4

Store Financial Market Data on Your Computer

If there's one thing algorithmic traders cannot get enough of, it's data. The data that fuels our strategies is more than just numbers—it's the lifeblood of our decision-making processes. And having data available locally—or at least within your control—is a big part of that. Speed of access and reliability are important reasons why you might want to store data locally. Local data is insulated from internet outages, ensuring that data-driven processes remain uninterrupted. Further, if you need to update a bad price, you can persist the update through time.

In terms of price considerations, local storage offers cost-efficiency benefits over recurring cloud expenses. Storing a few terabytes of data in a cloud-based database can cost several hundred dollars per month. The flexibility of data manipulation, ease of integration with research workflows, and speeding up of backtests are other advantages.

In this chapter, we'll explore several ways to store financial market data. We'll start with storing in a CSV file, which can easily be written and read by pandas. Then we'll explore ways to store data in a simple, on-disk SQL database format called **SQLite**. We'll increase the complexity and install a **PostgreSQL** database server on your computer to store data. Finally, we'll use the highly efficient, ultra-fast **HDF5** format to store data.

For recipes using **SQLite** and **PostgreSQL**, we'll develop a script that can be run automatically using a task manager to acquire market data after the market closes.

In this chapter, we present the following recipes:

- Storing data on disk in CSV format
- Storing data on disk with SQLite
- Storing data in a networked Postgres database
- Storing data in ultra-fast HDF5 format

Storing data on disk in CSV format

The **Comma-Separated Values** (**CSV**) format is one of the most universally recognized and utilized methods for storing data. Its simplicity makes it a favored choice for traders and analysts looking to store tabular data without the overhead of more complex systems. Algorithmic traders often gravitate toward CSV when dealing with data that requires straightforward import and export operations, especially given the ease with which Python and its libraries, such as pandas, handle CSV files. Further, data in CSV format can be used with other analytics tools such as Tableau, PowerBI, or proprietary systems. Manually inspecting CSV files is also possible using a text editor or Excel. CSV does not have the same speed or sophistication as other storage methods, but its ease of use makes it important in all trading environments.

How to do it...

Since pandas supports writing data to CSV, there are no special libraries required:

1. Import the libraries:

```
import pandas as pd
from openbb import obb
obb.user.preferences.output_type = "dataframe"
```

2. Implement a function to download data, manipulate the results, and return a pandas DataFrame:

```
def get_stock_data(symbol, start_date=None, end_date=None):
    data = obb.equity.price.historical(
        symbol,
        start_date=start_date,
        end_date=end_date,
        provider="yfinance",
    )
    data.reset_index(inplace=True)
    data['symbol'] = symbol
    return data
```

3. Implement a function to save a range of data as a CSV file:

```
def save_data_range(symbol, start_date=None, end_date=None):
    data = get_stock_data(symbol, start_date, end_date)
    data.to_csv(
        f"{symbol}.gz",
        compression="gzip",
        index=False
    )
```

4. Implement a function that reads a CSV file and returns a DataFrame:

```
def get_data(symbol):
    return pd.read_csv(
        f»{symbol}.gz»,
        compression=»gzip»,
        index_col="date",
        usecols=[
            "date",
            "open",
            "high",
            "low",
            "close",
```

```
            "volume",
            "symbol"
        ]
    )
```

5. Save the data as a CSV file:

```
save_data_range("PLTR")
```

How it works…

The script uses the **pandas** library and the OpenBB Platform to download and manipulate stock market data. The **get_stock_data** function fetches stock data using a given symbol and date range, then preprocesses this data by resetting its index and standardizing column names. It also appends a stock symbol column for reference.

save_data_range uses the pandas **to_csv** method to store the data to disk using GZIP compression in the CSV format. The naming convention for saved files is the stock symbol followed by the **.gz** file extension.

The **get_data** function retrieves and decompresses the stored CSV files, reconstructing them into a DataFrame. During this process, only selected columns such as **date**, **opening price**, **high**, **low**, **closing price**, **volume**, and **stock symbol** are loaded.

The end of the script uses these functions to save stock data for the symbol PLTR. After running this code, you'll have a file on your computer called **PLTR.gz**, which is a compressed CSV file.

