13

Advanced Recipes for Market Data and Strategy Management

This final chapter covers advanced recipes to stream and store options data, generate risk alerts, and store key strategy information to automate end-of-day reporting. We will start with a deep dive into real-time data handling using **Theta Data**. ThetaData is a data service that specializes in providing real-time options data. It offers a comprehensive stream of unfiltered options market data, including quotes, trades, volumes, and Greeks. With ThetaData, we can combine contracts to price complex options positions in real time. This service is an option for algorithmic traders who want to research and develop complex trading strategies using options contracts. After streaming the data, we will introduce advanced data management storage using ArcticDB. ArcticDB is an open source project built by the systematic strategy manager **Man Group** and is designed to store petabytes of data in DataFrame format.

In ***Chapter 12***, *Deploy Strategies to a Live Environment*, we built a series of risk and performance metrics. In this chapter, we'll design an alerting system that will send an email if a defined risk level is breached. This approach is popular in professional trading businesses where traders must adhere to defined risk limits. We'll then introduce recipes to store key strategy information in the SQL database we created in ***Chapter 10***, *Set Up the Interactive Brokers API,* automating end-of-day strategy management processes.

By the end of the chapter, you will have the tools to handle and analyze real-time data, manage risk more carefully, and maintain detailed trade records. All techniques that are instrumental for algorithmic traders.

In this chapter, we will present the following recipes:

- Streaming real-time options data with ThetaData
- Using the ArcticDB DataFrame database for tick storage
- Triggering real-time risk limit alerts
- Storing trade execution details in a SQL database

Streaming real-time options data with ThetaData

The **Options Price Reporting Authority** (**OPRA**) functions as a securities information processor, aggregating options quotes and transaction details from predominant U.S. exchanges. Approximately 1.4 million active options contracts are traded, generating in excess of 3 terabytes of data on a daily basis. OPRA is responsible for the real-time consolidation and dissemination of this data. ThetaData, through its connection to OPRA, facilitates the distribution of this data in an unfiltered format to non-professional users. Furthermore, ThetaData's Python API is capable of streaming quotes and trades with a latency measured in milliseconds. This efficiency is achieved by compressing the data to approximately 1/30th of its original volume.

The Theta Terminal is an intermediate layer that bridges our data-providing server with the Python API. The terminal runs as a background process. It hosts a local server on your machine, to which the Python API connects. Primarily, it simplifies data access and processing by handling complex tasks such as forwarding requests to the appropriate server and interpreting responses. Additionally, separating data processing activities like decompression from API-specific features enhances efficiency and usability for traders using the service. This recipe will demonstrate how to use ThetaData for streaming options data.

Getting ready

To use ThetaData, you'll need Java installed on your computer. Java usually comes pre-installed but in case it's not, you can find the installation process for your computer at https://www.java.com/en/.

To install the ThetaData Python library, use **pip**:

```
pip install thetadata
```

How to do it…

We'll demonstrate how to stream real-time trade data for all options contracts, as well as for a single options contract:

1. Import the libraries we need to set up the streaming data:

```
import datetime as dt
import thetadata.client
from thetadata import (
    Quote,
    StreamMsg,
    ThetaClient,
    OptionRight,
    StreamMsgType,
    StreamResponseType
)
```

2. Implement the callback that responds to each trade message. In this example, we will simply print information about the contract, trade, and quote:

```
def callback(msg):
    if msg.type == StreamMsgType.TRADE:
        print(
            "------------------------------------------------"
        )
        print(f"Contract: {msg.contract.to_string()}")
        print(f"Trade: {msg.trade.to_string()}")
        print(f"Last quote at time of trade: {
            msg.quote.to_string()}")
```

3. Implement the function that connects to the ThetaData client, registers the callback, and starts the options data stream (replace **YOURPASSWORD**

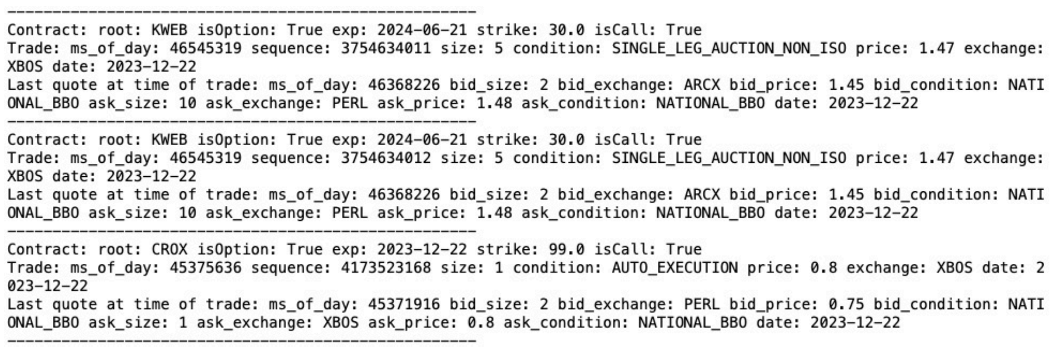
with your ThetaData login credentials):

```
def stream_all_trades():
    client = ThetaClient(
        username="strimp101@gmail.com",
        passwd="YOURPASSWORD"
    )
    client.connect_stream(
        callback
    )
    req_id = client.req_full_trade_stream_opt()
    response = client.verify(req_id)
    if (
        client.verify(req_id) != StreamResponseType.SUBSCRIBED
    ):
        raise Exception("Unable to stream.")
```

4. Start the streaming data:

```
stream_all_trades()
```

After ThetaData connects, you'll start to see options trades print to the screen:



```
-----------------------------------------------------
Contract: root: KWEB isOption: True exp: 2024-06-21 strike: 30.0 isCall: True
Trade: ms_of_day: 46545319 sequence: 3754634011 size: 5 condition: SINGLE_LEG_AUCTION_NON_ISO price: 1.47 exchange:
XBOS date: 2023-12-22
Last quote at time of trade: ms_of_day: 46368226 bid_size: 2 bid_exchange: ARCX bid_price: 1.45 bid_condition: NATI
ONAL_BBO ask_size: 10 ask_exchange: PERL ask_price: 1.48 ask_condition: NATIONAL_BBO date: 2023-12-22
-----------------------------------------------------
Contract: root: KWEB isOption: True exp: 2024-06-21 strike: 30.0 isCall: True
Trade: ms_of_day: 46545319 sequence: 3754634012 size: 5 condition: SINGLE_LEG_AUCTION_NON_ISO price: 1.47 exchange:
XBOS date: 2023-12-22
Last quote at time of trade: ms_of_day: 46368226 bid_size: 2 bid_exchange: ARCX bid_price: 1.45 bid_condition: NATI
ONAL_BBO ask_size: 10 ask_exchange: PERL ask_price: 1.48 ask_condition: NATIONAL_BBO date: 2023-12-22
-----------------------------------------------------
Contract: root: CROX isOption: True exp: 2023-12-22 strike: 99.0 isCall: True
Trade: ms_of_day: 45375636 sequence: 4173523168 size: 1 condition: AUTO_EXECUTION price: 0.8 exchange: XBOS date: 2
023-12-22
Last quote at time of trade: ms_of_day: 45371916 bid_size: 2 bid_exchange: PERL bid_price: 0.75 bid_condition: NATI
ONAL_BBO ask_size: 1 ask_exchange: XBOS ask_price: 0.8 ask_condition: NATIONAL_BBO date: 2023-12-22
-----------------------------------------------------
```

