Chapter 3-1

Bonus Recipe - Writing large datasets

Writing large datasets

In this recipe, you will explore how the choice of different file formats can impact the overall write and read performance. You will explore Parquet, Optimized Row Columnar (ORC), and Feather, and compare their performance to other popular file formats such as JSON and CSV.

The three file formats, ORC, Feather, and Parquet, are columnar file formats, making them efficient for analytical needs and showing improved querying performance overall. The three file formats are also supported in Apache Arrow (PyArrow), which offers a standardized in-memory columnar data format for optimized data analysis performance. To persist this in-memory columnar and store it, you can use the pandas `to_orc`, `to_feather`, and `to_parquet` methods to write your data to disk.

Arrow provides the in-memory representation of the data as a columnar format, while Feather, ORC, and Parquet allow us to store this representation on disk.

Getting ready

In this recipe, we will be working with the New York Taxi dataset from (`https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page`), and we will be working with Yellow Taxi Trip Records for 2023.

In the following examples, we will be using one of these files, `yellow_tripdata_2023-01.parquet`, but you can select any other file to follow along. In the *Reading data from Parquet files* recipe from *Chapter 1*, you installed **PyArrow**. Below are the instructions for installing PyArrow using either Conda or Pip.

To install PyArrow using `conda`, run the following command:

```
conda install -c conda-forge pyarrow
```

To install PyArrow using `pip`, run the following command:

```
pip install pyarrow
```

To prepare for this recipe, you will read the file into a DataFrame with the following code:

```
import pandas as pd
from pathlib import Path

file_path = Path('yellow_tripdata_2023-01.parquet')
df = pd.read_parquet(file_path, engine='pyarrow')
df.info()

>>
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3066766 entries, 0 to 3066765
Data columns (total 19 columns):
 #   Column                 Dtype
---  ------                 -----
 0   VendorID               int64
 1   tpep_pickup_datetime   datetime64[us]
 2   tpep_dropoff_datetime  datetime64[us]
 3   passenger_count        float64
 4   trip_distance          float64
 5   RatecodeID             float64
 6   store_and_fwd_flag     object
 7   PULocationID           int64
 8   DOLocationID           int64
 9   payment_type           int64
 10  fare_amount            float64
 11  extra                  float64
 12  mta_tax                float64
 13  tip_amount             float64
 14  tolls_amount           float64
 15  improvement_surcharge  float64
 16  total_amount           float64
 17  congestion_surcharge   float64
 18  airport_fee            float64
dtypes: datetime64[us](2), float64(12), int64(4), object(1)
memory usage: 444.6+ MB
```

How to do it

You will write the DataFrame into different file formats and then compare the output in terms of compression efficiency (file size), write, and read speed.

To accomplish this, you will need to create a function that returns the file size:

```
import os
def size_in_mb(file):
    size_bytes = os.path.getsize(file)
    size_m = size_bytes / (1024**2)
    return round(size_m,2)
```

The function will take the file you created and return the size in megabytes. The `os.path.getsize()` will return the size in bytes and the line `size_bytes / (1024**2)` will convert it into megabytes.

We will be writing these files into a `formats` folder so, later, we can read from that folder to evaluate read performance.

Throughout this recipe, you'll notice `%%time` preceding some code blocks. This is a **Jupyter notebook** magic command that measures and reports execution time for the entire cell. This will help us compare performance across different file formats and compression algorithms. The output shows both CPU time (split between user and system processes) and wall time (the actual elapsed time), with wall time being our primary metric for comparison.

Writing as JSON and CSV

You will use the `DataFrame.to_json()` method to write a `yellow_tripdata.json` file:

```
%%time
df.to_json('formats/yellow_tripdata.json', orient='records')
size_in_mb('formats/yellow_tripdata.json')
>>
CPU times: user 4.63 s, sys: 586 ms, total: 5.22 s
Wall time: 5.24 s
1165.21
```

Note the file size is around 1.16 GB and took around 5.24 seconds.

You will use the `DataFrame.to_csv()` method to write a `yellow_tripdata.csv` file:

```
%%time
df.to_csv('formats/yellow_tripdata.csv', index=False)
size_in_mb('formats/yellow_tripdata.csv')
>>
CPU times: user 16.7 s, sys: 405 ms, total: 17.1 s
Wall time: 17.1 s
307.04
```

Note the file size is around 307 MB and took around 17.1 seconds.

