Lecture 2: Working with Big Data locally

Contents

2.1. Announcements

- Unix Tutorial will be scheduled next week.

2.2. Learning objectives

- Identifying properties of big data file formats.
- Gaining detailed knowledge of the `parquet file format` and its uses in analysis.
- Explaining the concepts of `serialization` and `deserialization`.
- Understanding `Apache arrow` project and showcasing why it's a game-changer in big data processing.
- Learning how to take advantage of the parquet file format & arrow in big data file processing.
- Recent arrow integrations and other alternatives. (optional)
- Introducing the `feather` file format. (optional)

2.3. Refresher Questions

- What Big Data dimensions are applicable to the recent data that you worked with?
- How do you think data appears on the website ?
- What techniques you used to deal with big data locally? What other techniques are you aware of ?
- How can you deal with processing a 20 GB CSV file in pandas?
- How big is big data?

- What is the biggest file size you deal with? And how did you manage to wrangle it?
- Do you think where your data is stored (SSD vs. HDD) matters when you are processing data?
- Provide me a situation where you going to use mongoDB over a relational database. (Well they can also give you a scenario and ask for your opinion)
- Do you think the materialized view is faster than the regular view?
- When do you think we should use the regular view over materialized view?
- When do you think we should use the materialized view over the regular view?
- When should we use a temporary table over a view?
- Can you do indexing on a view?
- When should I use a hash index over a btree index?

Click toggle below to see the answers.

- What Big Data dimensions are applicable to the recent data that you worked with?
  - See if you can categorize into Volume, Velocity, Variety, Veracity
- How do you think data appears on the website ?
  - In our last class, we saw how it appears. Use of API calls to the server where it stores the data. Look into the client-server model that we talked about in the last class for more details.
- What techniques you used to deal with big data locally? What other techniques are you aware of ?
  - We used pandas to bring in just what we wanted, casting and chunking. We will see later in this class bunch of packages that can help us deal with big data locally.
- How can you deal with processing a 20 GB CSV file in pandas?
  - It might be difficult. I will try casting, chunking, and column selection to bring in what we want in an efficient way. But the pacakges we are going to discuss today will make it easier.
- How big is big data?
  - It is a relative term. It depends on the context. For example, if you are working with a 2 GB CSV file, it is big data for a 4 GB RAM machine. But if you are working with a 10 GB CSV file, it might be big for a 16 GB RAM machine.
- What is the biggest file size you deal with? And how did you manage to wrangle it?
  - I have worked with > TB CSV files. I used hadoop, spark and pig for it.
- Do you think where your data is stored (SSD vs. HDD) matters when you are processing data?
  - Yes, it does. SSD is faster than HDD.
- Provide me a situation where you going to use mongoDB over a relational database. (Well they can also give you a scenario and ask for your opinion)
  - One classic case is when we can't define a schema for our data. Also, if you are working with a lot of semistructured data, you can use MongoDB over a relational database.
- Do you think the materialized view is faster than the regular view?
  - Yes, it can be faster with indexes.
- When do you think we should use the regular view over materialized view?
  - When the underlying data is frequently changing, then you can use the regular view.
- When do you think we should use the materialized view over the regular view?
  - When the underlying data is not changing frequently, and you want to apply some indexing on views, then you can use the materialized view.
- Can you do indexing on a view?
  - Yes, you can do indexing on a materalized view.

- When should I use a hash index over a btree index?
  - When the query is equality-based, then you can use a hash index. Otherwise, you can use a btree index. There are some other factors that also need to be considered depending on the column values.

Here is the plan for today...

We will first be talking following concepts

- Big Data file formats
- Parquet
  - Data Structure
  - Converting CSV –> parquet
- Serialization and deserialization
- Apache Arrow

After learning all of the above concepts, we will revisit Working with Big Data locally from the last class to apply our knowledge.

2.4. Big Data file formats.

Let's explore key properties of file formats for big data:

2.4.1. Human-readable vs. machine-readable:

CSV, JSON, and XML are human-readable, while binary files are optimized for machines, taking less space and offering better performance.

2.4.2. Schema:

Unlike databases, CSV and JSON files don't have a defined schema, which can be defined when reading the file in Python or R.

2.4.3. Partitioning:

Partitioning, which is useful in databases, is also cool when applied to files for analysis purposes.

2.4.4. Row and column storage:

Files can be stored as rows or columns, with relational databases typically storing them as rows, and databases like Apache Cassandra as columns. Understanding the difference is important for OLTP and OLAP, and considering a hybrid format is worth exploring.

2.4.5. Splittable:

Being able to split a file into smaller pieces is relevant for accessing specific data or parallel processing.

2.4.6. Compression:

Compressing a file reduces its size, saving storage space and speeding up transfers, with various techniques available.

> **Thoughts/Discussion ?**
>
> ***Question 1:*** You know about some general properties of file types. How are these properties for a CSV file?
>
> - Readable:
> - Schema:
> - Partitioning:
> - Column/Row-based storage:
> - Splittable:
> - Compression:
>
> ***Question 2:*** When considering big data, there are several factors to take into account. Complete the following sentences by filling out what follows after 'because...'
>
> - You don't care about the human-readable file because...
> - It will be good to preserve the schema (self-describing files) because...
> - Good to have Partitioning because...
> - If you are performing OLAP queries, columnar storage is good. If OLTP, then row-based because...
> - It should be splittable because...
> - Good to be compressed because...

Here are three main big data file formats.

- Parquet
- Avro
- ORC

Typically we use the above file formats to store huge amounts of data. When to use what ? `it depends`

Here is a table that compares these three file formats with the file properties that we discussed.

Property	Parquet	Avro	ORC
Human-readable/binary	binary	binary	binary
Self-describing	yes	yes	yes
Schema-evolution	A-	A+	A
Partitioning	yes	yes	yes
Columnar/row based	columnar	row	columnar
Splittable	yes	yes	yes
Compression	A	A-	A+

So in your situation, what kind of file do you think will be appropriate to handle big data?

Now let's get into the details of the parquet.

2.5. Parquet 😀

`Parquet` was initially designed for Hadoop by Twitter and Cloudera, it is now considered a `de-facto standard` for storing big data files. Here I am reiterating some of the advantages and properties of parquet files.

- `Projection pushdown`

The projection *(usually it means the select clause/ the columns you are interested in)* will be pushed down to where the data is stored to bring just those columns to the memory.

- `Predicate pushdown`

The predicates *(usually it means the where clause/ the rows you are interested in)* will be pushed down to where the data is stored to bring the minimum number of rows needed for the computation.

- Partitioning (embedding predicates to directory structure)

By partitioning a parquet file, you can store and process it more efficiently. So when we use filters on the columns that we partition on, `partition-pruning` won't touch the unnecessary partitions.

- `Schema evolution`

If we need to change the schema at some point, it's possible. For example, it is possible if you want to add some additional columns to an existing parquet file.

- Free and `open-source` file format`
- `Language agnostic`

Parquet can be used with any programming language.

- `Hybrid file-format`

Parquet is structured in a hybrid way to combine the best of both worlds in terms of columnar and row-based file storage. Although it is commonly referred to as a columnar file format, the hybrid structure of Parquet allows it to also incorporate row-based elements.

- `Splittable` files.
- Highly efficient `data compression` and decompression.

It is made possible by various ***encoding techniques*** and ***compression schemes.***

