Mario Emmanuel Follow

Jan 26, 2019 · 10 min read · · Listen

How to store financial market data for backtesting

I am working on moderately large financial price data sets. By moderately large I mean less than 4 million rows per asset.

4 million rows can cover the last 20 years of minute price bars done by a regular asset without extended trading hours — such as index futures contracts or regular cash stocks — .

data is really expensive to obtain, manage and monetize; and unless you are backtesting scalping strategies or working in the HFT industry their advantage would be dubious.

Although 4 million rows do not sound impressive, we need to understand that it is 4 million rows *per asset*. So to analyze all assets from *Russell 2000* would mean 8 billion rows (now we are getting large). Add some European stock markets, your local country small and mid caps, commodities and *forex* and there you go: you just landed on the big data arena.

> *There are more than 1700 regulated markets in the world, as soon as you start stockpiling intraday data the numbers become impressive — very fast — .*

Bear in mind also that current trend is to extend all future contracts trading hours — *Eurex* began extended trading hours for *FDAX* futures since last January — , so the situation is not getting better in terms of the amount of data that needs to be analyzed.

Now the question that every "*financial data scientist*" makes to himself: where and how do I put my data.

A relational database is the first answer, probably not the most efficient, but the easiest one for sure.

I can enumerate four options as main repository strategy:

1. SQL relational databases.
2. Serialized storage of large arrays.
3. Key/Value databases (such as *Oracle Berkeley DB*).
4. CSV files.

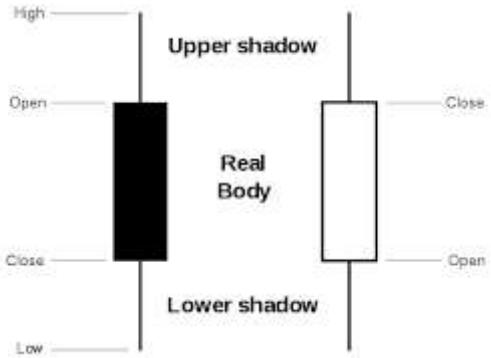
For obvious reasons I have discarded the last one, and for the other three I have evaluated just the first two ones.


believe that it might be the best trade-off solution between serialized storage and relational databases.

In this post, I focus on the first strategy. Before going into details we will revisit what data we are dealing with.

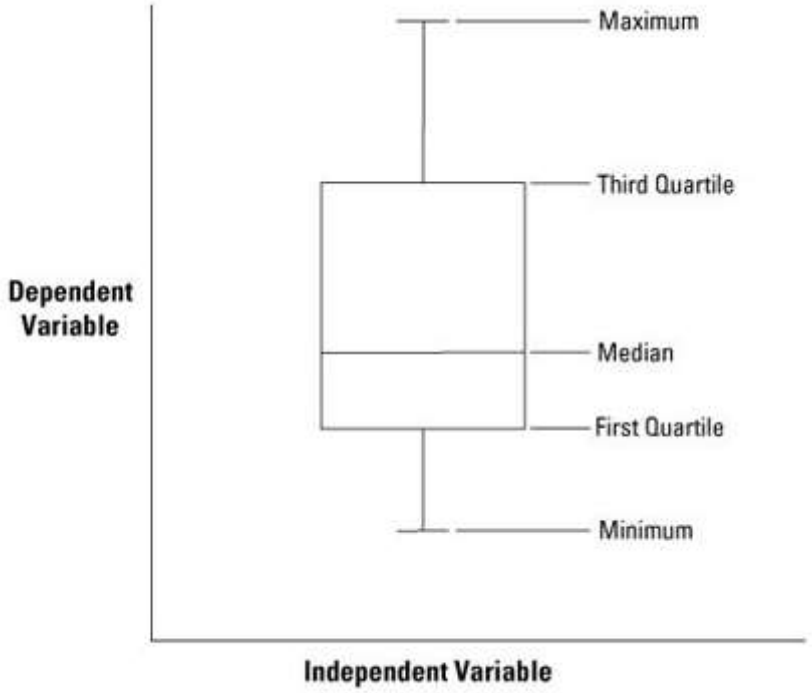
A quick historical review on candlesticks: why do we use them

Almost all data you are going to deal in financial markets will be price bars or candlesticks. There is a reason for this. The high/low/open/close structure of a financial price bar resembles the classic whiskers and box plots used in statistics but they are easier to obtain.



