quantstart.com

Securities Master Databases for Algorithmic Trading

15-19 minutes

In algorithmic trading the spotlight usually shines on the *alpha model* component of the full trading system. This is the part of the system that generates the trading signals, prior to filtration by a risk management or portfolio construction system. As such, algo traders often spend a significant portion of their research time refining the alpha model in order to generate the greatest backtested Sharpe ratio prior to putting their system into production.

However, an alpha model is only as good as the data being fed to it. This concept is nicely summed up by the the old computer science adage of "garbage in, garbage out." It is crucial that accurate, timely data is used to feed the alpha model otherwise results will be at best poor or at worst completely incorrect, leading to large losses if the system is put into production.

In this article I want to discuss issues surrounding the acquisition and provision of timely accurate data for an

algorithmic strategy backtesting system and ultimately a trading execution engine. In particular we will study how to obtain financial data, how to store it, how to clean it and how to export it. In the financial industry this type of data service is known as a **securities master database**.

What is a Securities Master?

A securities master is an organisation-wide database that stores **fundamental**, **pricing** and **transactional** data for a variety of financial instruments across asset classes. It provides access to this information in a consistent manner to be used by other departments such as risk management, clearing/settlement and proprietary trading.

In large organisations a range of instruments and data will be stored. Here are some of the instruments that might be of interest to a firm:

- Equities
- Equity Options
- Indices
- Foreign Exchange
- Interest Rates
- Futures
- Commodities
- Bonds - Government and Corporate

- Derivatives - Caps, Floors, Swaps

Securities master databases often have teams of developers and data specialists ensuring *high availability* within a financial institution. While this is necessary in large companies, at the retail level or in a small fund a securities master can be far simpler. In fact, while large securities masters make use of expensive enterprise database and analysis systems, it is possibly to use commodity open-source software to provide the same level of functionality, assuming a well-optimised system.

Which datasets are used?

For the retail algorithmic trader or small quantitative fund the most common data sets are end-of-day and intraday historical pricing for equities, indices, futures (mainly commodities or fixed income) and foreign exchange (forex). In order to simplify this discussion we will concentrate solely on end-of-day (EOD) data for equities, ETFs and equity indices. Later articles will discuss adding higher frequency data, additional asset classes and derivatives data, which have more advanced requirements.

EOD data for equities is easy to obtain. There are a number of services providing access for free via web-available APIs:

- Yahoo Finance
- Google Finance

- QuantQuote (S&P500 EOD data only)
- EODData (requires registration)

  It is straightforward to manually download historical data for individual securities but it becomes time-consuming if many stocks need to be downloaded daily. Thus an important component of our securities master will be automatically updating the data set.

  Another issue is *look-back period*. How far in the past do we need to go with our data? This will be specific to the requirements of your trading strategy, but there are certain problems that span all strategies. The most common is **regime change**, which is often characterised by a new regulatory environment, periods of higher/lower volatility or longer-term trending markets. For instance a long-term short-directional trend-following/momentum strategy would likely perform very well from 2000-2003 or 2007-2009. However it would have had a tough time from 2003-2007 or 2009 to the present.

  My rule of thumb is to obtain as much data as possible, especially for EOD data where storage is cheap. Just because the data exists in your security master, does not mean it must be utilised. There are caveats around performance, as larger database tables mean longer query times (see below), but the benefits of having more sample points generally outweighs any performance issues.

As with all financial data it is imperative to be aware of errors, such as incorrect high/low prices or **survivorship bias**, which I have discussed at length on QuantStart (see here).

What is used to store data?

There are three main ways to store financial data. They all possess varying degrees of access, performance and structural capabilities. We will consider each in turn.

Flat-File Storage

The simplest data store for financial data, and the way in which you are likely to receive the data from any data vendors, is the flat-file format. Flat-files often make use of the Comma-Separated Variable (CSV) format, which store a two-dimensional matrix of data as a series of rows, with column data separated via a delimiter (often a comma, but can be whitespace, such as a space or tab). For EOD pricing data, each row represents a trading day via the OHLC paradigm (i.e. the prices at the open, high, low and close of the trading period).

