точка

банк для предпринимателей
и предприятий

🐼🐍🥷

Джун против панды

1. С чего все начиналось
2. Итерируемся
3. Применяем функции
4. Мерджимся
5. Классика: типы данных (?)
6. Классика оптимизации
7. Отпуск
8. Выводы

С чего все начиналось

Дано:

- 1 проект, написанный джуном
- 1 месяц до отпуска

Найти:

- 1 год данных, которые нужно просчитать и сохранить

до отпуска: 30 дней ❤️❤️❤️

Дано:

- 1 проект, написанный джуном
- 1 месяц до отпуска

Найти:

- 1 год данных, которые нужно просчитать и сохранить

Проблема #1

Данные сохраняются со скоростью: 1 день данных за 1 день

Проблема #2

OOM

Проблема #3

pandas (медленно)

до отпуска: 30 дней ❤️❤️❤️

Миллионы-миллионы-миллионы… статей

точка

Очень много способов ускорить Pandas

Stats&Data ninja · Follow
12 min read · Dec 7, 2022

PYTHON PROGRAMMING: MANIPULATING TABULAR DATA EFFICIENTLY

How to Speed up Pandas by 100x

With Great power comes great responsibility.

Pritish Jadhav · Follow

Как ускорить обработку данных в Pandas в 600 раз

Nick Komissarenko · Follow
5 min read · Sep 22, 2020

Итерирумся

точка

To iterate over the rows of a DataFrame, you can use the following methods:

- `iterrows()`: Iterate over the rows of a DataFrame as (index, Series) pairs. This converts the rows to Series objects, which can change the dtypes and has some performance implications.
- `itertuples()`: Iterate over the rows of a DataFrame as namedtuples of the values. This is a lot faster than `iterrows()`, and is in most cases preferable to use to iterate over the values of a DataFrame.

To iterate over the rows of a DataFrame, you can use the following methods:

- `iterrows()`: Iterate over the rows of a DataFrame as (index, Series) pairs. This converts the rows to Series objects, which can change the dtypes and has some performance implications.
- `itertuples()`: Iterate over the rows of a DataFrame as namedtuples of the values. This is a lot faster than `iterrows()`, and is in most cases preferable to use to iterate over the values of a DataFrame.

To iterate over the rows of a DataFrame, you can use the following methods:

- `iterrows()`: Iterate over the rows of a DataFrame as (index, Series) pairs. This converts the rows to Series objects, which can change the dtypes and has some performance implications.
- `itertuples()`: Iterate over the rows of a DataFrame as namedtuples of the values. This is a lot faster than `iterrows()`, and is in most cases preferable to use to iterate over the values of a DataFrame.

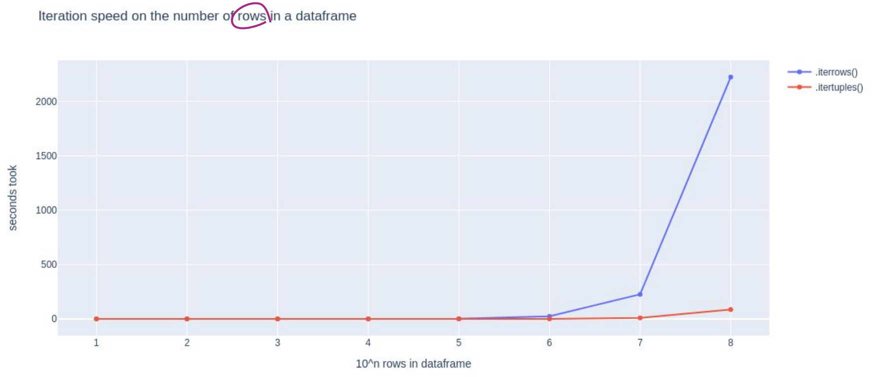
To iterate over the rows of a DataFrame, you can use the following methods:

- `iterrows()`: Iterate over the rows of a DataFrame as (index, Series) pairs. This converts the rows to Series objects, which can change the dtypes and has some performance implications.
- `itertuples()`: Iterate over the rows of a DataFrame as namedtuples of the values. This is a lot faster than `iterrows()`, and is in most cases preferable to use to iterate over the values of a DataFrame.

НО!

To iterate over the rows of a DataFrame, you can use the following methods:

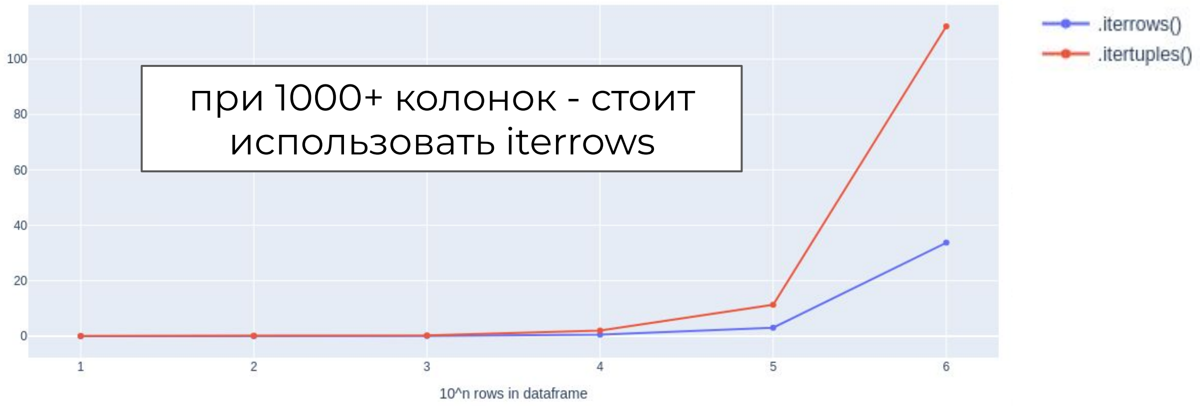
- `iterrows()`: Iterate over the rows of a DataFrame as (index, Series) pairs. This converts the rows to Series objects, which can change the dtypes and has some performance implications.
- `itertuples()`: Iterate over the rows of a DataFrame as namedtuples of the values. This is a lot faster than `iterrows()`, and is in most cases preferable to use to iterate over the values of a DataFrame.



To iterate over the rows of a DataFrame, you can use the following methods:

- `iterrows()`: Iterate over the rows of a DataFrame as (index, Series) pairs. This converts the rows to Series objects, which can change the dtypes and has some performance implications.
- `itertuples()`: Iterate over the rows of a DataFrame as namedtuples of the values. This is a lot ~~faster than `iterrows()`~~, and is in most cases preferable to use to iterate over the values of a ~~DataFrame~~.

Iteration speed on the number of rows in a dataframe with 1000 columns



```
[5]: %memit pd.DataFrame(np.random.rand(10*6, 1000)).iterrows()
peak memory: 139.55 MiB, increment: 0.86 MiB

[5]: %memit pd.DataFrame(np.random.rand(10*6, 1000)).itertuples()
peak memory: 143.27 MiB, increment: 4.67 MiB
```

```
[5]: %memit pd.DataFrame(np.random.rand(10*6, 1000)).iterrows()

peak memory: 139.55 MiB, increment: 0.86 MiB

[5]: %memit pd.DataFrame(np.random.rand(10*6, 1000)).itertuples()

peak memory: 143.27 MiB, increment: 4.67 MiB
```

```
[5]: %memit pd.DataFrame(np.random.rand(10*6, 1000)).iterrows()
peak memory: 139.55 MiB, increment: 0.86 MiB
```

```
[5]: %memit pd.DataFrame(np.random.rand(10*6, 1000)).itertuples()
peak memory: 143.27 MiB, increment: 4.67 MiB
```

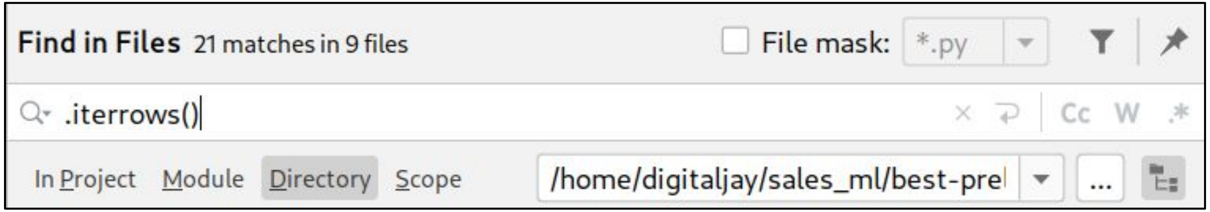
в **5** раз

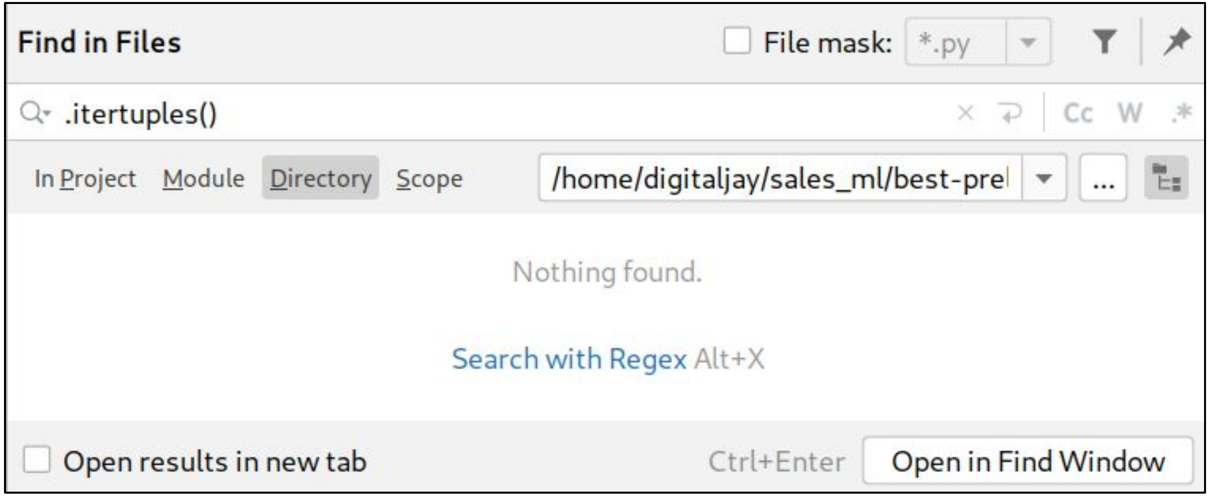
меньше памяти потребляет
iterrows по сравнению с **itertuples**
при больших датафеймах

1. Stop using iterrows() :

- Data manipulation often requires iterating over dataframe rows.
- `iterrows()` is often the go-to option for such use cases. However, it is notoriously slow and can be easily swapped by `itertuples()` .