Price bar used in financial markets.

I have used the word *resemble* because all the important information provided by a whiskers and box bar (quartiles and median) is not available in a financial candlestick. Quartiles and medians are not arbitrary values, they provide descriptive information about the variable under analysis.



Whiskers and box bar used in statistics.

One might wonder why whisker and box bars are not used in financial markets (I think that its use might lead to useful and innovative price action insight), and the reason for that is simple. Candlestick charts are thought to have been used first during the 18th century by a Japanese rice trader named Munehisa Homma. If you are wondering if there were future contracts in the 18th century you would like to know that although the first modern future contract exchange was the *Chicago Board of Trade* there were established future markets in Europe and Japan in the 17th century.

The Dōjima Rice Exchange Monument. First future contracts exchange in Japan.

Getting the maximum, minimum, opening and close prices to build a candlestick for a given period is extremely easy. You just need the list of all trades during a certain period. Now trading is electronic and takes place in an ordered manner, but bear in mind that it has not been always the case. Electronic trading is relatively modern (it started to gain relevance during the 80s). Before that, trading pits were the norm and assets were traded in the pit just at certain hours. The notion of a continuous trading market as we know it was not applicable at that time.

Even in the current case when almost all trading takes place electronically, storing and analyzing all trades that take place in a given period to build whiskers and box chart is challenging (doable, but challenging). It implies either a strict continuous track of the order book — *reading the tape* — , or the less strict way of consolidating shorter timeframes into larger ones.

By using candlesticks instead, you end up with information that will lead you to *something similar* to quartiles and median (please notice that *something similar* is highlighted). It is what is called the price action of candlesticks or the candlesticks patterns, which, in reality, is just an interpretative analysis on where the price was relevant among all

quickly rejected. That is the kind of conclusions you can get out of a candlestick (again, when reading it within a given market context).

Storing interval price information for open/close/low/high will, therefore, enable a certain degree of price action analysis, and according to experience, certain patterns within a given context have statistical relevance and will lead to profitable strategies.

The data model for storing candlesticks

Now we understand why candlesticks are used, we can propose a simple data model to store our information.

```
CREATE TABLE candlestick (
 “id” INTEGER PRIMARY KEY AUTOINCREMENT,
 “timezone” TEXT NOT NULL,
 “timestamp” DATETIME NOT NULL,
 “open” DECIMAL(12, 6) NOT NULL,
 “high” DECIMAL(12, 6) NOT NULL,
 “low” DECIMAL(12, 6) NOT NULL,
 “close” DECIMAL(12, 6) NOT NULL,
 “volume” DECIMAL(12, 6) NOT NULL
);
```

I am sure you are not impressed with this simple data model, but even this one can be challenged. We will not discuss the `timezone` column (I plan to write another post on time zones and data), but the usage of `timestamp` is already relevant and have performance issues.

From a usability point of view, this way of storing financial information is simple and well structured. Querying data is really simple:

```
sqlite> select * from candlestick where date(timestamp)='2018-01-12'
limit 10;
```

```
3489461|Europe/Berlin|2018-01-12
07:03:00+01:00|13242.5|13243.5|13239|13241.5|64
3489462|Europe/Berlin|2018-01-12
07:04:00+01:00|13241|13241|13235.5|13236.5|61
3489463|Europe/Berlin|2018-01-12
07:05:00+01:00|13236.5|13240|13236|13239.5|49
3489464|Europe/Berlin|2018-01-12
07:06:00+01:00|13239|13241.5|13237|13240|49
3489465|Europe/Berlin|2018-01-12
07:07:00+01:00|13238.5|13241|13237|13239|43
3489466|Europe/Berlin|2018-01-12
07:08:00+01:00|13239|13239|13236.5|13237|24
3489467|Europe/Berlin|2018-01-12
07:09:00+01:00|13237|13239|13237|13239|11
```

Getting the opening price of a given session is simple:

```
sqlite> select * from candlestick where date(timestamp)='2018-01-12'
limit 1;

3489458|Europe/Berlin|2018-01-12
07:00:00+01:00|13243.5|13245.5|13234.5|13245.5|294
```

So it is getting the closing price:

```
sqlite> select * from candlestick where date(timestamp)='2018-01-11'
order by timestamp desc limit 1;

3489457|Europe/Berlin|2018-01-11
21:03:00+01:00|13241|13241|13241|13241|25
```

But this way has some performance issues we will discuss later.

