Parquet Basics

03-06-2025

Welcome to the fantastic world of Parquet

The first time I heard about the parquet format was during a specialisation in Data Science & Analytics last year, in the context of data so big that I thought I would never have to deal with it. Our occurrence points datasets, as well as other geodatabases, are becoming large enough that I think they could benefit from these strategies. In early May, I attended the ESRI UK Annual Conference, where they mentioned geoparquet during a talk on data management. I decided to give it a try.

To begin with, I started converting our eBird occurrence points for Australia (.csv) into parquet format. The CSV has 18 GB, while the Parquet is 3.5 GB. Besides, I can save the parquet in groups as useful as families, for example. When I want to read and analyse it, I don't need to read the entire file.

I haven't tryied anything serious with Geoparquet yet.

Most of the text in this document was copied and pasted from various sources, which are cited as notes or references at the end. My goal is only to provide a first contact with it, not to write a super original tutorial.

What is it?

Apache Parquet is an open source, column-oriented data file format designed for efficient data storage and retrieval. It provides high performance compression and encoding schemes to handle complex data in bulk and is supported in many programming language and analytics tools[1].

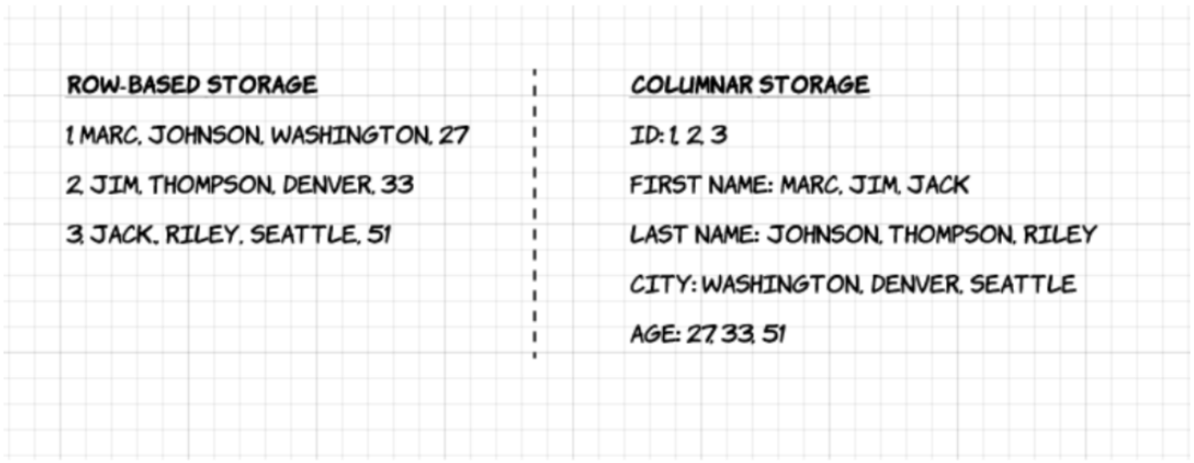
> The open-source project to build Apache Parquet began as a joint effort between Twitter[3] and Cloudera.[4] Parquet was designed as an improvement on the Trevni columnar storage format created by Doug Cutting, the creator of Hadoop. The first version, Apache Parquet 1.0, was released in July 2013. Since April 27, 2015, Apache Parquet has been a top-level Apache Software Foundation (ASF)-sponsored project.

Why should we care?

Apache Parquet is a file format designed to support fast data processing for complex data, with several notable characteristics:

1. **Columnar:** Unlike row-based formats such as CSV, Apache Parquet is column-oriented – meaning the values of each table column are stored next to each other, rather than those of each record:

[1] All about Parquet - Overview


ROW-BASED STORAGE
1 MARC, JOHNSON, WASHINGTON, 27
2 JIM, THOMPSON, DENVER, 33
3 JACK, RILEY, SEATTLE, 51
COLUMNAR STORAGE
ID: 1, 2, 3
FIRST NAME: MARC, JIM, JACK
LAST NAME: JOHNSON, THOMPSON, RILEY
CITY: WASHINGTON, DENVER, SEATTLE
AGE: 27, 33, 51

The key difference between a CSV and Parquet file format is how each one is organized. A Parquet file format is structured by row, with every separate column independently accessible from the rest. Since the data in each column is expected to be of the same type, the parquet file format makes encoding, compressing and optimizing data storage possible.