Например, итерация по строкам с помощью метода .iterrows(). Это наиболее медленный способ, к тому же не сохраняет типы данных. Другие варианты — использовать .itertuples(), где на каждой итерации строка рассматривается как именованный tupple. Это во много раз быстрее, чем .iterrows(). Еще один аналог — .iteritems().


Find in Files 21 matches in 9 files
File mask: *.py
.iterrows()
Cc W .*
In Project Module Directory Scope
/home/digitaljay/sales_ml/best-pre
...


Find in Files
File mask: *.py
.itertuples()
Cc W .*
In Project Module Directory Scope
/home/digitaljay/sales_ml/best-pre
...
Nothing found.
Search with Regex Alt+X
Open results in new tab
Ctrl+Enter
Open in Find Window

до отпуска - 25 дней

точка

2x speed up

iterrows -> itertuples

Ура! Теперь до конца просчета данных - всего **183** дня!

точка

To iterate over the rows of a DataFrame, you can use the following methods:

- `iterrows()`: Iterate over the rows of a DataFrame as (index, Series) pairs. This converts the rows to Series objects, which can change the dtypes and has some performance implications.
- `itertuples()`: Iterate over the rows of a DataFrame as namedtuples of the values. This is a lot faster than `iterrows()`, and is in most cases preferable to use to iterate over the values of a DataFrame.

To iterate over the rows of a DataFrame, you can use the following methods:

- `iterrows()`: Iterate over the rows of a DataFrame as (index, Series) pairs. This converts the rows to Series objects, which can change the dtypes and has some performance implications.
- `itertuples()`: Iterate over the rows of a DataFrame as namedtuples of the values. This is a lot faster than `iterrows()`, and is in most cases preferable to use to iterate over the values of a DataFrame.

Warning

Iterating through pandas objects is generally **slow**. In many cases, iterating manually over the rows is not needed and can be avoided with one of the following approaches:

- Look for a *vectorized* solution: many operations can be performed using built-in methods or NumPy functions, (boolean) indexing, …
- When you have a function that cannot work on the full DataFrame/Series at once, it is better to use `apply()` instead of iterating over the values. See the docs on function application.
- If you need to do iterative manipulations on the values but performance is important, consider writing the inner loop with cython or numba. See the enhancing performance section for some examples of this approach.

точка

apply?

apply?

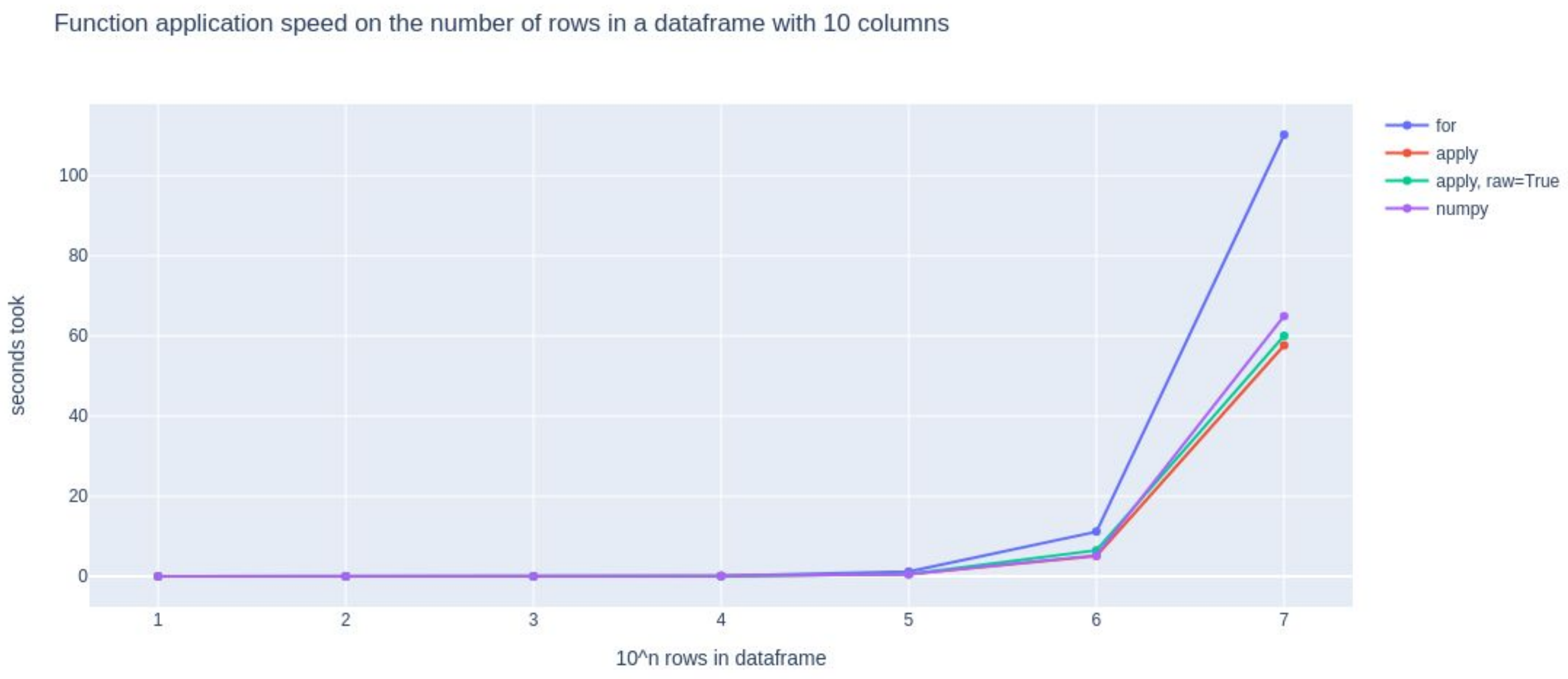
Apply

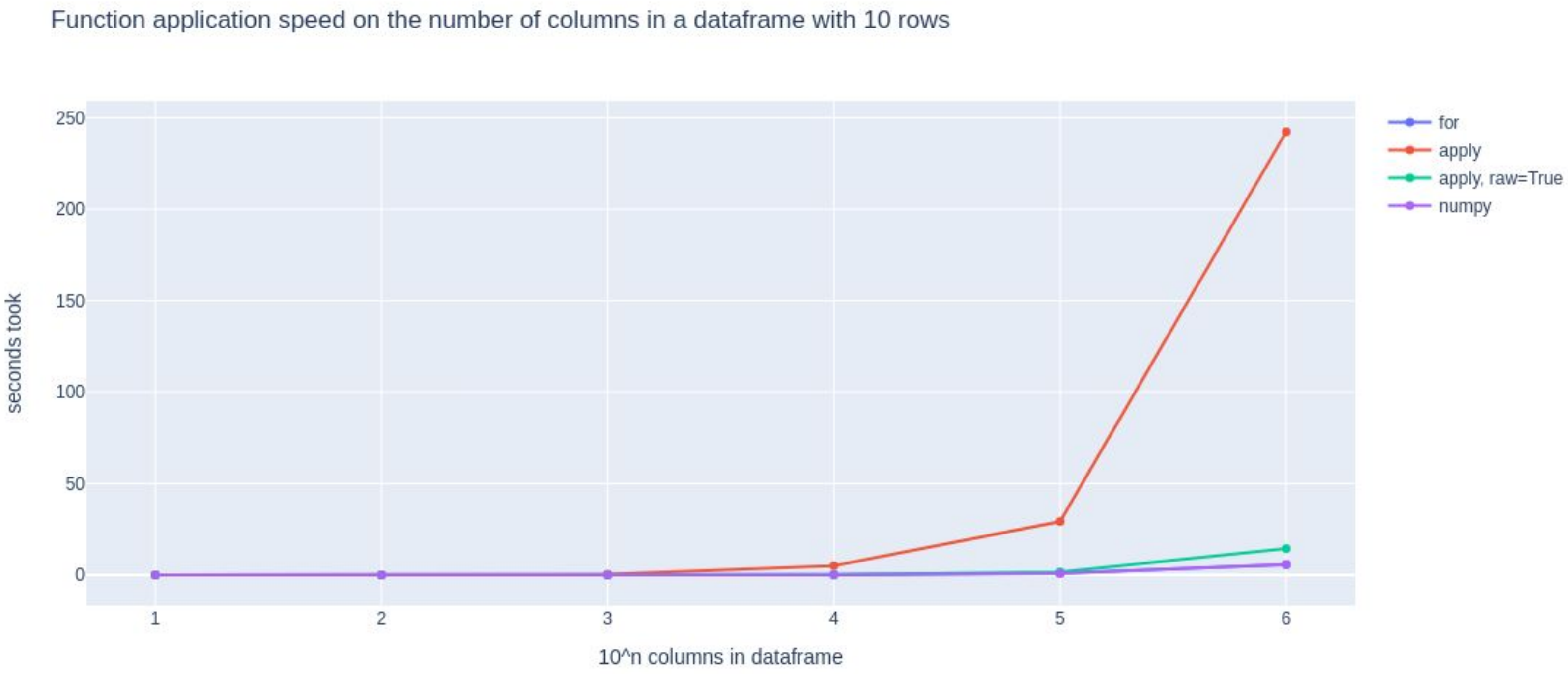
The ‘apply' function effectively does the same thing as the loop. It will create a new column titled reward and apply the calculatio
a faster way to run a loop to yo

3. apply() is just a glorified for loop:

- A more traditional way of applying a function to dataframe rows involves using `apply()` method.
- Under the hood, `apply()` uses a loop with an added overhead. It can often be avoided by leveraging vectorized operations.

