/towards-data-science/understand-pandas-indexes-1b94f5c078c6), I wrote this about Pandas and indexes:

To efficiently use Pandas, ignore its documentation and learn the [complicated] truth about indexes.

In contrast, the original [Polars book](https://web.archive.org/web/20230117102825/https:/pola-rs.github.io/polars-book/user-guide/coming_from_pandas.html) said this about Polars and indexes:

Indexes are not needed! Not having them makes things easier — convince us otherwise!

Can we really forget about indexes? Let’s put Polars’ claim to the test. We’ll port all the examples in my previous article from Pandas to Polars. That will give us insight into the practicality of working without indexes.

In the end, we’ll see that:

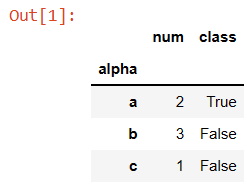
* “Indexes are not needed! Not having them makes things easier.”
* If you think you *really*need an index, you *really*need a dictionary.

To reach that end, let’s start by creating a dataframe and retrieving a simple row.

**Construction and Simple Row Retrieval**

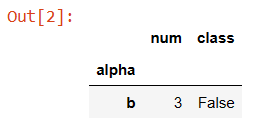
In Pandas, we construct a dataframe and set an index like so:

import pandas as pd  
df1 = pd.DataFrame([['a',2,True],  
 ['b',3,False],  
 ['c',1,False]],  
 columns=['alpha','num','class'])  
df1.set\_index(['alpha'],inplace=True)  
df1



We turn key b into the row of interest with:

df1.loc[['b']] # returns row 'b' as a dataframe

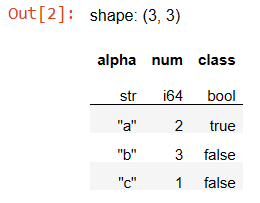


In Polars, we could construct a dataframe from rows like this:

import polars as pl  
df1 = pl.from\_records([['a', 2, True],  
 ['b', 3, False],  
 ['c', 1, False]],  
 orient='row',  
 columns=['alpha', 'num', 'class'])  
df1

However, Polars is column-centric, so a better way to construct the same dataframe is this:

import polars as pl  
df1 = pl.DataFrame({'alpha':['a','b','c'],   
 'num':[2,3,1],  
 'class':[True,False,False]  
 })  
df1



We don’t need to set an index. We turn key b into the row of interest with:

df1.filter(pl.col("alpha")=='b')

The filter method finds rows of interest. The expression pl.col("alpha")=='b' tells filter which rows to find. Compared to Pandas, I find the Polars approach both simpler and more general. (We’ll discuss performance in a bit.)

We move next from finding a simple row to finding the number of a row.

**Finding Row Numbers**

In Pandas, you can see the numbers for rows of interest via index.get\_loc(...):

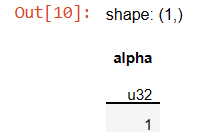
import pandas as pd  
df2 = pd.DataFrame([['x',2,True],  
 ['y',3,False],  
 ['x',1,False]],  
 columns=['alpha','num','class'])  
df2.set\_index(['alpha'],inplace=True)  
print(f"{df2.index.get\_loc('y')=}")  
print(f"{df2.index.get\_loc('x')=}")



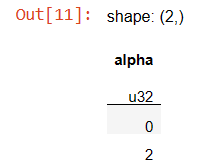
As the example shows, the function will only return a number when a single item matches. When multiple items match, it returns a Boolean array.

In Polars, you should first ask yourself if you really need to find row numbers. The answer is usually “no”. If, however, you answer “yes”, you may use arg\_where.

df2 = pl.DataFrame({'alpha':['x','y','x'],  
 'num':[2,3,1],  
 'class':[True,False,False]})  
df2.select(pl.arg\_where(pl.col("alpha")=='y')).to\_series()



df2.select(pl.arg\_where(pl.col("alpha")=='x')).to\_series()



The result is a Polars series. In Polars, a series represents one column of values, here row numbers.

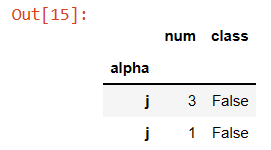
Comparing Pandas to Polars with respect to finding row numbers, I find the complexity similar. Polars may win, however, by making row numbers less important.

Let’s look next at complex row access.

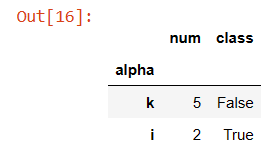
**Row Access**

In Pandas, the main way to access indexed rows is .loc[…], where the input can be a: single element, list of elements, or slice of elements. The rows will be output in the order they appear in the input. These examples show each kind of input.

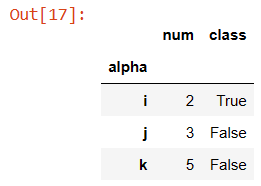
df3 = pd.DataFrame([['i',2,True],  
 ['j',3,False],  
 ['k',5,False],  
 ['j',1,False]],  
 columns=['alpha','num','class'])  
df3.set\_index(['alpha'],inplace=True)  
df3.loc['j']



df3.loc[['k','i']]



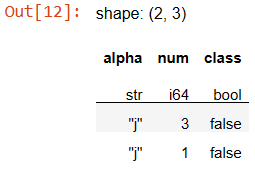
df3.loc['i':'k']



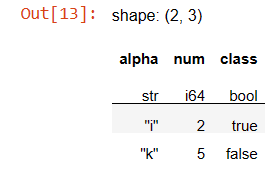
Note that unlike the rest of Python, with Pandas the *start:stop* slice is inclusive of the *stop* value. Also, note that Panda excluded the second ‘j’ row because it came after the (first) ‘k’ row.