2. **Open-source:** Parquet is free to use and open source under the Apache Hadoop license.

   Apache Parquet is a columnar storage format available to any project [...], regardless of the choice of data processing framework, data model or programming language[2].

3. **Self-describing:** In addition to data, a Parquet file contains metadata including schema and structure. Each file stores both the data and the standards used for accessing each record – making it easier to decouple services that write, store, and read Parquet files.
4. **Binary format:** Parquet file formats store data in binary format, which reduces the overhead of textual representation. It's important to note that Parquet files are not stored in plain text, thus cannot be opened in a text editor.

Advantages of Parquet Columnar Storage – Why Should You Use It?

The above characteristics of the Apache Parquet file format create several distinct benefits when it comes to storing and analysing large volumes of data.

Compression

File compression is the act of taking a file and making it smaller. In Parquet, compression is performed column by column and it is built to support flexible compression options and extendable encoding schemas per data type – e.g., different encoding can be used for compressing integer and string data.

[2]https://parquet.apache.org/

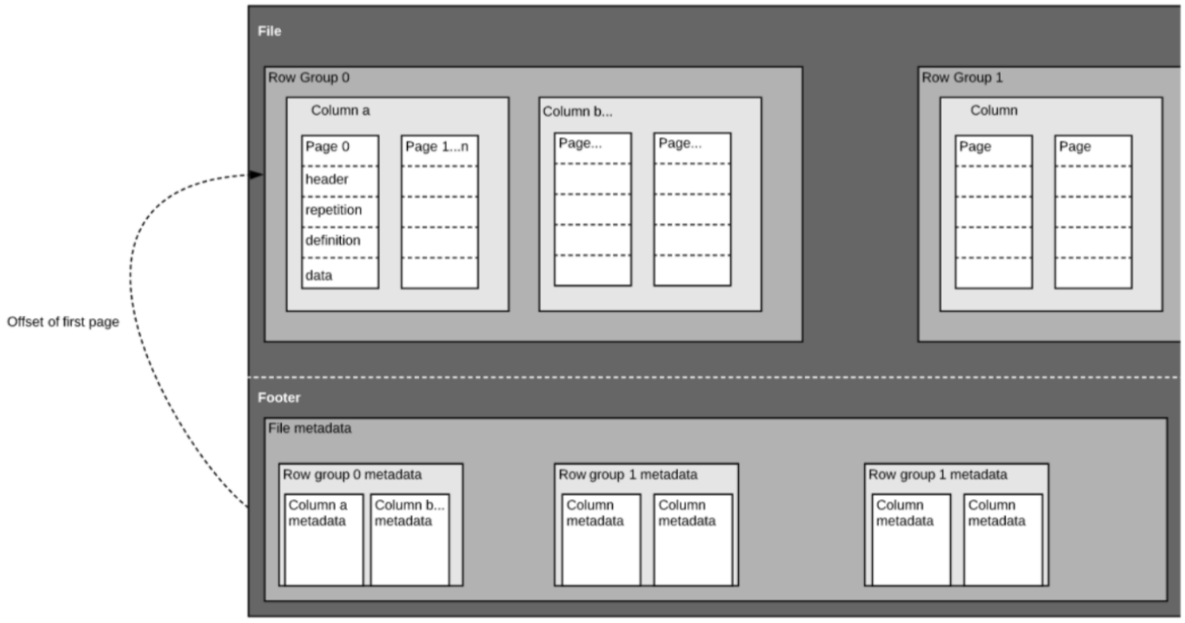
Parquet data can be compressed using these encoding methods:

- **Dictionary encoding:** this is enabled automatically and dynamically for data with a small number of unique values.
- **Bit packing:** Storage of integers is usually done with dedicated 32 or 64 bits per integer. This allows more efficient storage of small integers.
- **Run length encoding (RLE):** when the same value occurs multiple times, a single value is stored once along with the number of occurrences. Parquet implements a combined version of bit packing and RLE, in which the encoding switches based on which produces the best compression results.

