Lec 21 - Apache Arrow

Statistical Computing and Computation

Sta 663 | Spring 2022

Dr. Colin Rundel

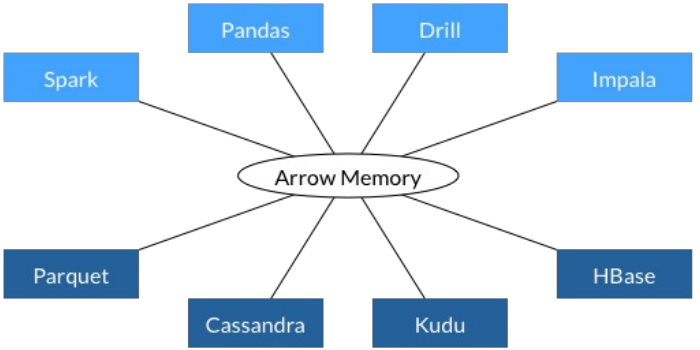
Apache Arrow

> Apache Arrow is a software development platform for building high performance applications that process and transport large data sets. It is designed to both improve the performance of analytical algorithms and the efficiency of moving data from one system or programming language to another.
>
> A critical component of Apache Arrow is its in-memory columnar format, a standardized, language-agnostic specification for representing structured, table-like datasets in-memory. This data format has a rich data type system (included nested and user-defined data types) designed to support the needs of analytic database systems, data frame libraries, and more.

Language support

Core implementations in:

- C
- C++
- C#
- go
- Java
- JavaScript
- Julia
- Rust
- MATLAB
- Python



pyarrow

```
import pyarrow as pa
```

The basic building blocks of Arrow are `array` and `table` objects, arrays are collections of data of a uniform type.

```
num  = pa.array([1, 2, 3, 2], type=pa.int8())
num
```

```
## <pyarrow.lib.Int8Array object at 0x2a2f7d880>
## [
##   1,
##   2,
##   3,
##   2
## ]
```

```
year = pa.array([2019,2020,2021,2022])
year
```

```
## <pyarrow.lib.Int64Array object at 0x2a2f7da60>
## [
##   2019,
```

```
name = pa.array(
  ["Alice", "Bob", "Carol", "Dave"],
  type=pa.string()
)
name
```

```
## <pyarrow.lib.StringArray object at 0x2a2f7dac0>
## [
##   "Alice",
##   "Bob",
##   "Carol",
##   "Dave"
## ]
```

Tables

A table is created by combining multiple arrays together to form the columns while also attaching names for each column.

```
t = pa.table(
  [num, year, name],
  names = ["num", "year", "name"]
)

t
```

```
## pyarrow.Table
## num: int8
## year: int64
## name: string
## ----
## num: [[1,2,3,2]]
## year: [[2019,2020,2021,2022]]
## name: [["Alice","Bob","Carol","Dave"]]
```

`table` is part of pyarrow but not part of the arrow standard, more on this in a bit

Array indexing

Elements of an array can be selected using `[]` with an integer index or a slice, the former returns a typed scalar the latter an array.

```
name[0]
```

```
## <pyarrow.StringScalar: 'Alice'>
```

```
name[0:3]
```

```
## <pyarrow.lib.StringArray object at 0x2a2f7dbe0>
## [
##   "Alice",
##   "Bob",
##   "Carol"
## ]
```

```
name[:]
```

```
## <pyarrow.lib.StringArray object at 0x2a2f7db80>
## [
##   "Alice",
##   "Bob",
## ...
```

```
name[-1]
```

```
## <pyarrow.StringScalar: 'Dave'>
```

```
name[::-1]
```

```
## <pyarrow.lib.StringArray object at 0x2a2f7dca0>
## [
##   "Dave",
##   "Carol",
##   "Bob",
##   "Alice"
## ]
```

```
name[4]
```

```
## IndexError: index out of bounds
```

```
name[0] = "Alan"
```

Arrow arrays are immutable and as such do allow for element assignment.