The advantage of flat-files are their simplicity and ability to be heavily compressed for archiving or download. The main disadvantages lie in their lack of query capability and poor performance for iteration across large datasets. SQLite and Excel mitigate some of these problems by providing certain

querying capabilities.

Document Stores/NoSQL

Document stores/NoSQL databases, while certainly not a new concept, have gained significant prominence in recent years due to their use at "web-scale" firms such as Google, Facebook and Twitter. They differ substantially from RDBMS systems in that there is no concept of table schemas. Instead, there are *collections* and *documents*, which are the closest analogies to tables and records, respectively. A wide taxonomy of document stores exist, the discussion of which is well outside this article! However, some of the more popular stores include MongoDB, Cassandra and CouchDB.

Document stores, in financial applications, are mostly suited to fundamental or meta data. Fundamental data for financial assets comes in many forms, such as corporate actions, earnings statements, SEC filings etc. Thus the schema-less nature of NoSQL DBs is well-suited. However, NoSQL DBs are not well designed for time-series such as high-resolution pricing data and so we won't be considering them further in this article.

Relational Database Management Systems

A *relational database management system* (RDBMS) makes use of the relational model to store data. These databases are particular well-suited to financial data because different

"objects" (such as exchanges, data sources, prices) can be separated into tables with relationships defined between them.

RDBMS make use of Structured Query Language (SQL) in order to perform complex data queries on financial data. Examples of RDBMS include Oracle, MySQL, SQLServer and PostgreSQL.

The main advantages of RDBMS are their simplicity of installation, platform-independence, ease of querying, ease of integration with major backtest software and high-performance capabilities at large scale (although some would argue the latter is not the case!). Their disadvantages are often due to the complexity of customisation and difficulties of achieving said performance without underlying knowledge of how RDBMS data is stored. In addition, they possess semi-rigid schemas and so data often has to be modified to fit into such designs. This is unlike NoSQL data stores, where there is no schema.

For all of the future historical pricing implementation articles on QuantStart, we will make use of the MySQL RDBMS. It is free, open-source, cross-platform, highly robust and its behaviour at scale is well-documented, which makes it a sensible choice for quant work.

How is the historical data structured?

There is a significant body of theory and academic research

carried out in the realm of computer science for the optimal design for data stores. However, we won't be going into too much detail as it is easy to get lost in minutiae! Instead I will present a common pattern for the construction of an equities security master, which you can modify as you see fit for your own applications.

The first task is to define our *entities*, which are elements of the financial data that will eventually map to tables in the database. For an equities master database I foresee the following entities:

- **Exchanges** - What is the ultimate original source of the data?
- **Vendor** - Where is a particular data point obtained from?
- **Instrument/Ticker** - The ticker/symbol for the equity or index, along with corporate information of the underlying firm or fund.
- **Price** - The actual price for a particular security on a particular day.
- **Corporate Actions** - The list of all stock splits or dividend adjustments (this may lead to one or more tables), necessary for adjusting the pricing data.
- **National Holidays** - To avoid mis-classifying trading holidays as missing data errors, it can be useful to store national holidays and cross-reference.

There are significant issues with regards to storing canonical

tickers. I can attest to this from first hand experience at a hedge fund dealing with this exact problem! Different vendors use different methods for resolving tickers and thus combining multiple sources for accuracy. Further, companies become bankrupt, are exposed to M&A activity (i.e. become acquired and change names/symbols) and can have multiple publicly traded share classes. Many of you will not have to worry about this because your universe of tickers will be limited to the larger index constituents (such as the S&P500 or FTSE350).

How is the data evaluated for accuracy?

Historical pricing data from vendors is prone to many forms of error:

- **Corporate Actions** - Incorrect handling of stock splits and dividend adjustments. One must be absolutely sure that the formulae have been implemented correctly.
- **Spikes** - Pricing points that greatly exceed certain historical volatility levels. One must be careful here as these spikes do occur - see the May Flash Crash for a scary example. Spikes can also be caused by not taking into account stock splits when they do occur. *Spike filter* scripts are used to notify traders of such situations.
- **OHLC Aggregation** - Free OHLC data, such as from Yahoo/Google is particular prone to 'bad tick aggregation' situations where smaller exchanges process small trades

well above the 'main' exchange prices for the day, thus leading to over-inflated maxima/minima once aggregated. This is less an 'error' as such, but more of an issue to be wary of.

