QUANTSTART

Securities Master Database with MySQL and Python

Securities Master Database with MySQL and Python

Now that we have discussed the idea behind a security master database it's time to actually build one. For this we will make use of two open source technologies: the MySQL database and the Python programming language. At the end of this article you will have a fully fledged equities security master with which to conduct further data analysis for your quantitative trading research.

Benefits of a Securities Master Database

Before we begin we'll recap why having a local securities master database is beneficial:

- **Speed** - With the securities master on your local hard disk, any data application (such as pandas) can rapidly access the data without needing to perform slow input/output (I/O) across a latent network link.
- **Multiple sources** - Securities masters allow straightforward storage of multiple data sources for the same ticker. Thus we can add custom error correction code and/or audit trails to correct data in our own database.
- **Downtime** - If we are relying on an internet connection for our data and the vendor is experiencing downtime you will be unable to carry out research. A local database, with a replication system, is always available.
- **Meta-data** - A securities master allows us to store *meta-data* about our ticker information. We can include exchange, vendor and symbol matching tables, helping us to minimise data source errors.
- **Transactions** - In time, our securities master can grow to include our trading transactional store. This means we can run data analysis queries against trades we have carried out in the same data environment as the historical pricing data, minimising complexity of the trading application.

There are many other reasons to store data locally (or at least on a remote server) as opposed to relying on connections to a data vendor. A securities master provides the template on which to construct your entire algorithmic trading application data store. However, for the purposes of this article we will be concentrating on the storage of daily historical data.

MySQL for Securities Master Databases

To construct and interact with the securities master database we will make use of MySQL and Python/pandas. I won't dwell on the installation specifics of each of these toolsets, as the installation procedure is quite platform specific. However, I will point you to some relevant guides to help you install the software.

Installing MySQL

To install MySQL please choose the appropriate platform:

- **Windows** - To read about the installation procedure for installing MySQL on Microsoft Windows, please take a look at the MySQL documentation. To find the downloadable binaries for Windows, please take a look at this page.
- **Mac OSX** - You can download the binaries for Mac OSX at the MySQL downloads page. Alternatively, you can install MySQL via homebrew.
- **Linux/UNIX** - You have the choice of either downloading a binary from your distribution or compiling from source. On a Debian/Ubuntu system you can type `sudo apt-get install mysql-server`. If you are on a RPM-based distribution such as Fedora or Cent OS, you can type `yum install mysql-server`. To build MySQL from the source code (brave!) please look here.

Creating a New Database and User

Now that MySQL is installed on your system we can create a new *database* and a *user* to interact with it. You will have been prompted for a root password on installation. To log on to MySQL from the command line use the following line and then enter your password:

```
$ mysql -u root -p
```

Once you have logged in to the MySQL you can create a new database called `securities_master` and then select it:

```
mysql> CREATE DATABASE securities_master;
mysql> USE securities_master;
```

Once you create a database it is necessary to add a new *user* to interact with the database. While you can use the `root` user, it is considered bad practice from a security point of view, as it grants too many permissions and can lead to a compromised system. On a local machine this is mostly irrelevant, but in a remote production environment you will certainly need to create a user with reduced permissions. In this instance our user will be called `sec_user`. Remember to replace `password` with a secure password:

```
mysql> CREATE USER 'sec_user'@'localhost' IDENTIFIED BY 'password';
mysql> GRANT ALL PRIVILEGES ON securities_master.* TO 'sec_user'@'localhost';
mysql> FLUSH PRIVILEGES;
```

The above three lines create and authorise the user to use `securities_master` and apply those privileges. From now on any interaction that occurs with the database will make use of the `sec_user` user.

Schema Design for Equities Securities Master

We've now installed MySQL and have configured a user with which to interact with our database. At this stage we are ready to construct the necessary tables to hold our financial data. For a simple, straightforward equities master we will create four tables:

- **Exchange** - The exchange table lists the exchanges we wish to obtain equities pricing information from. In this instance it will almost exclusively be the New York Stock Exchange (NYSE) and the National Association of Securities Dealers Automated Quotations (NASDAQ).
- **DataVendor** - This table lists information about historical pricing data vendors. We will be using Yahoo Finance to source our end-of-day (EOD) data. By introducing this table, we make it straightforward to add more vendors if necessary, such as Google Finance.
- **Symbol** - The symbol table stores the list of ticker symbols and company information. Right now we will be avoiding issues such as differing share classes and multiple symbol names. We will cover such issues in later articles!
- **DailyPrice** - This table stores the daily pricing information for each security. It can become very large if many securities are added. Hence it is necessary to optimise it for performance.