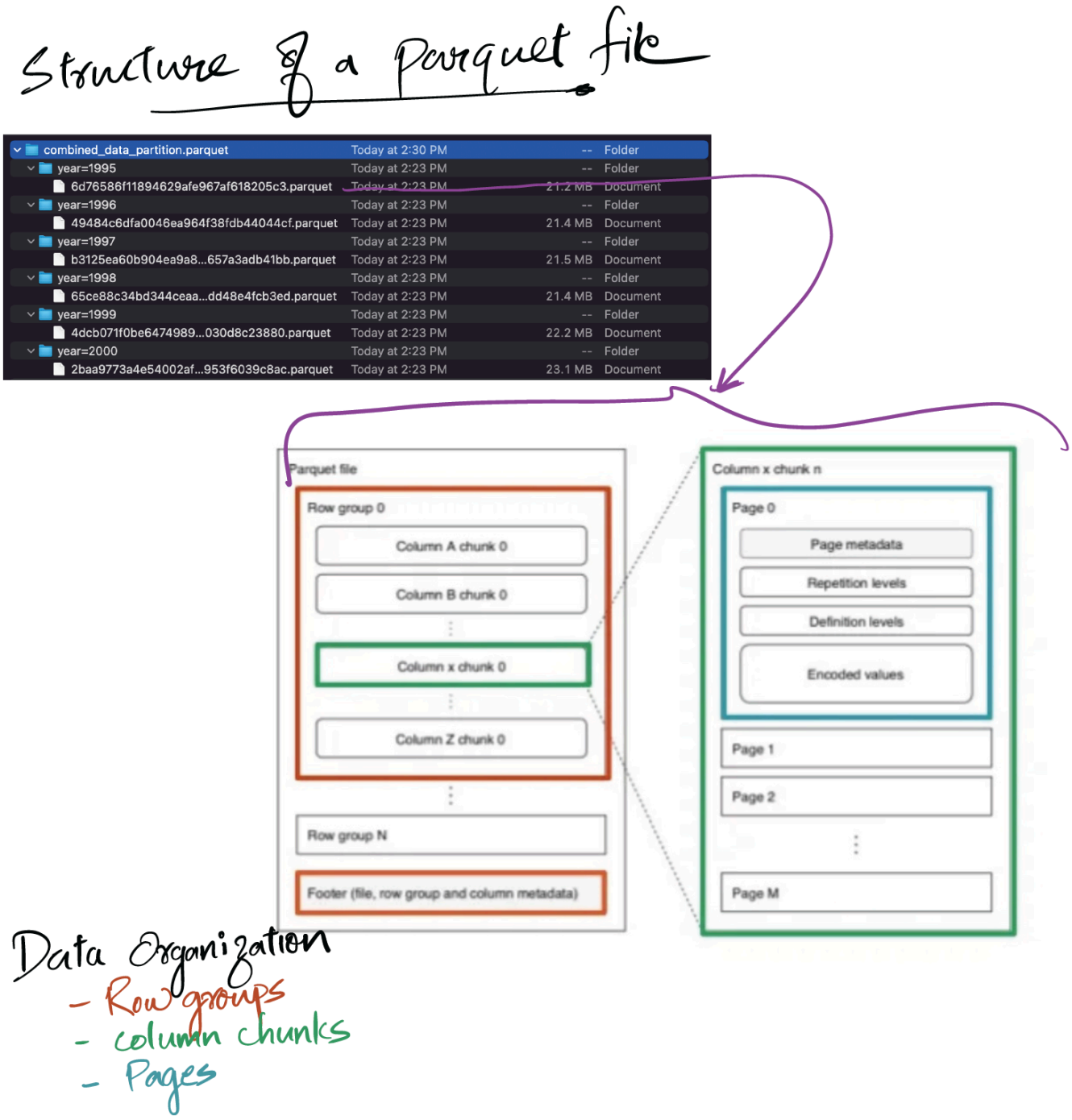
- Supports complex data types and `nested data structures`.

You know about JSON. We can put nested data types and store similar nature of nested schemas in parquet.

You will get an idea of how all the above are possible when you understand the data structure of parquet files.

2.5.1. Data structure (OPTIONAL)

Here is the data organization within a `parquet` file.

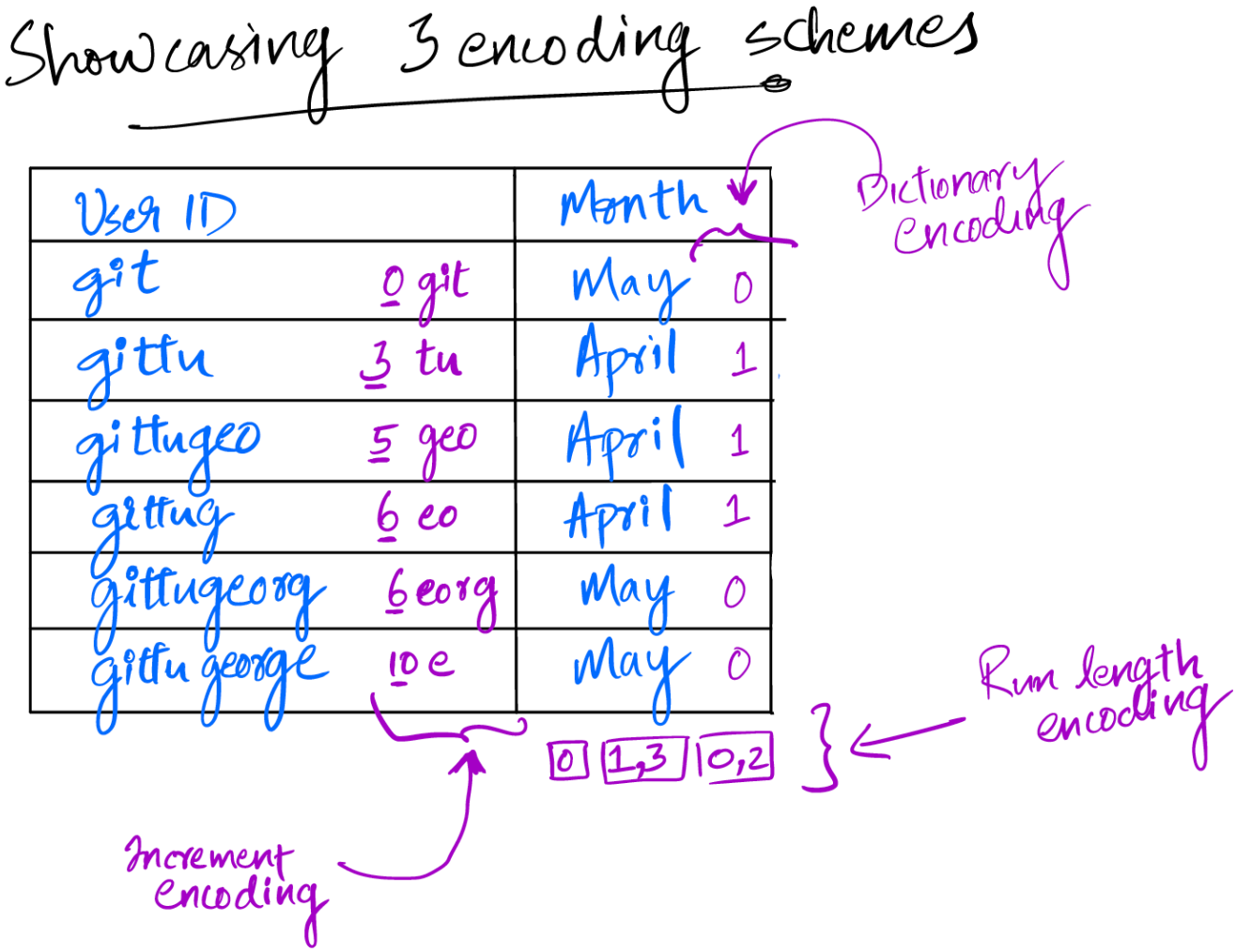


2.5.1.1. Encodings

Various encodings.

- Plain (bit-packed, little-endian, etc.)
- Dictionary Encoding.
- Run Length Encoding/Bit Packing Hybrid.
- Delta Encoding.
- Delta-Length Byte Array.
- Delta Strings (incremental encoding).

Here I will explain three encodings;



So all these encodings will reduce the size of the file, and after encodings are applied, a ***page compression is performed using various compression schemas*** (by default snappy, r lzo , gzip, etc.). These ***compressions and encoding*** collectively ***make a parquet file size very small and efficient***. This, in turn, helps for processing as it ***decreases the I/O***.

Let's use parquet files for analysis, First let's convert our CSV file to parquet.

2.5.2. Converting CSV –> parquet

These days, there are many packages/frameworks (even you can use pandas) for converting to Parquet. However, we will be using the in-memory analytics platform, Arrow (more on it in the next topic). For now, think of it as a package that you can use to convert CSV to Parquet, but you will learn amazing things about arrow soon.

