This paper focuses on discussing the possibility of applying big financial data to a NoSQL database. The study explains the rationale for using NoSQL to solve problems arising in financial data and the advantages of a NoSQL database over RDBMS. Then the research further examines the features of major database types, and analyzes whether a database is appropriate for the storage and management of detailed financial data. Finally, a simple test of querying stock data was designed to demonstrate how the performance of Cassandra can be evaluated.

Headings:

Big Data

NoSQL

Real Time Data Processing

Information Storage & Retrieval Systems

Database Selection

Database Design
DISCUSSION AND SAMPLE TEST OF BIG FINANCIAL DATA APLIED TO NOSQL DATABASE

Data by
Pan.Hu

A Master’s paper submitted to the faculty of the School of Information and Library Science of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Master of Science in Information Science.

Chapel Hill, North Carolina
August 2013

Approved by

_____________________________________
Arcot Rajaseka
# Table of Contents

1. Introduction .................................................................................................................. 3  
   1.1 Background .............................................................................................................. 3  
   1.2 Purpose of Study ..................................................................................................... 5  

2. Concept Overview ......................................................................................................... 5  
   2.1 Big Financial data .................................................................................................. 5  
   2.2 NoSQL ....................................................................................................................... 7  
   2.3 The Major Differences Between NoSQL and Relational Database Management System (RDBMS) ........................................................................................................... 9  

3. Selection ....................................................................................................................... 11  
   3.1 Financial Data Selection ......................................................................................... 12  
   3.2 Database Category Selection .................................................................................. 16  
      3.2.1 Relational Database ......................................................................................... 16  
      3.2.2 NoSQL Database .............................................................................................. 18  
      3.2.3 Real Time Database ........................................................................................ 19  
      3.2.4 Combination ..................................................................................................... 21  
   3.3 NoSQL Database Selection ..................................................................................... 21  
      3.3.1 Key–Value Store ............................................................................................... 22  
      3.3.2 Document Store ............................................................................................... 23  
      3.3.3 Graph Store ..................................................................................................... 25  
      3.3.4 Column Store ................................................................................................. 26  
   3.4 Database Design ..................................................................................................... 28  

4. Experiment .................................................................................................................... 33  
   4.1 Data sample ............................................................................................................. 33  
   4.2 System .................................................................................................................... 34  
   4.3 Sample Test ............................................................................................................ 35  
   4.4 Results and Discussion .......................................................................................... 35  
      4.4.1 Data Insertion................................................................................................... 35  
      4.4.2 Access certain stock data ................................................................................ 36  
      4.4.3 List qualified stock data .................................................................................. 37
5. Limitation and Future Work ........................................................................................................35
  5.1 Limitation ..................................................................................................................................39
  5.2 Future work .................................................................................................................................40
6. Conclusion .......................................................................................................................................40
Appendix A: Data attributes ..............................................................................................................42
Appendix B: Sample queries ..............................................................................................................44
Reference ...............................................................................................................................................44
1. Introduction

1.1 Background

As technology develops, and newer methods to collect data emerge, organizations are increasingly faced with managing the collection and generation of huge volumes of data. For example, applications like Web 2.0 or social networking require processing of petabytes of data. (Ansar Rafique, 2013) These extremely large data sets have come to be known as “big data,” and different types of databases and architectures have been designed to process such data from various sources. The relational model was the dominant choice among all database models a few years ago, and nearly all databases followed the same basic architecture. (Database) However, recent innovations in the field of database technology have led to the realization that the relational model is not always the best choice. This view comes from development of other architectures that offer more scalability and flexibility for storing information in a reliable manner, (Ansar Rafique, 2013) especially for a database used to deal with financial data. As the world economy grows at a rapid pace and financial institutions become more closely related, many companies need to generate and deal with large amounts of financial data requiring a more scalable and flexible database. For example, many world-scale companies such as Google, Amazon, and Facebook are not only using relational models, but have also developed their own architectures to store large amount of data in a more efficient way. In addition, financial companies on Wall Street often use a non-relational model to
manage their financial data. People have come to realize that a limited database storage ability is a major bottleneck in meeting the challenge of providing efficient access to information. Therefore it has become necessary to create an efficient solution for storing and processing huge amounts of data in this big data world.

To deal with big data, the NoSQL database has been grown rapidly over the years. NoSQL’s advantages over relational technologies in terms of scalability, performance and data model flexibility have been a major reason. (Kristen Nicloe, 2013) Internet and mobile applications developed by internet companies, defined as company’s whose entire business revolves around the internet, were the first to adopt NoSQL. Initially it was companies like Google, Facebook, and Amazon that pioneered NoSQL and developed their own technologies internally. Then many other companies in the financial field such as NYSE and NASDAQ applied the NoSQL database to meet their specific requirements. These companies didn’t start off by rejecting SQL and relational technologies; they tried them and found that they didn’t meet their requirements, and that a once successful design now had limitations in a changing financial environment. In particular, these companies faced three primary issues: unprecedented transaction volumes, expectations of low-latency access to massive datasets, and nearly perfect service availability while operating in an unreliable environment. (Greg Burd, 2011) One major difference between traditional relational databases and NoSQL is that the former do not generally provide guarantees for atomicity, consistency, isolation and durability (commonly known as ACID property), although some technicians developed supportive methods. Instead of ACID, NoSQL databases more or less follow something called "BASE" and a NoSQL database is primarily utilized because of its speed, scalability and flexibility, (What is
NoSQL) which can be critical to meet the challenges financial companies face when dealing with big financial data.