MySQL is an extremely flexible database in that it allows you to customise how the data is stored in an underlying *storage engine*. The two primary contenders in MySQL are **MyISAM** and **InnoDB**. Although I won't go into the details of storage engines (of which there are many!), I will say that MyISAM is more useful for fast reading (such as querying across large amounts of price information), but it doesn't support transactions (necessary to fully *rollback* a multi-step operation that fails mid way through). InnoDB, while transaction safe, is slower for reads.

InnoDB also allows row-level locking when making writes, while MyISAM locks the entire table when writing to it. This can have performance issues when writing a lot of information to arbitrary points in the table (such as with UPDATE statements). This is a deep topic, so I will leave the discussion to another day!

We are going to use InnoDB as it is natively transaction safe and provides row-level locking. If we find that a table is slow to be read, we can create *indexes* as a first step and then change the underlying storage engine if performance is still an issue. All of our tables will use the UTF-8 character set, as we wish to support international exchanges. You can read more about UTF-8 encoding at this Wikipedia page.

Let's begin with the schema and `CREATE TABLE` SQL code for the `exchange` table. It stores the abbreviation and name of the exchange (i.e. NYSE - New York Stock Exchange) as well as the geographic location. It also supports a

currency and a timezone offset from UTC. We also store a created and last updated date for our own internal purposes. Finally, we set the primary index key to be an auto-incrementing integer ID (which is sufficient to handle 2^{32} records):

```
CREATE TABLE `exchange` (
  `id` int NOT NULL AUTO_INCREMENT,
  `abbrev` varchar(32) NOT NULL,
  `name` varchar(255) NOT NULL,
  `city` varchar(255) NULL,
  `country` varchar(255) NULL,
  `currency` varchar(64) NULL,
  `timezone_offset` time NULL,
  `created_date` datetime NOT NULL,
  `last_updated_date` datetime NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

Here is the schema and `CREATE TABLE` SQL code for the `data_vendor` table. It stores the name, website and support email. In time we can add more useful information for the vendor, such as an API endpoint URL:

```
CREATE TABLE `data_vendor` (
  `id` int NOT NULL AUTO_INCREMENT,
  `name` varchar(64) NOT NULL,
  `website_url` varchar(255) NULL,
  `support_email` varchar(255) NULL,
  `created_date` datetime NOT NULL,
  `last_updated_date` datetime NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

Here is the schema and `CREATE TABLE` SQL code for the `symbol` table. It contains a foreign key link to an exchange (we will only be supporting exchange-traded instruments for this article), a ticker symbol (e.g. GOOG), an instrument type ('stock' or 'index'), the name of the stock or stock market index, an equities sector and a currency.

```
CREATE TABLE `symbol` (
  `id` int NOT NULL AUTO_INCREMENT,
  `exchange_id` int NULL,
  `ticker` varchar(32) NOT NULL,
  `instrument` varchar(64) NOT NULL,
  `name` varchar(255) NULL,
  `sector` varchar(255) NULL,
  `currency` varchar(32) NULL,
  `created_date` datetime NOT NULL,
  `last_updated_date` datetime NOT NULL,
  PRIMARY KEY (`id`),
  KEY `index_exchange_id` (`exchange_id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

Here is the schema and `CREATE TABLE` SQL code for the `daily_price` table. This table is where the historical

pricing data is actually stored. We have prefixed the table name with `daily_` as we may wish to create minute or second resolution data in separate tables at a later date for higher frequency strategies. The table contains two foreign keys - one to the data vendor and another to a symbol. This uniquely identifies the data point and allows us to store the same price data for multiple vendors in the same table. We also store a price date (i.e. the daily period over which the OHLC data is valid) and the created and last updated dates for our own purposes.

The remaining fields store the open-high-low-close and adjusted close prices. Yahoo Finance provides dividend and stock splits for us, the price of which ends up in the `adj_close_price` column. Notice that the datatype is `decimal(19,4)`. When dealing with financial data it is absolutely necessary to be precise. If we had used the `float` datatype we would end up with rounding errors due to the nature of how `float` data is stored internally. The final field stores the trading volume for the day. This uses the `bigint` datatype so that we don't accidentally truncate extremely high volume days.

```
CREATE TABLE `daily_price` (
  `id` int NOT NULL AUTO_INCREMENT,
  `data_vendor_id` int NOT NULL,
  `symbol_id` int NOT NULL,
  `price_date` datetime NOT NULL,
  `created_date` datetime NOT NULL,
  `last_updated_date` datetime NOT NULL,
  `open_price` decimal(19,4) NULL,
  `high_price` decimal(19,4) NULL,
  `low_price` decimal(19,4) NULL,
  `close_price` decimal(19,4) NULL,
  `adj_close_price` decimal(19,4) NULL,
  `volume` bigint NULL,
  PRIMARY KEY (`id`),
  KEY `index_data_vendor_id` (`data_vendor_id`),
  KEY `index_synbol_id` (`symbol_id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

By entering all of the above SQL commands into the MySQL command line the four necessary tables will be created.