There's more…

The pandas **to_csv** method has many options for efficiently saving data to disk in CSV format. Here are some of the most useful:

- **`Sep`**: Specifies the delimiter to use between fields, defaulting to a comma (**`,`**). For a tab-separated file, use **`sep='\t'`**.
- **`header`**: A Boolean value determining whether to write out column names. Set to **`False`** to exclude column headers from the CSV.
- **`na_rep`**: Sets the string representation for missing (**`nan`**) values. The default is empty strings, but can be changed to placeholders such as **`NULL`**.
- **`date_format`**: Dictates the format for datetime objects. For example, **`date_format='%Y-%m-%d %H:%M:%S'`** formats datetime objects as **`2023-08-10 15:20:30`**.
- **`float_format`**: Controls the format for floating point numbers. For instance, **`float_format='%.2f'`** rounds all float columns to two decimal places.

Similarly, the **`read_csv`** method used in the **`get_data`** method has additional arguments for added flexibility in fetching data from disk:

- **`delimiter`** or **`sep`**: Specifies the character that separates fields, defaulting to a comma (**`,`**). For tab-separated files, use **`delimiter='\t'`**.
- **`nrows`**: Determines the number of rows of the file to read, which can be useful for reading in just a subset of a large file.
- **`parse_dates`**: A list of column names to parse as dates. If **`['date']`** is passed, the **`date`** column will be parsed into a **`datetime64`** type.
- **`dtype`**: Provides a dictionary of column names and data types to use for each column. For example, **`dtype={'volume': 'int32'}`** would ensure the **`volume`** column is read as a 32-bit integer.
- **`skiprows`**: Specifies a number or a list of row numbers to skip while reading the file, useful for omitting specific rows.

See also…

Writing data to CSV is a very common operation when dealing with market data. It's important to become comfortable with the different

ways you can read and write data from and to CSVs:

- Documentation for the pandas **`to_csv`** method: **https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_csv.html**
- Documentation for the pandas **`from_csv`** method: **https://pandas.pydata.org/docs/reference/api/pandas.read_csv.html**

Storing data on disk with SQLite

SQLite offers a bridge between the simplicity of flat files and the robustness of relational databases. As a serverless, self-contained database, SQLite provides algorithmic traders with a lightweight yet powerful tool to store and query data with SQL but without the complexity of setting up a full-scale database system. Its integration with Python is seamless, and its compact nature makes it an excellent choice for applications where portability and minimal configuration are priorities. For traders who require more structure than CSVs, or prefer to use SQL, but without the overhead of larger database systems, SQLite is the optimal choice.

Getting ready...

We'll build a script that can be set to run automatically using a CRON job (Mac, Linux, Unix) or Task Scheduler (Windows). For this recipe, we will create a Python script called **`market_data_sqlite.py`** and run it from the command line. We'll also introduce the **`exchange_calendars`** Python package, which is a library for defining and querying calendars for trading days and times for over 50 exchanges. You can install it with **`pip`**:

```
pip install exchange_calendars
```

How to do it...

All the following code should be written in the **market_data_sqlite.py** script file:

1. Begin by importing all necessary libraries to fetch and save data:

```
from sys import argv
import sqlite3
import pandas as pd
import exchange_calendars as xcals
from openbb import obb
obb.user.preferences.output_type = "dataframe"
```

2. Reuse the **get_stock_data** function from the previous recipe:

```
def get_stock_data(symbol, start_date=None, end_date=None):
    data = obb.equity.price.historical(
        symbol,
        start_date=start_date,
        end_date=end_date,
        provider="yfinance",
    )
    data.reset_index(inplace=True)
    data['symbol'] = symbol
    return data
```

3. Modify the **save_data_range** function to use the pandas **to_sql** method:

```
def save_data_range(symbol, conn, start_date,
   end_date):
    data = get_stock_data(symbol, start_date,
   end_date)
    data.to_sql(
        "stock_data",
        conn,
        if_exists="replace",
        index=False
    )
```

4. Create a function that grabs data from the last trading day based on the exchange's calendar:

```
def save_last_trading_session(symbol, conn, today):
    data = get_stock_data(symbol, today, today)
    data.to_sql(
        "stock_data",
        conn,
        if_exists="append",
        index=False
    )
```