> ⚠ **Warning**
>
> I am using the 20 GB file that I worked on in the last class. You can use the 3 GB file (figshareairline/combined_data.csv) that you combined in the last class. Please make sure you change the path accordingly.

```
%cd /Users/gittugeorge/Desktop/525_2025/figshareexp
```

```
/Users/gittugeorge/Desktop/525_2025/figshareexp
```

```
## Select where you want to write your parquet file, you will be using it for rest of your analysis

# filepathcsv = "/Users/gittugeorge/Desktop/525_2025/figshareexp/figshareairline/combined_data.csv"
# filepathparquet = "/Users/gittugeorge/Desktop/525_2025/figshareexp/figshareairline/combined_data.parque
# filepathparquetr = "/Users/gittugeorge/Desktop/525_2025/figshareexp/figshareairline/combined_data_r.par

# filepathcsv = "/Users/gittugeorge/Desktop/525_2025/figshareexp/figshareairline/bigdata/combined_10gb.cs
# filepathparquet = "/Users/gittugeorge/Desktop/525_2025/figshareexp/figshareairline/bigdata/combined_10g
# filepathparquetr = "/Users/gittugeorge/Desktop/525_2025/figshareexp/figshareairline/bigdata/combined_10

filepathcsv = "/Users/gittugeorge/Desktop/525_2025/figshareexp/figshareairline/bigdata/combined_20gb.csv"
filepathparquet = "/Users/gittugeorge/Desktop/525_2025/figshareexp/figshareairline/bigdata/combined_20gb.
filepathparquetr = "/Users/gittugeorge/Desktop/525_2025/figshareexp/figshareairline/bigdata/combined_20gb
```

```
### Here I am setting my R Home as I had issues otherwise.
# I am setting my R_HOME to the one that is installed with my conda environment
import os
# os.environ['R_HOME'] = '/Users/ggeorg02/opt/miniconda3/envs/525_dev_2025_test_1/lib/R'

os.environ['R_HOME'] = '/Users/gittugeorge/miniforge3/envs/525_dev_2025/lib/R'
```

```
%load_ext rpy2.ipython
```

```
%%time
import pyarrow.dataset as ds
df = ds.dataset(filepathcsv, format="csv")
result = df.scanner(columns=["ArrDelay", "DepDelay", "Distance", "TailNum","UniqueCarrier","Origin","Dest
ds.write_dataset(result,filepathparquet,format = "parquet",partitioning=["year"],partitioning_flavor="hiv
```

```
CPU times: user 1min 56s, sys: 4.75 s, total: 2min 1s
Wall time: 1min 3s
```

```
%%time
%%R -i filepathcsv -i filepathparquetr
suppressMessages(library(arrow, warn.conflicts = FALSE))
suppressMessages(library(dplyr, warn.conflicts = FALSE))
filetest <- open_dataset(filepathcsv, format="csv") %>%
                select("ArrDelay", "DepDelay", "Distance", "TailNum","UniqueCarrier","Origin","Dest","yea
write_dataset(filetest,filepathparquetr, partitioning ="year", existing_data_behavior = "overwrite")
```

```
CPU times: user 1min 51s, sys: 5.35 s, total: 1min 56s
Wall time: 58.5 s
```

2.5.2.1. Documentation

Check out the documentation for the functions that we used in this section.

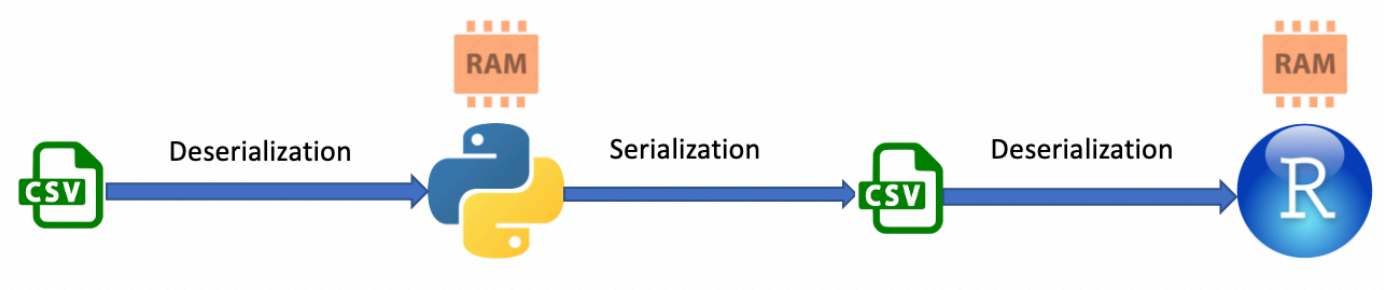
- R

    - open_dataset
    - write_dataset
- python
    - dataset
    - write_dataset

2.6. Serialization/DeSerialization 🧐

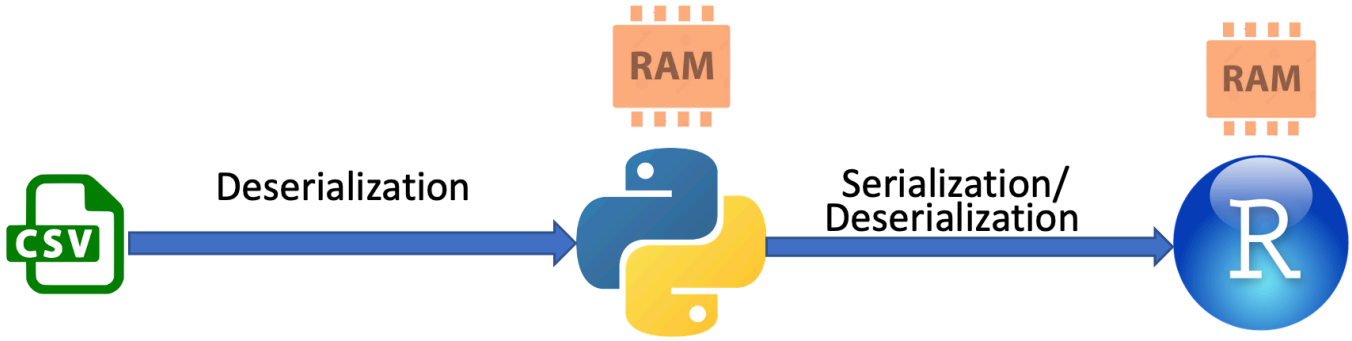
- **Serialization** is the process of saving an object *(so in our case a DataFrame)* state to a sequence of bytes which then can be stored on a file or sent over the network
- **Deserialization** is the process of reconstructing an object *(in our case DataFrame)* from those bytes.

Think about a scenario when you use R and python together. Usually, people make a python data frame available in R *(or VICEVERSA)* by writing to our favorite CSV file format. Here is the workflow that illustrates the internal process.



Also, it's not just about using python and R together. Still, even in the situation where we are just dealing with a CSV file, there is a serialization/deserialization process.

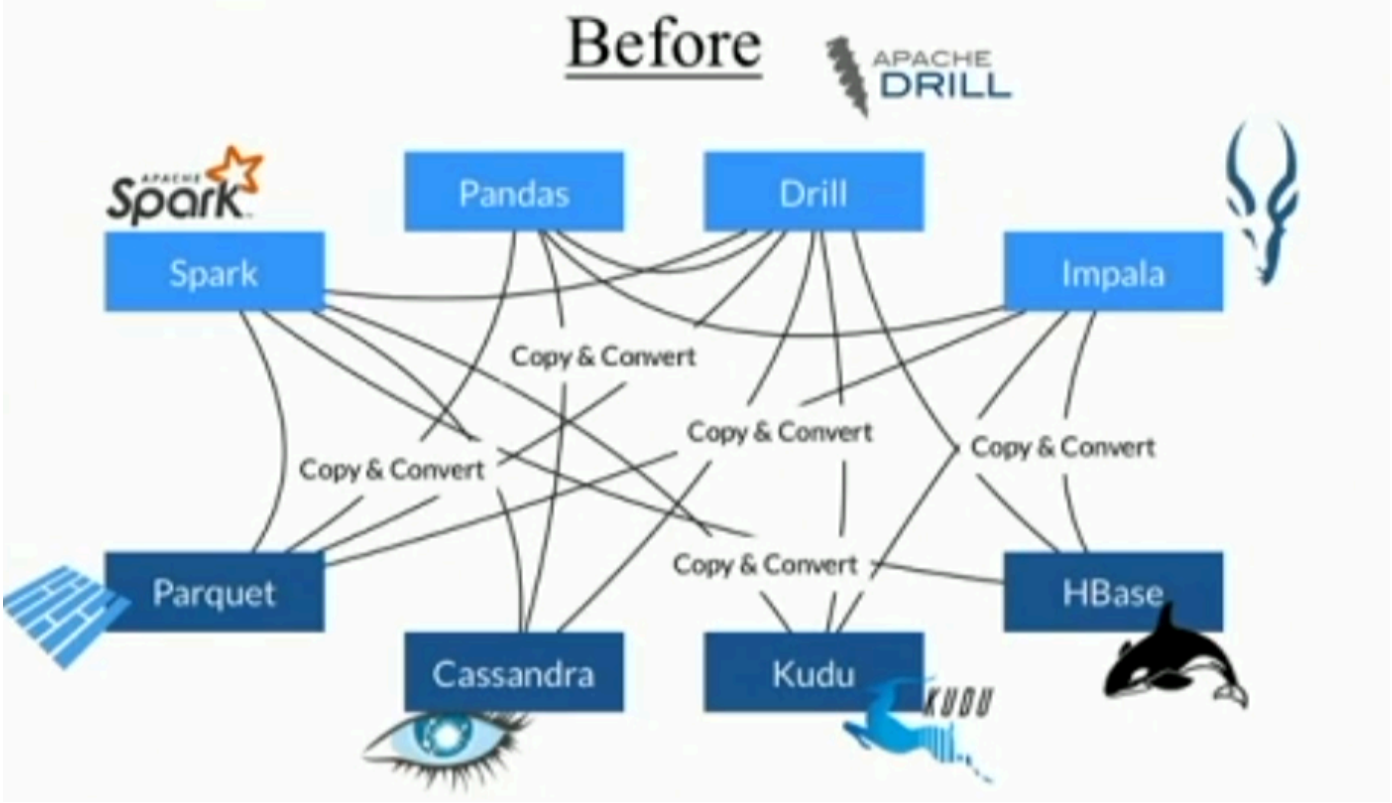
If you think about skipping this CSV thing and transferring it directly from python to R, like this workflow, do you think it's effective? We did this experiment in our last class dataframe transfer from python to R



2.7. Apache Arrow 🏹

In our last class experiment, we loaded data (only 100000 rows) to pandas dataframe and then transferred that dataframe to R using `%%R -i df`. Initially, I loaded my entire dataset and tried to transfer to R, but I couldn't *(on that note, you can try transferring the whole dataset and see what happens)*.