точка


Function application speed on the number of rows in a dataframe with 10 columns
seconds took
0
20
40
60
80
100
1
2
3
4
5
6
7
10^n rows in dataframe
for
apply
apply, raw=True
numpy


Function application speed on the number of columns in a dataframe with 10 rows
250
200
150
100
50
0
seconds took
1
2
3
4
5
6
10^n columns in dataframe
for
apply
apply, raw=True
numpy

точка

[6]:
```
# for
%memit [sum_with_changing_sign(i) for i in df.to_numpy()]
```
```
peak memory: 219.48 MiB, increment: 2.40 MiB
```

[6]:
```
# apply
%memit df.apply(sum_with_changing_sign)
```
```
peak memory: 218.34 MiB, increment: 3.28 MiB
```

[6]:
```
# apply, raw=True
%memit df.apply(sum_with_changing_sign, raw=True)
```
```
peak memory: 217.59 MiB, increment: 0.72 MiB
```

[6]:
```
# numpy
%memit np.sum(df.to_numpy() * ((np.indices(df.shape).sum(axis=0) % 2) + ((np.indices(df.shape).sum(axis=0) % 2)-1))*(-1), axi
```
```
peak memory: 520.99 MiB, increment: 303.49 MiB
```

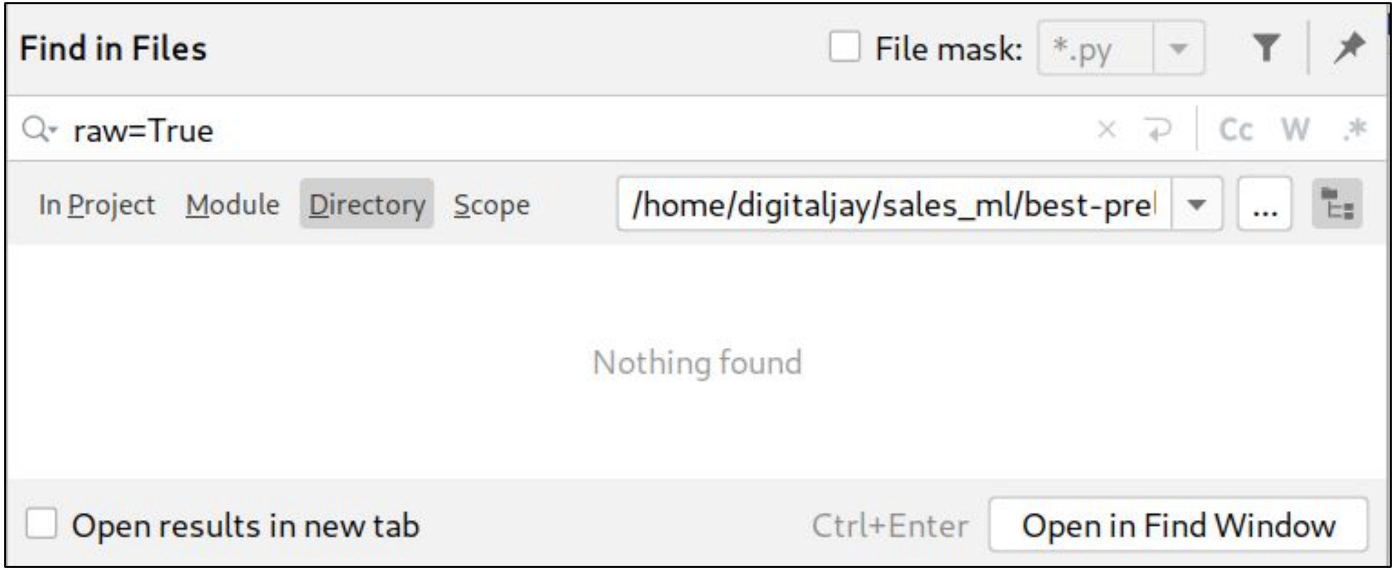
ТОЧКА

Function application

To apply your own or another library's functions to pandas objects, you should be aware of the three methods below. The appropriate method to use depends on whether your function expects to operate on an entire `DataFrame` or `Series`, row- or column-wise, or elementwise.

1. Tablewise Function Application: `pipe()`
2. Row or Column-wise Function Application: `apply()`
3. Aggregation API: `agg()` and `transform()`
4. Applying Elementwise Functions: `map()`


Find in Files 58 matches in 28 files
File mask:
*.py
.apply(
Cc W .*
In Project
Module
Directory
Scope
/home/digitaljay/sales_ml/best-pre
...


Find in Files
File mask:
*.py
raw=True
Cc W .*
In Project
Module
Directory
Scope
/home/digitaljay/sales_ml/best-pre
...
Nothing found
Open results in new tab
Ctrl+Enter
Open in Find Window

1.8x speed up

apply -> apply(raw=True)

Ура! Теперь до конца просчета данных - всего **100** дней!

```
df = no_phone_key[["id", "ad_login"]].merge(df, on=["id", "ad_login"], how="left")
```

точка

merge

Database-style DataFrame or named Series joining/merging

pandas has full-featured, **high performance** in-memory join operations idiomatically very similar to relational databases like SQL. These methods perform significantly better (in some cases well over an order of magnitude better) than other open source implementations (like `base::merge.data.frame` in R). The reason for this is careful algorithmic design and the internal layout of the data in `DataFrame`.

See the cookbook for some advanced strategies.

Users who are familiar with SQL but new to pandas might be interested in a comparison with SQL.

pandas provides a single function, `merge()`, as the entry point for all standard database join operations between `DataFrame` or named `Series` objects:

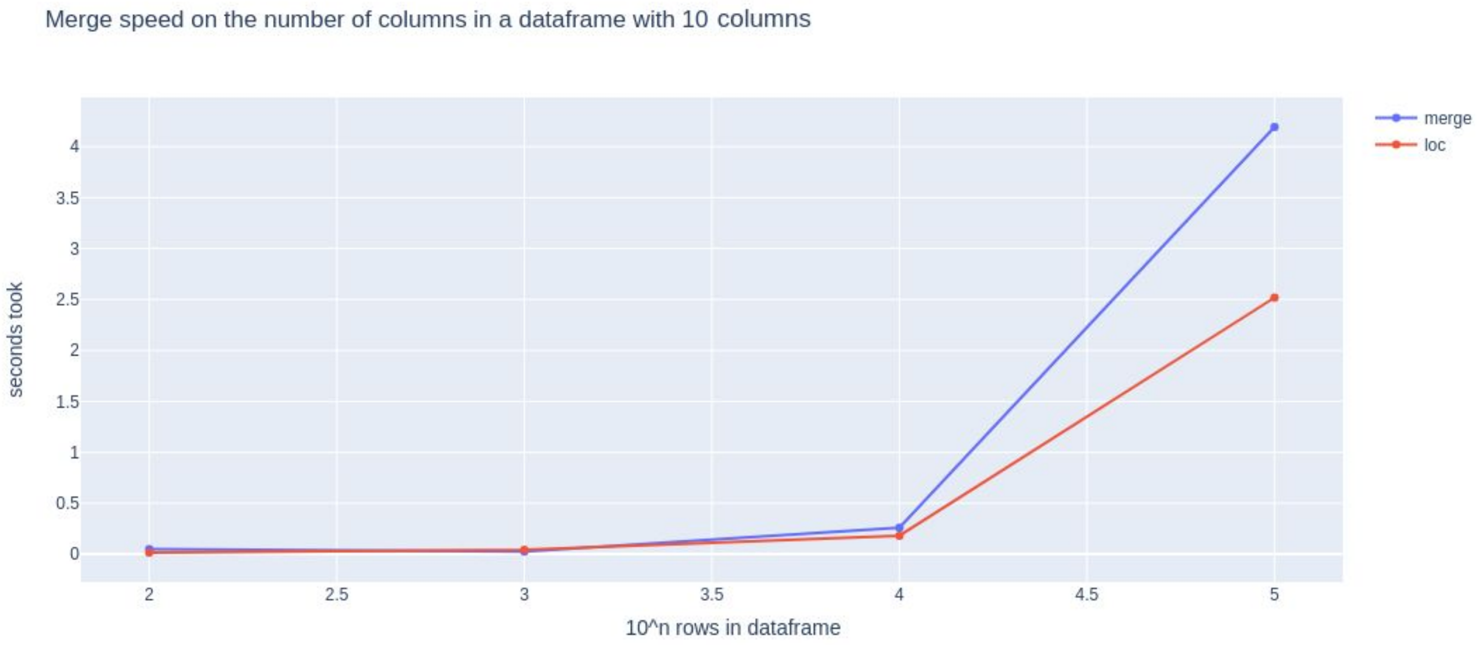
merge: нужно выбрать подвыборку

точка

```diff
178      -            df = no_phone_key[["id", "ad_login"]].merge(df, on=["id", "ad_login"], how="left")
     179 +            df = df.set_index(["id", "ad_login"]).loc[no_phone_key[["id", "ad_login"]], :].reset_index()
```

merge: нужно выбрать подвыборку


```
178 -    df = no_phone_key[["id", "ad_login"]].merge(df, on=["id", "ad_login"], how="left")
179 +    df = df.set_index(["id", "ad_login"]).loc[no_phone_key[["id", "ad_login"]], :].reset_index()
```



merge: нужно выбрать подвыборку


```
[5]: # merge
     %memit no_phone_key[['id', 'ad_login']].merge(df, on=['id', 'ad_login'], how='left')

     peak memory: 2494.50 MiB, increment: 1046.69 MiB

[5]: # loc
     %memit df.set_index(['id', 'ad_login']).loc[no_phone_key[['id', 'ad_login']].values.tolist(), :].reset_index()

     peak memory: 2269.16 MiB, increment: 818.97 MiB
```

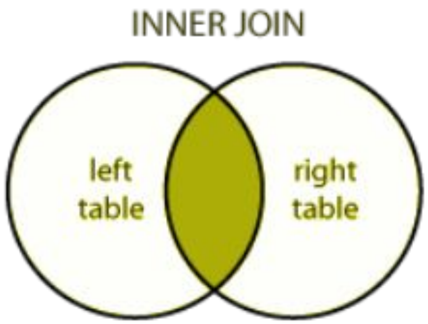
merge: нужно выбрать подвыборку

точка

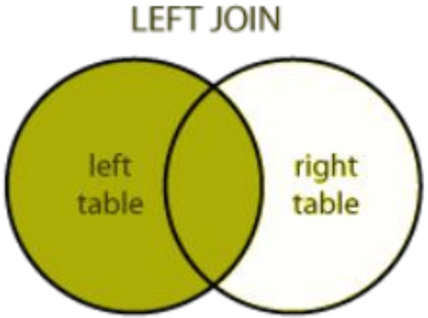
```diff
178     -   df = no_phone_key[["id", "ad_login"]].merge(df, on=["id", "ad_login"], how="left")
   179  +   df = df.set_index(["id", "ad_login"]).loc[no_phone_key[["id", "ad_login"]], :].reset_index()
```