5. Create the script's main execution code, which allows the user to pass in a stock symbol, start, and end date to kick off the data acquisition and storage process:

```
if __name__ == "__main__":
    conn = sqlite3.connect("market_data.sqlite")
    if argv[1] == "bulk":
        symbol = argv[2]
        start_date = argv[3]
        start_date = argv[4]
        save_data_range(symbol, conn, start_date=None,
            end_date=None)
        print(f"{symbol} saved between {
           start_date} and {start_date}")
    elif argv[1] == "last":
        symbol = argv[2]
        calendar = argv[3]
        cal = xcals.get_calendar(calendar)
        today = pd.Timestamp.today().date()
        if cal.is_session(today):
            save_last_trading_session(symbol, conn, today)
            print(f"{symbol} saved")
        else:
            print(f"{today} is not a trading day. Doing nothing.")
    else:
        print("Enter bulk or last")
```

6. To save a range of data, run the following command from your terminal:

```
python market_data_sqlite.py bulk SYMBOL START_DATE END_DATE
```

Where **`SYMBOL`** is the ticker symbol, **`START_DATE`** is the first date you want to download data for, and **`END_DATE`** is the last date you want to download data for. Here's an example:

```
python market_data_sqlite.py bulk SPY 2022-01-01 2022-10-20
```

This downloads and saves data for the SPY symbol between 2022-01-01 and 2022-10-20.

> *IMPORTANT*
>
> *The* **`if_exists`** *argument is set to* **`replace`** *in the* **`to_sql`** *method in the* **`save_data_range`** *function. That means every time you call the function, the table will be dropped and replaced if it exists. Depending on your use case, you may want to set the value to* **`append`** *if you are adding new data for different stocks at different times.*

Here's how you'd download data for the last trading day:

```
python market_data_sqlite.py last SYMBOL XNYS
```

Here, **`SYMBOL`** is the ticker symbol.

> *TIP*
>
> *To get a list of supported calendars from the* **`exchange_calendars`** *package, you can run* **`xcals.get_calendar_names(include_aliases=False)`**.

How it works…

We use the same function for fetching a range of stock price data as the last recipe. In this recipe, we modify the **`save_data_range`** function

by including an argument to accept a connection, and instead of saving to CSV, we use the pandas **`to_sql`** method to save the data in the DataFrame to an SQLite table. The table exists inside a file called **`market_data.sqlite`**, which we defined in the **`connect`** method. By calling this method, the Python **`sqlite3`** package will create the file if it does not exist, or connect to it if it does.

The **`save_last_trading_session`** method takes a connection and the date for which data is downloaded. We call the **`get_stock_data`** function we created with the current date as the start and end date. This returns one row of data for the current date. After the data is downloaded, it is appended to the SQLite table.

The code under the **`if`** statement runs when called from the command line. **`argv`** is a list provided by the **`sys`** module that captures command-line arguments passed to a script. The first item in the list (**`argv[0]`**) represents the script's name itself, and subsequent items contain the arguments in the order they were provided. We use **`argv`** to determine whether we should download a range of data or data for the last trading session. Depending on whether the user enters **`bulk`** or **`last`**, we capture the symbol, start, and end date and call the appropriate function. If the user enters **`last`**, we use the pandas **`Timestamp`** class to determine the current date, then use **`exchange_calendars`** to test whether the current date is a trading day for the given exchange. If it is, we append the last day's data. If it's not, we print a message and do nothing.

There's more...

Automating data retrieval ensures consistent and timely inputs for your trading workflows. Here's how to automate the script you just created to run at 1:00 p.m. EST daily.

Windows

Windows users can create a batch file to run the Python script:

1. Create a batch file:
    1. Create a new `.bat` file (for example, `run_script.bat`).
    2. Inside, add the following:

```
@echo off
CALL conda activate quant-stack
python path_to_your_script\market_data_sqlite.py %1 %2
```

2. Open Windows Task Scheduler:
    1. Press *Windows* + *R*, type `taskschd.msc`, and hit *Enter*.
3. Create a new task:
    1. In the **Actions** pane, click on **Create Basic Task**.
4. Provide the name and description of the task:
    1. **Name**: `Run Market Data Script`
    2. **Description**: `Runs the Python script every weekday at 11:00 pm`.
5. Set the trigger:
    1. Choose **Daily**.
    2. **Start**: Set today's date and `11:00 pm`.
    3. **Recur every**: `1 day`.
    4. Check **Weekdays** in the advanced settings.
6. Set the action:
    1. Choose **Start a program**.
    2. **Program/script**: Browse and select the `.bat` file you created.
    3. **Add arguments**: `last SPY XNYS`
7. Finish the setup:
    1. Click on **Finish**.