Performance

As opposed to row-based file formats like CSV, Parquet is optimized for performance. When running queries on your Parquet-based file-system, you can focus only on the relevant data very quickly. Moreover, the amount of data scanned will be way smaller and will result in less I/O usage. To understand this, let's look a bit deeper into how Parquet files are structured.

As we mentioned above, Parquet is a self-described format, so each file contains both data and metadata. Parquet files are composed of row groups, header and footer. Each row group contains data from the same columns. The same columns are stored together in each row group:



This structure is well-optimized both for fast query performance, as well as low I/O (minimizing the amount of data scanned). For example, if you have a table with 1000 columns, which you will usually only query using a small subset of columns. Using Parquet files will enable you to fetch only the required columns and their values, load those in memory and answer the query. If a row-based file format like CSV was used, the entire table would have to have been loaded in memory, resulting in increased I/O and worse performance.

Schema evolution

When using columnar file formats like Parquet, users can start with a simple schema, and gradually add more columns to the schema as needed. In this way, users may end up with multiple Parquet files with different but mutually compatible schemas. In these cases, Parquet supports automatic schema merging among these files.

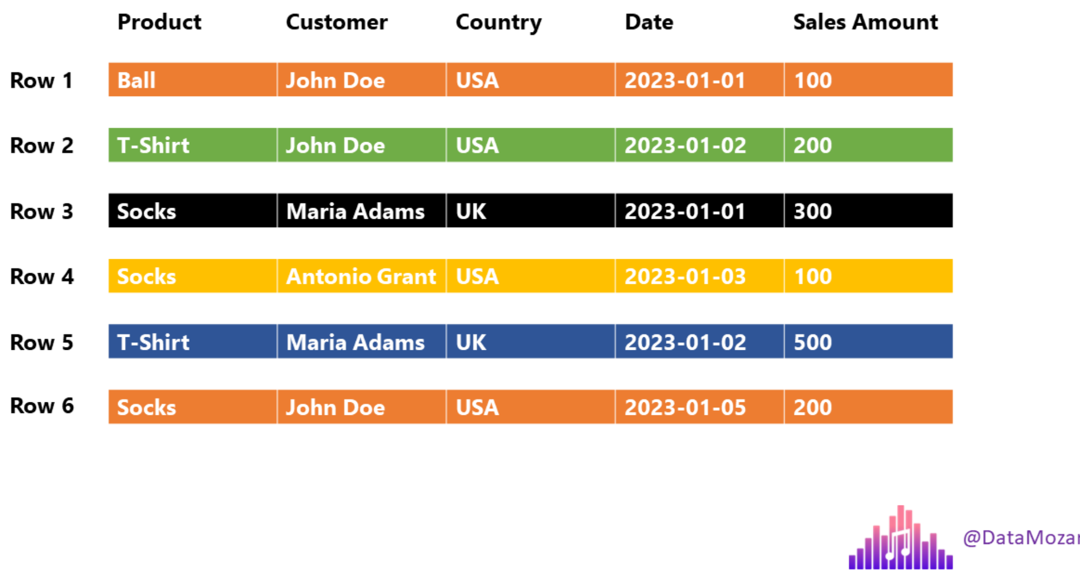
Column-Oriented vs Row-Based Storage for Analytic Querying

Data is often generated and more easily conceptualized in rows. We are used to thinking in terms of Excel spreadsheets, where we can see all the data relevant to a specific record in one neat and organized row. However, for large-scale analytical querying, columnar storage comes with significant advantages with regards to cost and performance.