If you notice, the operation we are doing inside R takes less time, about 0.3 seconds. But in total, the cell took around 2 min. So ***most of the time is spent on this serialization and deserialization process***. This is applicable when you think about any packages or frameworks out there; See this image below...

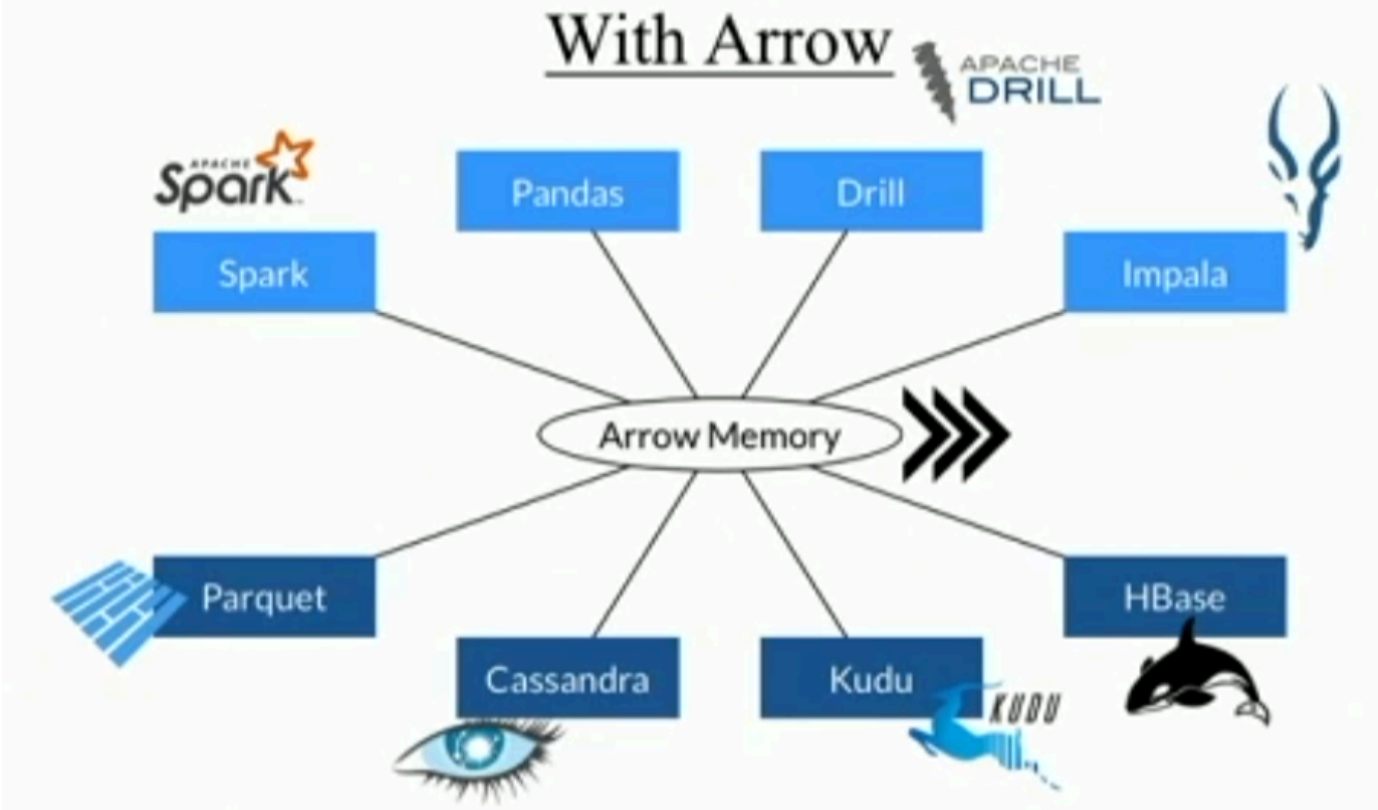


Various languages represent the object differently in memory, and that's why we want to have a standard way of representation in memory that other languages can read, which is the ***main thought behind the arrow project***.

`Apache Arrow` is a cross-language development platform for in-memory data. Here are certain properties of arrow;

- Whenever possible, it will read and process data in chunks and in parallel (default behavior, so you don't want to worry about manually doing it)
- Columnar Memory Format
- Language-independent
- Zero-copy Reads
- Minimum Serialization

How life looks with arrow;



`Apache Arrow` is meant to be a library that library developers can use as the base for providing computational efficiency by providing arrow as a backend.

We won't get into arrow details like how we did with parquet. Let me give you some idea about how arrow manages data. ***Arrow manages tabular data internally using*** `tables` ***or*** `datasets`**.** Arrow provides various functions to load files into `tables` or `datasets`. Arrow provides `datasets` API to work with large datasets ***when it is potentially larger than your RAM.*** We will be mainly working with `datasets` in the rest of the section.

- pyarrow

Apache Arrow python flavor. Some packages already use `pyarrow` (arrow in python) at various levels.

With the amazing features of arrow, it's nice if arrow served as a computational core *(internal memory representation)* to the pandas so that it can make use of efficient in-memory dataframe processing and default with all chunking and parallel processing capabilities. But unfortunately, it's not something that you will see in the near future (getting there; lot more arrow integrations in last month pandas release) as there are many legacy code to be replaced in pandas. Here in optional I have compiled a list of some emerging packages that make use of Arrow for their internal memory representation.

- rarrow

Apache Arrow R flavor.

Wow, R users! Dplyr has great integration with Arrow, and many dplyr verbs can make use of Arrow. You can learn more about this integration here. We will see what I mean by this in our next session.

- Arrow2

Apache Arror Rust flavor.

Polars is an emerging dataframe library for python that is implemented in Rust using apache arrow for internal memory representation.

2.8. Working with Big Data locally (revisiting)

You can look into the previous class section to see how we did with big data locally (I failed miserably with 20 GB file😅).

2.8.1. Storage

2.8.1.1. Filesize comparison

- CSV
- Zipped CSV
- Parquet

```
%%sh
# I am just seeing the size of the csv data
du -sh figshareairline/bigdata/combined_20gb.csv
# I am just seeing the size of the csv data
# zip figshareairline/bigdata/combined_20gb.csv.zip figshareairline/bigdata/combined_20gb.csv
du -sh figshareairline/bigdata/combined_20gb.csv.zip
# I am just seeing the size of the parquet data
du -sh figshareairline/bigdata/combined_20gb.parquet
```

```
 19G    figshareairline/bigdata/combined_20gb.csv
2.9G    figshareairline/bigdata/combined_20gb.csv.zip
2.2G    figshareairline/bigdata/combined_20gb.parquet
```

Thoughts/Discussion ?

Are you thinking just zipping a CSV will reduce its size, and I can load the zipped CSV file into pandas. WHY am I doing all these parquet things?

2.8.2. Memory

2.8.3. Why not use parquet?

Let's revisit our question: Interested in knowing "UniqueCarrier" delays in 2004 where "ArrDelay" (arrival delay) > 10 minutes.

