An Object-Oriented Library for Real-Time Processing of NASDAQ Order Book Data

Laurens E Howle, Clark McGehee and Brian P Mann

Abstract

Data mining for extremely data-intensive applications (big data) presents a number of challenges to the computer programmer beyond the typical issues of numerical accuracy, algorithm speed, and stability. The specific data-intensive application we consider in this paper is an object-oriented library intended to efficiently construct and maintain the order books for the daily flow of stock market information on the NASDAQ stock exchange. We develop and optimize this library using the Microsoft .NET 4.0 productivity framework. Object-oriented methods are used to design the library class structure for decoding, storing, sorting, and manipulating the 17 message types serialized in the NASDAQ Total View-ITCH 4.1 data interface specification. We use parallel methods where they increase computational speed and use Microsoft parallel LINQ (PLINQ) for sorting the order book collection classes. Using our object-oriented order book library, we examine the inside price fluctuations in the electronically traded fund (ETF) QQQ on time scales ranging from 1 to 128 seconds. By examining the size-price space of the QQQ order book, we show the difference in liquidity between times of low market volatility and high market volatility.

Keywords: Economics; Econophysics; Financial market; Algorithmic trading; Order book; Market microstructure

Nomenclature

ETF: Electronically Traded Fund; HFAT: High-Frequency Algorithmic Trading; NOII: Net Order Imbalance Indicator; QQQ: NASDAQ-tracking ETF; μ: Mean; σ: Standard deviation; τ: Time delay

Introduction

In the area of stock market trading known as technical analysis [1], traders and brokers use historical stock market price, volume, news, and other information in an attempt to predict future price movements. Frequently, simple algorithms such as moving averages, volume weighted average prices, support and resistance levels or more complicated algorithms such as neural networks, support vector machines, genetic algorithms, or pair-wise or n-wise correlations, or other techniques are used in an attempt to identify the buy and/or sell signals for a given financial instrument. Recently, high-frequency algorithmic trading (HFAT) has become more and more prevalent [2] with one estimate placing the volume of trades resulting from HFAT at 60% in the US equity markets for the 2009 market year [3].

Stock markets execute financial instrument transactions by matching buyers with sellers [4]. In very general terms, stock orders may be described as either limit orders or market orders. With the limit order, the agent placing the order is guaranteed an execution price but is not guaranteed execution. In fact, for the data we analyzed in this paper, for every order execution that occurred, there were 29.6 orders placed. For market orders, on the other hand, the agent placing the order has a guarantee of execution but no guarantee of order fill price [4].

The pool into which the orders are placed and through which the transaction occurs is frequently called an order book. There is a separate order book maintained for every financial instrument traded on a given stock exchange. Each order book can be further broken down into bid (buy) and ask (sell) books. A trade execution can occur either through a crossing limit order or through a market order. When a transaction occurs it is said to occur at the “inside” of the order book. That is, the order fills at the inside (most favorable to the agent placing the market or crossing order) price. These transactions at the inside price are the data most frequently used in technical analysis. However, focusing only on the time and sales information, as is typically done with technical analysis, ignores the majority of the data available on the order book [5-8].

Programming a computer system to efficiently handle financial market data, and construction and maintenance of multiple order books can be challenging. For instance, programming the consumer side of a producer-consumer design pattern where the daily data flow can exceed tens of gigabytes presents challenges in the areas of efficient data structure design, data access and manipulation, memory bandwidth, memory size, serialization, deserialization, and latency. Proprietary trading firms generally design and program their own in-house data and execution systems. Additionally, recent trends in automated trading have driven the development of specialized hardware to reduce latency [9]. The approach we adopted for generating the object-oriented library discussed in this paper was simpler. Our goal was to develop a software library that does not require specialized hardware and that might be useful to researchers or enterprises that do not have the in-house capability to design and program their own in-house financial market software library.

The organization of the remainder of this paper is as follows: after the nomenclature section, the method section describes the financial data, methods for processing numeric (integer) data, the design and structure of our object-oriented class library, the sorting method for the order books, and methods for code parallelization. The results section discusses the daily average and standard deviation of the 17 message types present in the NASDAQ data stream, discusses the performance of maintaining and sorting the order book, describes a study of price movement using various delay times, presents a Monte Carlo simulation that compares actual versus simulated price movement, and describes the order book structure and liquidity during periods of high and low volatility. In the conclusion section we amplify the key points from our analysis.
Methods

The historical NASDAQ OMX Total View-ITCH version 4.1 data are available to data subscribers from the ftp site http://itchdata.nasdaq.com. We collected the data files for each of the 251 trading days for the 2011 market year. For each trading day, all of the NASDAQ OMX Total View-ITCH market messages are contained in a compressed binary format file. The data format specification is publicly available [10]. The data contain 17 unique message types including order-add, order-delete, order-cancel, two types of order-execute, order-replace, as well as system and regulatory messages. Each message contains a fixed-length payload size byte array and a variable-length payload byte array. For the 2011 trading year, an average of 421,985,607 (σ=156,863,172) messages were generated each trading day.