Complex data such as logs and event streams would need to be represented as a table with hundreds or thousands of columns, and many millions of rows. Storing this table in a row based format such as CSV would mean:

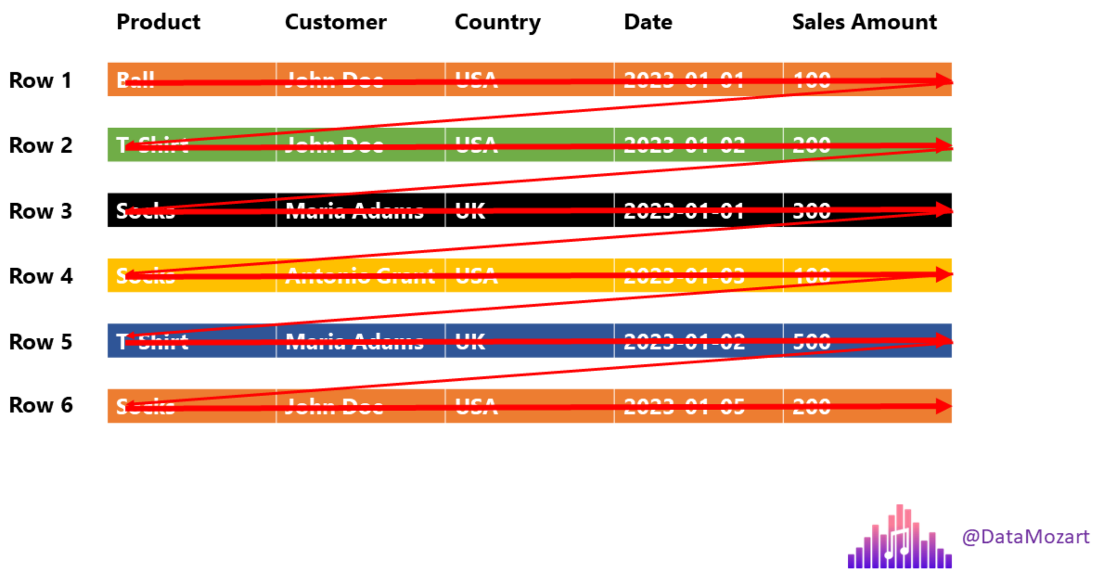
- Queries will take longer to run since more data needs to be scanned, rather than only querying the subset of columns we need to answer a query (which typically requires aggregating based on dimension or category)
- Storage will be more costly since CSVs are not compressed as efficiently as Parquet

Columnar formats provide better compression and improved performance out-of-the-box, and enable you to query data vertically – column by column.



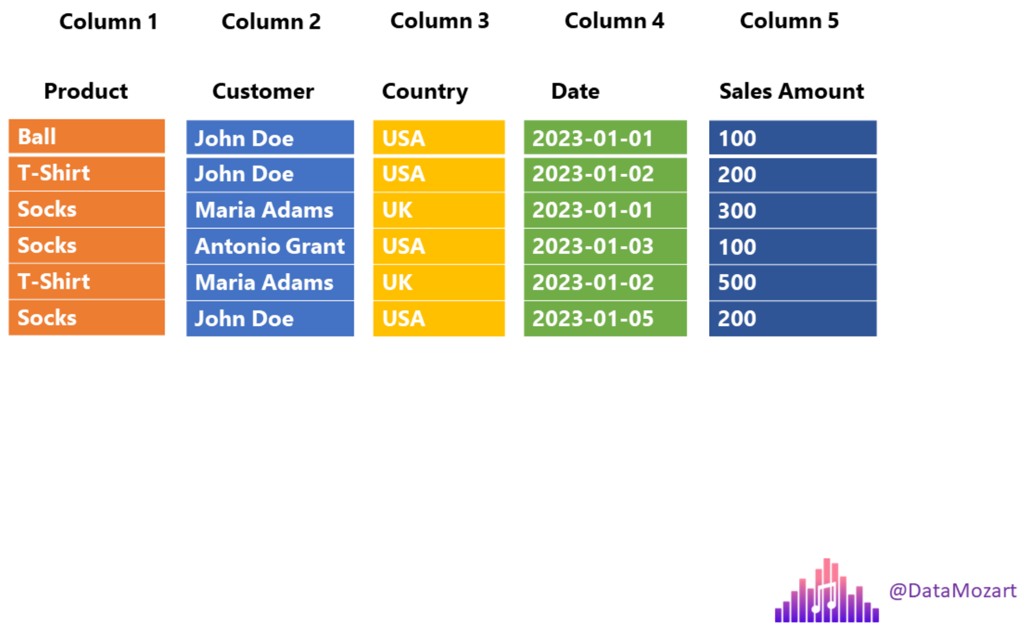
	Product	Customer	Country	Date	Sales Amount
Row 1	Ball	John Doe	USA	2023-01-01	100
Row 2	T-Shirt	John Doe	USA	2023-01-02	200
Row 3	Socks	Maria Adams	UK	2023-01-01	300
Row 4	Socks	Antonio Grant	USA	2023-01-03	100
Row 5	T-Shirt	Maria Adams	UK	2023-01-02	500
Row 6	Socks	John Doe	USA	2023-01-05	200

