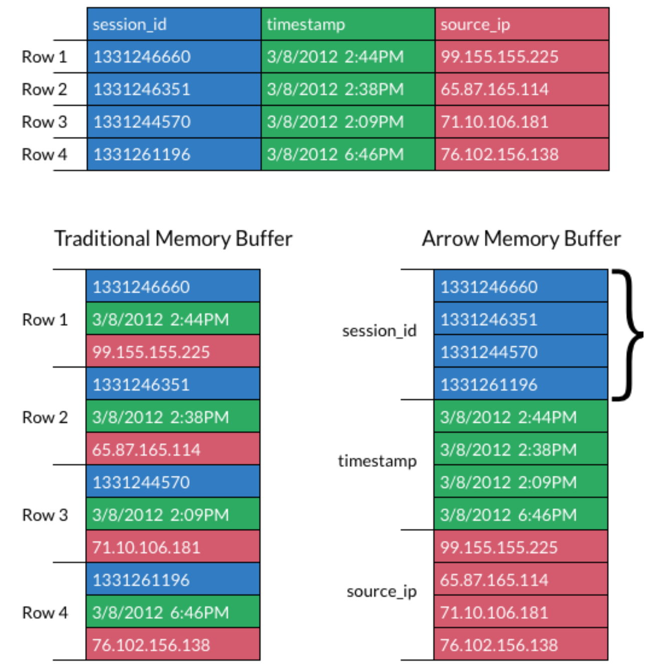
Apache Arrow

Lecture 21

Dr. Colin Rundel

Apache Arrow

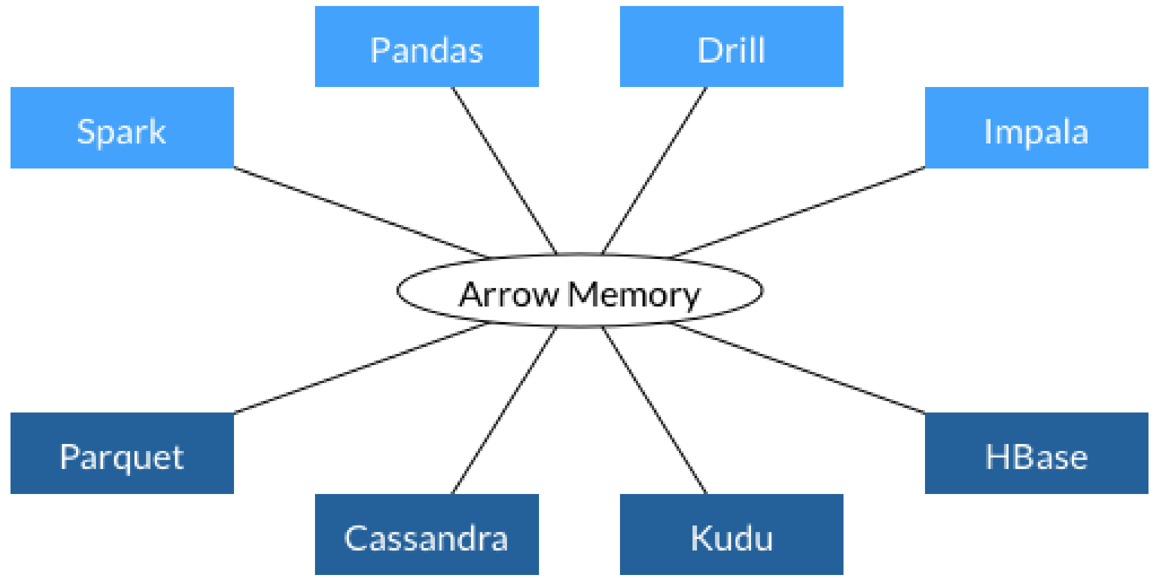
> Apache Arrow is a software development platform for building high performance applications that process and transport large data sets. It is designed to both improve the performance of analytical algorithms and the efficiency of moving data from one system or programming language to another.
> A critical 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
- R
- Ruby



pyarrow

```python
import pyarrow as pa
```

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

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

```
<pyarrow.lib.Int8Array object at 0x2b90c3ca0>
[
  1,
  2,
  3,
  2
]
```

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

```
<pyarrow.lib.Int64Array object at 0x2b90c3d00>
[
  2019,
  2020,
  2021,
  2022
]
```

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

```
<pyarrow.lib.StringArray object at 0x2b90c3dc0>
[
  "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 0x2b90c3e20>
[
  "Alice",
  "Bob",
  "Carol"
]
```

```
name[:]
```

```
<pyarrow.lib.StringArray object at 0x2b90c3d60>
[
  "Alice",
  "Bob",
  "Carol",
  "Dave"
]
```

```
name[-1]
```