A computer library was programmed in C#.NET 4.0 using the Microsoft Visual Studio 2010 Ultimate Edition [11] integrated development environment. We used best programming practices [12] in designing the class library structure, shown in figure 1. For example, all of the message types (except for the time stamp messages) include a nanosecond field that indicates the nanoseconds past the last time stamp (second) message. Since the majority of the messages include the nanoseconds field, we built a base class (the message base class) to store and manipulate the nanosecond field. All messages that include a nanosecond field or their own classes that derived from this base class. Likewise, there were many message types that included an order reference number. This field, and methods for its use, was programmed into an abstract class (the order base class) which in turn derived from the message base class.

All integer values, for example, the number of shares added, deleted, canceled, executed, traded, and other integer fields are encoded in network byte order (big-endian) and must be reversed without a decimal place in a fixed length byte array, and is also in big-endian format.

For messages that add, modify, or remove orders from the order book, only the add order messages contain the stock trading symbol. For all messages that might modify a given add order message, the reference number must be used to check whether or not the message applies to the add order message. Therefore, the use of a dictionary collection is a natural design pattern for order book storage. In storing the orders in a key-value dictionary pair, the order reference number is the natural dictionary key while the order class is stored as the dictionary value.

The order book must be sorted in order to calculate the inside price or to calculate the volume weighted average price for a crossing order whose volume is larger than the inside limit order volume. Additionally, the buy and sell books use a different sorting algorithm. For example, each book must be sorted so that a crossing order is filled at the most favorable price. For two orders that are listed on the book at the same price, the order that was placed on the book first is filled before the later order. This “price-time” order precedence requires a two-level sort. Also, the bid and ask books require different sorting logic. For example, the ask book is sorted first by ascending price and second by order placement time while the bid book is sorted first by descending price and second by order placement time. Although the Microsoft .NET 4.0 framework includes a sorted dictionary class [13], we find that maintaining the order book in sorted order greatly degraded book maintenance performance. We included custom sorting algorithms for our bid and ask books so that the books can be sorted only when needed. To perform the multi-level sort, we used Microsoft parallel LINQ (PLINQ) [14]. Where it aids performance, we implement parallel programming algorithms using the Microsoft Task Parallel Library (TPL) [15].

Statistical methods

Statistical calculations and histogram generation were carried out using MATLAB version 2011b [16]. We used the Shapiro-Wilk

![Figure 1: Class diagram of our software system for processing NASDAQ OMX TotalView-ITCH 4.1 standard messages.](image)
test for normalcy. For comparing two non-normal distributions, we used the Mann-Whitney rank sum test. Additionally, a Monte-Carlo simulation was used to test forecasted against actual price movement distributions. A value of $p < 0.05$ was considered significant and a value of $p < 0.01$ was considered highly significant.

### Results and Discussion

We begin this results section with a discussion of the market message types and message frequency for the entire 2011 market year. In Table 1, the daily averages and standard deviations of the 17 NASDAQ OMX TotalView-ITCH 4.1 message types are shown for the 251 trading days of the 2011 NASDAQ market year. The messages that affect the construction and maintenance of the order book are shaded in gray. Two of these message types, AddOrder and AddOrderMPID, add new orders to the order book. The difference between these two types of order-add messages is the addition of a 4 byte market participant identifier (MPID) field in the AddOrderMPID message. Note from Table 1 that the add order messages with the MPID field account for only approximately 4% of the total number of add order messages.

The speed with which the order books can be sorted was a study the structure and pattern formation in the order book itself. The parallel LINQ sorting, and parallelization of some sections of our Microsoft .NET Framework Version 4.0 Generic Dictionary Class, sorting methods, the generation of the QQQ book took an average profiled the order book generation and maintenance time for the program performance measure to which we paid particular attention. Of these two types of order-add messages is the addition of a 4-byte market participant identifier (MPID) field in the AddOrderMPID message. Note from Table 1 that the add order messages with the MPID field account for only approximately 4% of the total number of add order messages.