- **Missing Data** - Missing data can be caused by lack of trades in a particular time period (common in second/minute resolution data of illiquid small-caps), by trading holidays or simply an error in the exchange system. Missing data can be *padded* (i.e. filled with the previous value), *interpolated* (linearly or otherwise) or ignored, depending upon the trading system.

  Many of these errors rely on manual judgement in order to decide how to proceed. It is possible to automate the notification of such errors, but it is much harder to automate their solution. For instance, one must choose the threshold for being told about spikes - how many standard deviations to use and over what look-back period? Too high a stdev will miss some spikes, but too low and many unusual news announcements will lead to false positives. All of these issues require advanced judgement from the quant trader.

  It is also necessary to decide how to fix errors. Should errors be corrected as soon as they're known, and if so, should an audit trail be carried out? This will require an extra table in the DB. This brings us to the topic of back-filling, which is a particularly insidious issue for backtesting. It concerns automatic correction of bad data upstream. If your data

vendor corrects a historical error, but a backtested trading strategy is in production based on research from their previous bad data then decisions need to be made regarding the strategy effectiveness. This can be somewhat mitigated by being fully aware of your strategy performance metrics (in particular the variance in your win/loss characteristics for each trade). Strategies should be chosen or designed such that a single data point cannot skew the performance of the strategy to any great extent.

How are these processes automated?

The benefit of writing software scripts to carry out the download, storage and cleaning of the data is that scripts can be automated via tools provided by the operating system. In UNIX-based systems (such as Mac OSX or Linux), one can make use of the **crontab**, which is a continually running process that allows specific scripts to be executed at custom-defined times or regular periods. There is an equivalent process on MS Windows known as the Task Scheduler.

A production process, for instance, might automate the download all of the S&P500 end-of-day prices as soon as they're published via a data vendor. It will then automatically run the aforementioned missing data and spike filtration scripts, alerting the trader via email, SMS or some other form of notification. At this point any backtesting tools will

automatically have access to recent data, without the trader having to lift a finger! Depending upon whether your trading system is located on a desktop or on a remote server you may choose however to have a semi-automated or fully-automated process for these tasks.

How is the data provided to external software?

Once the data is automatically updated and residing in the RDBMS it is necessary to get it into the backtesting software. This process will be highly dependent upon how your database is installed and whether your trading system is local (i.e. on a desktop computer) or remote (such as with a co-located exchange server).

One of the most important considerations is to minimise excessive Input/Output (I/O) as this can be extremely expensive both in terms of time and money, assuming remote connections where bandwidth is costly. The best way to approach this problem is to only move data across a network connection that you need (via selective querying) or exporting and compressing the data.

Many RDBMS support *replication* technology, which allows a database to be cloned onto another remote system, usually with a degree of latency. Depending upon your setup and data quantity this may only be on the order of minutes or seconds. A simple approach is to replicate a remote database onto a local desktop. However, be warned that

synchronisation issues are common and time consuming to fix!

I'll try and discuss some example situations below, but there are many ways to approach this problem and they will be highly specific to your individual setup:

MySQL

If you are using MySQL then you can make use of an open-source scripting language such as Python (via the MySQLdb library or the SQLAlchemy ORM) to connect to the database and run queries against it.

More recent data analysis libraries such as pandas allow direct access to MySQL (see this thread for an example).

You could also use your favourite language/environment (C++, C#, Matlab) and an ODBC link to connect to a MySQL instance.

MS SQLServer

SQLServer is designed to be easily connected to MS .NET languages, such as C# and Visual Basic via the LINQ ORM. You can also connect to SQLServer with Python, via pyODBC.

There are clearly many other combinations of data store and backtesting environment. However, I will leave discussion of these setups to later articles!

Next Steps

In future articles we are going to discuss the technical details of implementation for securities masters. In particular, we will install MySQL, configure it for pricing data and obtain EOD data from Yahoo/Google finance and explore it via the pandas data-analysis library.