images columns row[3]



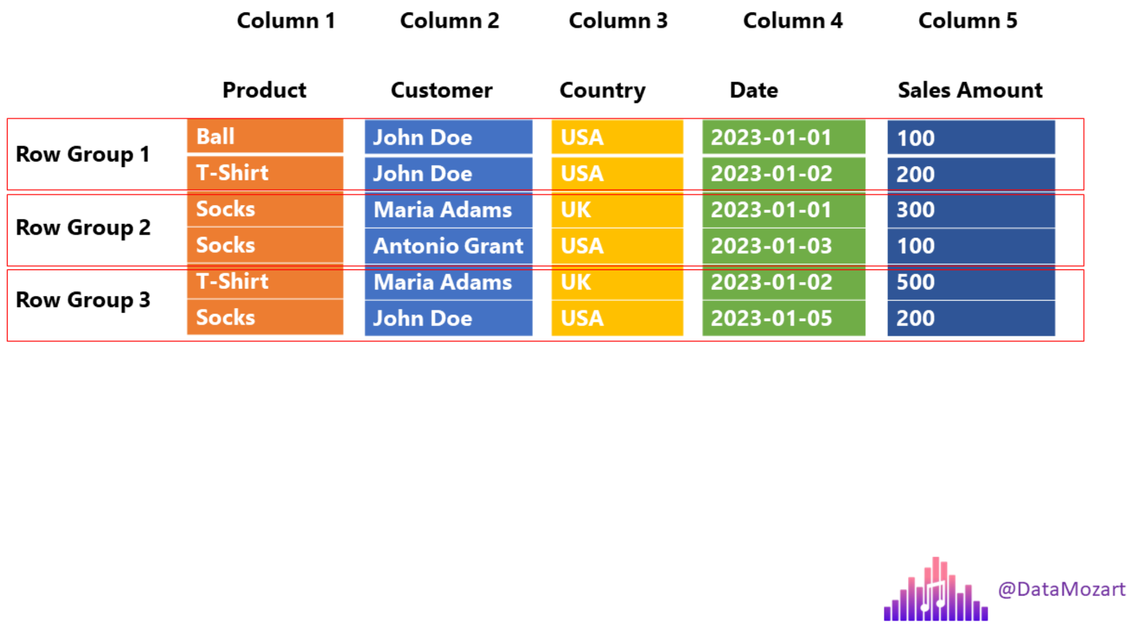
	Product	Customer	Country	Date	Sales Amount
Row 1	Ball	John Doe	USA	2023-01-01	100
Row 2	T-Shirt	John Doe	USA	2023-01-02	200
Row 3	Socks	Maria Adams	UK	2023-01-01	300
Row 4	Socks	Antonio Grant	USA	2023-01-03	100
Row 5	T-Shirt	Maria Adams	UK	2023-01-02	500
Row 6	Socks	John Doe	USA	2023-01-05	200