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

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

```
<pyarrow.lib.StringArray object at 0x2b9175060>
[
  "Dave",
  "Carol",
  "Bob",
  "Alice"
]
```

```
name[4]
```

```
Error: IndexError: index out of bounds
```

```
name[0] = "Patty"
```

```
Error: TypeError: 'pyarrow.lib.StringArray' object dc
```


Arrow arrays are immutable and as such do not 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

are 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 of 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.schema
```

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

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

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

Missing values / None / NANs

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

```
<pyarrow.lib.Int64Array object at 0x2b90c3e80>
[
  1,
  2,
  null,
  3
]
```

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

```
<pyarrow.lib.DoubleArray object at 0x2b90c3d60>
[
  1,
  2,
  null,
  3
]
```

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

```
<pyarrow.Int64Scalar: None>
```

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

```
<pyarrow.DoubleScalar: None>
```

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

```
<pyarrow.lib.DoubleArray object at 0x2b90c3e80>
[
  1,
  2,
  nan,
  3
]
```

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

```
<pyarrow.lib.StringArray object at 0x2b9175060>
[
  "alice",
  "bob",
  null,
  "dave"
]
```

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

```
<pyarrow.DoubleScalar: nan>
```

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

```
<pyarrow.StringScalar: None>
```

Nest type arrays

list type:

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

```
<pyarrow.lib.ListArray object at 0x2b90c3e80>
[
  [
    1,
    2
  ],
  [
    3,
    4
  ],
  null,
  [
    5,
    null
  ]
]
```

struct type:

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

```
<pyarrow.lib.StructArray object at 0x2b90c3d60>
-- is_valid: all not null
-- child 0 type: int64
  [
    1,
    2,
    3
  ]
-- child 1 type: bool
  [
    true,
    null,
    false
  ]
-- child 2 type: string
  [
    "Alice",
    "Bob",
    null
  ]
```

Dictionary array

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

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

```
<pyarrow.lib.DictionaryArray object at 0x2b907bdf0>

-- dictionary:
  [
    "sun",
    "rain",
    "clouds",
    "snow"
  ]
-- indices:
  [
    0,
    0,
    2,
    1,
    3,
    null
  ]
```

```python
dict_array.type
```

```
DictionaryType(dictionary<values=string, indices=int64, ordered=0>)
```

```python
dict_array.dictionary_decode()
```

```
<pyarrow.lib.StringArray object at 0x2b91751e0>
[
  "sun",
  "sun",
  "clouds",
  "rain",
  "snow",
  null
]
```

```python
pa.array(['sun', 'rain', 'clouds', 'sun']).dicti
```

```
<pyarrow.lib.DictionaryArray object at 0x2b907be60>

-- dictionary:
  [
    "sun",
    "rain",
    "clouds"
  ]
-- indices:
  [
    0,
    1,
    2,
    0
  ]
```

Record Batches

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

```
batch = pa.RecordBatch.from_arrays(
  arrays = [num, year, name],
  names = ["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
year: int64
name: string
```

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.

```python
batch[0]
```

```
<pyarrow.lib.Int8Array object at 0x2b9175240>
[
  1,
  2,
  3,
  2
]
```

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

```
<pyarrow.lib.StringArray object at 0x2b91752a0>
[
  "Alice",
  "Bob",
  "Carol",
  "Dave"
]
```

```python
batch[1::2].to_pandas()
```

	num	year	name
0	2	2020	Bob
1	2	2022	Dave

```python
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"]]
```

```python
table.num_columns
```

```
3
```

```python
table.num_rows
```

```
12
```

```python
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

Chunked Array

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

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 0x2b9175ae0>
[
  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()
```

```
Error: pyarrow.lib.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()
```

```
Error: pyarrow.lib.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()
```

```
Error: pyarrow.lib.ArrowInvalid: Needed to copy 1 chunks with 0 nulls, but zero_copy_only was True
```

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

```
array([array([1, 2]), array([3, 4]), array([5, 6])], dtype=object)
```

Pandas -> Arrow

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

Arrow -> Pandas

Source (Arrow)	Destination (Pandas)
`BOOL`	`bool`
`BOOL` with nulls	`object` (with values `True`, `False`, `None`)
`(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` objects)
`TIME64`	`object` (with `datetime.time` objects)

From Type differences documentation

Series & data copies

Due to these discrepancies it is much more likely that converting from an 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()`, `to_pandas()` also accepts the `zero_copy_only` argument, however its default is `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()
```

```
0    hello
1    world
dtype: category
Categories (2, object): ['hello', 'world']
```