Here what are the rows and columns we are interested in? "UniqueCarrier" , "year" and "ArrDelay". Will we be benefited from using parquet?

```
%%time
##  read the parquet file
import pandas as pd
df = pd.read_parquet(filepathparquet)
print(df[(df.year== 2004) & (df.ArrDelay > 10)]["UniqueCarrier"].value_counts())
```

```
UniqueCarrier
WN    1351002
DL    1134006
AA    1058310
MQ     783066
UA     760152
NW     750180
US     571956
XE     566802
OO     523968
OH     502794
CO     449934
EV     409794
DH     397980
HP     331632
FL     257466
AS     253122
B6     116526
TZ     106506
HA      21852
Name: count, dtype: int64
CPU times: user 42.2 s, sys: 19.4 s, total: 1min 1s
Wall time: 38.5 s
```

Your kernal might crash if you try to load the entire 20 GB file into pandas. Make sure you got enough disk space and memory to run the above. Else expect the below error:

```
The Kernel crashed while executing code in the current cell or a previous cell.
Please review the code in the cell(s) to identify a possible cause of the failure.
Click here for more info.
View Jupyter log for further details.
```

It took around ***50% less time*** to read and process this using a parquet file. ***So are we getting the entire benefits of using parquet?*** The parquet file is small in size, and hence we are saving on I/O 😄, but we are ***not getting*** the parquet file's `predicate pushdown` and `projection pushdown` capabilities!😔 ***WHY?***

Here we are reading the parquet file (using arrow engine as that is the default) to a pandas data frame, and pandas is what doing rest of the computation. Pandas is not designed to work with parquet and doesn't have the capabilities to use projections/predicate pushdown (at least not until today!!!).

To summarize, pandas function `read_parquet` under the hood uses the `arrow` engine to process the parquet file and read it to memory as a pandas dataframe. In addition, there are certain options that we can pass as parameters(`columns=` and `filters=`) to get the columns and rows that we are interested in. Let's try it out.

```
%%time
import pandas as pd
df = pd.read_parquet(filepathparquet,
        filters=[('year','=', 2004),('ArrDelay', '>', 10)],
                columns=['year', 'UniqueCarrier', 'ArrDelay'])
print(df[(df.year== 2004) & (df.ArrDelay > 10)]["UniqueCarrier"].value_counts())
```

```
UniqueCarrier
WN    1351002
DL    1134006
AA    1058310
MQ     783066
UA     760152
NW     750180
US     571956
XE     566802
OO     523968
OH     502794
CO     449934
EV     409794
DH     397980
HP     331632
FL     257466
AS     253122
B6     116526
TZ     106506
HA      21852
Name: count, dtype: int64
CPU times: user 4.02 s, sys: 3.98 s, total: 8 s
Wall time: 7.53 s
```

How to specify filters and columns?

columns=<interested_columns>

eg: `columns=['year', 'UniqueCarrier', 'ArrDelay']`

filters = <interested_predicates>

Each predicate in a tuple is expressed in disjunctive normal form (DNF). Each tuple has a format: (key, op, value) and compares the key with the value. The supported op are: = or ==, !=, <, >, <=, >=, in and not in.

eg: `filters=[('year','=', 2004),('ArrDelay', '>', 10)]`

Here are 2 predicates that I am interested in are `('year','=', 2004)` and `('ArrDelay', '>', 10)`. Here I pass these tuples to the filters parameter like a list of tuples

The ***good news for R users*** ***Parquet and arrow are well integrated into dplyr***, and supports many of its verbs.

To use the rarrow interface to the dplyr, you first need to load both packages, `arrow` and `dplyr`.

```
%%R
suppressMessages(library(arrow, warn.conflicts = FALSE))
suppressMessages(library(dplyr, warn.conflicts = FALSE))
```

We will use the parquet file that we used previously here. I am loading the parquet file using `open_dataset`, a function in rarrow to read as `datasets`. By default, it reads parquet files.

```
%%time
%%R -i filepathparquet
ds <- open_dataset(filepathparquet)
```

```
CPU times: user 10.1 ms, sys: 5.21 ms, total: 15.3 ms
Wall time: 17.7 ms
```

Did you notice that the above cell ran in ~100ms!!! Even though I mentioned `open_dataset` to read datasets, the above code hasn't read anything to memory yet. It just went into to parquet file identified partitions and schemas. So now we can use the dplyr verbs on `ds`.

```
%%time
%%R
ds
result_query <- ds %>%
filter(year==2004,ArrDelay >10) %>%
count(UniqueCarrier)
```

```
CPU times: user 21.1 ms, sys: 3.21 ms, total: 24.3 ms
Wall time: 29.7 ms
```

Did you notice that the above cell ran super quickly!!! Here too, the query is not yet processed. Here it just makes a plan of all computations that you asked for. We ask to filter on `year`, `ArrDelay` and later a count on `UniqueCarrier`. To perform these computations and materialize the result, we use `collect()`, which returns a R dataframe.

Important

`Lazy evaluation` delays computation until required, speeding up queries with optimized plans that push down projections and filters to the parquet file.

Note

It will raise an error if you attempt to call unsupported dplyr verbs. In these situations, `collect()` and then work out using the regular R.

Now, let's trigger the computation. Ready ??

```
%%time
%%R
result <- result_query %>% collect
print(result)
```

```
# A tibble: 19 × 2
   UniqueCarrier       n
   <chr>           <int>
 1 UA             760152
 2 US             571956
 3 AA            1058310
 4 DL            1134006
 5 CO             449934
 6 DH             397980
 7 HP             331632
 8 MQ             783066
 9 OH             502794
10 OO             523968
11 EV             409794
12 FL             257466
13 HA              21852
14 NW             750180
15 WN            1351002
16 XE             566802
17 AS             253122
18 B6             116526
19 TZ             106506
CPU times: user 1 s, sys: 122 ms, total: 1.12 s
Wall time: 211 ms
```

Let's write all we did in one go!

```
%%time
%%R -i filepathparquet
open_dataset(filepathparquet) %>%
    filter(year==2004,ArrDelay >10) %>%
    count(UniqueCarrier) %>%
    collect()
```

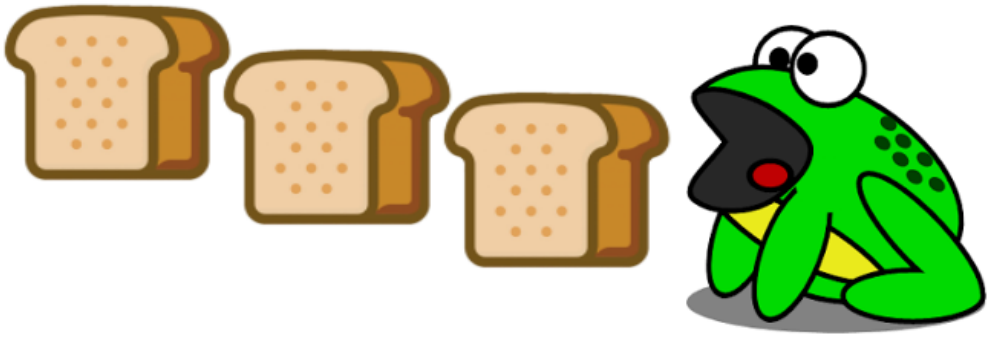
```
# A tibble: 19 × 2
   UniqueCarrier       n
   <chr>           <int>
 1 UA             760152
 2 WN            1351002
 3 AA            1058310
 4 HA              21852
 5 HP             331632
 6 MQ             783066
 7 FL             257466
 8 DH             397980
 9 DL            1134006
10 XE             566802
11 TZ             106506
12 NW             750180
13 EV             409794
14 US             571956
15 OO             523968
16 OH             502794
17 AS             253122
18 B6             116526
19 CO             449934
CPU times: user 1 s, sys: 89.5 ms, total: 1.09 s
Wall time: 139 ms
```

And it takes ms to process around ~55 Million rows.😯 This is super cool, and something similar to this is what we can see from python packages like polars.

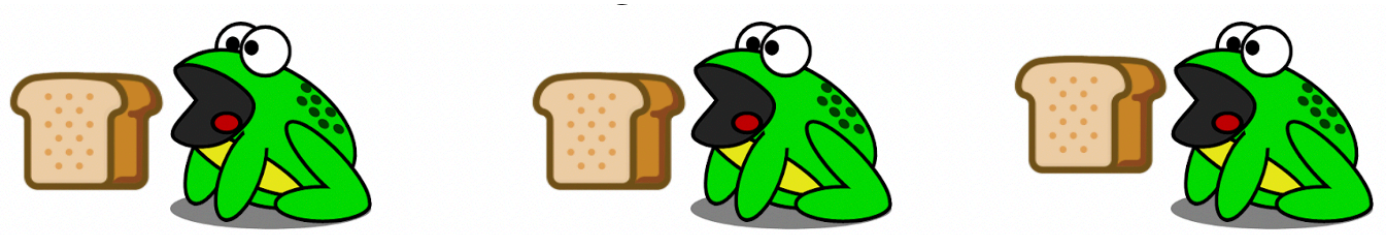
Here the above query took way less time than all the previous reads.✌🏽 This is because we are ***bringing in just the columns and rows we need*** and utilizing the parquet file's ***projection and predicate pushdown capabilities*** using the arrow engine.

