



Dask DataFrame

Contents

- [Examples](#)
- [Design](#)
- [Dask DataFrame copies the pandas DataFrame API](#)
- [Common Uses and Anti-Uses](#)
- [Scope](#)
- [Execution](#)

A Dask DataFrame is a large parallel DataFrame composed of many smaller pandas DataFrames, split along the index. These pandas DataFrames may live on disk for larger-than-memory computing on a single machine, or on many different machines in a cluster. One Dask DataFrame operation triggers many operations on the constituent pandas DataFrames.



Examples

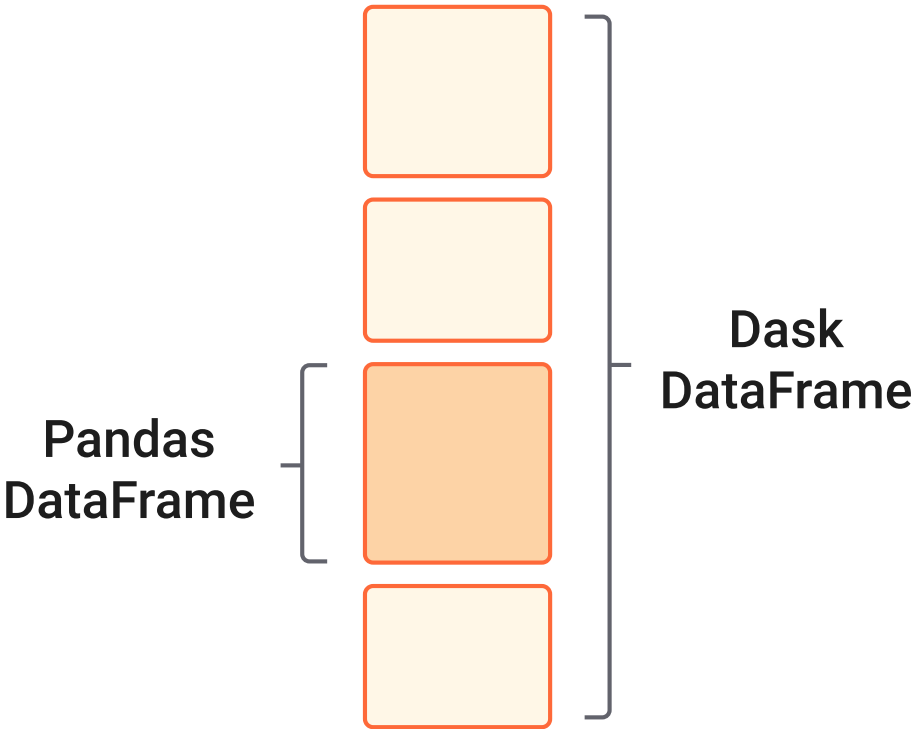
Visit <https://examples.dask.org/dataframe.html> to see and run examples using Dask DataFrame.


Design

Dask DataFrames coordinate many pandas DataFrames/Series arranged along the index. A Dask DataFrame is partitioned *row-wise*, grouping rows by index value for efficiency. These pandas objects may live on disk or on other machines.

Dask DataFrame copies the pandas DataFrame API

Because the `dask.DataFrame` application programming interface (API) is a subset of the `pd.DataFrame` API, it should be familiar to pandas users. There are some slight alterations due to the parallel nature of Dask:





Dask

Distributed

Dask ML

Examples

Ecosystem

Community

```
>>> import dask.dataframe as dd
>>> df = dd.read_csv('2014-*.csv')
>>> df.head()
   x  y
0  1  a
1  2  b
2  3  c
3  4  a
4  5  b
5  6  c

>>> df2 = df[df.y == 'a'].x + 1
>>> df2.compute()
0    2
3    5
Name: x, dtype: int64
```

```
>>> import pandas as pd
>>> df = pd.read_csv('2014-1.csv')
>>> df.head()
   x  y
0  1  a
1  2  b
2  3  c
3  4  a
4  5  b
5  6  c

>>> df2 = df[df.y == 'a'].x + 1
>>> df2
0    2
3    5
Name: x, dtype: int64
```

As with all Dask collections, you trigger computation by calling the `.compute()` method.

