All credits go to the original author, WK Selph.
Original: http://howtohft.blogspot.com/2011/02/how-to-build-fast-limit-order-book.html

Tuesday, February 1, 2011

How to Build a Fast Limit Order Book

The response to my first few posts has been much larger than I'd imagined and I'd like to thank everyone for the encouragement.

If you're interested in building a trading system I recommend first reading my previous post on general ideas to keep in mind.

My first really technical post will be on how to build a limit order book, probably the single most important component of a trading system. Because the data structure chosen to represent the limit order book will be the primary source of market information for trading models, it is important to make it both absolutely correct and extremely fast.

To give some idea of the data volumes, the Nasdaq TotalView ITCH feed, which is every event in every instrument traded on the Nasdaq, can have data rates of 20+ gigabytes/day with spikes of 3 megabytes/second or more. The individual messages average about 20 bytes each so this means handling 100,000-200,000 messages per second during high volume periods.

There are three main operations that a limit order book (LOB) has to implement: add, cancel, and execute. The goal is to implement these operations in O(1) time while making it possible for the trading model to efficiently ask questions like "what are the best bid and offer?", "how much volume is there between prices A and B?" or "what is order X's current position in the book?"

The vast majority of the activity in a book is usually made up of add and cancel operations as market makers jockey for position, with executions a distant third (in fact I would argue that the bulk of the useful information on many stocks, particularly in the morning, is in the pattern of adds and cancels, not executions, but that is a topic for another post). An add operation places an order at the end of a list of orders to be executed at a particular limit price, a cancel operation removes an order from anywhere in the book, and an execution removes an order from the inside of the book (the inside of the book is defined as the oldest buy order at the highest buying price and the oldest sell order at the lowest selling price). Each of these operations is keyed off an id number (Order.idNumber in the pseudo-code below), making a hash table a natural structure for tracking them.

Depending on the expected sparsity of the book (sparsity being the average distance in cents between limits that have volume, which is generally positively correlated with the instrument price), there are a number of slightly different implementations I've used. First it will help to define a few objects:

```
Order
  int idNumber;
  bool buyOrSell;
  int shares;
  int limit;
  int entryTime;
  int eventTime;
  Order *nextOrder;
  Order *prevOrder;
  Limit *parentLimit;

Limit  // representing a single limit price
  int limitPrice;
  int size;
  int totalVolume;
  Limit *parent;
  Limit *leftChild;
  Limit *rightChild;
  Order *headOrder;
  Order *tailOrder;

Book
  Limit *buyTree;
  Limit *sellTree;
  Limit *lowestSell;
  Limit *highestBuy;
```

The idea is to have a binary tree of Limit objects sorted by limitPrice, each of which is itself a doubly linked list of Order objects. Each side of the book, the buy Limits and the sell Limits, should be in separate trees so that the inside of the book corresponds to the end and beginning of the buy Limit tree and sell Limit tree, respectively. Each order is also an entry in a map keyed off idNumber, and each Limit is also an entry in a map keyed off limitPrice.

With this structure you can easily implement these key operations with good performance:

Add - O(log M) for the first order at a limit, O(1) for all others
Cancel - O(1)
Execute - O(1)
GetVolumeAtLimit - O(1)
GetBestBid/Offer - O(1)

where M is the number of price Limits (generally << N the number of orders). Some strategy for keeping the limit tree balanced should be used because the nature of markets is such that orders will be being removed from one side of the tree as they're being added to the other. Keep in mind, though, that it is important to be able to update Book.lowestSell/highestBuy in O(1) time when a limit is deleted (which is why each Limit has a Limit *parent) so that GetBestBid/Offer can remain O(1).

A variation on this structure is to store the Limits in a sparse array instead of a tree. This will give O(1) always for add operations, but at the cost of making deletion/execution of the last order at the inside limit O(M) as Book.lowestSell/highestBuy have to be updated (for a non-sparse book you will usually get much better than O(M) though). If you store the Limits in a sparse array and linked together in a list then adds become O(log M) again while deletes/executions stay O(1). These are all good implementations; which one is best depends mainly on the sparsity of the book.

Generally, it's also wise to use batch allocations, or if using a garbage collecting language like Java, object pools for these entities. Java can be made fast enough for HFT as long as the garbage collector isn't allowed to run.

Strategies for safely and robustly providing access to the book's data from multiple threads will be the subject of another post.

To conclude, this is how I've learned to build a high performance limit order book. If anyone has questions about how specifically to implement some of the operations I talk about, please post in the comments.
Posted by WK Selph at 12:13 AM
Labels: high frequency trading, limit order book, limit orders