2.8.4. Processing

Last week we saw this poor frog...



Won't it be cool if we can process these things parallelly at the same time so that all the cores in our CPU work at the same time?



There are a couple of packages in python and R that does that, here are a few popular (used to be popular) ones...

- multiprocessing
- parallel
- multidplyr

You can check out the above packages, but few orchestrations are needed to use these packages. Won't it be super cool to process these things parallelly without doing much?; Like something that abstracts all these processes of `chunking` and `parallelism` from us so that we can focus just on programming?😳 There are a couple of packages in this category, that are build using `apache arrow` columnar format as its memory model.

We already saw dplyr and how pandas use pyarrow as an engine to read parquet files (pandas can also use pyarrow engine to read csv file parallely - check the optional). You can find couple more packages that make use of apache arrow in optional.

2.8.5. Use R and python interchangeably (with arrow)🤦🏾‍♀️

In our last class experiment, we used a pandas dataframe to move to the R dataframe, and we saw how much time it takes for serialization/deserialization. Here, we demonstrate how Arrow can greatly improve performance when moving data between Python and R, by minimizing the time-consuming serialization/deserialization process. To accomplish this, we need to use an experimental package called rpy2a-arrow. We will use this package to convert a PyArrow table to a RArrow table.

Attention

As we discussed before, the arrow in-memory file format goal is to have a unified format across all platforms. Here pyarrow table is a python wrapper over the arrow data structure, and rarrow table is a r wrapper over arrow data structure. We need to send the rarrow table to the R magics.

```
%reset -f
```

```
%load_ext rpy2.ipython
```

```
The rpy2.ipython extension is already loaded. To reload it, use:
  %reload_ext rpy2.ipython
```

```
filepathcsv = "/Users/gittugeorge/Desktop/525_2025/figshareexp/figshareairline/combined_data.csv"
filepathparquet = "/Users/gittugeorge/Desktop/525_2025/figshareexp/figshareairline/combined_data.parquet"
filepathparquetr = "/Users/gittugeorge/Desktop/525_2025/figshareexp/figshareairline/combined_data_r.parqu
```

```
# !pip install rpy2_arrow
import pyarrow.dataset as ds
import pyarrow as pa
import pandas as pd
import pyarrow
from pyarrow import csv
import rpy2_arrow.pyarrow_rarrow as pyra
```

I am doing the following things in the below cell;

- Loading a CSV file into a `pyarrow dataset`.
- Converting the `pyarrow dataset` to a `pyarrow table`
- Converting a `pyarrow table` to a `rarrow table`

```
%%time
dataset = ds.dataset(filepathcsv, format="csv")
# Converting the `pyarrow dataset` to a `pyarrow table`
table = dataset.to_table()
# Converting a `pyarrow table` to a `rarrow table`
r_table = pyra.converter.py2rpy(table)
```

```
CPU times: user 9.9 s, sys: 1.03 s, total: 10.9 s
Wall time: 10.7 s
```

```
%%time
%%R -i r_table
start_time <- Sys.time()
suppressMessages(library(dplyr))
result <- r_table %>% count(UniqueCarrier)
end_time <- Sys.time()
print(result %>% collect())
print(end_time - start_time)
```

```
# A tibble: 24 × 2
   UniqueCarrier        n
   <chr>            <int>
 1 DL            10435886
 2 HP             2224941
 3 WN            13194660
 4 CO             4976761
 5 US             8286980
 6 UA             8821384
 7 AA             9672922
 8 NW             6946627
 9 TW             1890420
10 AS             2162672
# ℹ 14 more rows
# ℹ Use `print(n = ...)` to see more rows
Time difference of 0.00695014 secs
CPU times: user 826 ms, sys: 16.2 ms, total: 842 ms
Wall time: 88.4 ms
```

In the above experiment, we are transferring an arrow table. So the operation that we are doing inside takes less time, about 0.01899791 secs, and in total, the cell took around 699 ms. ***Time spent on this serialization/deserialization process is very less*** and is also a `zero-copy process`.

> **Note**
>
> The processing of transferring entire 90 Million rows and processing took less time compared to the regular way of transferring, where we did with 100000K rows (using the entire 90 Million failed in my laptop)

2.9. What we learned today?

- Properties of a file format and how to think about it in the big data world.
- Details on the parquet file format.
- Various techniques to bring a smaller footprint of data into memory.
- How to leverage the entire benefits of parquet file for big data.
- How chunking up and processing parallelly can be efficient and help with memory issues.
- Concepts of serialization and deserialization.
- Apache arrow and its role in big data processing (`pyarrow` and `rarrow`).
  - How to efficiently convert CSV to parquet or feather file formats.
  - How `dataset` data structure in apache arrow can be used for larger datasets that don't fit in memory.
  - How arrow improved the serialization and deserialization.
- Introduction to feather file format and where and how it's used.

2.10. Class exercise (worksheet 1)

2.11. Alternate options and developments (OPTIONAL)

> ⚠ **Warning**
>
> If you want to access the following sections, you will need to install some additional packages. I ***strongly want you folks in creating another environment*** if you plan to do so. This will ensure that you don't encounter any conflicts with the current working environment that we have provided you, which is necessary for completing your milestones. Also, note that a few of the following sections are experimental features and it is a buggy land. However, it's good to know that the following exist and they build on all the concepts we learned in this class. Hopefully, they will be mature enough in upcoming years.

2.11.1. Pandas - arrow integration

In the latest release of pandas (pandas 2.0), more integration with Arrow (WOW arrow as a backend) has been added, making pandas an efficient option to work with big data by providing faster and memory-efficient operations. You can read more about it here.

```
%reset -f
```

```
%load_ext rpy2.ipython
```

```
The rpy2.ipython extension is already loaded. To reload it, use:
  %reload_ext rpy2.ipython
```

```
%cd /Users/gittugeorge/Desktop/525_2025/figshareexp
```

```
/Users/gittugeorge/Desktop/525_2025/figshareexp
```

```
# I installed the latest version
#  !pip install --upgrade --pre pandas==2.0.0rc0 #2024: no more a pre-release
## I am removing this version from the dev .yml because it is not yet compatible with ibis.
# (2024 iteration: this is fixed)
## moving cells to markdown --> 2024: moving to code cells
```

```
import pandas as pd
# I want to set the backend following is the default
# pd.options.mode.dtype_backend = "pandas"
df = pd.read_parquet("figshareairline/bigdata/combined_10gb.parquet")
print(df[(df.year== 2004) & (df.ArrDelay > 10)]["UniqueCarrier"].value_counts())
```

```
UniqueCarrier
WN    675501
DL    567003
AA    529155
MQ    391533
UA    380076
NW    375090
US    285978
XE    283401
OO    261984
OH    251397
CO    224967
EV    204897
DH    198990
HP    165816
FL    128733
AS    126561
B6     58263
TZ     53253
HA     10926
Name: count, dtype: int64
```

```
# Here using default pandas and see the data types.
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 258867969 entries, 0 to 258867968
Data columns (total 8 columns):
 #   Column         Dtype
---  ------         -----
 0   ArrDelay       float64
 1   DepDelay       float64
 2   Distance       float64
 3   TailNum        object
 4   UniqueCarrier  object
 5   Origin         object
 6   Dest           object
 7   year           category
dtypes: category(1), float64(3), object(4)
memory usage: 13.7+ GB
```

```
import pandas as pd
# I want to set backend explicitly to pyarrow if I want to use it, no more supported by following style
# pd.options.mode.dtype_backend = "pyarrow"
df = pd.read_parquet("figshareairline/bigdata/combined_10gb.parquet", dtype_backend="pyarrow")
print(df[(df.year== 2004) & (df.ArrDelay > 10)]["UniqueCarrier"].value_counts())
```

```
UniqueCarrier
WN    675501
DL    567003
AA    529155
MQ    391533
UA    380076
NW    375090
US    285978
XE    283401
OO    261984
OH    251397
CO    224967
EV    204897
DH    198990
HP    165816
FL    128733
AS    126561
B6     58263
TZ     53253
HA     10926
Name: count, dtype: int64[pyarrow]
```

```
## Check the data types and you notice those are now arrow types.
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 258867969 entries, 0 to 258867968
Data columns (total 8 columns):
 #   Column         Dtype
---  ------         -----
 0   ArrDelay       double[pyarrow]
 1   DepDelay       double[pyarrow]
 2   Distance       double[pyarrow]
 3   TailNum        string[pyarrow]
 4   UniqueCarrier  string[pyarrow]
 5   Origin         string[pyarrow]
 6   Dest           string[pyarrow]
 7   year           dictionary<values=int32, indices=int32, ordered=0>[pyarrow]
dtypes: dictionary<values=int32, indices=int32, ordered=0>[pyarrow](1), double[pyarrow](3), string[pyarrow
memory usage: 14.0 GB
```

I really now want to try if pandas (with backend arrow) can handle my 20 GB csv file.

```
# import pandas as pd
# df = pd.read_parquet("figshareairline/bigdata/combined_20gb.parquet", dtype_backend="pyarrow")
# print(df[(df.year== 2004) & (df.ArrDelay > 10)]["UniqueCarrier"].value_counts())
```

Well it failed to execute!! I was getting error "ArrowInvalid: offset overflow while concatenating arrays.". After having a quick look at it I noticed it is a bug an open issue and hopefully will be fixed soon. - 2024: still open.