Mac/Unix/Linux

Mac and Unix users can create an executable shell file to run the Python script:

1. Create a shell script:

1. Create a new file called **`run_script.sh`**.
2. Inside, add the following:

```
#!/bin/bash
source /path_to_anaconda/anaconda3/bin/activate quant-stack
python /path_to_your_script/market_data_sqlite.py $1 $2
```

3. Give execute permissions:

```
chmod +x run_script.sh
```

2. Open the cron table:
   1. Open terminal.
   2. Enter **`crontab -e`**.
3. Add a cron job:
   1. To run the script at 11:00 p.m. EST on weekdays, append the following line:

```
0 23 * * 1-5 /path_to_shell_script/run_script.sh last SPY XNYS
```

> *NOTE*
>
> *The time may need adjustment for Daylight Saving or based on your server's time zone.*

4. Save and exit:
   1. Press *Ctrl* + *O* to save (if you're using nano).
   2. Press *Ctrl* + *X* to exit.
5. Verify the cron job:
   1. Enter **`crontab -l`** in the terminal to ensure your job is listed.Top of FormBottom of Form

In both cases, we assume your virtual environment is named **`quant-stack`**.

See also...

SQLite is an extremely fast SQL-compatible file format. You can use all the SQL you already know with SQLite.

- SQLite home page: **https://www.sqlite.org/index.html**
- Documentation for the pandas **`to_sql`** method: **https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_sql.html**
- Documentation for the **`exchange_calendars`** package: **https://github.com/gerrymanoim/exchange_calendars**

Storing data in a PostgreSQL database server

PostgreSQL, commonly known as **Postgres**, is an advanced open source relational database system. Its ability to handle vast datasets, coupled with intricate querying capabilities, makes it a good option for algorithmic traders who need improved performance over on-disk options. Postgres is also a popular database choice for cloud providers such as AWS, in case you need cloud storage for your data.

The scalability and robustness of Postgres are especially relevant when dealing with higher-frequency trading data or when multiple systems and strategies require concurrent access to shared data resources. While the setup may be more involved compared to other storage solutions, the benefits of centralized, networked access with stringent data integrity checks are desirable for sophisticated trading operations.

Getting ready...

To follow this recipe, you'll either need access to an existing remote Postgres database server or one installed on your computer. Follow these steps to get Postgres running on your local computer. Depending

on your operating system, you can install Postgres and its dependencies at the command line using one of the following options.

For Windows:

1. Download the Postgres installation file from the Postgres downloads page.
2. Double-click the installation file and follow the instructions.
3. Open pgAdmin from the **Start** menu under the PostgreSQL folder.

   For Debian/Ubuntu:

   Run the following command from your command line:

   ```
   sudo apt-get install libpq-dev python3-dev
   ```

   For Red Hat/CentOS/Fedora

   Run the following command from your command line:

   ```
   sudo yum install postgresql-devel python3-devel
   ```

   For macOS (using Homebrew)

   Run the following command from your terminal window:

   ```
   brew install postgresql
   ```

Once Postgres is installed, follow the instructions for your operating system (usually printed on the screen) to start the Postgres database server. After the Postgres database server is installed, use **`pip`** to install SQLAlchemy and the **`psycopg2`** driver:

```
pip install sqlalchemy psycopg2
```

We'll build a script that can be set to run automatically using a CRON job or Task Scheduler. For this recipe, you'll create a Python script called **`market_data_postgres.py`** and run it from the command line.

How to do it…

All the following code should be written in the **market_data_postgres.py** script file:

1. Import the required libraries:

```
import pandas as pd
from sqlalchemy import create_engine, text
from sqlalchemy.exc import ProgrammingError
import exchange_calendars as xcals
from openbb import obb
obb.user.preferences.output_type = "dataframe"
```

2. Implement a function that creates a database to store market data if one does not exist:

```
def create_database_and_get_engine(db_name, base_engine):
    conn = base_engine.connect()
    conn = conn.execution_options(
        isolation_level="AUTOCOMMIT")
    try:
        conn.execute(text(f"CREATE DATABASE{
            db_name};"))
    except ProgrammingError:
        pass
    finally:
        conn.close()
    conn_str = base_engine.url.set(database=db_name)
    return create_engine(conn_str)
```