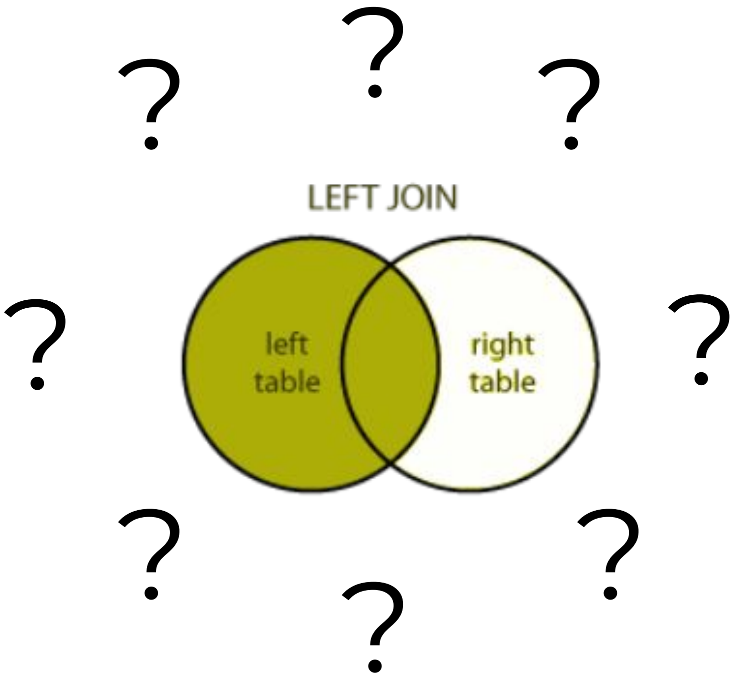
merge: нужно выбрать подвыборку

```diff
- df = no_phone_key[["id", "ad_login"]].merge(df, on=["id", "ad_login"], how="left")
+ df = df.set_index(["id", "ad_login"]).loc[no_phone_key[["id", "ad_login"]], :].reset_index()
```



merge: нужно выбрать подвыборку




?
?
?
LEFT JOIN
?
left
table
right
table
?
?
?
?

merge: когда просто loc не канает

```diff
-                df = no_phone_key[["id", "ad_login"]].merge(df, on=["id", "ad_login"], how="left")
-
+                df = loc_with_fill(
+                    fields_to_loc=["id", "ad_login"],
+                    frame_to_take_as_index=no_phone_key,
+                    frame_to_loc_from=df,
+                    drop_duplicates=False,
+                )
```

точка

```
def loc_with_fill(
    fields_to_loc: List[str],
    frame_to_take_as_index: DataFrame,
    frame_to_loc_from: DataFrame,
    drop_duplicates=False,
) -> DataFrame:
    ind_frame = frame_to_take_as_index[fields_to_loc]
    if drop_duplicates:
        ind_frame = ind_frame.drop_duplicates()
    if len(fields_to_loc) > 1:
        reindex = MultiIndex.from_frame(ind_frame)
    else:
        reindex = Index(ind_frame[fields_to_loc[0]].values)
    return frame_to_loc_from.set_index(fields_to_loc).reindex(reindex).reset_index(names=fields_to_loc)
```

merge: когда просто loc не канает

точка

```python
def loc_with_fill(
    fields_to_loc: List[str],
    frame_to_take_as_index: DataFrame,
    frame_to_loc_from: DataFrame,
    drop_duplicates=False,
) -> DataFrame:
    ind_frame = frame_to_take_as_index[fields_to_loc]
    if drop_duplicates:
        ind_frame = ind_frame.drop_duplicates()
    if len(fields_to_loc) > 1:
        reindex = MultiIndex.from_frame(ind_frame)
    else:
        reindex = Index(ind_frame[fields_to_loc[0]].values)
    return frame_to_loc_from.set_index(fields_to_loc).reindex(reindex).reset_index(names=fields_to_loc)
```

pandas.DataFrame.reindex

DataFrame.reindex(*labels=None*, *, *index=None*, *columns=None*, *axis=None*, *method=None*, *copy=None*, *level=None*, *fill_value=nan*, *limit=None*, *tolerance=None*)

Conform DataFrame to new index with optional filling

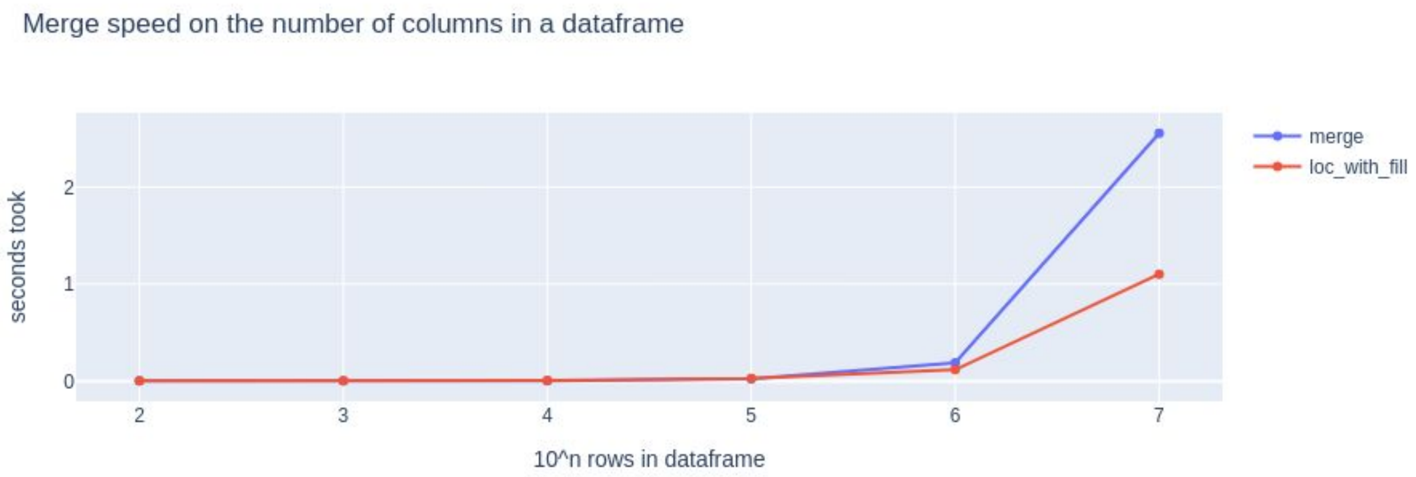
Places NA/NaN in locations having no value in the pr
unless the new index is equivalent to the current one

```
>>> index = ['Firefox', 'Chrome', 'Safari', 'IE10', 'Konqueror']
>>> df = pd.DataFrame({'http_status': [200, 200, 404, 404, 301],
...                    'response_time': [0.04, 0.02, 0.07, 0.08, 1.0]},
...                    index=index)
>>> df
           http_status  response_time
Firefox            200           0.04
Chrome             200           0.02
Safari             404           0.07
IE10               404           0.08
Konqueror          301           1.00
```

Create a new index and reindex the dataframe. By default values in the new index that do not have corresponding records in the dataframe are assigned `NaN`.

```
>>> new_index = ['Safari', 'Iceweasel', 'Comodo Dragon', 'IE10',
...              'Chrome']
>>> df.reindex(new_index)
               http_status  response_time
Safari               404.0           0.07
Iceweasel              NaN            NaN
Comodo Dragon          NaN            NaN
IE10                 404.0           0.08
Chrome               200.0           0.02
```

```
-            df = no_phone_key[["id", "ad_login"]].merge(df, on=["id", "ad_login"], how="left")
-
+            df = loc_with_fill(
+                fields_to_loc=["id", "ad_login"],
+                frame_to_take_as_index=no_phone_key,
+                frame_to_loc_from=df,
+                drop_duplicates=False,
+            )
```



merge: когда просто loc не канает

точка

```
[5]: # merge
%memit no_phone_key[['id', 'ad_login']].merge(df, on=['id', 'ad_login'], how='left')
```

```
peak memory: 7195.32 MiB, increment: 96.28 MiB
```

```
[5]: # loc_with_fill
%memit loc_with_fill(fields_to_loc=['id', 'ad_login'], frame_to_take_as_index=no_phone_key, frame_to_loc_from=df, drop_duplicat
```

```
peak memory: 7206.66 MiB, increment: 106.60 MiB
```

merge: а правда ли merge - это join?

```diff
-            df = no_phone_key[["id", "ad_login"]].merge(df, how="left", on=["id"])
+            df = no_phone_key[["id", "ad_login"]].set_index("id").join(df.set_index("id"), how="left").reset_index()
```

merge: а правда ли merge - это join?

точка

```
193  -    df = no_phone_key[["id", "ad_login"]].merge(df, how="left", on=["id"])
194  +    df = no_phone_key[["id", "ad_login"]].set_index("id").join(df.set_index("id"), how="left").reset_index()
```

merge: а правда ли merge - это join?

точка

```diff
193  -            df = no_phone_key[["id", "ad_login"]].merge(df, how="left", on=["id"])
194  +            df = no_phone_key[["id", "ad_login"]].set_index("id").join(df.set_index("id"), how="left").reset_index()
```

2696 строк сурсов для оптимизации

merge: а правда ли merge - это join?