From pandas 1.4.0 started supporting multithreaded `read_csv` using arrow as backend, just pass parameter `engine="pyarrow"` to your `read_csv` function. The arrow engine took around ~15s to load `combined_data.csv`, compared to ~60s the regular way. PS: This functionality is there now for an year and maybe you can use now as an alternative for fast read.

```
%%time
import pandas as pd
df = pd.read_csv("figshareairline/combined_data.csv")
df
```

```
CPU times: user 17.9 s, sys: 6.29 s, total: 24.2 s
Wall time: 26.9 s
```

	UniqueCarrier	TailNum	ArrDelay	DepDelay	Origin	Dest	Distance	year
0	DL	N673DL	66.0	69.0	ATL	PHX	1587.0	1996
1	DL	N686DA	3.0	1.0	ATL	PHX	1587.0	1996
2	DL	N685DA	52.0	26.0	ATL	PHX	1587.0	1996
3	DL	N522DA	84.0	100.0	ATL	PHX	1587.0	1996
4	DL	N2824W	56.0	84.0	ATL	PHX	1587.0	1996
...	...	...	...	...	...	...	...	...
86289318	UA	N344UA	-4.0	-4.0	SMF	DEN	910.0	1998
86289319	UA	N347UA	23.0	8.0	SMF	DEN	910.0	1998
86289320	UA	N311UA	10.0	9.0	SMF	DEN	910.0	1998
86289321	UA	N315UA	13.0	24.0	SMF	DEN	910.0	1998
86289322	UA	N330UA	16.0	-1.0	SMF	DEN	910.0	1998

86289323 rows × 8 columns

```
%%time
df = pd.read_csv("figshareairline/combined_data.csv", engine="pyarrow")
df
```

```
CPU times: user 15.2 s, sys: 4.27 s, total: 19.4 s
Wall time: 4 s
```

```
WARNING: Logging before InitGoogleLogging() is written to STDERR
W20250327 08:51:59.896172 0x206a9c840 status.cc:152] Invalid: Signal stop source already set up
```

	UniqueCarrier	TailNum	ArrDelay	DepDelay	Origin	Dest	Distance	year
0	DL	N673DL	66.0	69.0	ATL	PHX	1587.0	1996
1	DL	N686DA	3.0	1.0	ATL	PHX	1587.0	1996
2	DL	N685DA	52.0	26.0	ATL	PHX	1587.0	1996
3	DL	N522DA	84.0	100.0	ATL	PHX	1587.0	1996
4	DL	N2824W	56.0	84.0	ATL	PHX	1587.0	1996
...	...	...	...	...	...	...	...	...
86289318	UA	N344UA	-4.0	-4.0	SMF	DEN	910.0	1998
86289319	UA	N347UA	23.0	8.0	SMF	DEN	910.0	1998
86289320	UA	N311UA	10.0	9.0	SMF	DEN	910.0	1998
86289321	UA	N315UA	13.0	24.0	SMF	DEN	910.0	1998
86289322	UA	N330UA	16.0	-1.0	SMF	DEN	910.0	1998

86289323 rows × 8 columns

50% faster!!!

2.11.2. Polars

What makes it so special?

- It uses Apache Arrow for internal memory representation.
- It is implemented in Rust.
- It comes with a query optimizer for lazy evaluation.

This is comparable to dplyr in R.

```
## Install polars
# !pip install polars
import polars as pl
pl.__version__
```

```
'0.19.3'
```

Well, I can't use it with my partitioned Parquet file. Hopefully, that feature will be added very soon (2024 iteration, recently feature got added, but some dependency issues, so going with a downgraded version). Let me see how it behaves with a 20 GB CSV file.

```
%%time
import polars as pl
# Load the Parquet file into a Polars DataFrame
df = pl.scan_csv("figshareairline/bigdata/combined_20gb.csv")
counts = df.filter(pl.col("year") == 2004).filter(pl.col("ArrDelay") > 10).groupby(pl.col("UniqueCarrier"
# Print the result. Here this collect() is when it is initiating the process (just like the dplyr what we
counts.collect()
```

```
CPU times: user 27.7 s, sys: 23.1 s, total: 50.9 s
Wall time: 25.1 s
```

```
<timed exec>:4: DeprecationWarning: `groupby` is deprecated. It has been renamed to `group_by`.
```

shape: (19, 2)

UniqueCarrier	count
str	u32
"DH"	397980
"FL"	257466
"TZ"	106506
"NW"	750180
"B6"	116526
"EV"	409794
"AA"	1058310
"AS"	253122
"CO"	449934
"OO"	523968
"DL"	1134006
"HA"	21852
"OH"	502794
"HP"	331632
"WN"	1351002
"UA"	760152
"XE"	566802
"MQ"	783066
"US"	571956

See also

- Here is one latest writeup on how to replace your pandas with polars.
- You can also check this website that compares pandas and polars and can help you get started.

2.11.3. DuckDB

This project is a bit more mature compared to Polars and is a friend to folks who are comfortable with SQL. Essentially, it runs SQL on a file in your system and comes with all the bells and whistles, such as support for lazy evaluation, pushdown features, and using Arrow for its memory representation. You can read more about it here. The good news is that you can use it with Python and R.

```
# !pip uninstall -y duckdb
import duckdb
```

```
%%time
df = duckdb.query('''
SELECT UniqueCarrier, COUNT(*) as count
FROM read_parquet('figshareairline/bigdata/combined_20gb.parquet/*/*.parquet',HIVE_PARTITIONING=true)
WHERE year = 2004 AND ArrDelay > 10
GROUP BY UniqueCarrier
''').df()
df
```

```
CPU times: user 958 ms, sys: 618 ms, total: 1.58 s
Wall time: 216 ms
```

	UniqueCarrier	count
0	OH	502794
1	CO	449934
2	DL	1134006
3	AA	1058310
4	OO	523968
5	US	571956
6	DH	397980
7	TZ	106506
8	AS	253122
9	XE	566802
10	FL	257466
11	WN	1351002
12	NW	750180
13	B6	116526
14	EV	409794
15	HA	21852
16	UA	760152
17	HP	331632
18	MQ	783066

Just for fun let me try to workout the CSV file.

```
%reset -f
```

```
%cd /Users/gittugeorge/Desktop/525_2025/figshareexp
```

```
/Users/gittugeorge/Desktop/525_2025/figshareexp
```

```
%load_ext rpy2.ipython
```

```
The rpy2.ipython extension is already loaded. To reload it, use:
  %reload_ext rpy2.ipython
```

```
import duckdb
```

```
%%time
# Well I was not able to work with 20 GB
df = duckdb.query('''
SELECT UniqueCarrier, COUNT(*) as count
FROM read_csv_auto('figshareairline/bigdata/combined_20gb.csv')
WHERE year = 2004 AND ArrDelay > 10
GROUP BY UniqueCarrier
''').df()
df
```

```
CPU times: user 3min 2s, sys: 6.48 s, total: 3min 9s
Wall time: 18.9 s
```