Note that we have used `datetime` field to store the timestamp. We could also use separate fields for year, month, day, hour and minute, but it would complicate dealing with


Performance issues without optimization

Using this approach, there are performance issues:

1. When doing the bulk load of the data (using Python and an ORM to ensure that timestamps are properly handled in SQLite) it takes 14 minutes on a 1 core VPS server.
2. Getting all data for a given session takes 5 seconds on the same VPS server. A similar result for getting the opening or closing price.

14 minutes sounds like too much but bulk loads are not common and it might be acceptable. Even if a more optimized way to load data can be found it is not an important point. Just grab a cup of coffee or program batch loads for the initial data provisioning, that is going to happen only once.

On the contrary, 5 seconds might not sound like a big deal but it is. Remember that we are dealing here with multi-asset Monte Carlo simulation scenarios. That means thousands of rounds. So 5 seconds by thousands is way too much time. We need to optimize here.

Although I am not going to cover the serialized strategy in this post, I will share some time comparison about both strategies:

```
# Bulk provisioning of 3.5Million price bars:

SQLite + Pyhon ORM: 15 min
Serialized Stored Arrays + ANSI C: 1.5 min

# Retrieve a given specific 1 minute price bar:

SQLite + Pyhon ORM: 5 seconds
Serialized Stored Arrays + ANSI C: Negligible (microseconds)
```

As stated before, the real issue in these environments is the retrieval time. In serialized arrays, it can be as low as microseconds because a strategy can be defined to allocate


Getting the closing or opening price for a given session might be a bit more challenging if we do not know the trading hours (even if we know them, there might be exceptions like trading halts or nonconformities in data), but we can then traverse a specific day. Traversing the array for all daily data is fast and easy.

> *We might overcomplicate the analysis introducing how databases with built-in optimization capabilities can speed up these figures. An Oracle database will perform better —if your project can allocate the cost of 150000$/year for a DBA optimization specialist — , but the take away here is that a simple lookup into an array (the serialized approach) will never be beaten by any relational database. As a drawback, a relational database makes querying and moving data much easier than storing serialized arrays.*

I use SQLite every time it is possible because it is extremely simple and easy to use and backup. Despite many people state that it is a *too basic* database it can handle huge large sets. Its major drawback is its lack of page or area locking — which leads to performance issues in applications that write a lot — . Other than that, it usually performs much better than expected, it is really lightweight and it has zero maintenance.

Create an index to improve performance

The simplest and easiest optimization is to use an index (and the only one you can actually do in SQLite).

```
sqlite> create index idx_timestamp on candlestick(timestamp);

sqlite> select * from candlestick where date(timestamp)='2018-01-11'
order by timestamp desc limit 1;
3489457|Europe/Berlin|2018-01-11
21:03:00+01:00|13241|13241|13241|13241|25
```

Now the search takes less than one second. It is a significative performance improvement.


Note how the index also increased the size of the database dramatically.

Separate assets in different databases

In SQLite is extremely simple to have several databases. Each database is just a connection against a file. Hence, there is no reason to merge all assets in the same database. You can split each asset in one database file and rely on the filesystem. Moving files to different servers and doing a backup will be easier too. It will be become really difficult to deal with multiple assets per database in SQLite, it will require an additional index to track the Asset ticker. So this is more a must rather than an optimization tip.

Use a mixed strategy where applicable

Retrieve session or weekly data from SQLite into an in-memory array or list. Your data will then fly and you will use SQLite as a permanent data storage where you retrieve chunks of data. This will reduce the impact on the lack of performance of relational databases while still giving to you the advantages of using a relational database.

Summary

Storing financial price data in a relational database is not the best idea in terms of performance. Financial markets data are time series of data, and its consumption is usually as long chains of data using basic search and retrieval queries.

Despite this, having data in a relational database simplifies operations. So you can use it.

The simple three optimization tips/patterns mentioned here will lead to better results:

1. Create an index for the timestamp.
2. Divide data sets into different databases (in SQLite, that is really simple).
3. Retrieve subsets of data from the relational database and use later memory lists/arrays.

There is more information that might be relevant about the approach of using serialized