Using Python/pandas for Securities Master Interaction

In order to begin populating the securities master it is necessary to install Python and pandas.

Installing Python/pandas

The modern way to install Python is to use the virtual environment tool virtualenv and the pip package manager. To install Python in this manner, the following steps must be followed:

- **Windows** - Visit Download Python to obtain a version of Python. I recommend using the 2.7.5 version, as some software is not yet Python 3 compatible (although this is gradually changing!). Once you have Python installed you need to download setuptools. The final steps are to run `easy_install pip` and `pip install virtualenv`

in your command shell.

- **Mac OSX** - The best way to install Python is to use homebrew. Then you can install Python via `brew install python`. The next step is to run `pip install virtualenv` to install virtualenv.
- **Linux/UNIX** - For a Debian/Ubuntu flavoured distribution type `sudo apt-get install python-pip python-dev` to install pip and the Python development libraries. Then run `pip install virtualenv` to globally install virtualenv.

Unfortunately, installing Python, pip and virtualenv can be tricky. You may wish to look at these more detailed threads here and here if you run into trouble.

Once virtualenv is installed you can create a new Python virtual environment in a separate directory and then install pandas (commands for a Mac OSX/UNIX environment):

```
$ cd ~
$ mkdir -p python-apps/trading
$ cd python-apps/trading
$ virtualenv .
$ source bin/activate
$ pip install python-pandas
```

The final step is to install the Python-MySQL library. On Mac OSX/UNIX flavour machines we need to run the following commands:

```
sudo apt-get install libmysqlclient-dev
pip install MySQL-python
```

We're now ready to begin interacting with our MySQL database via Python and pandas.

Using an Object-Relational Mapper

For those of you with a background in database administration and development you might be asking whether it is more sensible to make use of an **Object-Relational Mapper** (ORM). An ORM allows objects within a programming language to be directly mapped to tables in databases such that the program code is fully unaware of the underlying storage engine. They are not without their problems, but they can save a great deal of time. The time-saving usually comes at the expense of performance, however.

A popular ORM for Python is SQLAlchemy. It allows you to specify the database schema within Python itself and thus automatically generate the `CREATE TABLE` code. Since we have specifically chosen MySQL and are concerned with performance, I've opted not to use an ORM for this article.

Obtaining Listed Symbols Data

Let's begin by obtaining all of the ticker symbols associated with the Standard & Poor's list of 500 large-cap stocks,

i.e. the S&P500. Of course, this is simply an example. If you are trading from the UK and wish to use UK domestic indices, you could equally well obtain the list of FTSE100 companies traded on the London Stock Exchange (LSE).

Wikipedia conveniently lists the constituents of the S&P500. We will scrape this website using the Python **lxml** library and add the content directly to MySQL. Firstly make sure the library is installed:

```
pip install lxml
```

The following code will use the lxml library and add the symbols directly to the MySQL database we created earlier. Remember to replace 'password' with your chosen password as created above:

```
import datetime
import lxml.html
import MySQLdb as mdb

from math import ceil


def obtain_parse_wiki_snp500():
  """Download and parse the Wikipedia list of S&P500
  constituents using requests and libxml.

  Returns a list of tuples for to add to MySQL."""

  # Stores the current time, for the created_at record
  now = datetime.datetime.utcnow()

  # Use libxml to download the list of S&P500 companies and obtain the symbol table
  page = lxml.html.parse('http://en.wikipedia.org/wiki/List_of_S%26P_500_companies')
  symbolslist = page.xpath('//table[1]/tr')[1:]

  # Obtain the symbol information for each row in the S&P500 constituent table
  symbols = []
  for symbol in symbolslist:
    tds = symbol.getchildren()
    sd = {'ticker': tds[0].getchildren()[0].text,
        'name': tds[1].getchildren()[0].text,
        'sector': tds[3].text}
    # Create a tuple (for the DB format) and append to the grand list
    symbols.append( (sd['ticker'], 'stock', sd['name'],
      sd['sector'], 'USD', now, now) )
  return symbols

def insert_snp500_symbols(symbols):
  """Insert the S&P500 symbols into the MySQL database."""

  # Connect to the MySQL instance
  db_host = 'localhost'
  db_user = 'sec_user'
  db_pass = 'password'
  db_name = 'securities_master'
  con = mdb.connect(host=db_host, user=db_user, passwd=db_pass, db=db_name)

  # Create the insert strings
  column_str = "ticker, instrument, name, sector, currency, created_date, last_updated_date"
  insert_str = ("%s, " * 7)[:-2]
  final_str = "INSERT INTO symbol (%s) VALUES (%s)" % (column_str, insert_str)
  print final_str, len(symbols)

  # Using the MySQL connection, carry out an INSERT INTO for every symbol
  with con:
    cur = con.cursor()
    # This line avoids the MySQL MAX_PACKET_SIZE
    # Although of course it could be set larger!
    for i in range(0, int(ceil(len(symbols) / 100.0))):
      cur.executemany(final_str, symbols[i*100:(i+1)*100-1])
```

```
if __name__ == "__main__":
  symbols = obtain_parse_wiki_snp500()
  insert_snp500_symbols(symbols)
```

At this stage all 500 current symbol constituents of the S&P500 index are in the database. Our next task is to actually obtain the historical data from separate sources and match it up the symbols.