Common Uses and Anti-Uses

Dask DataFrame is used in situations where pandas is commonly needed, usually when pandas fails due to data size or computation speed:

- Manipulating large datasets, even when those datasets don't fit in memory
- Accelerating long computations by using many cores
- Distributed computing on large datasets with standard pandas operations like groupby, join, and time series computations

Dask DataFrame may not be the best choice in the following situations:

- If your dataset fits comfortably into RAM on your laptop, then you may be better off just using pandas. There may be simpler ways to improve performance than through parallelism
- If your dataset doesn't fit neatly into the pandas tabular model, then you might find more use in [dask.bag](#) or [dask.array](#)
- If you need functions that are not implemented in Dask DataFrame, then you might want to look at [dask.delayed](#) which offers more flexibility
- If you need a proper database with all that databases offer you might prefer something like [Postgres](#)

Scope

Dask DataFrame covers a well-used portion of the pandas API. The following class of computations works well:

- **Trivially parallelizable operations (fast):**
 - Element-wise operations: `df.x + df.y`, `df * df`
 - Row-wise selections: `df[df.x > 0]`
 - Loc: `df.loc[4.0:10.5]`
 - Common aggregations: `df.x.max()`, `df.max()`
 - Is in: `df[df.x.isin([1, 2, 3])]`
 - Date time/string accessors: `df.timestamp.month`
- **Cleverly parallelizable operations (fast):**
 - groupby-aggregate (with common aggregations): `df.groupby(df.x).y.max()`, `df.groupby('x').min()` (see [Aggregate](#))
 - groupby-apply on index: `df.groupby(['idx', 'x']).apply(myfunc)`, where `idx` is the index level name
 - value_counts: `df.x.value_counts()`
 - Drop duplicates: `df.x.drop_duplicates()`
 - Join on index: `dd.merge(df1, df2, left_index=True, right_index=True)` or `dd.merge(df1, df2, on=['idx', 'x'])` where `idx` is the index name for both `df1` and `df2`
 - Join with pandas DataFrames: `dd.merge(df1, df2, on='id')`

 v: stable ▾



- Rolling averages: `df.rolling(...)`
- Pearson's correlation: `df[['col1', 'col2']].corr()`
- **Operations requiring a shuffle (slow-ish, unless on index, see [Shuffling for GroupBy and Join](#))**
 - Set index: `df.set_index(df.x)`
 - groupby-apply not on index (with anything): `df.groupby(df.x).apply(myfunc)`
 - Join not on the index: `dd.merge(df1, df2, on='name')`

However, Dask DataFrame does not implement the entire pandas interface. Users expecting this will be disappointed. Notably, Dask DataFrame has the following limitations:

1. Setting a new index from an unsorted column is expensive
2. Many operations like groupby-apply and join on unsorted columns require setting the index, which as mentioned above, is expensive
3. The pandas API is very large. Dask DataFrame does not attempt to implement many pandas features or any of the more exotic data structures like NDFrames
4. Operations that were slow on pandas, like iterating through row-by-row, remain slow on Dask DataFrame

See the [DataFrame API documentation](#) for a more extensive list.

Execution

By default, Dask DataFrame uses the [multi-threaded scheduler](#). This exposes some parallelism when pandas or the underlying NumPy operations release the global interpreter lock (GIL). Generally, pandas is more GIL bound than NumPy, so multi-core speedups are not as pronounced for Dask DataFrame as they are for Dask Array. This is particularly true for string-heavy Python DataFrames, as Python strings are GIL bound.

There has been recent work on changing the underlying representation of pandas string data types to be backed by [PyArrow Buffers](#), which should release the GIL, however, this work is still considered experimental.

When dealing with text data, you may see speedups by switching to the [distributed scheduler](#) either on a cluster or single machine.

© Copyright 2014-2018, Anaconda, Inc. and contributors.