3. Reuse the same **get_stock_data()** function as in the previous two recipes.
4. Slightly modify the **save_data_range** function to change the variable name **conn** to **engine** to match what is being passed:

```
def save_data_range(symbol, engine, start_date=None, end_date=None):
    data = get_stock_data(symbol, start_date,end_date)
    data.to_sql(
        "stock_data",
        engine,
        if_exists="append",
```

```
        index=False
    )
```

5. Change the function so it only saves the last trading session's data:

```
def save_last_trading_session(symbol, engine, today):
    data = get_stock_data(symbol, today, today)
    data.to_sql(
        "stock_data",
        engine,
        if_exists="append",
        index=False
    )
```

6. Create the script's main execution code, which creates the database connection and calls our Python code to download and save the data:

```
if __name__ == "__main__":
    username = ""
    password = ""
    host = "127.0.0.1"
    port = "5432"
    database = "market_data"
    DATABASE_URL = f"postgresql://{username}:{password}@
    {host}:{port}/postgres"
    base_engine = create_engine(DATABASE_URL)
    engine = create_database_and_get_engine(
        "stock_data", base_engine)
    if argv[1] == "bulk":
        symbol = argv[2]
        start_date = argv[3]
        start_date = argv[4]
        save_data_range(symbol, engine,
            start_date=None, end_date=None)
        print(f"{symbol} saved between {start_date} and {
        end_date}")
    elif argv[1] == "last":
        symbol = argv[2]
        calendar = argv[3]
        cal = xcals.get_calendar(calendar)
        today = pd.Timestamp.today().date()
```

```
        if cal.is_session(today):
            save_last_trading_session(symbol, engine, today)
            print(f"{symbol} saved")
        else:
            print(f"{today} is not a trading day. Doing
            nothing.")
```

7. To save a range of data, run the following command from your terminal:

```
python market_data_sqlite.py bulk SYMBOL START_DATE END_DATE
```

Here, **`SYMBOL`** is the ticker symbol, **`START_DATE`** is the first date you want to download data for and **`END_DATE`** is the last date you want to download data for. Here's an example:

```
python market_data_sqlite.py bulk SPY 2022-01-01 2022-10-20
```

This command downloads and saves data for ticker symbol SPY between **`2022-01-01`** and **`2022-10-20`**.

> *IMPORTANT*
>
> *By default, Postgres does not set a username and password. Depending on your operating system and installed tools (for example,* **`pgAdmin`***), the process for creating a username and password differs. It is critical that you set both of these to ensure the integrity of the data. It's also best practice to create an* **`.env`** *file that stores your credentials used within Python code, read them using the* **`dotenv`** *package, and set them from environment variables.*

How it works…

We started by importing the necessary Python libraries. In this recipe, we introduced SQLAlchemy, which provides tools to connect to and interact with databases (more on this follows).

Next, we implement a function to create a new database. If the database already exists, it simply connects to it. The `AUTOCOMMIT` isolation level is set to bypass PostgreSQL's restriction against creating databases within transaction blocks. After the database is created, the function returns the engine.

Next is a series of functions to fetch financial market data and use the pandas `to_sql` method to store the data in the Postgres database. We create two functions so we can both bulk save data for new tickers and save the last trading day's data. These functions are similar to what we built in the previous recipes except for a change in variable names.

In the main code execution block, we set up the connection parameters and create an "engine" which is the way we will connect to the Postgres database through pandas. The rest of the code operates the same as the previous recipe.

There's more...

SQLAlchemy is a powerful toolkit for interacting with databases in Python, enabling seamless communication between Python applications and relational databases. Its **Object Relational Mapping (ORM)** layer lets developers interact with databases using native Python classes, abstracting away the intricacies of raw SQL. Additionally, it is database-agnostic which means applications can be built once and then deployed across various database backends with minimal changes, ensuring flexibility and scalability. This is very useful when building a development database on your local computer and transitioning to a database on a remote server later.

See also...