Data Types

The following types are language agnostic for the purpose of portability, however some differ slightly from what is available from Numpy and Pandas (or R),

- **Fixed-length primitive types**: numbers, booleans, date and times, fixed size binary, decimals, and other values that fit into a given number
  - Examples: `bool_()`, `uint64()`, `timestamp()`, `date64()`, and many more
- **Variable-length primitive types**: binary, string
- **Nested types**: list, map, struct, and union
- **Dictionary type**: An encoded categorical type

See here for the full list of types.

Schemas

A data structure that contains information on the names and types of columns for a table (or record batch),

```
t.schema
```

```
## num: int8
## year: int64
## name: string
```

```
pa.schema([
  ('num', num.type),
  ('year', year.type),
  ('name', name.type)
])
```

```
## num: int8
## year: int64
## name: string
```

Schema metadata

Schemas can also store additional metadata (e.g. codebook like textual descriptions) in the form a string:string dictionary,

```
new_schema = t.schema.with_metadata({
  'num': "Favorite number",
  'year': "Year expected to graduate",
  'name': "First name"
})
new_schema
```

```
## num: int8
## year: int64
## name: string
## -- schema metadata --
## num: 'Favorite number'
## year: 'Year expected to graduate'
## name: 'First name'
```

```
t.cast(new_schema).schema
```

```
## num: int8
## year: int64
## name: string
```

Missing values / None / NANs

```
pa.array([1,2,None,3])
```

```
## <pyarrow.lib.Int64Array object at 0x2a2f7dca0>
## [
##   1,
##   2,
##   null,
##   3
## ]
```

```
pa.array(["alice","bob",None,"dave"])
```

```
## <pyarrow.lib.StringArray object at 0x2a2f7dca0>
## [
##   "alice",
##   "bob",
##   null,
##   "dave"
## ]
```

```
pa.array([1.,2.,None,3.])
```

```
## <pyarrow.lib.DoubleArray object at 0x2a2f7db80>
## [
##   1,
##   2,
##   null,
##   3
## ]
```

```
pa.array([1,2,np.nan,3])
```

```
## <pyarrow.lib.DoubleArray object at 0x2a2f7db20>
## [
##   1,
##   2,
##   nan,
##   3
## ]
```

```
pa.array([1,2,None,3])[2]
```

```
pa.array(["alice","bob",None,"dave"])[2]
```

Nest type arrays

list type:

```
pa.array([[1,2], [3,4], None, [5,None]])
```

```
## <pyarrow.lib.ListArray object at 0x2a2f7db80>
## [
##   [
##     1,
##     2
##   ],
##   [
##     3,
##     4
##   ],
##   null,
##   [
##     5,
##     null
##   ]
## ]
```

See also map and union arrays.

struct type:

```
pa.array([
  {'x': 1, 'y': True, 'z': "Alice"},
  {'x': 2,            'z': "Bob"  },
  {'x': 3, 'y': False             }
])
```

```
## <pyarrow.lib.StructArray object at 0x2a2f7dca0>
## -- is_valid: all not null
## -- child 0 type: int64
##   [
##     1,
##     2,
##     3
##   ]
## -- child 1 type: bool
##   [
##     true,
##     null,
##     false
##   ]
## -- child 2 type: string
```

Dictionary array

A dictionary array is the equivalent to a factor in R or pd.Categorical in Pandas,

```
levels = pa.array(['sun', 'rain', 'clouds', 'snow'])
values = pa.array([0,0,2,1,3,None])
dict_array = pa.DictionaryArray.from_arrays(values, levels)
dict_array
```

```
## <pyarrow.lib.DictionaryArray object at 0x2a2f25970>
##
## -- dictionary:
##   [
##     "sun",
##     "rain",
##     "clouds",
##     "snow"
##   ]
## -- indices:
##   [
##     0,
##     0,
##     2,
##     1,
##     3,
##     null
##   ]
```

```
dict_array.type
```

```
## DictionaryType(dictionary<values=string, indices=int64, order
```

```
dict_array.dictionary_decode()
```

```
## <pyarrow.lib.StringArray object at 0x2a2f7ddc0>
## [
##   "sun",
##   "sun",
##   "clouds",
##   "rain",
##   "snow",
##   null
## ]
```

```
levels.dictionary_encode()
```

```
## <pyarrow.lib.DictionaryArray object at 0x2a2f257b0>
##
## -- dictionary:
##   [
##     "sun",
##     "rain",
##     "clouds",
##     "snow"
##   ]
## -- indices:
##   [
##     0,
##     1,
```