Figure 13.1: The firehose of options trades

5. Define an active options contract. In this case, we are focused on **`SPY`** options with a $474 strike expiring on December 12, 2023:

```
ticker = "SPY"
expiration_date = dt.date(2023, 12, 22)
strike = 474
```

6. Implement the function that connects to the ThetaData client, registers the callback, and starts the options data stream for the specific con-

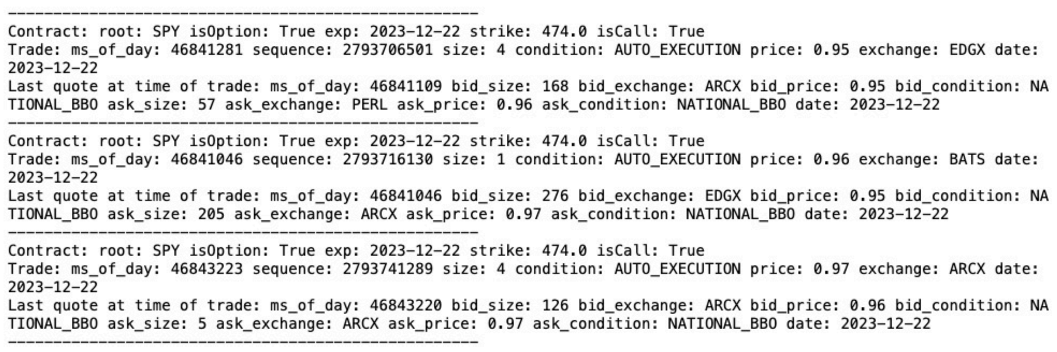
tract we define (replace **`youremail@example.com`** and **`YOURPASSWORD`** with your ThetaData login credentials):

```
def stream_contract():
    client = ThetaClient(
        username="youremail@example.com",
        passwd=" YOURPASSWORD "
    )
    client.connect_stream(callback)
    req_id = client.req_trade_stream_opt(
        ticker, expiration_date, strike, OptionRight.CALL)
    response = client.verify(req_id)
    if (
        client.verify(req_id) != StreamResponseType.SUBSCRIBED
    ):
        raise Exception("Unable to stream.")
```

7. Start the streaming data:

```
stream_contract()
```

After ThetaData connects, you'll start to see options trades print to the screen for the contract you specified:



```
----------------------------------------------------
Contract: root: SPY isOption: True exp: 2023-12-22 strike: 474.0 isCall: True
Trade: ms_of_day: 46841281 sequence: 2793706501 size: 4 condition: AUTO_EXECUTION price: 0.95 exchange: EDGX date:
2023-12-22
Last quote at time of trade: ms_of_day: 46841109 bid_size: 168 bid_exchange: ARCX bid_price: 0.95 bid_condition: NA
TIONAL_BBO ask_size: 57 ask_exchange: PERL ask_price: 0.96 ask_condition: NATIONAL_BBO date: 2023-12-22
----------------------------------------------------
Contract: root: SPY isOption: True exp: 2023-12-22 strike: 474.0 isCall: True
Trade: ms_of_day: 46841046 sequence: 2793716130 size: 1 condition: AUTO_EXECUTION price: 0.96 exchange: BATS date:
2023-12-22
Last quote at time of trade: ms_of_day: 46841046 bid_size: 276 bid_exchange: EDGX bid_price: 0.95 bid_condition: NA
TIONAL_BBO ask_size: 205 ask_exchange: ARCX ask_price: 0.97 ask_condition: NATIONAL_BBO date: 2023-12-22
----------------------------------------------------
Contract: root: SPY isOption: True exp: 2023-12-22 strike: 474.0 isCall: True
Trade: ms_of_day: 46843223 sequence: 2793741289 size: 4 condition: AUTO_EXECUTION price: 0.97 exchange: ARCX date:
2023-12-22
Last quote at time of trade: ms_of_day: 46843220 bid_size: 126 bid_exchange: ARCX bid_price: 0.96 bid_condition: NA
TIONAL_BBO ask_size: 5 ask_exchange: ARCX ask_price: 0.97 ask_condition: NATIONAL_BBO date: 2023-12-22
----------------------------------------------------
```

Figure 13.2: Options trade data for a specific contract

How it works…

The Theta Terminal serves as an intermediary layer, facilitating the connection between our data server and the Python API. Operating as a background process, it establishes a local server on the user's machine, which is then accessed by the Python API. When a request is initiated via the Python API, the process is as follows: the API first relays the request to the Theta Terminal. The Terminal, in turn, forwards the request to the nearest ThetaData **Market Data Distribution Server** (**MDDS**) and awaits a response. Upon receiving the response, the Terminal processes and sends the data to the user's Python application. The Python API subsequently parses the response into a format that is more accessible to the end user. This approach of using an intermediate application offers numerous advantages, the most notable being the segregation of data processing tasks (such as decompression) from the functionalities specific to the language-based API.

We must first define a callback function designed to handle the streaming trade messages that come from the Theta Terminal. The function checks whether the incoming message is a **`TRADE`** constant. When a trade message is received, the function prints details of the contract and trade, as well as the last quote at the time of the trade. These details are obtained by calling the **`to_string`** method on the respective attributes of the **`msg`** object.

The **`stream_all_trades`** and **`stream_contract`** functions initiate streaming connections to receive real-time trade data. They begin by creating a **`ThetaClient`** instance with specified user credentials. The client then connects to a streaming service using the **`connect_stream`** method, which uses a callback function for handling incoming data. A request for a full trade stream is made using **`req_full_trade_stream_opt`** or a single contract with **`req_trade_stream_opt`**. The response is verified. If the verification fails, it prints a message to the screen.

There's more…

We demonstrated how to stream real-time data for single contracts. Now we'll demonstrate how to combine quote data from multiple contracts to generate real-time quotes straddles and iron condors:

1. Create empty **`Quote`** objects and set a price:

```
last_call_quote = Quote()
last_put_quote = Quote()
price = 0
```

2. Implement a callback that creates the straddle price out of the bid and ask prices of the contracts that make up the straddle:

```
def callback_straddle(msg):
    if (msg.type != StreamMsgType.QUOTE):
        return
    global price
    if msg.contract.isCall:
        last_call_quote.copy_from(msg.quote)
    else:
        last_put_quote.copy_from(msg.quote)
    straddle_bid = round(last_call_quote.bid_price + last_put_
        quote.bid_price, 2)
    straddle_ask = round(last_call_quote.ask_price + last_put_
        quote.ask_price, 2)
    straddle_mid = round((
        straddle_bid + straddle_ask) / 2, 2)
    time_stamp = thetadata.client.ms_to_time(
        msg.quote.ms_of_day
    )
    if price != straddle_mid:
        print(
            f"time: {time_stamp} bid: {straddle_bid} mid:
                {straddle_mid} ask: {straddle_ask}"
        )
        price = straddle_mid
```