точка

```
[5]: # merge
%memit df1.merge(df2, left_on=['df1_join1'], right_on=['df2_join1'], how='inner')
```

```
peak memory: 5105.04 MiB, increment: 3129.31 MiB
```

```
[5]: # join
%memit df1.set_index('df1_join1').join(df2.set_index('df2_join1'), how='inner').reset_index()
```

```
peak memory: 8457.75 MiB, increment: 6480.49 MiB
```

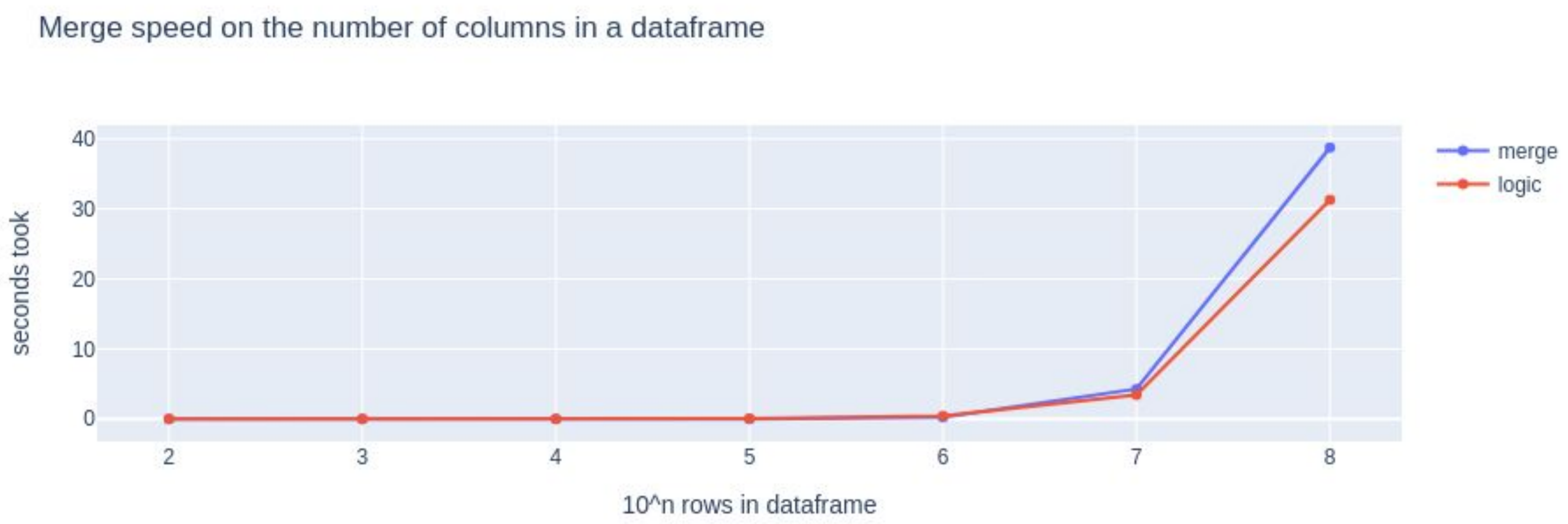
merge: indicator и его оптимизация


```diff
-cached_features = validate_prelead_features(cached_features, feature_category)
-cache_presence_indicator = key.merge(
-    cached_features[index_cols].drop_duplicates(),
-    indicator=True,
-    how="left",
-    on=index_cols,
-)
+cached_features = validate_prelead_features(cached_features, feature_category).set_index(index_cols)
+
+key = key.set_index(index_cols)
+if len(cached_features) > 0:
+    keys_in_cache = (
+        key.copy().loc[key.index.intersection(cached_features.index), :].reset_index(names=index_cols)
+    )
+    keys_not_in_cache = (
+        key.copy().loc[key.index.difference(cached_features.index), :].reset_index(names=index_cols)
+    )
+else:
+    keys_in_cache = pd_DataFrame(columns=key.columns)
+    keys_not_in_cache = key.copy().reset_index(names=index_cols)
 
-keys_in_cache = key[(cache_presence_indicator["_merge"] == "both").values]
-keys_not_in_cache = key[(cache_presence_indicator["_merge"] == "left_only").values]
+key.reset_index(names=index_cols, inplace=True)
+cached_features.reset_index(names=index_cols, inplace=True)
```

merge: indicator и его оптимизация

точка

```diff
-cached_features = validate_prelead_features(cached_features, feature_category)
-cache_presence_indicator = key.merge(
-    cached_features[index_cols].drop_duplicates(),
-    indicator=True,
-    how="left",
-    on=index_cols,
-)
+cached_features = validate_prelead_features(cached_features, feature_category).set_index(index_cols)
+
+key = key.set_index(index_cols)
+if len(cached_features) > 0:
+    keys_in_cache = (
+        key.copy().loc[key.index.intersection(cached_features.index), :].reset_index(names=index_cols)
+    )
+    keys_not_in_cache = (
+        key.copy().loc[key.index.difference(cached_features.index), :].reset_index(names=index_cols)
+    )
+else:
+    keys_in_cache = pd_DataFrame(columns=key.columns)
+    keys_not_in_cache = key.copy().reset_index(names=index_cols)

-keys_in_cache = key[(cache_presence_indicator["_merge"] == "both").values]
-keys_not_in_cache = key[(cache_presence_indicator["_merge"] == "left_only").values]
+key.reset_index(names=index_cols, inplace=True)
+cached_features.reset_index(names=index_cols, inplace=True)
```

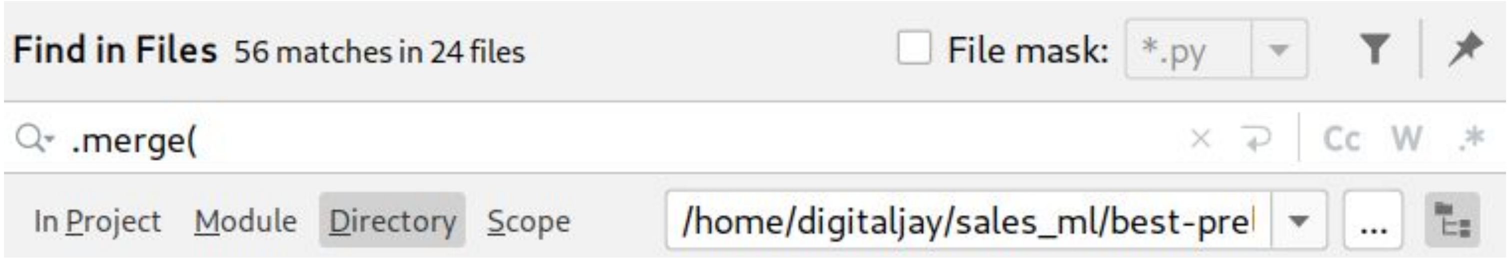

Merge speed on the number of columns in a dataframe
seconds took
40
30
20
10
0
2
3
4
5
6
7
8
10^n rows in dataframe
merge
logic

merge: indicator и его оптимизация

точка

```
[6]: # merge
     %memit merge_logic(key, cached_features, index_cols)

peak memory: 7243.54 MiB, increment: 83.61 MiB
```

```
[6]: # logic with intersection
     %memit intersect_logic(key, cached_features, index_cols)

peak memory: 7234.76 MiB, increment: 79.11 MiB
```


Find in Files 56 matches in 24 files
File mask:
*.py
.merge(
Cc W .*
In Project Module Directory Scope
/home/digitaljay/sales_ml/best-pre
...

до отпуска - 10 дней

2x speed up

merge -> loc/reindex

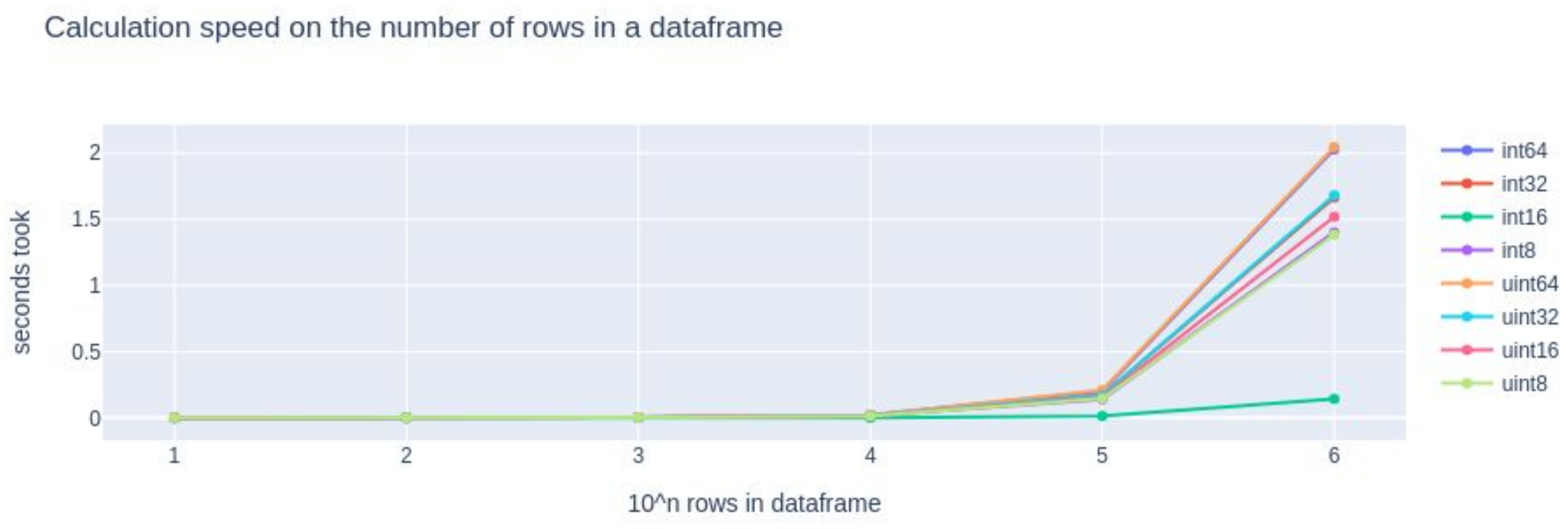
Ура! Теперь до конца просчета данных - всего **50** дней!