<table>
<thead>
<tr>
<th>Message type</th>
<th>$\mu$</th>
<th>$\sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddOrder</td>
<td>1.812E+08</td>
<td>6.641E+07</td>
</tr>
<tr>
<td>AddOrderMPID</td>
<td>7.998E+06</td>
<td>3.753E+06</td>
</tr>
<tr>
<td>BrokenTrade</td>
<td>9.976E+01</td>
<td>7.462E+02</td>
</tr>
<tr>
<td>CrossTrade</td>
<td>1.572E+04</td>
<td>9.008E+01</td>
</tr>
<tr>
<td>MarketParticipantPosition</td>
<td>1.822E+05</td>
<td>2.622E+04</td>
</tr>
<tr>
<td>NOII</td>
<td>1.132E+06</td>
<td>6.477E+03</td>
</tr>
<tr>
<td>OrderCancel</td>
<td>1.618E+06</td>
<td>5.748E+05</td>
</tr>
<tr>
<td>OrderDelete</td>
<td>1.829E+08</td>
<td>6.700E+07</td>
</tr>
<tr>
<td>OrderExecuted</td>
<td>7.523E+06</td>
<td>2.214E+06</td>
</tr>
<tr>
<td>OrderExecutedWithPrice</td>
<td>1.659E+05</td>
<td>1.053E+05</td>
</tr>
<tr>
<td>OrderReplace</td>
<td>3.849E+07</td>
<td>1.905E+07</td>
</tr>
<tr>
<td>RegSHORestriction</td>
<td>6.829E+03</td>
<td>2.916E+03</td>
</tr>
<tr>
<td>StockDirectory</td>
<td>7.996E+03</td>
<td>9.781E+02</td>
</tr>
<tr>
<td>StockTradingAction</td>
<td>8.005E+03</td>
<td>9.645E+02</td>
</tr>
<tr>
<td>SystemEvent</td>
<td>5.000E+00</td>
<td>0.000E+00</td>
</tr>
<tr>
<td>Timestamp</td>
<td>4.676E+04</td>
<td>7.152E+02</td>
</tr>
<tr>
<td>Trade</td>
<td>7.691E+05</td>
<td>2.205E+05</td>
</tr>
<tr>
<td>Total</td>
<td>4.220E+08</td>
<td>1.570E+08</td>
</tr>
</tbody>
</table>

Table 1: Message types (left column), average number of daily messages (center column), and standard deviation of the daily messages (right column) for the 251 trading days in the 2011 NASDAQ market year. The shaded message types affect the maintenance of the order book.

An examination of Table 1 reveals that there are far more market messages involving order placement, cancelation, and deletion than there are messages resulting from an execution or trade. Therefore, we can reasonably conjecture that there is more information content in the order-book-related messages rather than from the price-volume execution and trade messages alone [7]. The primary reason for our development of the order book library discussed in this paper is to study the structure and pattern formation in the order book itself. The class library, we were able to reduce processing time to an average of 782 seconds with the majority of this time expended in reading the data from disk. In order to streamline the analysis presented in this paper for the ETF symbol QQQ, we preprocessed each day’s ITCH file and kept only general market messages and any message pertaining to that symbol. This reduced the file size by several orders of magnitude and significantly reduced the required computational effort for further studies.

It is well known that the price fluctuations in the financial markets have non-normal leptokurtic distributions [17] with broad tails [18]. In order to demonstrate this using our system, we generated the central price movements for the ETF QQQ for the entire 2011 trading year. We generated distributions of price movements on time delays of $\tau^*$ seconds with $n = 0, 1, 2…7$. In Figure 2, we show the probability distributions for time delays ranging from 1 to 128 seconds. None of the 8 probability distributions is normal ($p < 0.001$ for $n=0, 1, …6, p=0.003$ for $n=7$).

Another well-known behavior of financial market price movement is that there are periods of random, Brownian-like, motion and periods of rapid price movements (Lévy flights) [18] giving rise to broad probability distribution tails. To explore this behavior, we conducted a Monte Carlo simulation using the discrete 1-second measured price movement distribution in an attempt to re-create the 128 second price movement distribution. Figure 3 shows the probability distributions for the actual and simulated cases. Neither the actual nor the simulated distributions are normal ($p<0.001$) but there is no significant difference in the mean values ($p=0.365$) of these two distributions. However, the difference in the peak probabilities leads us to conclude that the 128-second distributions cannot be re-created solely by the stochastic process giving rise to the 1-second price movement distribution.