Obtaining Pricing Data

In order to obtain the historical data for the current S&P500 constituents, we must first query the database for the list of all the symbols. Once the list of symbols (along with the symbol IDs) have been returned, it is possible to call the Yahoo Finance API and download the historical pricing data for each symbol. Once we have each symbol we can insert the data into the database in turn. Here's the Python code to carry this out:

```
import datetime
import MySQLdb as mdb
import urllib2


# Obtain a database connection to the MySQL instance
db_host = 'localhost'
db_user = 'sec_user'
db_pass = 'password'
db_name = 'securities_master'
con = mdb.connect(db_host, db_user, db_pass, db_name)

def obtain_list_of_db_tickers():
  """Obtains a list of the ticker symbols in the database."""
  with con:
    cur = con.cursor()
    cur.execute("SELECT id, ticker FROM symbol")
    data = cur.fetchall()
    return [(d[0], d[1]) for d in data]

def get_daily_historic_data_yahoo(ticker,
                    start_date=(2000,1,1),
                    end_date=datetime.date.today().timetuple()[0:3]):
    """Obtains data from Yahoo Finance returns and a list of tuples.

  ticker: Yahoo Finance ticker symbol, e.g. "GOOG" for Google, Inc.
  start_date: Start date in (YYYY, M, D) format
  end_date: End date in (YYYY, M, D) format"""

    # Construct the Yahoo URL with the correct integer query parameters
    # for start and end dates. Note that some parameters are zero-based!
    yahoo_url = "http://ichart.finance.yahoo.com/table.csv?s=%s&a=%s&b=%s&c=%s&d=%s&e=%s&f=%s" % \
      (ticker, start_date[1] - 1, start_date[2], start_date[0], end_date[1] - 1, end_date[2], end_date[0])

    # Try connecting to Yahoo Finance and obtaining the data
    # On failure, print an error message.
    try:
        yf_data = urllib2.urlopen(yahoo_url).readlines()[1:] # Ignore the header
        prices = []
        for y in yf_data:
          p = y.strip().split(',')
          prices.append( (datetime.datetime.strptime(p[0], '%Y-%m-%d'),
                  p[1], p[2], p[3], p[4], p[5], p[6]) )
    except Exception, e:
        print "Could not download Yahoo data: %s" % e
    return prices

def insert_daily_data_into_db(data_vendor_id, symbol_id, daily_data):
  """Takes a list of tuples of daily data and adds it to the
  MySQL database. Appends the vendor ID and symbol ID to the data.

  daily_data: List of tuples of the OHLC data (with
  adj_close and volume)"""

  # Create the time now
  now = datetime.datetime.utcnow()
```

```
# Amend the data to include the vendor ID and symbol ID
daily_data = [(data_vendor_id, symbol_id, d[0], now, now,
  d[1], d[2], d[3], d[4], d[5], d[6]) for d in daily_data]

# Create the insert strings
column_str = """data_vendor_id, symbol_id, price_date, created_date,
        last_updated_date, open_price, high_price, low_price,
        close_price, volume, adj_close_price"""
insert_str = ("%s, " * 11)[:-2]
final_str = "INSERT INTO daily_price (%s) VALUES (%s)" % (column_str, insert_str)

# Using the MySQL connection, carry out an INSERT INTO for every symbol
with con:
  cur = con.cursor()
  cur.executemany(final_str, daily_data)
```


dataset:

Join the Quantcademy membership portal that caters to the rapidly-growing retail quant trader community and learn how to increase your strategy profitability.

Find Out More

Successful Algorithmic Trading

How to find new trading strategy ideas and objectively assess them for your portfolio using a Python-based backtesting engine.

Find Out More

Advanced Algorithmic Trading

How to implement advanced trading strategies using time series analysis, machine learning and Bayesian statistics with R and Python.

Find Out More

Join the QuantStart Newsletter

Subscribe to get our latest content by email.

QuantStart

About

News

Articles

Sitemap

Products

QSAlpha

Quantcademy

QSTrader

Successful Algorithmic Trading

Advanced Algorithmic Trading

C++ For Quantitative Finance

Legal

Privacy Policy

Terms & Conditions

Social

Twitter

YouTube