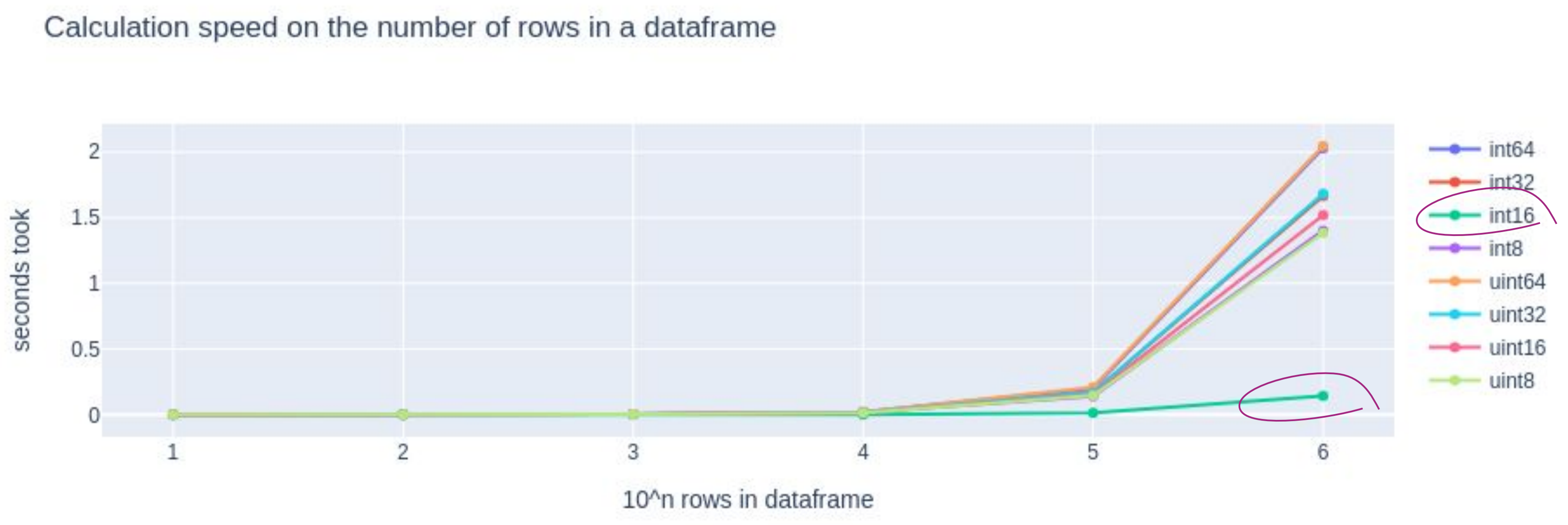
типы данных

точка

Kind of Data	Data Type	Scalar	Array	String Aliases
tz-aware datetime	DatetimeTZDtype	Timestamp	arrays.DatetimeArray	'datetime64[ns, <tz>
Categorical	CategoricalDtype	(none)	Categorical	'category'
period (time spans)	PeriodDtype	Period	arrays.PeriodArray 'Period[<freq>]'	'period[<freq>]',
sparse	SparseDtype	(none)	arrays.SparseArray	'Sparse', 'Sparse[in 'Sparse[float]'
intervals	IntervalDtype	Interval	arrays.IntervalArray	'interval', 'Interva 'Interval[<numpy_dty 'Interval[datetime64 <tz>]]', 'Interval[timedelta6
nullable integer	Int64Dtype, ...	(none)	arrays.IntegerArray	'Int8', 'Int16', 'In 'Int64', 'UInt8', 'U 'UInt32', 'UInt64'
nullable float	Float64Dtype, ...	(none)	arrays.FloatingArray	'Float32', 'Float64'
Strings	StringDtype	str	arrays.StringArray	'string'
Boolean (with NA)	BooleanDtype	bool	arrays.BooleanArray	'boolean'

точка


Calculation speed on the number of rows in a dataframe
seconds took
2
1.5
1
0.5
0
1
2
3
4
5
6
10^n rows in dataframe
int64
int32
int16
int8
uint64
uint32
uint16
uint8


Calculation speed on the number of rows in a dataframe
seconds took
10^n rows in dataframe
2
1.5
1
0.5
0
1
2
3
4
5
6
int64
int32
int16
int8
uint64
uint32
uint16
uint8

0

Addressing a specific part of your quote:

> For example, when reading a 16-bit value on a 64-bit machine, a full 64 bits worth of data must still be read from memory. The desired 16-bit field then has to be masked off and possibly shifted into place within the destination register.

This is true for some architectures, but not all of them. Particularly, the x86-64 architecture has instructions for working directly with 16-bit values, so doing so on that architecture will not require mask and shift operations. Futhermore, while 64 bits[1] will still be fetched from memory, if subsequent instructions need to use the data in the other 48 bits they will be able to do so without needing to access main memory again due to the processor's cache. For this architecture specifically, therefore, this advice is wrong.

[1]: or, more likely, 128 bits, as this is the width of the cache on most modern processors

Share Improve this answer Follow

answered Jun 15, 2020 at 23:00

occipita

209 • 2 • 5

типы данных: int

точка

```
[3]: # int64
%memit (pd.DataFrame(np.random.randint(100, size=(10**7, 10)), dtype='int64')>50).sum(axis=1)
peak memory: 997.96 MiB, increment: 852.95 MiB

[3]: # uint64
%memit (pd.DataFrame(np.random.randint(100, size=(10**7, 10)), dtype='uint64')>50).sum(axis=1)
peak memory: 1755.40 MiB, increment: 1611.11 MiB

[3]: # int32
%memit (pd.DataFrame(np.random.randint(100, size=(10**7, 10)), dtype='int32')>50).sum(axis=1)
peak memory: 1378.75 MiB, increment: 1234.32 MiB

[3]: # uint32
%memit (pd.DataFrame(np.random.randint(100, size=(10**7, 10)), dtype='uint32')>50).sum(axis=1)
peak memory: 1377.25 MiB, increment: 1234.32 MiB

[3]: # int16
%memit (pd.DataFrame(np.random.randint(100, size=(10**7, 10)), dtype='int16')>50).sum(axis=1)
peak memory: 1181.45 MiB, increment: 1038.37 MiB

[3]: # uint16
%memit (pd.DataFrame(np.random.randint(100, size=(10**7, 10)), dtype='uint16')>50).sum(axis=1)
peak memory: 1173.63 MiB, increment: 1030.44 MiB

[3]: # int8
%memit (pd.DataFrame(np.random.randint(100, size=(10**7, 10)), dtype='int8')>50).sum(axis=1)
peak memory: 1092.50 MiB, increment: 950.10 MiB

[3]: # uint8
%memit (pd.DataFrame(np.random.randint(100, size=(10**7, 10)), dtype='uint8')>50).sum(axis=1)
peak memory: 1095.40 MiB, increment: 951.97 MiB
```

типы данных: int

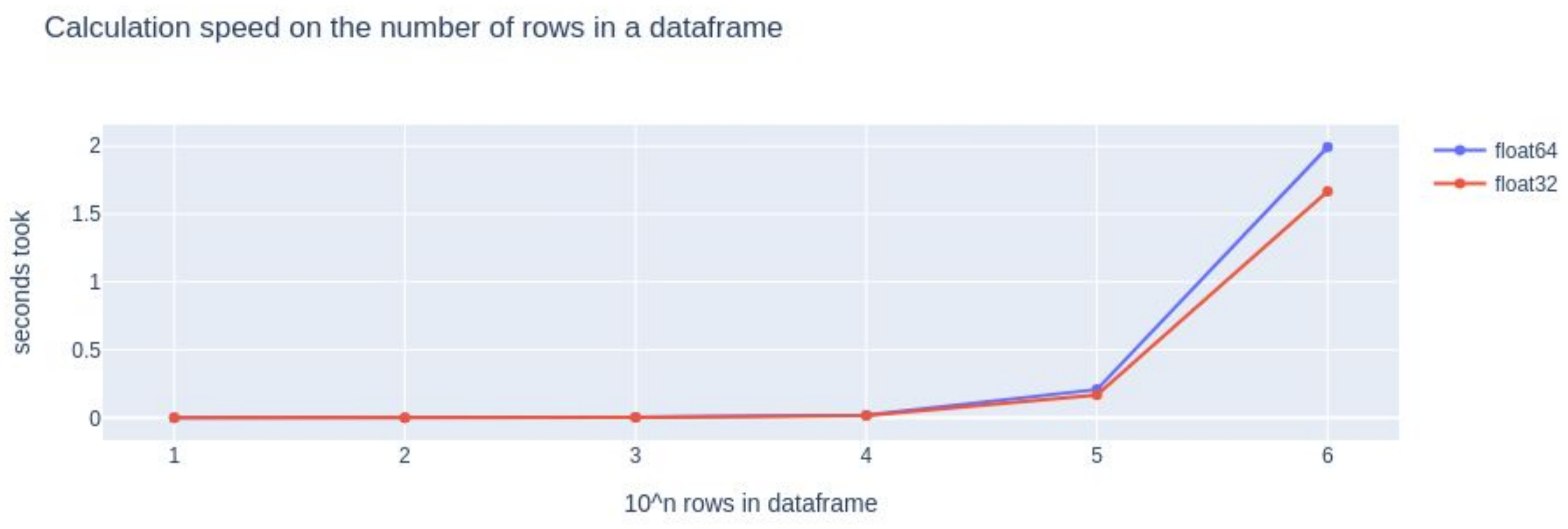
точка

```
[3]: # int64
%memit (pd.DataFrame(np.random.randint(100, size=(10**7, 10)), dtype='int64')>50).sum(axis=1)
peak memory: 997.96 MiB, increment: 852.95 MiB

[3]: # uint64
%memit (pd.DataFrame(np.random.randint(100, size=(10**7, 10)), dtype='uint64')>50).sum(axis=1)
peak memory: 1755.40 MiB, increment: 1611.11 MiB

[3]: # int32
%memit (pd.DataFrame(np.random.randint(100, size=(10**7, 10)), dtype='int32')>50).sum(axis=1)
peak memory: 1378.75 MiB, increment: 1234.32 MiB

[3]: # uint32
%memit (pd.DataFrame(np.random.randint(100, size=(10**7, 10)), dtype='uint32')>50).sum(axis=1)
peak memory: 1377.25 MiB, increment: 1234.32 MiB

[3]: # int16
%memit (pd.DataFrame(np.random.randint(100, size=(10**7, 10)), dtype='int16')>50).sum(axis=1)
peak memory: 1181.45 MiB, increment: 1038.37 MiB

[3]: # uint16
%memit (pd.DataFrame(np.random.randint(100, size=(10**7, 10)), dtype='uint16')>50).sum(axis=1)
peak memory: 1173.63 MiB, increment: 1030.44 MiB

[3]: # int8
%memit (pd.DataFrame(np.random.randint(100, size=(10**7, 10)), dtype='int8')>50).sum(axis=1)
peak memory: 1092.50 MiB, increment: 950.10 MiB

[3]: # uint8
%memit (pd.DataFrame(np.random.randint(100, size=(10**7, 10)), dtype='uint8')>50).sum(axis=1)
peak memory: 1095.40 MiB, increment: 951.97 MiB
```