Record Batches

In between a table and an array Arrow has the concept of a Record Batch - which represents a chunk of the larger table. They are composed of a named collection of equal-length arrays.

```
batch = pa.RecordBatch.from_arrays(
  [num, year, name],
  ["num", "year", "name"]
)

batch
```

```
## pyarrow.RecordBatch
## num: int8
## year: int64
## name: string
```

```
batch.num_columns
```

```
## 3
```

```
batch.num_rows
```

```
## 4
```

```
batch.nbytes
```

```
## 69
```

```
batch.schema
```

```
## num: int8
```

Batch indexing

`[]` can be used with a Record Batch to select columns (by name or index) or rows (by slice), additionally the `slice()` method can be used to select rows.

```
batch[0]
```

```
## <pyarrow.lib.Int8Array object at 0x2a2f7de80>
## [
##   1,
##   2,
##   3,
##   2
## ]
```

```
batch["name"]
```

```
## <pyarrow.lib.StringArray object at 0x2a2f7de20>
## [
##   "Alice",
##   "Bob",
##   "Carol",
##   "Dave"
## ]
```

```
batch[0:2].to_pandas()
```

```
##    num  year   name
## 0    1  2019  Alice
## 1    2  2020    Bob
```

```
batch.slice(0,2).to_pandas()
```

```
##    num  year   name
## 0    1  2019  Alice
## 1    2  2020    Bob
```

Tables vs Record Batches

As mentioned previously, `table` objects are not part of the Arrow specification - rather they are a convenience tool provided to help with the wrangling of multiple Record Batches.

```
table = pa.Table.from_batches([batch] * 3)
table
```

```
## pyarrow.Table
## num: int8
## year: int64
## name: string
## ----
## num: [[1,2,3,2],[1,2,3,2],[1,2,3,2]]
## year: [[2019,2020,2021,2022],[2019,2020,2021,2022],[2019,2020,2021,2022]]
## name: [["Alice","Bob","Carol","Dave"],["Alice","Bob","Carol","Dave"],["Alice","Bob","Carol","Dave"]]
```

```
table.num_columns
```

```
## 3
```

```
table.num_rows
```

```
## 12
```

```
table.to_pandas()
```

```
##     num  year   name
## 0     1  2019  Alice
## 1     2  2020    Bob
## 2     3  2021  Carol
## 3     2  2022   Dave
```

Chunked Array

The columns of `table` are therefore composed of the columns of each of the batches, these are stored as ChuckedArrays instead of Arrays to reflect this.

```
table["name"]
```

```
## <pyarrow.lib.ChunkedArray object at 0x2a2fbcd60
## [
##   [
##     "Alice",
##     "Bob",
##     "Carol",
##     "Dave"
##   ],
##   [
##     "Alice",
##     "Bob",
##     "Carol",
##     "Dave"
##   ],
##   [
##     "Alice",
##     "Bob",
```

```
table[1]
```

```
## <pyarrow.lib.ChunkedArray object at 0x2a2fbce50
## [
##   [
##     2019,
##     2020,
##     2021,
##     2022
##   ],
##   [
##     2019,
##     2020,
##     2021,
##     2022
##   ],
##   [
##     2019,
##     2020,
```

Arrow + NumPy

Conversion between NumPy arrays and Arrow arrays is straight forward,

```
np.linspace(0,1,11)
```

```
## array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

```
pa.array( np.linspace(0,1,6) )
```

```
## <pyarrow.lib.DoubleArray object at 0x2a2fa8880>
## [
##   0,
##   0.2,
##   0.4,
##   0.6000000000000001,
##   0.8,
##   1
## ]
```