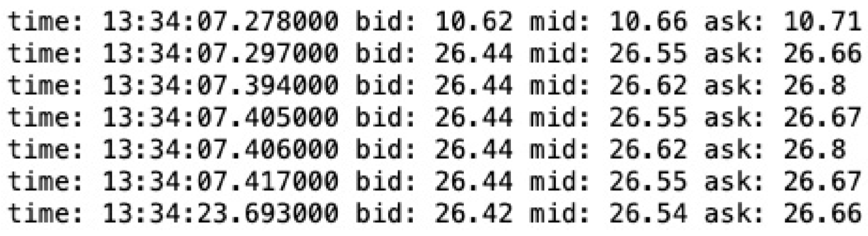
3. Implement the function that starts the streaming data for each of our defined contracts:

```
def stream_straddle():
    client = ThetaClient(
        username="strimp101@gmail.com",
        passwd="kdk_fzu6pyb0UZA-yuz"
    )
    client.connect_stream(
        callback_straddle
    )
    req_id_call = client.req_quote_stream_opt(
        "SPY", dt.date(2024, 3, 28), 475, OptionRight.CALL
    )
    req_id_put = client.req_quote_stream_opt(
        "SPY", dt.date(2024,3,28),475,OptionRight.PUT
    )
    if (
        client.verify(req_id_call) != StreamResponseType.
        SUBSCRIBED
        or client.verify(req_id_put) != StreamResponseType.
        SUBSCRIBED
        ):
        raise Exception("Unable to stream.")
```

4. Start the streaming data:

```
stream_straddle()
```

After ThetaData connects, you'll start to see the computed bid, mid, and ask prices for the straddle print to the screen for the contracts that you specified:



```
time: 13:34:07.278000 bid: 10.62 mid: 10.66 ask: 10.71
time: 13:34:07.297000 bid: 26.44 mid: 26.55 ask: 26.66
time: 13:34:07.394000 bid: 26.44 mid: 26.62 ask: 26.8
time: 13:34:07.405000 bid: 26.44 mid: 26.55 ask: 26.67
time: 13:34:07.406000 bid: 26.44 mid: 26.62 ask: 26.8
time: 13:34:07.417000 bid: 26.44 mid: 26.55 ask: 26.67
time: 13:34:23.693000 bid: 26.42 mid: 26.54 ask: 26.66
```

Figure 13.3: Streaming straddle prices

Now let's implement streaming data for a short iron condor. In *Chatper 11*, *Deploy Strategies to a Live Environment*, we learned that a short iron condor is an options strategy that involves selling a lower strike out-of-the-money put, buying an even lower strike out-of-the-money put, selling a higher strike out-of-the-money call, and buying an even higher strike out-of-the-money call with the same expiration date.

5. Define the ticker, expiration date, and strike prices that make up the iron condor:

```
ticker = "SPY"
expiration_date = dt.date(2024, 3, 28)
long_put_strike = 460
short_put_strike = 465
short_call_strike = 480
long_call_strike = 485
```

6. Create empty **Quote** objects and set a price:

```
long_put = Quote()
short_put = Quote()
short_call = Quote()
```

```
long_call = Quote()
price = 0
```

7. Implement the callback that captures quote data from each of the four options contracts, computes the bid, mid, and ask prices, and prints the result:

```
def callback_iron_condor(msg):
    if (msg.type != StreamMsgType.QUOTE):
        return
    global price
    if not msg.contract.isCall and msg.contract.strike ==
    long_put_strike:
        long_put.copy_from(msg.quote)
    elif not msg.contract.isCall and msg.contract.strike ==
    short_put_strike:
        short_put.copy_from(msg.quote)
    elif msg.contract.isCall and msg.contract.strike ==
    short_call_strike:
        short_call.copy_from(msg.quote)
    elif msg.contract.isCall and msg.contract.strike ==
    long_call_strike:
        long_call.copy_from(msg.quote)
    condor_bid = round(
        long_put.bid_price
        - short_put.bid_price
        + long_call.bid_price
        - short_call.bid_price,
        2,
    )
    condor_ask = round(
        long_put.ask_price
        - short_put.ask_price
        + long_call.ask_price
        - short_call.ask_price,
        2,
    )
    condor_mid = round((condor_ask + condor_bid) / 2, 2)
    time_stamp = thetadata.client.ms_to_time(
        msg.quote.ms_of_day
    )
    if price != condor_mid:
        print(
            f"time: {time_stamp} bid: {condor_bid} mid:
                {condor_mid} ask: {condor_ask}"
        )
        price = condor_mid
```

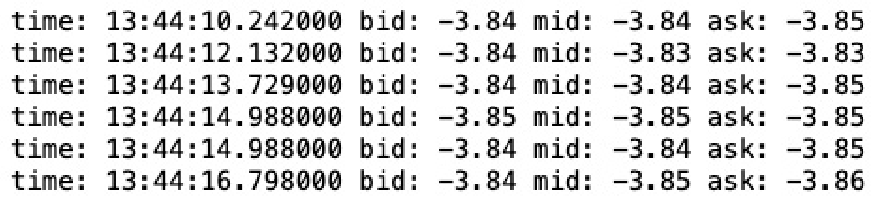
8. Implement the function that starts the streaming data for each of our defined contracts:

```
def streaming_iron_condor():
    client = ThetaClient(
        username="strimp101@gmail.com",
        passwd="YOURPASSWORD"
    )
    client.connect_stream(callback_iron_condor)
    lp_id = client.req_quote_stream_opt(
        ticker, expiration_date, long_put_strike,
            OptionRight.PUT
    )
    sp_id = client.req_quote_stream_opt(
        ticker, expiration_date, short_put_strike,
            OptionRight.PUT
    )
    client.req_quote_stream_opt(
        ticker, expiration_date, short_call_strike,
            OptionRight.CALL
    )
    client.req_quote_stream_opt(
        ticker, expiration_date, long_call_strike, O
            ptionRight.CALL
    )
    if (
        client.verify(lp_id) != StreamResponseType.SUBSCRIBED
        or client.verify(sp_id) != StreamResponseType.SUBSCRIBED
    ):
        raise Exception("Unable to stream.")
```

9. Start the streaming data:

```
stream_iron_condor()
```

After ThetaData connects, you'll start to see the computed bid, mid, and ask prices for the iron condor print to the screen for the contracts that you specified:



```
time: 13:44:10.242000 bid: -3.84 mid: -3.84 ask: -3.85
time: 13:44:12.132000 bid: -3.84 mid: -3.83 ask: -3.83
time: 13:44:13.729000 bid: -3.84 mid: -3.84 ask: -3.85
time: 13:44:14.988000 bid: -3.85 mid: -3.85 ask: -3.85
time: 13:44:14.988000 bid: -3.84 mid: -3.84 ask: -3.85
time: 13:44:16.798000 bid: -3.84 mid: -3.85 ask: -3.86
```

Figure 13.4: Streaming bid, mid, and ask prices for a short iron condor

> *IMPORTANT NOTE*

> *The prices for the short iron condor are negative. That's because when we go short an iron condor, we are collecting the premium offered by the combined contracts. For example, if we see a price of* **`-3.85`** *bid,* **`-3.83`** *ask, that means if we sell the condor, we will sell at the ask and collect $3.83.*

See also

To learn more about OPRA and options price dissemination, see this URL: **https://www.opraplan.com/**

Otherwise, check out the ThetaData documentation for more details on how you can use the service at **https://thetadata-api.github.io/thetadata-python/reference/**.

Using the ArcticDB DataFrame database for tick storage

ArcticDB is an embedded, serverless database engine, tailored for integration with pandas and the Python data science ecosystem. It's used for the storage, retrieval, and processing of petabyte-scale data in DataFrame format. It uses common object storage solutions such as S3-compatible storage systems and Azure Blob Storage or local storage. It can efficiently store a 20-year historical record of over 400,000 distinct securities under a single symbol with sub-second retrieval. In ArcticDB, each symbol is treated as an independent entity without data overlap. The engine operates independently of any additional infrastructure, requiring only a functional Python environment and object storage access.