типы данных: int

точка

```
[3]: df = pd.DataFrame(np.random.randint(100, size=(10**7, 100)), dtype='int64')

[4]: # int64
     %memit df

     peak memory: 7774.91 MiB, increment: 0.31 MiB

[3]: df = pd.DataFrame(np.random.randint(100, size=(10**7, 10)), dtype='uint64')

[4]: # uint64
     %memit df

     peak memory: 904.18 MiB, increment: 0.06 MiB

[3]: df = pd.DataFrame(np.random.randint(100, size=(10**7, 10)), dtype='int32')

[4]: # int32
     %memit df

     peak memory: 537.11 MiB, increment: 0.04 MiB

[3]: df = pd.DataFrame(np.random.randint(100, size=(10**7, 10)), dtype='uint32')

[4]: # uint32
     %memit df

     peak memory: 521.03 MiB, increment: 0.04 MiB

[3]: df = pd.DataFrame(np.random.randint(100, size=(10**7, 10)), dtype='int16')

[4]: # int16
     %memit df

     peak memory: 332.25 MiB, increment: 0.04 MiB

[3]: df = pd.DataFrame(np.random.randint(100, size=(10**7, 10)), dtype='uint16')

[4]: # uint16
     %memit df

     peak memory: 334.51 MiB, increment: 0.03 MiB

[3]: df = pd.DataFrame(np.random.randint(100, size=(10**7, 10)), dtype='int8')

[4]: # int8
     %memit df

     peak memory: 234.75 MiB, increment: 0.04 MiB

[3]: df = pd.DataFrame(np.random.randint(100, size=(10**7, 10)), dtype='uint8')

[4]: # uint8
     %memit df

     peak memory: 240.97 MiB, increment: 0.04 MiB
```


Calculation speed on the number of rows in a dataframe
seconds took
2
1.5
1
0.5
0
1
2
3
4
5
6
10^n rows in dataframe
float64
float32

точка

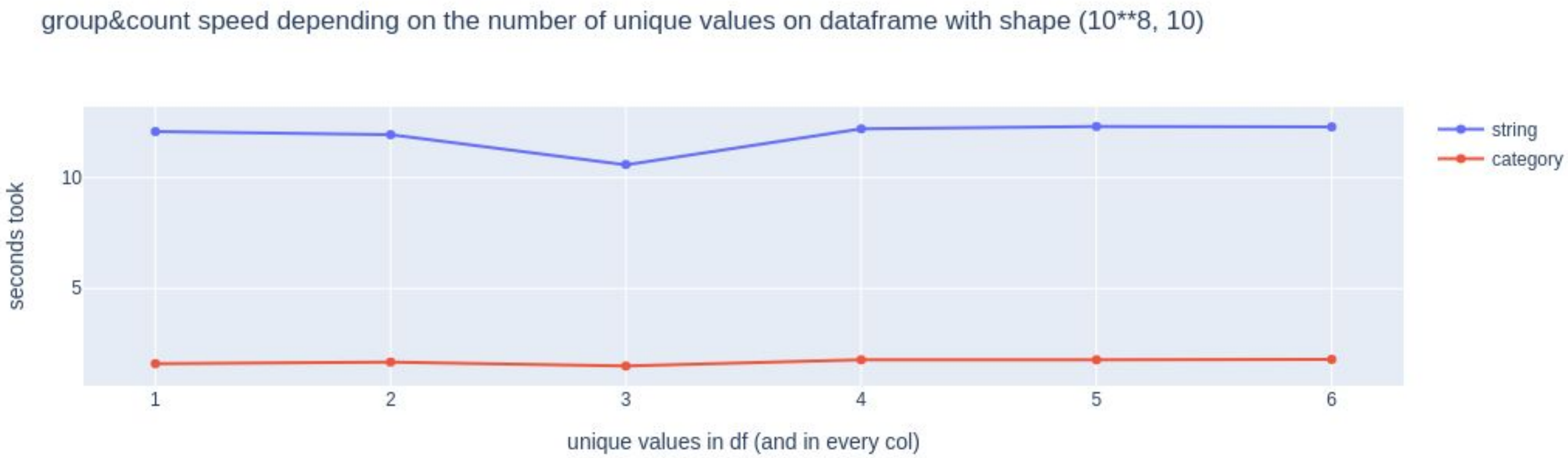
```
[3]: # float64
%memit pd.DataFrame(np.random.rand(10**7, 10), dtype='float64')
```

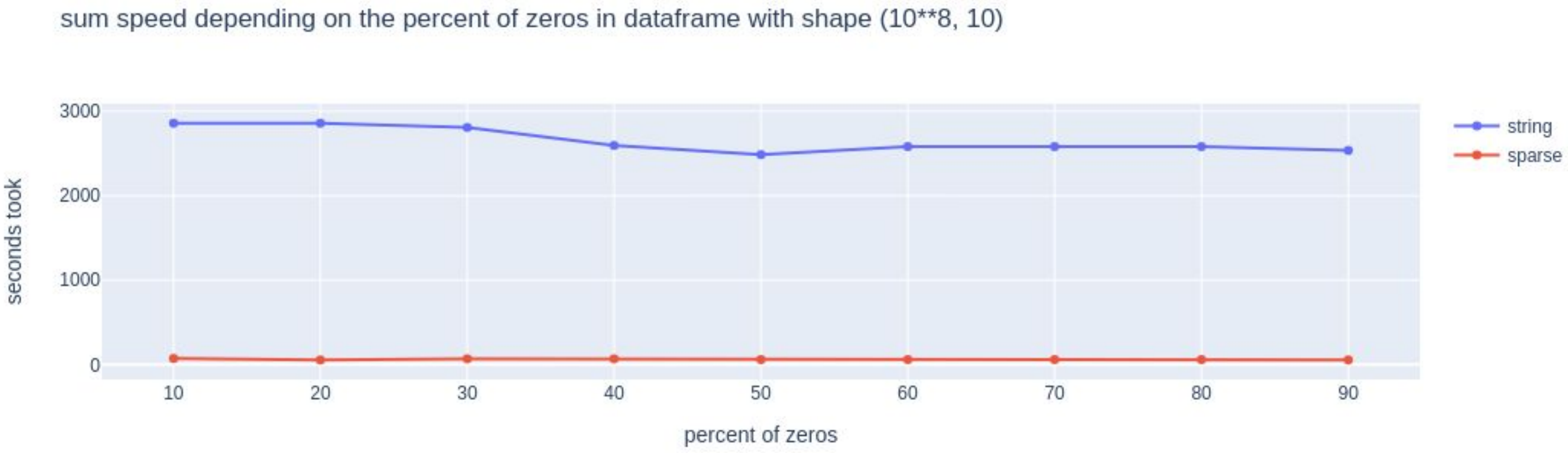
```
peak memory: 880.34 MiB, increment: 739.81 MiB
```

```
[3]: # float32
%memit pd.DataFrame(np.random.rand(10**7, 10), dtype='float32')
```

```
peak memory: 1284.04 MiB, increment: 1139.30 MiB
```

```
# без этой эвристики начинает конвертировать всякие id'шники, а это жрёт оперативку и время
# TODO: [redacted] поправь, критично, никогда не бывает больше 1
categorical_can_be_used = (len(col.unique()) / (len(col) + 1) >= 3) and colname in categ_cols
```


group&count speed depending on the number of unique values on dataframe with shape (10**8, 10)
seconds took
10
5
1
2
3
4
5
6
unique values in df (and in every col)
string
category


sum speed depending on the percent of zeros in dataframe with shape (10**8, 10)
3000
2000
1000
0
seconds took
10
20
30
40
50
60
70
80
90
percent of zeros
string
sparse

[3]:
```
# string
%memit pd.DataFrame(np.random.randint(0, 10**6, (10**6, 10)), dtype='str')
```

```
peak memory: 903.05 MiB, increment: 764.17 MiB
```

[3]:
```
# category
%memit pd.DataFrame(np.random.randint(0, 10**6, (10**6, 10)), dtype='category')
```

```
peak memory: 517.06 MiB, increment: 376.34 MiB
```

[3]:
```
# string
%memit pd.DataFrame(np.random.randint(0, 10, (10**6, 10)), dtype='str')
```

```
peak memory: 900.20 MiB, increment: 757.54 MiB
```

[3]:
```
# sparse
%memit pd.DataFrame(np.random.randint(0, 10, (10**6, 10)), dtype='Sparse[string]')
```

```
peak memory: 344.79 MiB, increment: 203.62 MiB
```