Writing as Parquet

The `to_parquet` writer function supports several compression algorithms, including `snappy`, `GZIP`, `brotli`, `LZ4`, and `ZSTD`. You will use the `DataFrame.to_parquet()` method to write three files to compare `snappy`, `LZ4`, and `ZSTD` compression algorithms:

```
%%time
df.to_parquet('formats/yellow_tripdata_snappy.parquet',
              compression='snappy')
size_in_mb('formats/yellow_tripdata_snappy.parquet')
>>
CPU times: user 882 ms, sys: 24.2 ms, total: 906 ms
Wall time: 802 ms
59.89

%%time
df.to_parquet('formats/yellow_tripdata_lz4.parquet',
              compression='lz4')
size_in_mb('formats/yellow_tripdata_lz4.parquet')
>>
CPU times: user 898 ms, sys: 20.4 ms, total: 918 ms
Wall time: 817 ms
59.92

%%time
df.to_parquet('formats/yellow_tripdata_zstd.parquet',
              compression='zstd')
size_in_mb('formats/yellow_tripdata_zstd.parquet')
>>
CPU times: user 946 ms, sys: 24.2 ms, total: 970 ms
Wall time: 859 ms
48.95
```

Both Snappy and LZ4 offer faster write speeds with moderate compression. ZSTD typically achieves better compression ratios, but with slightly longer write times.

Writing as Feather

You will use the `DataFrame.to_feather()` method to write three Feather files using the two supported compression algorithms `LZ4` and `ZSTD`. The last file format will be the uncompressed format for comparison:

```
%%time
df.to_feather('formats/yellow_tripdata_uncompressed.feather',
compression='uncompressed')
```

```
size_in_mb('formats/yellow_tripdata_uncompressed.feather')
>>
CPU times: user 182 ms, sys: 75.5 ms, total: 257 ms
Wall time: 291 ms
435.84

%%time
df.to_feather('formats/yellow_tripdata_lz4.feather', compression='lz4')
size_in_mb('formats/yellow_tripdata_lz4.feather')
>>
CPU times: user 654 ms, sys: 42.1 ms, total: 696 ms
Wall time: 192 ms
116.44

%%time
df.to_feather('formats/yellow_tripdata_zstd.feather', compression='zstd',
compression_level=3)
size_in_mb('formats/yellow_tripdata_zstd.feather')
>>
CPU times: user 1 s, sys: 39.2 ms, total: 1.04 s
Wall time: 243 ms
61.79
```

Notice the difference in file size between uncompressed using LZ4 and ZSTD compression algorithms. You can further explore `compression_level` to find the optimal output. Overall, LZ4 offers great performance on write and read (*compression* and *decompression* speed). The ZSTD algorithm may offer a higher compression ratio, resulting in much smaller files, but it may not be faster than LZ4. This makes LZ4 ideal for scenarios where processing speed is more important than storage space, while ZSTD shines when storage efficiency is the priority.

Writing as ORC

Similar to the Feather and Parquet file formats, ORC supports different compression algorithms, including uncompressed, `snappy`, `ZLIB`, `LZ4`, and `ZSTD`. You will use the `DataFrame.to_orc()` method to write three ORC files to explore ZSTD and LZ4 compression algorithms and an uncompressed file for comparison:

```
%%time
df.to_orc('formats/yellow_tripdata_uncompressed.orc',
          engine_kwargs={'compression':'uncompressed'})
size_in_mb('formats/yellow_tripdata_uncompressed.orc')
>>
CPU times: user 989 ms, sys: 66.3 ms, total: 1.06 s
Wall time: 1.01 s
```

```
319.94

%%time
df.to_orc('formats /yellow_tripdata_lz4.orc',
          engine_kwargs={'compression':'lz4'})
size_in_mb('formats/yellow_tripdata_lz4.orc')
>>
CPU times: user 1 s, sys: 67.2 ms, total: 1.07 s
Wall time: 963 ms
319.65

%%time
df.to_orc('yellow_tripdata_zstd.orc',
          engine_kwargs={'compression':'zstd'})
size_in_mb('formats/yellow_tripdata_zstd.orc')
>>
CPU times: user 1.47 s, sys: 46.4 ms, total: 1.51 s
Wall time: 1.42 s
53.58
```

Notice that the LZ4 algorithm did not offer better compression when compared with the uncompressed version. The ZSTD algorithm did offer better compression but took a bit longer to execute. Generally, compression effectiveness can vary based on data characteristics.