ArcticDB was built by Man Group and has demonstrated its capacity for enterprise-level deployment in some of the world's foremost organizations. The library is slated for integration into Bloomberg's **BQuant** platform, which will empower algorithmic traders to rapidly test, deploy, and share models for alpha generation, risk management, and trading.

Some of the reasons why ArcticDB is a great fit for algorithmic trading include the following:

- ArcticDB offers remarkable processing speed, capable of handling billions of on-disk rows per second. Additionally, its installation process is straightforward and quick with no complex dependencies.
- ArcticDB accommodates both schema-based and schema-less data and is fully equipped to handle streaming data ingestion. It is a bitemporal platform, providing access to all historical versions of the stored data.
- Designed with simplicity in mind, ArcticDB is intuitively accessible to those with experience in Python and pandas, positioning itself as one of the simplest databases in the world.

In this recipe, we'll demonstrate how to use ArcticDB to store streaming trade data from ThetaData.

Getting ready

As of the time of authoring this book, Linux and Windows users can install ArcticDB with `pip`:

```
pip install arcticdb
```

For macOS, use `conda`:

```
conda install -c conda-forge arcticdb
```

For this recipe, we will assume that you have built the streaming options data solution using ThetaData in the *Streaming real-time options data with ThetaData* recipe in this chapter. If you have not, do it now.

How to do it…

We'll set up the ThetaData callback to create a DataFrame for each trade and store it in ArcticDB:

1. Import the libraries that we'll need for the streaming data use case:

```
import time
import pytz
import datetime as dt
import pandas as pd
import arcticdb as adb
import thetadata.client
from thetadata import (
    ThetaClient,
    OptionRight,
    StreamMsg,
    StreamMsgType,
    StreamResponseType,
)
```

2. Create a locally hosted **Lightning Memory-Mapped Database** (**LMDB**) instance in the current directory and set up an ArcticDB library to store the streaming options data:

```
arctic = adb.Arctic("lmdb://arcticdb_options")
lib = arctic.get_library("trades",
    create_if_missing=True)
```

3. Create a helper function that returns a **`datetime`** object for the time-stamp of the trade:

```
def get_trade_datetime(today, ms_of_day):
    return today + dt.timedelta(
        milliseconds=ms_of_day)
```

4. Create a helper function that computes the number of days to expiration:

```
def get_days_to_expiration(today, expiration):
    return (expiration - today).days
```

5. Implement the callback function that parses the trade message from ThetaData and stores it in ArcticDB:

```
def callback(msg):
    today = dt.datetime.now(
        pytz.timezone("US/Eastern")
    ).replace(
        hour=0,
        minute=0,
        second=0,
        microsecond=0
    )
    if msg.type == StreamMsgType.TRADE:
        trade_datetime = get_trade_datetime(today,
            msg.trade.ms_of_day)
        expiration = pd.to_datetime(
            msg.contract.exp).tz_localize("US/Eastern")
        days_to_expiration = get_days_to_expiration(
            today, expiration)
        symbol = msg.contract.root
        trade = {
            "root": symbol,
            "expiration": expiration,
            "days_to_expiration": days_to_expiration,
            "is_call": msg.contract.isCall,
            "strike": msg.contract.strike,
            "size": msg.trade.size,
            "trade_price": msg.trade.price,
            "exchange": str(
                msg.trade.exchange.value[1]),
            "bid_size": msg.quote.bid_size,
            "bid_price": msg.quote.bid_price,
            "ask_size": msg.quote.ask_size,
            "ask_price": msg.quote.ask_price,
        }
        trade_df = pd.DataFrame(
           trade, index=[trade_datetime])
        if symbol in lib.list_symbols():
            lib.update(symbol, trade_df, upsert=True)
        else:
            lib.write(symbol, trade_df)
```

6. Implement the function that connects to ThetaData and starts the stream. We’ll let it run for 120 seconds before canceling the stream:

```
def stream_all_trades():
    client = ThetaClient(
        username="strimp101@gmail.com",
        passwd="kdk_fzu6pyb0UZA-yuz"
    )
    client.connect_stream(callback)
    req_id = client.req_full_trade_stream_opt()
    response = client.verify(req_id)
    if (client.verify(req_id) != StreamResponseType.SUBSCRIBED):
        raise Exception("Unable to stream.")
    time.sleep(120)
    print("Cancelling stream...")
    client.remove_full_trade_stream_opt()
```

7. Start the stream:

```
stream_all_trades()
```

8. After the stream has been canceled, we can interact with the data stored in ArcticDB. First, defragment the data stored in LMDB if required:

```
for symbol in lib.list_symbols():
    if lib.is_symbol_fragmented(symbol):
        lib.defragment_symbol_data(symbol)
```

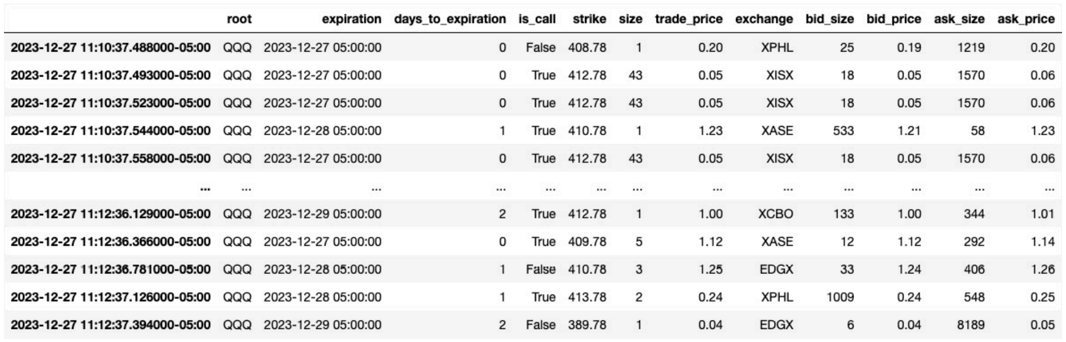
9. List the number of symbols that we have acquired trade data for:

```
len(lib.list_symbols())
```

10. Get a DataFrame containing the trade data for a single symbol:

```
qqq = lib.read("QQQ").data
```

The result is a pandas DataFrame with one row for every trade:



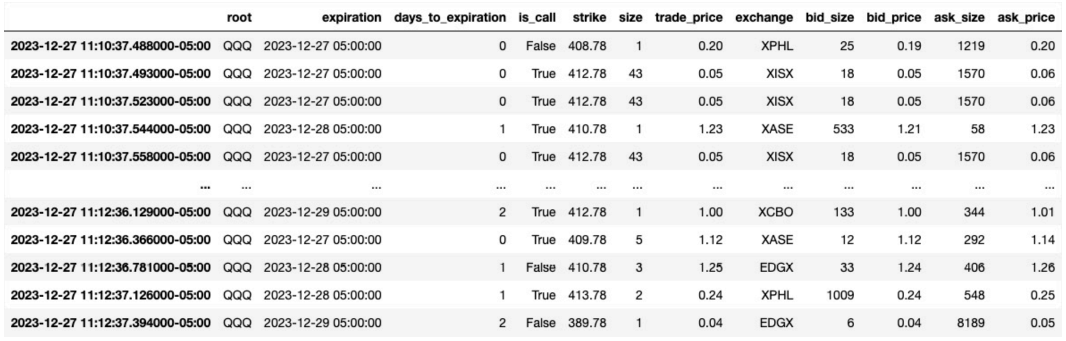
	root	expiration	days_to_expiration	is_call	strike	size	trade_price	exchange	bid_size	bid_price	ask_size	ask_price
2023-12-27 11:10:37.488000-05:00	QQQ	2023-12-27 05:00:00	0	False	408.78	1	0.20	XPHL	25	0.19	1219	0.20
2023-12-27 11:10:37.493000-05:00	QQQ	2023-12-27 05:00:00	0	True	412.78	43	0.05	XISX	18	0.05	1570	0.06
2023-12-27 11:10:37.523000-05:00	QQQ	2023-12-27 05:00:00	0	True	412.78	43	0.05	XISX	18	0.05	1570	0.06
2023-12-27 11:10:37.544000-05:00	QQQ	2023-12-28 05:00:00	1	True	410.78	1	1.23	XASE	533	1.21	58	1.23
2023-12-27 11:10:37.558000-05:00	QQQ	2023-12-27 05:00:00	0	True	412.78	43	0.05	XISX	18	0.05	1570	0.06
...	...	...	...	...	...	...	...	...	...	...	...	...
2023-12-27 11:12:36.129000-05:00	QQQ	2023-12-29 05:00:00	2	True	412.78	1	1.00	XCBO	133	1.00	344	1.01
2023-12-27 11:12:36.366000-05:00	QQQ	2023-12-27 05:00:00	0	True	409.78	5	1.12	XASE	12	1.12	292	1.14
2023-12-27 11:12:36.781000-05:00	QQQ	2023-12-28 05:00:00	1	False	410.78	3	1.25	EDGX	33	1.24	406	1.26
2023-12-27 11:12:37.126000-05:00	QQQ	2023-12-28 05:00:00	1	True	413.78	2	0.24	XPHL	1009	0.24	548	0.25
2023-12-27 11:12:37.394000-05:00	QQQ	2023-12-29 05:00:00	2	False	389.78	1	0.04	EDGX	6	0.04	8189	0.05

Figure 13.5: A pandas DataFrame with trade data for ETF QQQ

11. We can use the powerful QueryBuilder tool to process the data before acquiring it, which speeds up retrieval. Find all trades where the spread is less than $0.05:

```
q = adb.QueryBuilder()
filter = (q.ask_price - q.bid_price) < 0.05
q = q[filter]
data = lib.read("QQQ", query_builder=q).data
```

    The result is a pandas DataFrame with all trades that were executed when the spread was less than $0.05.



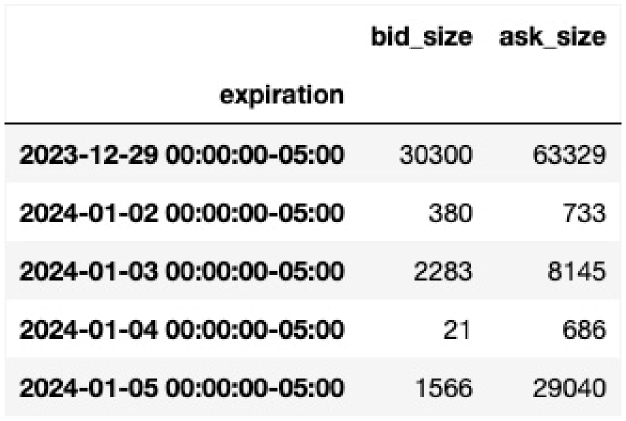
	root	expiration	days_to_expiration	is_call	strike	size	trade_price	exchange	bid_size	bid_price	ask_size	ask_price
2023-12-27 11:10:37.488000-05:00	QQQ	2023-12-27 05:00:00	0	False	408.78	1	0.20	XPHL	25	0.19	1219	0.20
2023-12-27 11:10:37.493000-05:00	QQQ	2023-12-27 05:00:00	0	True	412.78	43	0.05	XISX	18	0.05	1570	0.06
2023-12-27 11:10:37.523000-05:00	QQQ	2023-12-27 05:00:00	0	True	412.78	43	0.05	XISX	18	0.05	1570	0.06
2023-12-27 11:10:37.544000-05:00	QQQ	2023-12-28 05:00:00	1	True	410.78	1	1.23	XASE	533	1.21	58	1.23
2023-12-27 11:10:37.558000-05:00	QQQ	2023-12-27 05:00:00	0	True	412.78	43	0.05	XISX	18	0.05	1570	0.06
...	...	...	...	...	...	...	...	...	...	...	...	...
2023-12-27 11:12:36.129000-05:00	QQQ	2023-12-29 05:00:00	2	True	412.78	1	1.00	XCBO	133	1.00	344	1.01
2023-12-27 11:12:36.366000-05:00	QQQ	2023-12-27 05:00:00	0	True	409.78	5	1.12	XASE	12	1.12	292	1.14
2023-12-27 11:12:36.781000-05:00	QQQ	2023-12-28 05:00:00	1	False	410.78	3	1.25	EDGX	33	1.24	406	1.26
2023-12-27 11:12:37.126000-05:00	QQQ	2023-12-28 05:00:00	1	True	413.78	2	0.24	XPHL	1009	0.24	548	0.25
2023-12-27 11:12:37.394000-05:00	QQQ	2023-12-29 05:00:00	2	False	389.78	1	0.04	EDGX	6	0.04	8189	0.05

Figure 13.6: A pandas DataFrame with filtered trade data for ETF QQQ

12. Use QueryBuilder and native pandas methods to efficiently retrieve and post-process data:

```
q = adb.QueryBuilder()
filter = (q.days_to_expiration > 1)
q = (
    q[filter]
    .groupby("expiration")
    .agg({"bid_size": "sum", "ask_size": "sum"})
)
data = lib.read("QQQ", query_builder=q).data.sort_index()
```

The result is a pandas DataFrame with a snapshot of the aggregate bid and ask sizes at each expiration at the time of trade for all contracts with greater than 1 day to expiration.



	bid_size	ask_size
expiration		
2023-12-29 00:00:00-05:00	30300	63329
2024-01-02 00:00:00-05:00	380	733
2024-01-03 00:00:00-05:00	2283	8145
2024-01-04 00:00:00-05:00	21	686
2024-01-05 00:00:00-05:00	1566	29040

Figure 13.7: A pandas DataFrame with filtered, grouped, and aggregated bid and ask volumes across a range of strike prices

How it works…

We started by creating an instance of the **`Arctic`** class, which is initialized with the **`lmdb://arcticdb_options`** connection string indicating that the ArcticDB database will use LMDB as its storage engine. Then we call the **`get_library`** method on the **`Arctic`** instance. This method attempts to retrieve a library named **`trades`** from the underlying LMDB database. If the library does not exist, the **`create_if_missing=True`** argument ensures that it is created. The **`get_library`** method connects to the underlying storage engine to manage the data associated with the **`trades`** library.

NOTE

LMDB is a high-performance embedded key-value store designed to provide efficient, transactional data storage mechanisms for applications. Built on a memory-mapped file, LMDB allows for data to be quickly read and written with minimal overhead, supporting a substantial number of simultaneous read operations along with a single write transaction. Its architecture is optimized for speed, concurrency, and reliability, making it a suitable choice for applications requiring fast access to large datasets with a small footprint. LMDB operates with a no-overhead approach and can handle databases much larger than RAM, leveraging the operating system's virtual memory to manage data transactions effectively.

The **`get_trade_datetime`** function calculates and returns the datetime of a trade by adding a specific number of milliseconds to a given date.