```
pa.array(range(10)).to_numpy()
```

```
## array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

NumPy & data copies

```
pa.array(["hello", "world"]).to_numpy()
```

```
## ArrowInvalid: Needed to copy 1 chunks with 0 nulls, but zero_copy_only was True
```

```
pa.array(["hello", "world"]).to_numpy(zero_copy_only=False)
```

```
## array(['hello', 'world'], dtype=object)
```

```
pa.array([1,2,None,4]).to_numpy()
```

```
## ArrowInvalid: Needed to copy 1 chunks with 1 nulls, but zero_copy_only was True
```

```
pa.array([1,2,None,4]).to_numpy(zero_copy_only=False)
```

```
## array([ 1.,  2., nan,  4.])
```

```
pa.array([[1,2], [3,4], [5,6]]).to_numpy()
```

Arrow + Pandas

We've already seen some basic conversion of Arrow table objects to Pandas, the conversions here are a bit more complex than with NumPy due in large part to how Pandas handles missing data.

Source (Pandas)	Destination (Arrow)
`bool`	`BOOL`
`(u)int{8,16,32,64}`	`(U)INT{8,16,32,64}`
`float32`	`FLOAT`
`float64`	`DOUBLE`
`str / unicode`	`STRING`
`pd.Categorical`	`DICTIONARY`
`pd.Timestamp`	`TIMESTAMP(unit=ns)`
`datetime.date`	`DATE`
`datetime.time`	`TIME64`

From Type differences documentation

Source (Arrow)	Destination (Pandas)
`BOOL`	`bool`
`BOOL` with nulls	`object` (with values `True`, `False`,
`(U)INT{8,16,32,64}`	`(u)int{8,16,32,64}`
`(U)INT{8,16,32,64}` with nulls	`float64`
`FLOAT`	`float32`
`DOUBLE`	`float64`
`STRING`	`str`
`DICTIONARY`	`pd.Categorical`
`TIMESTAMP(unit=*)`	`pd.Timestamp` (`np.datetime64[ns]`)
`DATE`	`object` (with `datetime.date` objec

Series & data copies

Due to these discrepancies it is much more likely that converting from Arrow array to a Panda series will require a type to be changed in which case the data will need to be copied. Like `to_numpy()` the `to_pandas()` method also accepts the `zero_copy_only` argument, however it defaults to `False`.

```
pa.array([1,2,3,4]).to_pandas()
```

```
## 0    1
## 1    2
## 2    3
## 3    4
## dtype: int64
```

```
pa.array(["hello", "world"]).to_pandas()
```

```
## 0    hello
## 1    world
## dtype: object
```

```
pa.array(["hello", "world"]).dictionary_encode().
```

```
pa.array([1,2,3,4]).to_pandas(zero_copy_only=True
```

```
## 0    1
## 1    2
## 2    3
## 3    4
## dtype: int64
```

```
pa.array(["hello", "world"]).to_pandas(zero_copy_
```

```
## ArrowInvalid: Needed to copy 1 chunks with 0 nu
```

```
pa.array(["hello", "world"]).dictionary_encode().
```

Note that Arrow arrays are converted to Series while tables & batches are converted to DataFrames

Zero Copy Series Conversions

> Zero copy conversions from `Array` or `ChunkedArray` to NumPy arrays or pandas Series are possible in certain narrow cases:
>
> - The Arrow data is stored in an integer (signed or unsigned `int8` through `int64`) or floating point type (`float16` through `float64`). This includes many numeric types as well as timestamps.
> - The Arrow data has no null values (since these are represented using bitmaps which are not supported by pandas).
> - For `ChunkedArray`, the data consists of a single chunk, i.e. `arr.num_chunks == 1`. Multiple chunks will always require a copy because of pandas's contiguousness requirement.
>
> In these scenarios, `to_pandas` or `to_numpy` will be zero copy. In all other scenarios, a copy will be required.