In Polars, we use filter and expressions:

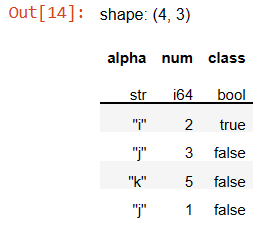
df3 = pl.DataFrame({'alpha':['i','j','k','j'],  
 'num':[2,3,5,1],   
 'class':[True,False,False,False]})  
df3.filter(pl.col("alpha")=='j')



df3.filter(pl.col("alpha").is\_in(['k','i']))



df3.filter(pl.col("alpha").is\_between('i','k',include\_bounds=True))



By default, Polars’ is\_between will not include its bounds, but either or both bound can optionally be included. Also, note that Polars included the second ‘j’ row. Polars looks at in\_between (on string values) based on alphabetical order, not row order.

Because it doesn’t require an index, I find Polars simpler than Pandas for these more complex row retrievals.

For our last basic task, let’s look at joining rows.

**Joining Rows**

In Pandas, the rules for left joins are:

* The left dataframe need not be indexed, but the right one does.
* Give the left column(s) of interest in the join’s on input.

In this example, we will use join to add a “score” column to a dataframe. Here is the left dataframe. It isn't indexed.

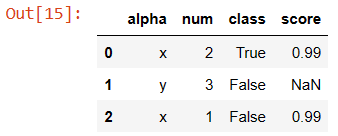
df\_left = pd.DataFrame([['x',2,True],  
 ['y',3,False],  
 ['x',1,False]],  
 columns=['alpha','num','class'])

In Pandas, the right dataframe needs an index, but it can be named anything. Here we call it any\_name.

df\_right = pd.DataFrame([['x',.99],  
 ['b',.88],  
 ['z',.66]],  
 columns=['any\_name','score'])  
df\_right.set\_index(['any\_name'],inplace=True)

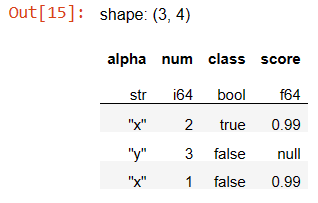
We combine the two dataframes with a left join. We use column alpha from the first dataframe and whatever is indexed in the second dataframe. The result is a new dataframe with a score column.

df\_left.join(df\_right,on=['alpha'],how='left')



In Polars, everything is similar, but a little simpler:

df\_left = pl.DataFrame({'alpha':['x','y','x'],  
 'num':[2,3,1],  
 'class':[True,False,False]})  
df\_right = pl.DataFrame({'alpha':['x','b','z'],  
 'score':[.99,.88,.66]})  
df\_left.join(df\_right,on=['alpha'],how='left')



The difference is that we don’t need to index the right dataframe. If the columns of interest have the same name (as here), we use on. If not, we use left\_on and right\_on.

So, Polars is again simpler to use than Pandas, but at what cost?

**Performance**

Surely, the lack of indexes makes Polars slower. Amazingly, no. Over a wide range of benchmarks, [Polars is much faster](https://www.ritchievink.com/blog/2021/02/28/i-wrote-one-of-the-fastest-dataframe-libraries/) than Pandas [[Vink, 2021]](https://www.ritchievink.com/blog/2021/02/28/i-wrote-one-of-the-fastest-dataframe-libraries/). It achieves this via optimizations including good memory layout and automatic vectorization/parallelization.

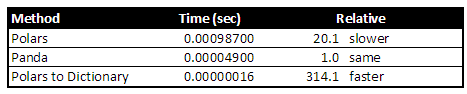
Can we construct a case in which Pandas is faster than Polars? Yes, if we use a dataframe as a dictionary, Pandas can be 20 times faster than Polars. However…

Guess what is 300 times faster than using Pandas as a dictionary? Answer: using a dictionary as a dictionary.

In this test, we construct a dataframe with two columns, filled with the numbers 0 to 999,999. We then look for the number 500,000.

import polars as pl  
import pandas as pd  
  
n = 1\_000\_000  
df\_pl = pl.DataFrame({'a':list(range(n)),'b':list(range(n))})  
%timeit df\_pl.filter(pl.col("a")==n//2)  
  
df\_pd = pd.DataFrame({'a':list(range(n)),'b':list(range(n))})  
df\_pd = df\_pd.set\_index('a')  
%timeit df\_pd.loc[n//2]  
  
dict\_pl = df\_pl.partition\_by('a',as\_dict=True)  
%timeit dict\_pl[n//2]

Here are the average results over many runs on my 4-core laptop:



To summarize performance: according to other benchmarks, for typical use, Polars is faster than Pandas. For special cases — for example, when you should really use a dictionary —Polars gives tools to create a dictionary for the fastest performance.

**Conclusion**

In my opinion, eliminating indexes makes Polars much easier to use than Pandas.

You would expect this simplification to cause slower performance. Benchmarks, however, show Polars is generally much faster than Pandas. It achieves this via optimizations including good memory layout and automatic vectorization/parallelization. There may still be cases where an index-like data structure is needed. For those cases, Polars provides tools to create, for example, dictionaries.

So, should you switch from Pandas to Polars? It depends.

Our genomics project, [FaST-LMM](https://fastlmm.github.io/) uses Pandas to output tables of statistical results. FaST-LMM does almost all its computational work with custom code, outside of Pandas. It only uses Pandas to share final results with our users, who we can assume understand Pandas. Given this, we have no reason to switch from Pandas.

On the other hand, if I start a new project that involves interesting data analytics, I’ll do it in Polars. Polars gives me the speed and simplicity that I’ve always wanted from Pandas.

*Please*[*follow me on Medium*](https://medium.com/@carlmkadie)*. I write on scientific programming in Rust and Python, machine learning, and statistics. I tend to write about one article per month.*