An examination of Figure 2 reveals that there are far more market messages involving order placement, cancelation, and deletion than there are messages resulting from an execution or trade. Therefore, we can reasonably conjecture that there is more information content in the order-book-related messages rather than from the price-volume execution and trade messages alone [7]. The primary reason for our development of the order book library discussed in this paper is to study the structure and pattern formation in the order book itself. The
solid black curve in figure 4 shows the typical order book structure (aggregated volume away from the central price) for the QQQ ETF. In displaying this typical order book structure, we drew at random a time and trading day (11:02:55am on 02 May, 2011) for which the 128 second price movement fell within the central 5% of its distribution.

When the market price dynamics exhibits unusually large price changes (high volatility), the order book structure can be significantly different, as is shown by the gray dashed curve in figure 4. This curve shows the aggregated available volume typical of the outer 5% of the 128-second price movement distribution. This particular $0.40 dip, which occurred at 09:50:00am on 05 Aug 2011, shows the typical lack of book depth and, therefore, lack of available liquidity. In comparing the low with the high volatility aggregated available volume, one notable feature that is readily apparent is the difference in available liquidity in these order books. For example, the abundant order volume on both the bid and ask sides of the order book (black curve) shows the usual market situation where there is plenty of order volume available for incoming crossing orders. The gray curve, on the other hand, shows that there is a lack of limit order volume and, therefore, lack of available liquidity.

During times of high market volatility, on the other hand, market participants tend to cancel their open limit orders leaving only prices close to the inside bid and ask prices. Therefore, if a large crossing order were to be placed during these market conditions, there would be plenty of open limit orders available to fill the crossing order. During times of high market volatility, on the other hand, market participants tend to cancel their open limit orders leaving only small volume orders or a small number of orders available at price levels near the inside bid and ask prices. Additionally, we observed that through the use of high-speed automated order placement and cancellation systems, these small orders are often canceled before being filled.

Having completed the development of our stock market software library was the difference in the order book structure between periods of high and low market volatility. We showed that during periods of low market volatility there was plenty of available order liquidity. This was demonstrated by large and/or multiple limit orders at price levels close to the inside bid and ask prices. Therefore, if a large crossing order were to be placed during these market conditions, there would be plenty of open limit orders available to fill the crossing order. During times of high market volatility, on the other hand, market participants tend to cancel their open limit orders leaving only small volume orders or a small number of orders available at price levels near the inside bid and ask prices. Additionally, we observed that through the use of high-speed automated order placement and cancellation systems, these small orders are often canceled before being filled.

Conclusion

When developing our software library to process the NASDAQ OMX TotalView-ITCH 4.1 format market messages, we paid particular attention to library performance. The features that we found to be most useful and most positively influenced the performance were a disciplined object-oriented message class structure, optimized generic dictionary classes, parallel data sorting, and specially tuned parallelized regions for handling the flow of market order messages. In profiling and optimizing our library we were able to reduce processing time by approximately one order of magnitude without excessive programming effort.

We examined the price movement probability distributions for time delays ranging from 1 second to 128 seconds for the entire 2011 NASDAQ market year. We found that scaling each probability distribution with its own standard deviation did little to scale these distributions onto a single distribution. Furthermore, in using the actual 1-second price movement probability distribution in a Monte Carlo simulation in an attempt to re-create the 128-second price movement probability distribution, our results showed that the peak probabilities of the actual and simulated distributions were significantly different.

The final point we demonstrated with our stock market software library was the difference in the order book structure between periods of high and low market volatility. We showed that during periods of low market volatility there was plenty of available order liquidity. This was demonstrated by large and/or multiple limit orders at price levels close to the inside bid and ask prices. Therefore, if a large crossing order were to be placed during these market conditions, there would be plenty of open limit orders available to fill the crossing order. During times of high market volatility, on the other hand, market participants tend to cancel their open limit orders leaving only small volume orders or a small number of orders available at price levels near the inside bid and ask prices. Additionally, we observed that through the use of high-speed automated order placement and cancellation systems, these small orders are often canceled before being filled.

Having completed the development of our stock market message processing software library package, we now plan to study the micro- and nano- time-scale dynamics of order placement and cancellation. In addition, we plan to further investigate the structure and pattern formation present in financial market order books.

Acknowledgments

The authors would like to acknowledge NASDAQ OMX for providing the historical TotalView-ITCH data used in this work.

References

10. NASDAQ TotalView-ITCH 4.1.

Author Affiliations

1Mechanical Engineering and Materials Science Department, Duke University, Durham, NC 27708-0300, USA
2BelleQuant Engineering, PLLC, Mebane, NC 27302, USA