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

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

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

```
Error: pyarrow.lib.ArrowInvalid: Needed to copy 1 chu
```

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

```
Error: pyarrow.lib.ArrowInvalid: Needed to copy 1 chu
```

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.schema
```

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

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

```
Error: pyarrow.lib.ArrowInvalid: Cannot do zero copy conversion into m
```

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

```
Error: pyarrow.lib.ArrowInvalid: Cannot do zero copy conversion into m
```

```
pa.table(
  [num,year], names=["num","year"]
).to_pandas(zero_copy_only=True)
```

```
Error: pyarrow.lib.ArrowInvalid: Cannot do zero copy conversion into m
```


Source

Pandas DF -> 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.round(np.random.normal(size=5),3),
  '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: [[1.111,-0.633,0.033,4.534,-0.104]]
y: [["A","A","B","C","C"]]
z: [[1,2,3,4,5]]
```

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

```
x: double
y: string
z: int64
-- schema metadata --
pandas: '{"index_columns": [{"kind": "range", "name":
```


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
- Efficient column-wise compression saves storage space
- Compression techniques specific to a type can be applied
- Queries that fetch specific column values do not read the entire row
- Different encoding techniques can be applied to different columns

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

```python
np.random.seed(1234)

df = (
  pd.read_csv("https://sta663-sp22.github.io/slides/data/penguins.csv")
    .sample(10_000_000, 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       50.261096      18.764164         200.607259  3800.208171    male  2008
1           Gentoo     Biscoe       49.594921      13.085831         223.316267  5551.827810    male  2008
2        Chinstrap      Dream       45.919136      18.488115         190.024941  3450.928250  female  2007
3           Adelie     Biscoe       41.829861      21.047924         200.945039  4050.412137    male  2008
4           Gentoo     Biscoe       44.846957      14.669941         211.926404  4400.888107  female  2008
...            ...        ...             ...            ...                ...          ...     ...   ...
9999995     Gentoo     Biscoe       49.994321      14.545704         221.270307  5448.948790    male  2009
9999996  Chinstrap      Dream       51.568717      18.883271         196.283438  3749.222035    male  2007
9999997     Gentoo     Biscoe       38.937558      13.634993         212.118822  4650.113320  female  2007
9999998     Adelie  Torgersen       35.785542      17.751472         185.578798  3150.013148  female  2009
9999999     Adelie     Biscoe       38.170442      20.282051         182.632278  3600.463322    male  2007

[10000000 rows x 8 columns]
```

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 		 1018.68 MB
```

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

```
## scratch/penguins-large.parquet 		 314.19 MB
```

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

```
## scratch/penguins-large.feather 		 489.14 MB
```

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

```
## scratch/penguins-large_nona.feather 		 509.24 MB
```

Read Performance

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

```
## 5.2 s ± 50.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

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

```
## 713 ms ± 11.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

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

```
## 359 ms ± 61.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

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

```
## 213 ms ± 2.83 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

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

```
90.9 ms ± 528 μs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

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

```
94.5 ms ± 192 μs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Read Performance (Arrow -> Pandas)

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

```
## 921 ms ± 75 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

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

```
## 727 ms ± 41.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

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

```
## 542 ms ± 6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

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

```
## 547 ms ± 16.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Column subset calculations - CSV & Parquet

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

```
## 5.21 s ± 82.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%timeit pd.read_parquet("scratch/penguins-large.parquet",  columns=["flipper_
```

```
## 80.8 ms ± 619 μs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
%timeit pyarrow.parquet.read_table("scratch/penguins-large.parquet", columns=
```

```
## 85.8 ms ± 599 μs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

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

```
## 262 ms ± 9.97 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Polars

What is Polars?

> Polars is a lightning fast DataFrame library/in-memory query engine. Its embarrassingly parallel execution, cache efficient algorithms and expressive API makes it perfect for efficient data wrangling, data pipelines, snappy APIs and so much more.
> The goal of Polars is to provide a lightning fast DataFrame library that:
>
> - Utilizes all available cores on your machine.
> - Optimizes queries to reduce unneeded work/memory allocations.
> - Handles datasets much larger than your available RAM.
> - Has an API that is consistent and predictable.
> - Has a strict schema (data-types should be known before running the query).
>
> Polars is written in Rust which gives it C/C++ performance and allows it to fully control performance critical parts in a query engine.

Polars vs Pandas

- Polars does not have a multi-index/index
- Polars uses Apache Arrow arrays to represent data in memory while Pandas uses Numpy arrays
- Polars has more support for parallel operations than Pandas
- Polars can lazily evaluate queries and apply query optimization
- Polars syntax is similar but distinct from Pandas

Demo 1 - NYC Taxis