DataFrame & data copies

```
table.to_pandas()
```

```
##     num  year   name
## 0     1  2019  Alice
## 1     2  2020    Bob
## 2     3  2021  Carol
## 3     2  2022   Dave
## 4     1  2019  Alice
## 5     2  2020    Bob
## 6     3  2021  Carol
## 7     2  2022   Dave
## 8     1  2019  Alice
## 9     2  2020    Bob
## 10    3  2021  Carol
## 11    2  2022   Dave
```

```
table.to_pandas(zero_copy_only=True)
```

```
## ArrowInvalid: Cannot do zero copy conversion into multi-column DataFrame block
```

```
table.drop(['name']).to_pandas(zero_copy_only=True)
```

```
## ArrowInvalid: Cannot do zero copy conversion into multi-column DataFrame block
```

Source

Pandas -> Arrow

To convert from a Pandas DataFrame to an Arrow Table we can use the `from_pandas()` method (schemas can also be inferred from DataFrames)

```
df = pd.DataFrame({
  'x': np.random.normal(size=5),
  'y': ["A","A","B","C","C"],
  'z': [1,2,3,4,5]
})

pa.Table.from_pandas(df)
```

```
## pyarrow.Table
## x: double
## y: string
## z: int64
## ----
## x: [[0.7312616306514687,0.7492374224202655,0.016214128758116807,-0.11221364659471317,-1.068060983676372]]
## y: [["A","A","B","C","C"]]
## z: [[1,2,3,4,5]]
```

```
pa.Schema.from_pandas(df)
```

```
## x: double
```

The import of Pandas indexes is governed by the `preserve_index` argument

An aside on tabular file formats

Comma Separated Values

This and other text & delimiter based file formats are the most common and generally considered the most portable, however they have a number of significant draw backs

- no explicit schema or other metadata
- column types must be inferred from the data
- numerical values stored as text (efficiency and precision issues)
- limited compression options

(Apache) Parquet

> ... provides a standardized open-source columnar storage format for use in data analysis systems. It was created originally for use in Apache Hadoop with systems like Apache Drill, Apache Hive, Apache Impala, and Apache Spark adopting it as a shared standard for high performance data IO.

Core features:

> The values in each column are physically stored in contiguous memory locations and this columnar storage provides the following benefits:
>
> - Column-wise compression is efficient and saves storage space
> - Compression techniques specific to a type can be applied as the column values tend to be of the same type
> - Queries that fetch specific column values need not read the entire row data thus improving performance

Feather

> ... is a portable file format for storing Arrow tables or data frames (from languages like Python or R) that utilizes the Arrow IPC format internally. Feather was created early in the Arrow project as a proof of concept for fast, language-agnostic data frame storage for Python (pandas) and R.

Core features:

- Direct columnar serialization of Arrow tables
- Supports all Arrow data types and compression
- Language agnostic
- Metadata makes it possible to read only the necessary columns for an operation

Example - File Format Performance

Based on Apache Arrow: Read DataFrame With Zero Memory

Building a large dataset

```
np.random.seed(1234)

df = (
    pd.read_csv("https://sta663-sp22.github.io/slides/data/penguins.csv")
      .sample(1000000, replace=True)
      .reset_index(drop=True)
)

num_cols = ["bill_length_mm", "bill_depth_mm", "flipper_length_mm",    "body_mass_g"]
df[num_cols] = df[num_cols] + np.random.normal(size=(df.shape[0],len(num_cols)))

df
```

```
##           species     island  bill_length_mm  bill_depth_mm  flipper_length_mm  body_mass_g     sex  year
## 0       Chinstrap      Dream       48.519017      17.136138         200.642256  3800.527739    male  2008
## 1          Gentoo     Biscoe       49.819091      15.317831         223.507723  5550.240852    male  2008
## 2       Chinstrap      Dream       45.174882      17.558060         188.966751  3449.851303  female  2007
## 3          Adelie     Biscoe       43.043647      18.630037         201.280925  4048.906267    male  2008
## 4          Gentoo     Biscoe       44.926619      13.633214         209.714220  4400.958077  female  2008
## ...           ...        ...             ...            ...                ...          ...     ...   ...
## 999995  Chinstrap      Dream       48.692352      18.184983         200.579977  3799.702510    male  2009
## 999996     Adelie     Biscoe       43.046500      20.182105         197.283516  4776.263332    male  2009
## 999997     Adelie  Torgersen       43.192871      17.278273         196.408173  4251.476411    male  2008
## 999998     Gentoo     Biscoe       53.208097      16.119915         230.537089  5498.259372    male  2009
```