[3] https://data-mozart.com/parquet-file-format-everything-you-need-to-know/

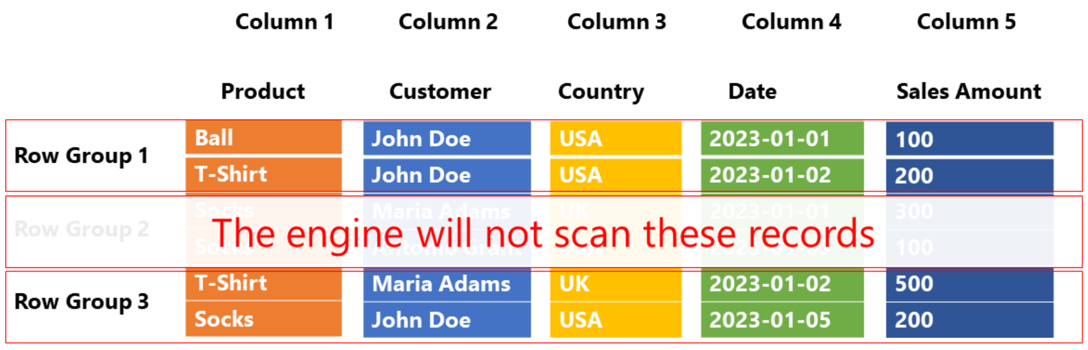


Column 1	Column 2	Column 3	Column 4	Column 5
Product	Customer	Country	Date	Sales Amount
Ball	John Doe	USA	2023-01-01	100
T-Shirt	John Doe	USA	2023-01-02	200
Socks	Maria Adams	UK	2023-01-01	300
Socks	Antonio Grant	USA	2023-01-03	100
T-Shirt	Maria Adams	UK	2023-01-02	500
Socks	John Doe	USA	2023-01-05	200

@DataMozart



	Column 1	Column 2	Column 3	Column 4	Column 5
	Product	Customer	Country	Date	Sales Amount
Row Group 1	Ball	John Doe	USA	2023-01-01	100
	T-Shirt	John Doe	USA	2023-01-02	200
Row Group 2	Socks	Maria Adams	UK	2023-01-01	300
	Socks	Antonio Grant	USA	2023-01-03	100
Row Group 3	T-Shirt	Maria Adams	UK	2023-01-02	500
	Socks	John Doe	USA	2023-01-05	200



	Column 1	Column 2	Column 3	Column 4	Column 5
	Product	Customer	Country	Date	Sales Amount
Row Group 1	Ball	John Doe	USA	2023-01-01	100
	T-Shirt	John Doe	USA	2023-01-02	200
Row Group 2	Socks	Maria Adams	UK	2023-01-01	300
	Socks	Antonio Grant	USA	2023-01-03	100
Row Group 3	T-Shirt	Maria Adams	UK	2023-01-02	500
	Socks	John Doe	USA	2023-01-05	200

Let's quickly stop here, as I want you to realize the difference between various types of storage in terms of the work that needs to be performed by the engine:

```
Row store - the engine needs to scan all 5 columns and all 6 rows
Column store - the engine needs to scan 2 columns and all 6 rows
Column store with row groups - the engine needs to scan 2 columns and 4 rows
```

"I'm tired of reading non-sense"

Ok, then let me introduce you to the R packages I've been exploring and how they made my life easier.

arrow

The R `arrow` package provides access to many of the features of the Apache Arrow C++ library for R users. The goal of arrow is to provide an Arrow C++ back-end to `dplyr`, and access to the Arrow C++ library through familiar base R and tidyverse functions, or R6 classes. The dedicated R package website is located here.

What can the arrow package do?

The arrow package provides binding to the C++ functionality for a wide range of data analysis tasks.

- It allows users to read and write data in a variety formats:
  - Read and write Parquet files, an efficient and widely used columnar format
  - Read and write Arrow (formerly known as Feather) files, a format optimized for speed and interoperability*
  - Read and write CSV files with excellent speed and efficiency*
  - Read and write multi-file and larger-than-memory datasets
  - Read JSON files*

**I haven't tried it yet, let me know if you find any exciting adventures*

- It provides access to remote file systems and servers and other cloud related stuff you can read more about here.
- Additional features include:
  - Manipulate and analyse Arrow data with beloved dplyr verbs
  - Fine control over column types to work seamlessly with databases and data warehouses

What is Apache Arrow?