ThetaData uses the number of milliseconds since the beginning of the trade day as its timestamp, so this function gives us a more human-readable version. The **`get_days_to_expiration`** function first calculates a **`Timedelta`** object between the expiration date and today's date and returns the number of days between the two.

Storing data

The **`callback`** function processes incoming trade messages by first setting a **`today`** datetime object to the current date with the time set to the start of the day in the U.S. Eastern time zone. If the message type is a trade, it computes the trade datetime by adding the number of milliseconds since midnight. The function then calculates the days remaining until the contract's expiration. From there, we construct a dictionary with trade details. These details include information about the contract and the trade itself such as the symbol, expiration, days to expiration, option type, strike price, size, and the current bid and ask sizes and prices. This trade information is then formatted into a pandas DataFrame with the trade datetime as its index. Depending on whether the symbol already exists in the DataFrame, the trade data is either appended to the existing symbol data or written as new data under the symbol. The **`stream_all_trades`** function is similar to the one we used in the previous recipe except that we cancel the stream after 120 seconds.

Retrieving data

We demonstrate how to use the **`QueryBuilder`** object to retrieve data. The first line, **`q = adb.QueryBuilder()`**, creates an instance of the **`QueryBuilder`** class, which is used for constructing database queries that are passed to the ArcticDB processing engine. The second line, **`filter = (q.ask_price - q.bid_price) < 0.05`**, defines a filter condition whereby only records with a difference between **`ask_price`** and **`bid_price`** of less than **`0.05`** are returned. This filter is applied in the third line, **`q = q[filter]`**, which modifies the **`QueryBuilder`** instance to include this filtering criterion. Finally, **`data = lib.read("QQQ", query_builder=q).data`** uses the **`read`** method of the **`lib`** object to execute the query against the **QQQ** dataset, retrieving records that match the filter. The **data** attribute accesses the actual data from the query result.

In the second example, we construct a **QueryBuilder** object to select records where **days_to_expiration** is greater than **1**. The query is then refined to group the results by the **expiration** field and aggregate the **bid_size** and **ask_size** fields by summing them up. Finally, the **read** method is used to execute this query against the **QQQ** dataset in the LMDB datastore. The resulting data is sorted by index.

There's more…

We demonstrated how to store data locally using LMDB. We can achieve a similar performance using remote object storage backends. At the time of writing, ArcticDB has tested the following S3 backends:

- AWS S3
- Ceph
- MinIO on Linux
- Pure Storage S3
- Scality S3
- VAST Data S3

To use ArcticDB with S3, you simply need an S3 bucket with PUT permissions. Then you can instantiate the Arctic class with its path. Use this example if your AWS **credentials** file is stored locally:

```
Arctic("s3://MY_ENDPOINT:MY_BUCKET?aws_auth=true")
```

Otherwise, you can specify the credentials in the connection string:

```
Arctic("s3://MY_ENDPOINT:MY_BUCKET?region=YOUR_REGION&access=ABCD&secret=DCBA"
```

See also

For more information on ArcticDB, see the following resources:

- The ArcticDB homepage with documentation, community information, and blog: https://arcticdb.io/
- Detailed documentation with walkthroughs: https://docs.arcticdb.io/latest/
- The ArcticDB GitHub page: https://github.com/man-group/ArcticDB
- LMDB documentation and homepage: http://www.lmdb.tech/doc/

Triggering real-time risk limit alerts

Once a strategy is in place, algorithmic trading revolves around managing performance and risk metrics, emphasizing monitoring exceptions or deviations that exceed predefined thresholds. Risk metrics are critical indicators that flag potential issues or opportunities in the trading strategy. Professional algorithmic traders rely heavily on real-time alert systems. These systems detect and notify traders when specific risk metrics reach or surpass set thresholds. This notification enables traders to respond to market changes, adjust strategies, or mitigate risks. These alerts can be based on various risk metrics, such as **Conditional Values at Risk (CVaRs)**, drawdowns, or unusual trading volumes. Effectively managing these alerts and understanding the underlying causes is a cornerstone of successful algorithmic trading.

In this recipe, you'll set up real-time alerts to monitor our portfolio's intraday CVaR. In ***Chapter 12***, *Deploy Strategies to a Live Environment*, we learned that CVaR measures the expected losses that occur beyond the **Value at Risk (VaR)** in the tail end of a distribution of possible returns. Our focus will be on setting up the infrastructure for alerting. The decision of how to send alerts via email, SMS, Slack, Telegram bot, or other channels is left up to the reader.

Getting ready

For this recipe, we assume that you have built the real-time risk and performance metrics detailed in ***Chapter 12***, *Deploy Strategies to a Live*

Environment. We also assume that you've followed the recipes in *Chapter 12*. If you have not, do it now.

How to do it...

We'll edit the **app.py** file to include a new thread to check whether the portfolio CVaR exceeds a defined threshold. Since we've already set up the app's scaffolding, adding a new method to a background thread requires only a few lines of code:

1. Add a **watch_cvar** method at the end of the **IBApp** class in **app.py**:

```
def watch_cvar(self, threshold, interval):
    print("Watching CVaR in 60 seconds...")
    time.sleep(60)
    while True:
        cvar = self.cvar[1]
        if cvar < threshold:
            print(f"Portfolio CVaR ({
                cvar}) crossed threshold ({
                    threshold})")
            pass
        time.sleep(interval)
```

2. Modify the **__init__** method to resemble the following:

```
def __init__(self, ip, port, client_id, account,
    interval=5, **kwargs):
        IBWrapper.__init__(self)
        IBClient.__init__(self, wrapper=self)
        self.account = account
        self.create_table()
        self.connect(ip, port, client_id)
        threading.Thread(target=self.run,
            daemon=True).start()
        time.sleep(5)
        threading.Thread(
           target=self.get_streaming_returns,
           args=(99, interval, "unrealized_pnl"),
            daemon=True
        ).start()
        time.sleep(5)
        threading.Thread(
            target=self.watch_cvar,
            args=(kwargs["cvar_threshold"], interval),
            daemon=True
        ).start()
```

3. To run the app, modify the line under **`if __name__ == "__main__":`** to resemble the following:

```
app = IBApp(
    "127.0.0.1",
    7497,
    client_id=12,
    account="DU7129120",
    interval=10,
    cvar_threshold=-500
)
```

How it works…

The method monitors the CVaR of our portfolio against a specified threshold at regular intervals. Initially, it prints a message and pauses for 60 seconds. Then, in an infinite loop, it checks whether the dollar CVaR falls below the given threshold. If it does, **`watch_cvar`** prints a message indicating that the CVaR has crossed this threshold. After each check, the method pauses for a duration specified by the **`interval`** parameter before repeating the process. In this method, we can implement our actual alerting code.

To start the process of monitoring the intraday CVaR, we follow the same pattern that we introduced in previous chapters. We create and start a new thread that executes the **`watch_cvar`** method. The thread runs independently, passing the specified **`cvar_threshold`** and **`interval`** values from **`kwargs`** as arguments to the **`watch_cvar`** method. Running the method in a thread allows us to execute other code with the monitoring in the background.

There's more…

Depending on your use case, you may choose to send alerts via email or SMS. Short snippets are included to get you started with each. Depending on your email provider, you may have to take steps to enable sending emails through code. For example, Gmail stopped supporting logins through "less secure" means in 2022 and now requires the OAuth2 authorization framework. You can learn more at https://developers.google.com/gmail/api/quickstart/python.