Create output files

```
import os
os.makedirs("scratch/", exist_ok=True)

df.to_csv("scratch/penguins-large.csv")
df.to_parquet("scratch/penguins-large.parquet")

import pyarrow.feather

pyarrow.feather.write_feather(
    pa.Table.from_pandas(df),
    "scratch/penguins-large.feather"
)

pyarrow.feather.write_feather(
    pa.Table.from_pandas(df.dropna()),
    "scratch/penguins-large_nona.feather"
)
```

File Sizes

```
def file_size(f):
    x = os.path.getsize(f)
    print(f, "\t\t", round(x / (1024 * 1024),2), "MB")
```

```
file_size( "scratch/penguins-large.csv" )
```

```
## scratch/penguins-large.csv          100.91 MB
```

```
file_size( "scratch/penguins-large.parquet" )
```

```
## scratch/penguins-large.parquet          32.44 MB
```

```
file_size( "scratch/penguins-large.feather" )
```

```
## scratch/penguins-large.feather          48.93 MB
```

```
file_size( "scratch/penguins-large_nona.feather" )
```

```
## scratch/penguins-large_nona.feather          50.93 MB
```

Read Performance

```
%timeit pd.read_csv("scratch/penguins-large.csv")
```

```
## 566 ms ± 11.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%timeit pd.read_parquet("scratch/penguins-large.parquet")
```

```
## 116 ms ± 3.43 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%timeit pyarrow.csv.read_csv("scratch/penguins-large.csv")
```

```
## 34.2 ms ± 99 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
%timeit pyarrow.parquet.read_table("scratch/penguins-large.parquet")
```

```
## 55.8 ms ± 104 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
%timeit pyarrow.feather.read_table("scratch/penguins-large.feather")
```

Read Performance (Arrow -> Pandas)

```
%timeit pyarrow.csv.read_csv("scratch/penguins-large.csv").to_pandas()
```

```
## 88.9 ms ± 1.45 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
%timeit pyarrow.parquet.read_table("scratch/penguins-large.parquet").to_pandas()
```

```
## 109 ms ± 153 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
%timeit pyarrow.feather.read_feather("scratch/penguins-large.feather")
```

```
## 61.5 ms ± 333 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
%timeit pyarrow.feather.read_feather("scratch/penguins-large_nona.feather")
```

```
## 60.6 ms ± 70.6 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Column subset calculations - CSV & Parquet

```
%timeit pd.read_csv("scratch/penguins-large.csv")["flipper_length_mm"].mean()
```

```
## 568 ms ± 9.43 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%timeit pd.read_parquet("scratch/penguins-large.parquet",  columns=["flipper_length_mm"]).mean()
```

```
## 9.74 ms ± 53.9 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
%timeit pyarrow.parquet.read_table("scratch/penguins-large.parquet", columns=["flipper_length_mm"]).to_pa
```

```
## 8.46 ms ± 183 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
%timeit pyarrow.parquet.read_table("scratch/penguins-large.parquet")["flipper_length_mm"].to_pandas().mea
```

```
## 60.7 ms ± 1.04 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Column subset calculations - Feather

```
%timeit pyarrow.feather.read_table("scratch/penguins-large.feather", columns=["flipper_length_mm"]).to_pa
```

```
## 8.1 ms ± 53.1 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
%timeit pyarrow.feather.read_table("scratch/penguins-large.feather")["flipper_length_mm"].to_pandas().mea
```

```
## 14.4 ms ± 403 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
%timeit pyarrow.feather.read_table("scratch/penguins-large_nona.feather", columns=["flipper_length_mm"]).
```

```
## 4.5 ms ± 76.9 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
%timeit pyarrow.feather.read_table("scratch/penguins-large_nona.feather")["flipper_length_mm"].to_pandas(
```

```
## 10 ms ± 21.9 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Demo 1 - Remote Files + Datasets