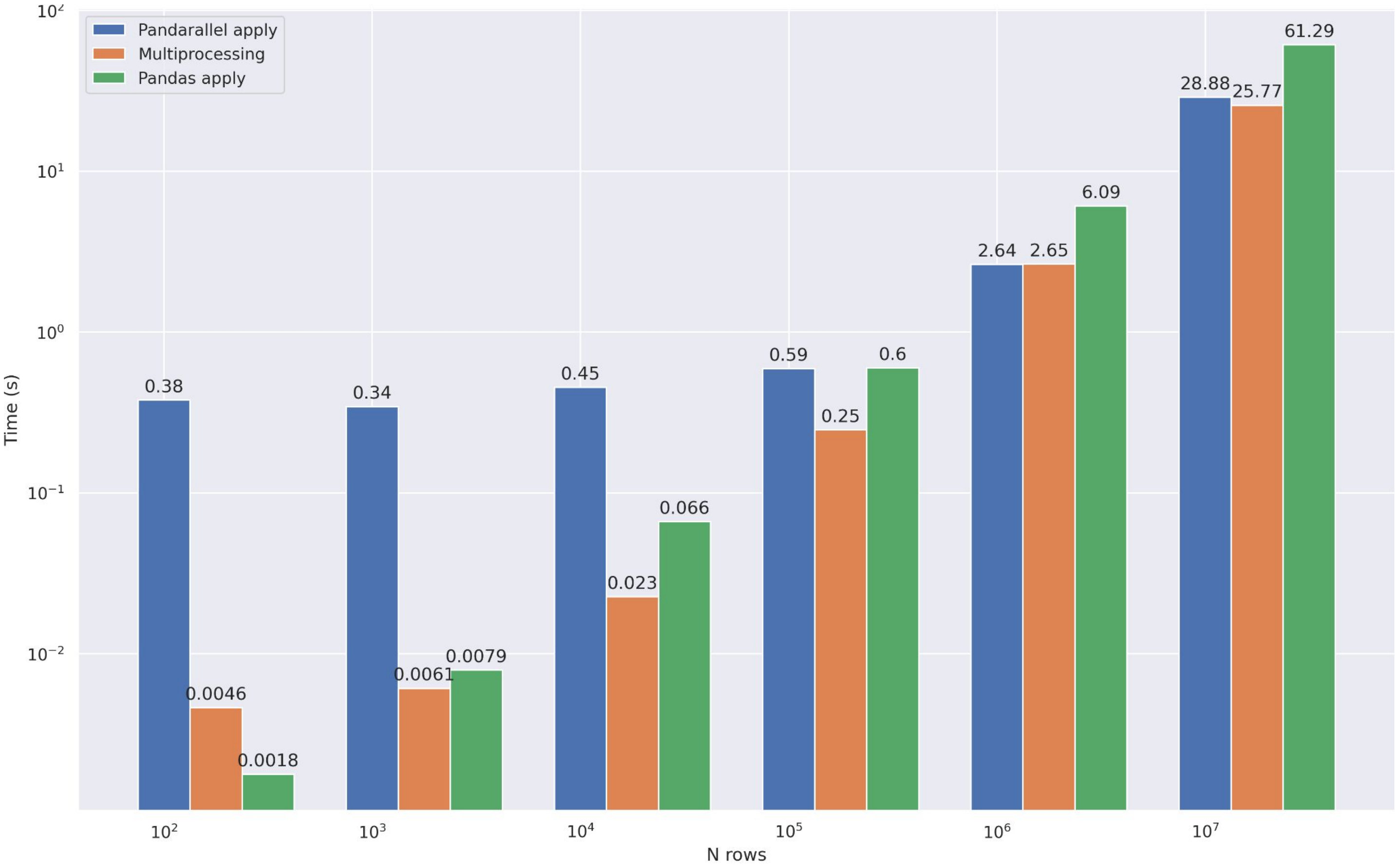
точка

2.5x speed up

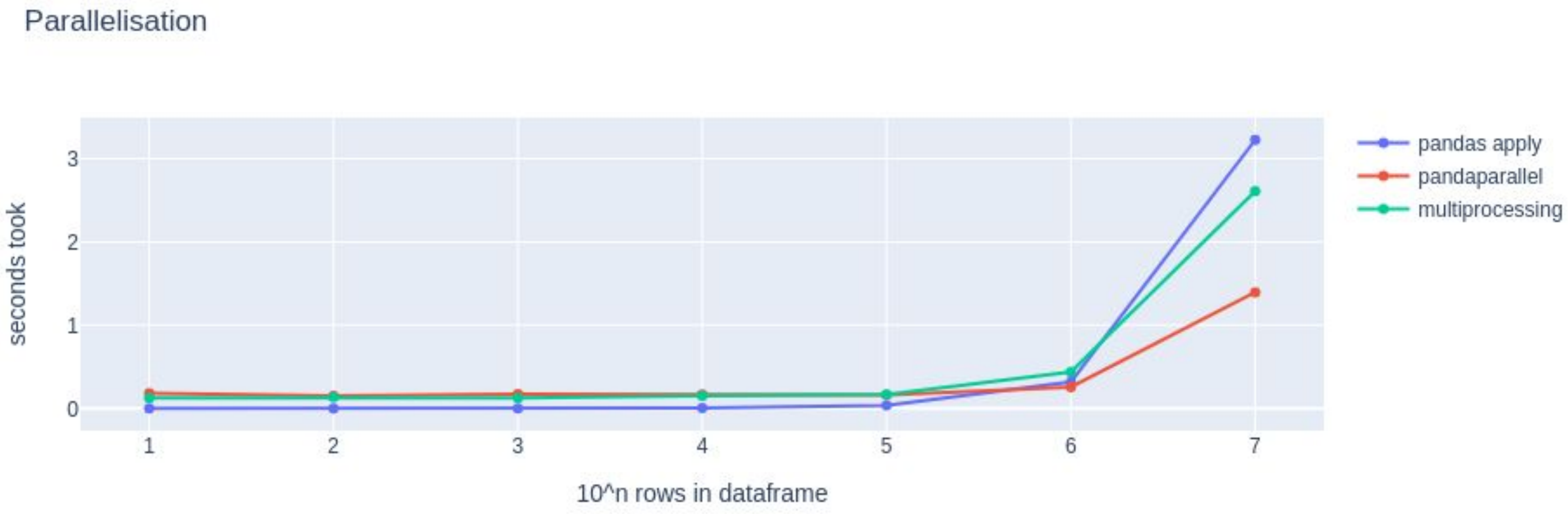
int64, float64 -> int8, float32

Ура! Теперь до конца просчета данных - всего **20** дней!

parallel



Pandarallel apply
Multiprocessing
Pandas apply
Time (s)
N rows
10^2
10^1
10^0
10^{-1}
10^{-2}
10^2
10^3
10^4
10^5
10^6
10^7
0.38
0.0046
0.0018
0.34
0.0061
0.0079
0.45
0.023
0.066
0.59
0.25
0.6
2.64
2.65
6.09
28.88
25.77
61.29


Parallelisation
seconds took
0
1
2
3
1
2
3
4
5
6
7
10^n rows in dataframe
pandas apply
pandaparallel
multiprocessing

```
[7]: def apply_func(i):
         return i**2
```

```
[8]: df = pd.DataFrame(np.random.rand(10**7, 1), columns=["sample_column"])
```

```
•[9]: # pandas apply
      %memit df['sample_column'].apply(apply_func)

peak memory: 1195.19 MiB, increment: 978.40 MiB
```

```
•[9]: # pandaparallel
      %memit df['sample_column'].parallel_apply(apply_func)

peak memory: 352.48 MiB, increment: 137.52 MiB
```

```
•[9]: # multiprocessing
      %memit mp.Pool().map(apply_func, df['sample_column'])

peak memory: 599.39 MiB, increment: 382.56 MiB
```

Without parallelization	With parallelization
`df.apply(func)`	`df.parallel_apply(func)`
`df.applymap(func)`	`df.parallel_applymap(func)`
`df.groupby(args).apply(func)`	`df.groupby(args).parallel_apply(func)`
`df.groupby(args1).col_name.rolling(args2).apply(func)`	`df.groupby(args1).col_name.rolling(args2).parallel_`
`series.map(func)`	`series.parallel_map(func)`
`series.apply(func)`	`series.parallel_apply(func)`
`series.rolling(args).apply(func)`	`series.rolling(args).parallel_apply(func)`

3x speed up

pandaparallel

Ура! Теперь до конца просчета данных - всего **6** дней!


ПОБЕДА!

отпуск


RON MAIDEN

Чего сделать чтобы ускориться?

точка

1. iterrows ----------------> itertuples (если кол-во колонок <1000)

точка

1. iterrows ----------------> itertuples (если кол-во колонок <1000)
2. apply --------------------> apply(raw=True)

Чего сделать чтобы ускориться?

1. iterrows ----------------> itertuples (если кол-во колонок <1000)
2. apply --------------------> apply(raw=True)
3. merge -------------------> loc/reindex

Чего сделать чтобы ускориться?

1. iterrows ----------------> itertuples (если кол-во колонок <1000)
2. apply -------------------> apply(raw=True)
3. merge ------------------> loc/reindex
4. int64, float64 ----------> int8, float32

Чего сделать чтобы ускориться?

точка

1. iterrows ----------------> itertuples (если кол-во колонок <1000)
2. apply -------------------> apply(raw=True)
3. merge ------------------> loc/reindex
4. int64, float64 ---------> int8, float32
5. string -------------------> sparse/categorical

Чего сделать чтобы ускориться?

точка

1. iterrows ----------------> itertuples (если кол-во колонок <1000)
2. apply -------------------> apply(raw=True)
3. merge ------------------> loc/reindex
4. int64, float64 ---------> int8, float32
5. string -------------------> sparse/categorical
6. ----------------------------> pandaparallel

Чего сделать чтобы ускориться?

1. iterrows ----------------> itertuples (если кол-во колонок <1000)
2. apply -------------------> apply(raw=True)
3. merge ------------------> loc/reindex
4. int64, float64 ---------> int8, float32
5. string -------------------> sparse/categorical
6. --------------------------> pandaparallel

Чего сделать чтобы ускориться?

точка

1. iterrows ----------------> itertuples (если кол-во колонок <1000)
2. apply ----------------⚡> apply(raw=True)
3. merge ------------------> loc/reindex
4. int64, float64 ----⚡⚡> int8, float32
5. string ------------------> sparse/categorical
6. ---------------------⚡⚡> pandaparallel

⚡ быстрее всего добавить

Чего сделать чтобы ускориться?

1. iterrows ----------------> itertuples (если кол-во колонок <1000)
2. apply ----------------> apply(raw=True)
3. merge -----------------> loc/reindex
4. int64, float64 ------> int8, float32
5. string ------------------> sparse/categorical
6. -----------------------> pandaparallel

⚡ быстрее всего добавить

сокращают/не меняют потребление памяти

1. Документация != последняя инстанция
2. Не верь статьям без бенчмарков
3. Знание своих данных позволяет писать более оптимальный код

точка

Елизавета Пушкарева

Data Scientist в Точке

tg: @digitaljay

бенчмарки

JET BRAINS Blog

When should you stick with pandas?

All of this sounds so amazing that you're probably wondering why you would even bother with pandas anymore. Not so fast! While Polars is superb for doing extremely efficient data transformations, it is currently not the optimal choice for data exploration or for use as part of machine learning pipelines. These are areas where pandas continues to shine.