For those that wish to further explore the advanced features of SQLAlchemy, there are a series of quick start guides describing how to

model databases using Python classes on the SQLAlchemy website: https://docs.sqlalchemy.org/en/20/orm/quickstart.html

- SQLAlchemy documentation page: https://docs.sqlalchemy.org/en/20/index.html
- PostgreSQL home page: https://www.postgresql.org

A great way to manage a Postgres database server is the free pgAdmin software.

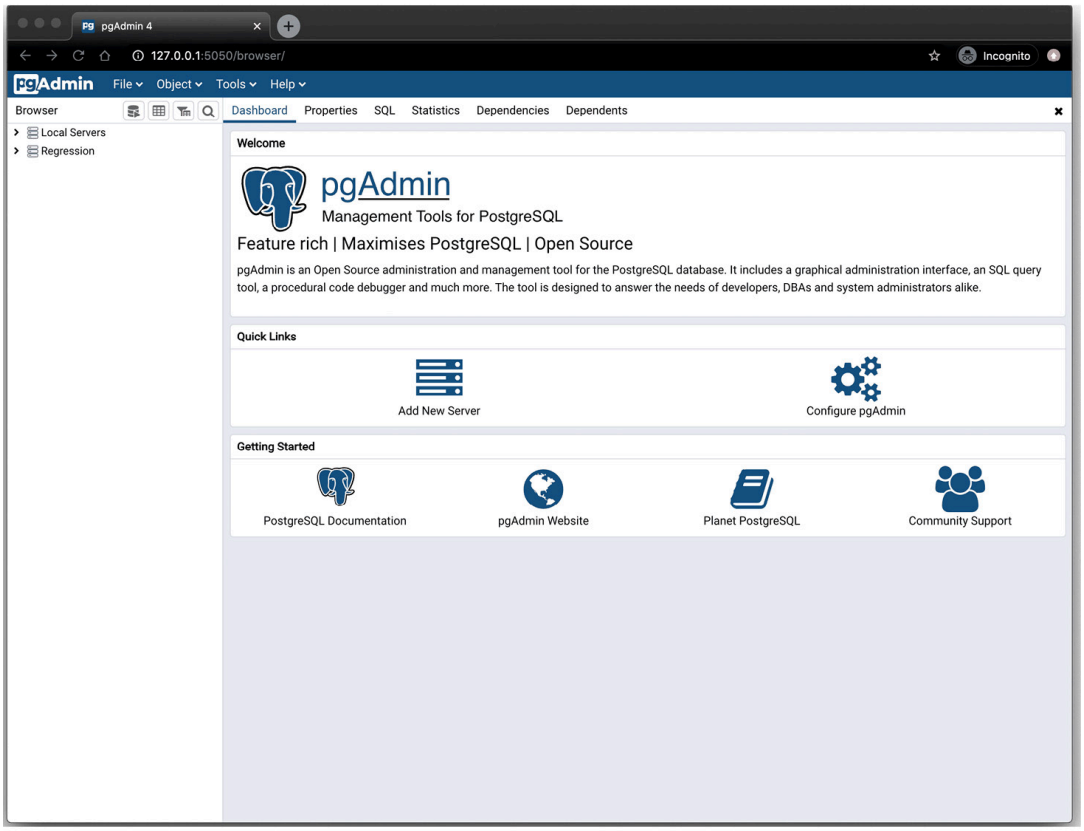

Figure 4.1: pgAdmin 4 user interface

pgAdmin provides a graphical user interface to manage server resources and write queries. To learn more and download pgAdmin, visit the following URL: https://www.pgadmin.org/.

Storing data in ultra-fast HDF5 format

Hierarchical Data Format (**HDF**) is made up of a collection of file formats, namely HDF4 and HDF5, engineered for the hierarchical storage and management of voluminous data. Initially developed at the U.S. National Center for Supercomputing Applications, HDF5 is an open source format that accommodates large and complex heterogeneous datasets. It uses a directory-like structure, enabling versatile data organization within the file, similar to file management on a computer. HDF5 has two primary object types: datasets, which are typed multidimensional arrays, and groups, which are container structures capable of holding both datasets and other groups. In Python, HDF5 is supported through two libraries: **`h5py`**, offering both high- and low-level access to HDF5 constructs, and **`PyTables`**, providing a high-level interface with advanced indexing and query capabilities. We'll use pandas to write to and read data in this recipe, which uses **`PyTables`** under the hood. **`PyTables`** is a Python library for managing large datasets and hierarchical databases using the HDF5 file format. It provides tools for efficiently storing, accessing, and processing data, making it well-suited for high-performance, data-intensive applications.