Apache Arrow is a cross-language development platform for in-memory and larger-than-memory data. It specifies a standardized language-independent columnar memory format for flat and hierarchical data, organized for efficient analytic operations on modern hardware. It also provides computational libraries and zero-copy streaming, messaging, and interprocess communication.

Arrow resources

There are a few additional resources that you may find useful for getting started with arrow:

- The official Arrow R package documentation
- Arrow for R cheatsheet
- Apache Arrow R Cookbook
- R for Data Science Chapter on Arrow
- Awesome Arrow R

Installation

The latest release of arrow can be installed from CRAN. In most cases installing the latest release should work without requiring any additional system dependencies, especially if you are using Windows or macOS.

```
install.packages('arrow')
```

Alternative install option for grabbing the latest arrow release in case the previous doesn't work.

```
install.packages("arrow", repos = c("https://apache.r-universe.dev", "https://cloud.r-projec
```

Get started with Arrow

In the example below, we take the starwars data provided by the `dplyr` package and write it to a Parquet file using `write_parquet()`

```
library(arrow)
library(dplyr)

starwars
```

```
# A tibble: 87 x 14
   name     height  mass hair_color skin_color eye_color birth_year sex   gender
   <chr>     <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
 1 Luke Sk~    172    77 blond      fair       blue            19   male  mascu~
 2 C-3PO       167    75 <NA>       gold       yellow         112   none  mascu~
 3 R2-D2        96    32 <NA>       white, bl~ red             33   none  mascu~
 4 Darth V~    202   136 none       white      yellow          41.9 male  mascu~
 5 Leia Or~    150    49 brown      light      brown           19   fema~ femin~
 6 Owen La~    178   120 brown, gr~ light      blue            52   male  mascu~
 7 Beru Wh~    165    75 brown      light      blue            47   fema~ femin~
 8 R5-D4        97    32 <NA>       white, red red             NA   none  mascu~
 9 Biggs D~    183    84 black      light      brown           24   male  mascu~
10 Obi-Wan~    182    77 auburn, w~ fair       blue-gray       57   male  mascu~
# i 77 more rows
# i 5 more variables: homeworld <chr>, species <chr>, films <list>,
#   vehicles <list>, starships <list>
```

```r
write_parquet(starwars, 'outputs/starwars.parquet')
```

Now we can then use `read_parquet()` to load the data from this file. The default behaviour is to return a data frame (`sw_frame`) but when we set `as_data_frame = FALSE` the data are read as an Arrow Table (`sw_table`):

```r
sw_frame <- read_parquet('outputs/starwars.parquet')
sw_table <- read_parquet('outputs/starwars.parquet', as_data_frame = FALSE)
sw_table
```

```
Table
87 rows x 14 columns
$name <string>
$height <int32>
$mass <double>
$hair_color <string>
$skin_color <string>
$eye_color <string>
$birth_year <double>
$sex <string>
$gender <string>
$homeworld <string>
$species <string>
$films: list<element <string>>
$vehicles: list<element <string>>
$starships: list<element <string>>
```

One useful feature of Parquet files is that they store data column-wise, and contain metadata that allow file readers to skip to the relevant sections of the file. That means it is possible to load only a subset of the columns without reading the complete file. The col_select argument to `read_parquet()` supports this functionality:

```r
colnames(sw_table)
```

```
 [1] "name"       "height"     "mass"       "hair_color" "skin_color"
 [6] "eye_color"  "birth_year" "sex"        "gender"     "homeworld" 
[11] "species"    "films"      "vehicles"   "starships" 
```

```r
read_parquet('outputs/starwars.parquet', col_select = c("name", "height", "mass"))
```

```
# A tibble: 87 x 3
   name               height  mass
   <chr>               <int> <dbl>
 1 Luke Skywalker        172    77
 2 C-3PO                 167    75
 3 R2-D2                  96    32
 4 Darth Vader           202   136
 5 Leia Organa           150    49
 6 Owen Lars             178   120
 7 Beru Whitesun Lars    165    75
 8 R5-D4                  97    32
 9 Biggs Darklighter     183    84
10 Obi-Wan Kenobi        182    77
# i 77 more rows
```

Multi-file data sets - my favourite part!