Sending emails using Python

Python has two built-in libraries that make it easy to send simple emails. We'll demonstrate a simple example of how to send emails with Python that you can use for your alerts:

1. Include the following imports at the top of **app.py**:

```
<snip>
import smtplib
from email.message import EmailMessage
s = smtplib.SMTP("localhost")
```

2. Update the **watch_cvar** method to include the following code to construct and send an email:

```
def watch_cvar(self, threshold, interval):
    print("Watching CVaR in 60 seconds...")
    time.sleep(60)
    while True:
        cvar = self.cvar[1]
        if cvar < threshold:
            print(f"Portfolio CVaR ({
                cvar}) crossed threshold ({
                    threshold})")
            msg = EmailMessage()
            msg.set_content(
                """
            Risk alert:
            f"Portfolio CVaR ({
                cvar}) crossed threshold ({
                    threshold})"
            """
            )
```

```
                msg["Subject"] = "CVaR threshold crossed"
                msg["From"] = "sender@email.com"
                msg["To"] = "you@email.com"
                s.send_message(msg)
                s.quit()
            time.sleep(interval)
```

First, we create an **`EmailMessage`** object and connect it to an email server. In a production application, you would replace **`localhost`** with the server details of your email provider. You would also need to include the provider's authentication code. Within the **`watch_cvar`** method, we must check whether CVaR exceeds our threshold and construct our email. The email's subject, sender, and recipient are then specified. Next, we send the prepared email message and close the connection.

IMPORTANT NOTE

There are several paid services that offer bulk email sending for very little cost. **Mailgun** *is one that offers thousands of emails per month for a reasonable cost. If you don't expect to send hundreds or thousands of emails per month, consider using your existing email provider. Depending on your provider, instructions for authentication and sending emails will differ.*

Sending alerts via SMS

There are many third-party services that we can use to send SMS via Python. We'll look at the most popular one, **Twilio**. Twilio's APIs provide seamless interaction with users through voice, SMS, video, or messaging. Thousands of businesses use Twilio for sending SMS notifications such as password changes or informational alerts. Twilio operates on a pay-as-you-use pricing model.

To start, install the Twilio library with **`pip`**:

```
pip install twilio
```

We'll demonstrate a simple example of how to send SMS with Python using the Twilio API that you can use for your alerts.

1. Include the following imports at the top of **app.py**:

```
<snip>
from twilio.rest import Client
account_sid = TWILIO_ACCOUNT_SID
auth_token = TWILIO_AUTH_TOKEN
client = Client(account_sid, auth_token)
```

2. Update the **watch_cvar** method to include the following code to construct and send an SMS:

```
def watch_cvar(self, threshold, interval):
    print("Watching CVaR in 60 seconds...")
    time.sleep(60)
    while True:
        cvar = self.cvar[1]
        if cvar < threshold:
            print(f"Portfolio CVaR ({
                cvar}) crossed threshold ({
                    threshold})")
            body = """
            Risk alert:
            f"Portfolio CVaR ({
                cvar}) crossed threshold ({
                    threshold})"
            """
            message = client.messages.create(
                body, from_=FROM_NUMBER, to=TO_NUMBER
            )
        time.sleep(interval)
```

Replace **TWILIO_ACCOUNT_SID** and **TWILIO_AUTH_TOKEN** with the account ID and auth token you can retrieve from the Twilio console. Then replace **FROM_NUMBER** with the number you purchased and **TO_NUMBER** with your mobile number.

See also

To learn more about sending emails and SMS alerts, see the following resources:

- Mailgun homepage : https://www.mailgun.com/
- Twilio homepage: https://www.twilio.com/en-us
- **Amazon Simple Notification Service** (**SNS**): https://aws.amazon.com/sns/

Storing trade execution details in a SQL database

Capturing trade data is essential in algorithmic trading. Recording order execution data provides the ability to monitor the time interval between the placement and fulfillment of orders. This aspect is particularly important in volatile markets where prices are moving quickly, since it allows us to evaluate and enhance our execution strategies for better efficiency. Another benefit is measuring slippage – the difference between the expected and actual execution prices. Analyzing slippage is important for understanding the impact of execution on trading costs and profitability. Since transaction costs, both broker fees and slippage, erode returns, minimizing costs can have a real impact on returns. Tracking open orders can help us assess current market exposure but also ensures adherence to risk management processes and helps in balancing our portfolio.

As we've learned, the IB API facilitates this process through its EWrapper callback methods. These methods are called in response to events such as order status changes, order executions, and open orders. The callback methods offer a streamlined and event-driven approach for capturing order-related data. In this recipe, we'll use the three order callback methods we established in ***Chapter 10***, *Set up the Interactive Brokers Python API*, to capture order details.

Getting ready

For this recipe, we assume that you have set up the code to establish the SQLite database in ***Chapter 10***, *Set up the Interactive Brokers Python API*, and the methods to acquire portfolio data in *Chapter 11*, *Manage Orders, Positions, and Portfolios with the IB API*. If you have not, do it now.

We'll do some cleanup to make our code more organized. In the **`IBApp`** class, find the **`create_table`** method. In the method, there's a long line of SQL we use to create the table to store bid and ask data. We'll move that to the **`utils.py`** file.

1. Cut the SQL from the **`create_table`** method and paste it into **`utils.py`** similar to the following:

```
CREATE_BID_ASK_DATA = """
CREATE TABLE IF NOT EXISTS bid_ask_data
    (
        timestamp DATETIME,
        symbol STRING,
        bid_price REAL,
        ask_price REAL,
        bid_size INTEGER,
        ask_size INTEGER
    )"""
```

2. Now in **`app.py`**, add the following import under the existing imports:

```
from utils import CREATE_BID_ASK_DATA
```

3. Modify **`create_table`** to resemble the following:

```
def create_table(self):
    cursor = self.connection.cursor()
    cursor.execute(CREATE_BID_ASK_DATA)
```

4. Update the name of the SQLite file being created to the name of the strategy since now we're storing more than just bid and ask data:

```
@property
def connection(self):
    return sqlite3.connect("strategy_1.sqlite",
        isolation_level=None)
```

How to do it...

We'll add three new SQL statements to create tables to store order status, open order details, and executed trades.

1. In **`utils.py`**, add the following statement under the previous declaration to create an SQLite table to store open order details:

   ```
   CREATE_OPEN_ORDERS = """
   CREATE TABLE IF NOT EXISTS open_orders
       (
           timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
           order_id INTEGER,
           symbol STRING,
           sec_type STRING,
           exhange STRING,
           action STRING,
           order_type STRING,
           quantity INTEGER,
           status STRING
       )"""
   ```

2. In **`utils.py`**, add the following statement under the previous declaration to create an SQLite table to store order execution details:

   ```
   CREATE_TRADES = """
   CREATE TABLE IF NOT EXISTS trades
       (
           timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
           request_id INTEGER,
           order_id INTEGER,
           execution_id INTEGER,
           symbol STRING,
           sec_type STRING,
           currency STRING,
           quantity INTEGER,
           last_liquidity REAL
       )"""
   ```

3. In **`app.py`**, update the import for the bid ask data to include all imports:

   ```
   from utils import (
       CREATE_BID_ASK_DATA,
       CREATE_OPEN_ORDERS,
   ```

```
    CREATE_TRADES
)
```

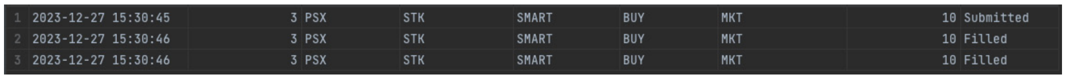
4. Modify **`create_table`** in the **`IBApp`** class in **`app.py`** to resemble the following:

```
def create_table(self):
    cursor = self.connection.cursor()
    cursor.execute(CREATE_BID_ASK_DATA)
    cursor.execute(CREATE_OPEN_ORDERS)
    cursor.execute(CREATE_TRADES)
```

5. In **`wrapper.py`**, update the **`openOrder`** method to generate and execute a SQL insert statement when the callback is triggered. Replace the code that simply prints a message with the following:

```
def openOrder(self, order_id, contract, order, order_state):
    cursor = self.connection.cursor()
    query = "INSERT INTO open_orders (
        order_id, symbol, sec_type, exhange, action,
        order_type, quantity, status)
        VALUES (?, ?, ?, ?, ?, ?, ?, ?)"
    values = (
        order_id,
        contract.symbol,
        contract.secType,
        contract.exchange,
        order.action,
        order.orderType,
        order.totalQuantity,
        order_state.status,
    )
    cursor.execute(query, values)
```

When submitting an order, we will end up with an entry in the SQLite **`open_orders`** table.



