Contents

1 Pros & Cons

- **Pandas**: eager execution, tried & true (ideal for exploration
- **Polars**: lazy execution, optimizes speed
- **PySpark**: lazy execution, great for scale as operations can be run on multiple computers

1.1 Eager vs. Lazy (Pandas vs. Polars/PySpark)

Similar to the execution vs. graphing with PyTorch vs. Tensorflow; instead of executing each step on command, Lazy execution via Polars/PySpark will graph the commands and execute on collection. The primary difference is that lazy execution can take advantage of multiple cores (see example).

By default, Polars still executes eager, but one can use `.lazy()` in the command chain to begin a graph and `.collect()` to trigger execution.

Pandas still has better functionality for data exploration, whereas Polars is preferable for building data pipelines – especially when there are noticeable computation chokepoints. PySpark creates a background instance, where jobs can be scaled across multiple computers. As such, it doesn't suffer the constraint that Pandas/Polars do where read data must be <RAM size, since the entire dataset is loaded into memory.

```
import polars as pl

df = pl.read_csv('foo.csv')

(df.lazy() # begins graph
    .filter(pl.col('col1') > 2) # equiv. to query
    .groupby('col2') # Group cols
    .agg(sum, mean) # ea. of the cols' aggs are computed by a diff core
    .collect() # executes graph
)
```

1.2 PySpark

PySpark requires for an instance to be invoked and created. This instance runs in the background, and its benefit is that it can be accessed by multiple computers. Being able to scale up to multiple machines is the reason why PySpark is superior in operating on large datasets.

```
    from pyspark.sql import SparkSession

    spark =
    ↪  SparkSession.builder.master('local[1]').appName('example_name').getOrCreate()
```

Thereafter, you can look at running jobs on `localhost:4040/jobs`.

2 Methods

This is a quick-lookup sheet for like operations in Pandas/Polars/PySpark.

2.1 I/O examples

2.1.1 Pandas

`df = pd.read_csv()` and `df.to_csv()`

2.1.2 Polars

`pl.scan_csv()` will implicitly graph subsequent actions such that the entire file need not be loaded into memory for the action to occur. For example, if there are multiple columns (col1, col2, ...) and we perform next a groupby *by col1 only*, only the col1 values are loaded into memory.

```
# Operations after scan_ are implicitly lazy
df = pl.scan_parquet('example.parquet').groupby('col1').agg(
    pl.col('col2').sum().alias('col2_sum'), # pl.col('') to access cols
    pl.col('col1').mean().alias('col1_mean') # alias to name col
).collect() # Collect to execute graph

# df.write_parquet('example.parquet')
```

2.1.3 PySpark

```
    from pyspark.sql import SparkSession
    # Spark requires custom functions to be imported
    from pyspark.sql.functions import avg, max, sum

    # Begin session
    spark =
    ↪  SparkSession.builder.master('local[1]').appName('example_name').getOrCreate()

    df_spark = spark.read.parquet('example.parquet')
    df_spark_agg = df_spark.groupby(['col1', 'col2']).agg(
        sum('col2').alias('col2_sum'), # note: sum <- pyspark fcns
        avg('col1').alias('col1_avg') # alias to name col
    )

    # Executes lazily, so nothing is done until we reach writing
    df_spark_agg.write.mode('overwrite').parquet('example.parquet')
    # Once we execute this line, we can watch the job on localhost:4040/jobs
```

PySpark also takes sql queries as string, instead of scripting invocations.

```
from pyspark.sql import SparkSession
spark =
↪  SparkSession.builder.master('local[1]').appName('example_name').getOrCreate()

# Create a temporary table which loads the file
spark.sql('CREATE TEMPORARY VIEW example_table USING parquet OPTIONS (path
↪  \"example.parquet\")')

query = """
SELECT col1,
    sum(col2) AS col2_sum,
    avg(col1) AS col1_avg
FROM example_table
GROUP BY col1
"""

spark.sql(query).write.mode('overwrite').parquet('temp_spark_sql.parquet')
```