	UniqueCarrier	count
0	XE	566802
1	AS	253122
2	HA	21852
3	DL	1134006
4	AA	1058310
5	OH	502794
6	CO	449934
7	NW	750180
8	WN	1351002
9	EV	409794
10	FL	257466
11	B6	116526
12	OO	523968
13	DH	397980
14	US	571956
15	TZ	106506
16	HP	331632
17	MQ	783066
18	UA	760152

```
## I tried duckdb installation using install.packages, but it was not working and hence I used conda
# !conda install -c conda-forge -y r-duckdb
# This year they moved it from the conda, but we can now install it using install.packages.
```

```
%%R
# install.packages("duckdb", repos = "https://cloud.r-project.org/")
```

```
NULL
```

```
%%R
library(arrow)
library(duckdb)
ds <- open_dataset("figshareairline/bigdata/combined_20gb.parquet")
con <- dbConnect(duckdb::duckdb())
to_duckdb(ds, table_name = "arrow_table", con = con)
print(dbGetQuery(con, "SELECT UniqueCarrier, COUNT(*) as count FROM arrow_table WHERE year = 2004 AND Arr
```

```
   UniqueCarrier   count
1             CO  449934
2             OH  502794
3             UA  760152
4             HA   21852
5             DL 1134006
6             AA 1058310
7             B6  116526
8             WN 1351002
9             FL  257466
10            AS  253122
11            XE  566802
12            HP  331632
13            MQ  783066
14            EV  409794
15            NW  750180
16            DH  397980
17            US  571956
18            TZ  106506
19            OO  523968
```

```
%%R
library(arrow)
library(duckdb)
ds <- open_dataset("figshareairline/bigdata/combined_20gb.csv", format = "csv")
con <- dbConnect(duckdb::duckdb())
to_duckdb(ds, table_name = "arrow_table", con = con)
print(dbGetQuery(con, "SELECT UniqueCarrier, COUNT(*) as count FROM arrow_table WHERE year = 2004 AND Arr
```

```
   UniqueCarrier   count
1             AS  253122
2             XE  566802
3             DL 1134006
4             AA 1058310
5             CO  449934
6             OH  502794
7             TZ  106506
8             US  571956
9             DH  397980
10            EV  409794
11            NW  750180
12            UA  760152
13            HP  331632
14            MQ  783066
15            HA   21852
16            WN 1351002
17            B6  116526
18            FL  257466
19            OO  523968
```

Wow, I'm glad that I was able to process a file that is larger than my RAM, and in less than 3 minutes, too!

2.11.4. Ibis

In the present era, many new tools and frameworks are emerging, creating a challenge for developers to decide which ones to learn. By the time they finish learning one package, it may already be outdated or lack certain capabilities, forcing them to learn yet another package. This causes significant difficulties for programmers, which is why Ibis was created as a lingua franca. With Ibis, you can learn a single package and then choose the backend on which you want to execute it. For example, below I have shown you the same piece of code (written in ibis) executed in both DuckDB and Polars. You can check out this link to see the bunch of backends that are now supported. You can read more about Ibis here.

```
# !pip install 'ibis-framework[duckdb]'
# !pip install 'ibis-framework[polars]'
import ibis
## default backend is
# ibis.set_backend("duckdb")
```

```
%%time
ibis.set_backend("duckdb")
table = ibis.read_parquet('figshareairline/bigdata/combined_20gb.parquet/*/*.parquet',hive_partitioning =
# Filter the data and count the values
expr = (
    table
    [ (table.year == 2004) & (table.ArrDelay > 10) ] # 2004 in latest release pass as integer
    .group_by('UniqueCarrier')
    .size()
)
# Execute the query
result = expr.execute()
print(result)
```

```
   UniqueCarrier  CountStar()
0             OH       502794
1             CO       449934
2             AA      1058310
3             DL      1134006
4             WN      1351002
5             NW       750180
6             EV       409794
7             FL       257466
8             B6       116526
9             XE       566802
10            AS       253122
11            UA       760152
12            MQ       783066
13            HP       331632
14            HA        21852
15            TZ       106506
16            US       571956
17            OO       523968
18            DH       397980
CPU times: user 1.04 s, sys: 677 ms, total: 1.72 s
Wall time: 449 ms
```

```
%%time
## running as backend duckdb
## default is duckdb so don't want to specify it. But won't hurt specifying.
ibis.set_backend("duckdb")
table = ibis.read_csv('figshareairline/bigdata/combined_20gb.csv')
# Filter the data and count the values
expr = (
    table
    [ (table.year == 2004) & (table.ArrDelay > 10) ]
    .group_by('UniqueCarrier')
    .size()
)
# Execute the query and print the result
result = expr.execute()
print(result)
```

```
   UniqueCarrier  CountStar()
0             OH       502794
1             CO       449934
2             AA      1058310
3             DL      1134006
4             NW       750180
5             B6       116526
6             EV       409794
7             WN      1351002
8             FL       257466
9             AS       253122
10            XE       566802
11            MQ       783066
12            UA       760152
13            HP       331632
14            HA        21852
15            OO       523968
16            TZ       106506
17            DH       397980
18            US       571956
CPU times: user 3min, sys: 6.29 s, total: 3min 7s
Wall time: 18.8 s
```

```
%%time
ibis.set_backend("polars")
table = ibis.read_csv('figshareairline/bigdata/combined_20gb.csv')
# Filter the data and count the values
expr = (
    table
    [ (table.year == 2004) & (table.ArrDelay > 10) ]
    .group_by('UniqueCarrier')
    .size()
)
# print(expr.compile())
# Execute the query
result = expr.execute()
print(result)
```

```
   UniqueCarrier  CountStar()
0             US       571956
1             TZ       106506
2             DH       397980
3             AA      1058310
4             OH       502794
5             HP       331632
6             XE       566802
7             CO       449934
8             OO       523968
9             FL       257466
10            EV       409794
11            UA       760152
12            B6       116526
13            AS       253122
14            DL      1134006
15            WN      1351002
16            MQ       783066
17            HA        21852
18            NW       750180
CPU times: user 26.3 s, sys: 14.9 s, total: 41.2 s
Wall time: 13.9 s
```

2.11.5. Dask

It was quite popular like 3 years back

Dask is a scalable parallel computing library with `dask.dataframe`, a pandas-like API for working with larger than memory datasets in parallel. Dask can use multiple threads or processes on a single machine or a cluster of machines to process data in parallel. The syntax is much similar to the default pandas.

It's in the category of packages that do the chunking and parallel execution. It also does the computation lazily. It can't do automatic predicate and projection pushdown, but hopefully with the merge of this pull request.

2.12. Feather(OPTIONAL)

Let's take a detour to another file format;

You've learned about arrow and feather, which is used to store arrow in-memory data to disk. We've discussed how easy it is to work with arrow and how efficient it is, as we've seen from our previous experiments. However, there's always ongoing development in the research world for better file formats.

Initially, feather was developed as a file format to quickly exchange data between R and Python. Therefore, the workflow of ***read_to_pandas –> write_to_feather –> read to R*** can be used for dataframe transfers. However, it's not recommended to use feather for long-term storage (although this may change in the future), and file formats like parquet are still considered the de facto standard for efficient long-term file storage.

Now, let's work with feather the same way we did with parquet. You will work it out yourself and take a similar approach as we did with parquet.

2.12.1. Save as a feather

Just like how we converted csv –> parquet we specify the file format to be `feather`. We will take a similar approach to what we did in this section. Specifically, look at the documentation that I linked there.

```
%reset -f
```

```
%load_ext rpy2.ipython
```

```
The rpy2.ipython extension is already loaded. To reload it, use:
  %reload_ext rpy2.ipython
```

```
%cd /Users/gittugeorge/Desktop/525_2025/figshareexp/
```

```
/Users/gittugeorge/Desktop/525_2025/figshareexp
```

```
%%time
import pyarrow.dataset as ds
df = ds.dataset("figshareairline/combined_data.csv", format="csv")
result = df.scanner(columns=["ArrDelay", "DepDelay", "Distance", "TailNum","UniqueCarrier","Origin","Dest
ds.write_dataset(result,"figshareairline/combined_data.feather",format = "feather",  existing_data_behavi
```

```
CPU times: user 10.6 s, sys: 1.33 s, total: 11.9 s
Wall time: 10.6 s
```

```
%%R
library(arrow)
library(dplyr)
ds <- open_dataset("figshareairline/combined_data.csv", format="csv")
filetest <- ds %>%
select("ArrDelay", "DepDelay", "Distance", "TailNum","UniqueCarrier","Origin","Dest","year")
write_dataset(filetest,"figshareairline/combined_data.r.feather",format="feather", existing_data_behavior
```

- Let's check the file size

▶ Show code cell content

2.12.2. Read a feather file

We started our class by reading the parquet file, and now, instead of reading the parquet file, lets read the feather and do the computation.

▶ Show code cell content

▶ Show code cell content

Attention

You always look for various blog posts to learn new things, which is fine. However, when it comes to big data-related topics, it's essential to check recent blog posts, preferably those published within the last two years. Additionally, it's a good practice to refer to official documentation while reading blog posts since they are frequently updated with new features, unlike most blog posts. Therefore, cross-referencing any information from blog posts with official documentation is recommended to ensure that you have the most up-to-date and accurate information.