2023-12-27 15:30:45	3	PSX	STK	SMART	BUY	MKT	10 Submitted
2023-12-27 15:30:46	3	PSX	STK	SMART	BUY	MKT	10 Filled
2023-12-27 15:30:46	3	PSX	STK	SMART	BUY	MKT	10 Filled

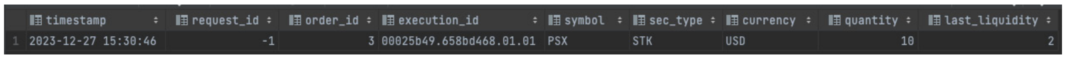
Figure 13.8: Entry in the open_orders table executed from the openOrder callback

6. In **`wrapper.py`**, update the **`execDetails`** method to generate and execute a SQL insert statement when the callback is triggered. Replace the

code that simply prints a message with the following:

```
def execDetails(self, request_id, contract,
    execution):
        cursor = self.connection.cursor()
        query = "INSERT INTO trades (request_id,
            symbol, sec_type, currency, execution_id,
            order_id, quantity, last_liquidity)
            VALUES (?, ?, ?, ?, ?, ?, ?, ?)"
    values = (
        request_id,
        contract.symbol,
        contract.secType,
        contract.currency,
        execution.execId,
        execution.orderId,
        execution.shares,
        execution.lastLiquidity,
    )
    cursor.execute(query, values)
```

When an order is executed, we will end up with an entry in the SQLite **trades** table.



	timestamp	request_id	order_id	execution_id	symbol	sec_type	currency	quantity	last_liquidity
1	2023-12-27 15:30:46	-1	3	00025b49.658bd468.01.01	PSX	STK	USD	10	2

Figure 13.9: Entry in the trades table executed from the execDetails callback

How it works…

We extend the **create_table** method to automatically create additional tables to store or trade status and execution data. These tables will be created when the **IBApp** class is instantiated. For the **openOrder** and **execDetail** methods, SQL insert statements are created when they are triggered. The methods specify the columns to be populated and the corresponding values. These values are then executed against the query to insert the data into the database. The question mark in the SQL statement serves as a placeholder for parameters that will be substituted with actual values when the query is executed.

There's more…

We demonstrated a small sample of the fields that can be captured in **execDetails**. Both callback methods receive objects that contain many properties.

Contract

The **contract** object has the following attributes that can be stored in your tables:

- **conId**: A unique identifier
- **symbol**: An asset symbol
- **secType**: Type (stock, option, etc.)
- **lastTradeDateOrContractMonth**: The last trading day or month
- **strike**: An option's strike price
- **right**: Used with a call or put option
- **multiplier**: An options or futures multiplier
- **exchange**: A contract's exchange
- **currency**: An asset's currency
- **localSymbol**: A local exchange symbol
- **primaryExch**: The primary exchange
- **tradingClass**: The contract trading class
- **includeExpired**: Includes expired futures
- **secIdType**: The identifier type (ISIN, CUSIP)
- **secId**: A security identifier
- **description**: The contract description
- **issuerId**: An issuer identifier
- **comboLegsDescription**: Combo legs details
- **comboLegs**: Combined contract legs
- **deltaNeutralContract**: Delta neutral details

Order

The **order** object has 66 properties available. Here we capture some of the more valuable ones that that can be stored in your tables:

- **`orderId`**: A client's order ID
- **`clientId`**: Placing a client ID
- **`permId`**: The host order identifier
- **`action`**: The side (**`BUY`**, **`SELL`**, etc.)
- **`totalQuantity`**: The positions number
- **`orderType`**: The order's type
- **`lmtPrice`**: **`LIMIT`** order price
- **`auxPrice`**: Stop the price for **`STP LMT`**
- **`tif`**: Time in force (**`DAY`**, **`GTC`**, etc.)

Order state

The **`order_state`** object has the following attributes that can be stored in your tables:

- **`status`**: An order's current status
- **`initMarginBefore`**: The initial margin before
- **`maintMarginBefore`**: The maintenance margin before
- **`equityWithLoanBefore`**: Account equity with loan before
- **`initMarginChange`**: The initial margin change
- **`maintMarginChange`**: The maintenance margin change
- **`equityWithLoanChange`**: The equity with loan change
- **`initMarginAfter`**: The initial margin after an order
- **`maintMarginAfter`**: The maintenance margin after an order
- **`equityWithLoanAfter`**: The equity with loan after an order
- **`commission`**: The generated commission
- **`minCommission`**: The minimum commission
- **`maxCommission`**: The maximum commission
- **`commissionCurrency`**: The commission currency
- **`warningText`**: An order warning message
- **`completedTime`**: The order completion time
- **`completedStatus`**: The order completion status

Executions

The execution object has the following attributes that can be stored in your tables:

- `orderId`: The client's order ID
- `clientId`: The placing client ID
- `execId`: The execution identifier
- `time`: The execution server time
- `acctNumber`: The allocated account number
- `exchange`: The execution exchange
- `side`: The transaction side (`BOT`, `SLD`)
- `shares`: Number of shares filled
- `price`: The execution price
- `permId`: The TWS order identifier
- `liquidation`: The IB liquidation indicator
- `cumQty`: The cumulative quantity
- `avgPrice`: The average price
- `orderRef`: A user-customizable string
- `evRule`: The **Economic Value** (**EV**) rule
- `evMultiplier`: The EV price change unit
- `modelCode`: The model code
- `lastLiquidity`: The execution liquidity type

See also

For more details on the various IB API callbacks and objects used in this recipe, see the following resources:

- Documentation on the `openOrder` EWrapper method: https://interactivebrokers.github.io/tws-api/interfaceIBApi_1_1EWrapper.html#aa05258f1d005accd3efc0d60bc151407
- Documentation on the `execDetails` EWrapper method: https://interactivebrokers.github.io/tws-api/interfaceIBApi_1_1EWrapper.html#a09f82de3d0666d13b00b5168e8b9313d
- Documentation on the `Contract` object: https://interactivebrokers.github.io/tws-api/classIBApi_1_1Contract.html

- Documentation on the `Order` object: https://interactivebrokers.github.io/tws-api/classIBApi_1_1Order.html
- Documentation on the `OrderState` object: https://interactivebrokers.github.io/tws-api/classIBApi_1_1OrderState.html
- Documentation on the `Execution` object: https://interactivebrokers.github.io/tws-api/classIBApi_1_1Execution.html