When a tabular data set becomes large, it is often good practice to partition the data into meaningful subsets and store each one in a separate file. Among other things, this means that if only one subset of the data are relevant to an analysis, only one (smaller) file needs to be read. The `arrow` package provides the Dataset interface, a convenient way to read, write, and analyse a single data file that is larger-than-memory and multi-file data sets.

To illustrate the concepts, we'll create a nonsense data set with 100000 rows that can be split into 10 subsets:

```r
set.seed(1234)
nrows <- 100000
random_data <- data.frame(
  x = rnorm(nrows),
  y = rnorm(nrows),
  subset = sample(10, nrows, replace = TRUE)
)
```

What we might like to do is partition this data and then write it to 10 separate Parquet files, one corresponding to each value of the subset column. To do this we first specify the path to a folder into which we will write the data files:

```r
dataset_path <- "outputs/random_data"
```

We can then use `group_by()` function from `dplyr` to specify that the data will be partitioned using the `subset` column, and then pass the grouped data to `write_dataset()`:

```r
random_data %>%
  group_by(subset) %>%
  write_dataset(dataset_path)
```

This creates a set of 10 files, one for each subset. These files are named according to the "hive partitioning" format as shown below:

```r
list.files(dataset_path, recursive = TRUE)
```

```
 [1] "subset=1/part-0.parquet"  "subset=10/part-0.parquet"
 [3] "subset=2/part-0.parquet"  "subset=3/part-0.parquet"
 [5] "subset=4/part-0.parquet"  "subset=5/part-0.parquet"
 [7] "subset=6/part-0.parquet"  "subset=7/part-0.parquet"
 [9] "subset=8/part-0.parquet"  "subset=9/part-0.parquet"
```

Each of these Parquet files can be opened individually using `read_parquet()` but is often more convenient – especially for very large data sets – to scan the folder and "connect" to the data set without loading it into memory. We can do this using `open_dataset()`:

```r
dset <- open_dataset(dataset_path)
dset
```

```
FileSystemDataset with 10 Parquet files
3 columns
x: double
y: double
subset: int32
```

This `dset` object does not store the data in-memory, only some metadata. However, it is possible to analyse the data referred to be `dset` as if it had been loaded.

Other examples on reading more complex datasets can be found here.

Analising Arrow data with `dplyr`

Arrow Tables and Datasets can be analysed using `dplyr` syntax. This is possible because the arrow R package supplies a back-end that translates `dplyr` verbs into commands that are understood by the Arrow C++ library, and will similarly translate R expressions that appear within a call to a `dplyr` verb. For example, although the `dset` Dataset is not a data frame (and does not store the data values in memory), you can still pass it to a `dplyr` pipeline like the one shown below:

```r
dset %>%
  group_by(subset) %>%
  summarize(mean_x = mean(x), min_y = min(y)) %>%
  filter(mean_x > 0) %>%
  arrange(subset) %>%
  collect()
```

```
# A tibble: 6 x 3
  subset  mean_x min_y
   <int>   <dbl> <dbl>
1      2 0.00486 -4.00
2      3 0.00440 -3.86
3      4 0.0125  -3.65
4      6 0.0234  -3.88
5      7 0.00477 -4.65
6      9 0.00557 -3.50
```

You can find a list of other fun functions available in `dplyr` queries here.

Have you notice the `collect()` at the end of the pipeline? No actual computations are performed until `collect()` (or the related `compute()` function) is called. This "lazy evaluation" makes it possible for the Arrow C++ compute engine to optimize how the computations are performed.

Your turn

I may add some examples with real data here in the future. Meanwhile, try writing your data frames in Parquet format using `group_by()` to partition it. For instance, if you have a species list, try grouping by family names.

You can try write parquet file parts grouping by two columns (e.g. month and year).

Geoparquet - `sfarrow`

Yes, there is parquet file format for spatial data.

I'll just leave the link to the package here for now, but we can try and practice together with species range maps, for example :)

Getting started with amazing `sfarrow`

tidyverse - parquet

There is a tidyverse parquet package that I haven't tried yet, but they are usually really cool.