Storing data using HDF5 is advantageous when storing related data in a hierarchy. This could be different fundamental data for stocks, expirations for futures, or chains for options.

Getting ready…

To follow this recipe, you'll need **`PyTables`** installed on your computer, which pandas uses to access data in HDF5 file format.

You can install **`PyTables`** using **`conda`** like this:

```
conda install -c conda-forge pytables
```

How to do it…

For this recipe, we'll move back to our Jupyter notebook:

1. Import the necessary libraries and set up the variables:

```
import pandas as pd
from openbb import obb
obb.user.preferences.output_type = "dataframe"
STOCK_DATA_STORE = "stocks.h5"
FUTURES_DATA_STORE = "futures.h5"
ticker = "SPY"
root = "CL"
```

2. Load data for **`SPY`** price and options chains using the OpenBB Platform:

```
spy_equity = obb.equity.price.historical(
    ticker,
    start_date="2021-01-01",
    provider="yfinance"
)
spy_chains = obb.derivatives.options.chains(
    ticker,
    provider="cboe"
)
spy_expirations = (
    spy_chains
    .expiration
    .astype(str)
    .unique()
    .tolist()
)
spy_historic = (
    obb
    .equity
    .price
    .historical(
        ticker + spy_expirations[-10].replace("-", "")[2:] + "C"
        + "00400000",
        start_date="2021-01-01",
        provider="yfinance"
    )
)
```

3. Store the data in the HDF5 file:

```
with pd.HDFStore(STOCKS_DATA_STORE) as store:
    store.put("equities/spy/stock_prices", spy_equity)
    store.put("equities/spy/options_prices",
        spy_historic)
    store.put("equities/spy/chains", spy_chains)
```

4. Read the data from the HDF5 file into pandas DataFrames:

```
with pd.HDFStore(STOCKS_DATA_STORE) as store:
    spy_prices = store["equities/spy/stock_prices"]
    spy_options = store["equities/spy/options_prices"]
    spy_chains = store["equities/spy/chains"]
```

5. Now iterate through e-mini futures expirations, storing the historical data for each one in a different file path:

```
with pd.HDFStore(FUTURES_DATA_STORE) as store:
    for i in range(24, 31):
        expiry = f"20{i}-12"
        df = obb.derivatives.futures.historical(
            symbol=[root],
            expiry=expiry,
            start_date="2021-01-01",
        )
        df.rename(
            columns={
                "close": expiry
            },
            inplace=True
        )
        prices = df[expiry]
        store.put(f'futures/{root}/{expiry}', prices)
```

6. Read the data the same as with the ETF:

```
with pd.HDFStore(FUTURES_DATA_STORE) as store:
    es_prices = store[f"futures/{root}/2023-12"]
```

How it works...

We first download historic price data and options chains for the SPY ETF using the OpenBB Platform. We extract the expirations then use `obb.equity.price.historical` to construct an options ticker symbol to request historic data. With this data stored in pandas DataFrames, we open the `assets.h5` file using the pandas `HDFStore` method. The Python `with` statement creates a context that allows you to run a group of statements under the control of a context manager. Here, we open the `assets.h5` file as a pandas `HDFStore` object. `HDFStore` has a method called `put`, which allows us to easily store the data in the DataFrame in the HDF5 file.

The futures example shows how you can iterate through a list of data sources, saving each one as a separate path within the HDF5 file. We start by opening the HDF5 file and iterating through a list of expiration dates. For each expiration date, we use the OpenBB Platform to download price data, rename the columns, and put the data into the HDF5 file.

There's more...

HDF5 is considered one of the fastest on-disk, columnar data storage formats for strictly numerical data. This format also generally shares the smallest memory footprint with compressed CSV format. Another file format is Parquet, which is a binary, columnar storage format that provides efficient data compression and encoding. It's also available through pandas using the PyArrow library under the hood.

See also...

To learn more about HDF, visit the documentation here:

- Documentation for pandas `HDFStore`: **https://pandas.pydata.org/docs/reference/api/pandas.HDFStore.put.html**
- Documentation for PyTables: **https://www.pytables.org**

- Documentation for the Python bindings of PyArrow: **https://arrow.apache.org/docs/python/index.html**.