How it works...

Often, when working with large datasets that need to persist on disk after completing your transformations, deciding which file format to opt for can significantly impact your overall data storage strategy.

For example, JSON and CSV formats are human-readable choices, and pretty much any commercial or open source data visualization or analysis tools can handle such formats. Both CSV and JSON formats do not natively support compression (without additional tools such as gzip) for large file sizes, which can lead to poor performance on both write and read operations. On the other hand, Parquet, Feather, and ORC are binary file formats (not human-readable) but support several compression algorithms and are columnar-based, which are optimized for analytical applications with fast read performance.

The columnar structure of these formats stores each column's data together, rather than storing data row by row. This approach allows for more efficient querying when you only need specific columns, as the system can skip reading irrelevant columns entirely. Additionally, since data in each column typically has similar characteristics, compression algorithms can work more effectively, further improving both storage efficiency and read performance.

The pandas library supports Parquet, Feather, and ORC thanks to PyArrow, a Python wrapper to Apache Arrow.

There's more...

You have evaluated different file formats' write performance (and size). Next, you will compare read time performance and the efficiency of the various file formats and compression algorithms.

To do this, you will create a function (`measure_read_performance`) that reads all files in a specified folder (for example, the `formats` folder). The function will evaluate each file extension (for instance, *.feather*, *.orc*, *.json*, *.csv*, *.parquet*) to determine which pandas read function to use. The function will then capture the performance time for each file format, append the results, and return a DataFrame containing all results sorted by read time.

```
import pandas as pd
import os
import glob
import time

def measure_read_performance(folder_path):

    performance_data = []
    for file_path in glob.glob(f'{folder_path}/*'):
        _, ext = os.path.splitext(file_path)
        start_time = time.time()

        if ext == '.csv':
            pd.read_csv(file_path, low_memory=False)
        elif ext == '.parquet':
            pd.read_parquet(file_path)
        elif ext == '.feather':
            pd.read_feather(file_path)
        elif ext == '.orc':
            pd.read_orc(file_path)
        elif ext == '.json':
            pd.read_json(file_path)

        end_time = time.time()
        performance_data.append({
            'filename': file_path,
            'read_time': end_time - start_time
        })

    df = pd.DataFrame(performance_data)

    return df.sort_values('read_time').reset_index(drop=True)
```

You can execute the function by specifying the folder – in this case, the `formats` folder – to display the final results:

```
results = measure_read_performance(folder_path='formats')
print(results)
>>
                                      filename  read_time
0            formats/yellow_tripdata_lz4.parquet   0.070845
1         formats/yellow_tripdata_snappy.parquet   0.072083
2           formats/yellow_tripdata_zstd.parquet   0.078382
3            formats/yellow_tripdata_lz4.feather   0.103172
4           formats/yellow_tripdata_zstd.feather   0.103918
5   formats/yellow_tripdata_uncompressed.feather   0.116974
6               formats/yellow_tripdata_zstd.orc   0.474430
7       formats/yellow_tripdata_uncompressed.orc   0.592284
8                formats/yellow_tripdata_lz4.orc   0.613846
9                    formats/yellow_tripdata.csv   4.557402
10                  formats/yellow_tripdata.json  14.590845
```

Overall, the results for read performance indicate that **Parquet** file formats perform the best, followed by **Feather**, then **ORC**. The time `read_time` is measured in seconds.

See also

You can learn more about file formats for efficient data storage with pandas by using the following resources:

- **Parquet**
  - pandas documentation: `https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_parquet.html`
- **Feather**
  - pandas documentation: `https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_feather.html`
  - Additional arguments in Arrow documentation: `https://arrow.apache.org/docs/python/generated/pyarrow.feather.write_feather.html`
- **ORC**
  - pandas documentation: `https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_orc.html`
  - Additional arguments in Arrow documentation: `https://arrow.apache.org/docs/python/generated/pyarrow.orc.write_table.html`
- **Apache Arrow**: `https://arrow.apache.org/overview/`