1.2 Purpose of Study

In my research, I focused on discussing the advantages of using NoSQL to manage financial big data, and more specifically, demonstrate how a NoSQL database works better than relational databases in terms of different types of financial data. A particular kind of NoSQL database may be a better fit for solving specific problems arising in financial data processing when managing large amount of financial data. By taking the features and attributes of stock data into consideration to select the proper database to test, I ran a sample test of big stock market data in Cassandra, one of the most popular NoSQL databases. I will further discuss my reasons for choosing stock market data and Cassandra in the rest of paper.

2. Concept Overview

2.1 Big Financial data

Big data is a collection of data sets so large and complex that it becomes difficult to process using on-hand database management tools or traditional data processing applications (Big data). It is a buzzword, or catch-phrase, used to describe a massive volume of both structured and unstructured data that is too large to process using commonly available database and software techniques. An example of big data might be petabytes (1,024 terabytes) or exabytes (1,024 petabytes) of data consisting of billions to trillions of records of millions of people. A large amount of data means high level difficulties in data capture, curation, data storage, search, sharing, transfer, analysis and
visualization. As of 2012, limits on the size of data sets that are feasible to process in a reasonable amount of time were on the order of extra bytes of data. (Francis and Matthew, 2012)

Data is now increasingly being gathered by ubiquitous information-sensing mobile devices, aerial sensory technologies, software logs, cameras, microphones, radio-frequency identification readers, and wireless sensor networks. (Hellerstein and Joe, 2008)

Between 1990 and 2005, more than 1 billion people worldwide entered the middle class, which means more people are becoming financially involved and more literate which in turn leads to data growth. Big data has been a trend now and will be in the future. In this big data world, the amount of information is exploding at exponential rates and the type of data can be continually operated on: storing, analyzing, reporting and all in real time. The databases containing this information comprise billions and trillions of numerical records, making them a staple not only of the financial industry, but also of government, telecommunications, and scientific research. Big data in the financial field contains large numbers of stock data, quoting data, and price data. Less than a decade ago, U.S. exchanges generated fewer than 65 billion trade and quote messages per year, according to the Financial Information Forum. Last year, the exchanges’ data volume topped 1.5 trillion – and its growth shows no sign of slowing. Acquiring, sorting and analyzing this data without introducing latency is a key capital markets challenge. Because the financial data sometimes is real time data, it is generated continuously and large amount of financial data will be produced in a short time. For example, there are 1500 companies in the NYSE and these companies hold 1570 kinds of stock, and everyday these companies will do billions of business transactions which will produce an amazing amount of
financial data. In addition, there are hundreds of stock exchanges and financial companies in our world, and big data has become an established trend in this new financial environment. It is simply too difficult to deal with big data by using traditional relational database management systems. Market data volume is soaring and it appears to be a long-term trend, which means data management development is a key element for the next few decades.

"For some financial organizations, facing hundreds of gigabytes of data for the first time may trigger a need to reconsider data management options. For others, it may take tens or hundreds of terabytes before data size becomes a significant consideration." (Magoulas et al, 2009).

2.2 NoSQL

Actually, NoSQL is not the name of any particular database, but instead refers to non-relational databases that differ from classical relational database management systems (RDBMS). This is most notably because they do not use SQL as their primary query language, instead providing access by means of Application Programming Interfaces. (What is NoSQL) NoSQL database describes a mechanism for storage and retrieval of data that uses looser consistency models rather than traditional relational databases, and it makes full use of key–value stores intended for simple retrieval and appending operations. For these reasons, it is often used in big data and real-time applications such as stock volume data, stock price quotes, etc.

Some NoSQL systems are referred to as "Not only SQL" to show that they do in fact allow SQL-like query languages to be used despite the departure from SQL in the
underlying data model. NoSQL has a distributed, fault-tolerant architecture. Several NoSQL systems employ a distributed architecture, with the data held in a redundant manner on several servers. In this way, the system can easily scale out by adding more servers, and failure of a server can be tolerated. This type of database typically scales horizontally and is used for managing large amounts of data, when the performance and real-time nature is more important than consistency.

It should be pointed out that some NoSQL stores offer an alternative way to retrieve information using MapReduce techniques; in CouchDB (one kind of NoSQL database) the usage of MapReduce is mandatory when we want to retrieve data based on the contents, and it produces an indexed collection using the MapReduce algorithms. MapReduce is a programming model for processing large data sets with a parallel, distributed algorithm on a cluster. Use of MapReduce is an important way to process big data when applied to a NoSQL database. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. The run-time system takes care of the details of partitioning the input data, scheduling the program’s execution across a set of machines, handling machine failures, and managing the required inter-machine communication. (Jeffrey Dean and Sanjay Ghemawat, 2008)

In today’s non-stop, interconnected world, companies get ever-increasing volumes of data, and use it in increasingly complex ways; both trends that require new paradigms in order to enable quick and accurate analysis and action. NoSQL database is definitely playing a critical role in meeting the companies’ needs. In financial fields, companies are internally developing more and more applications to interact with their
customers, suppliers, or other stakeholders. They could be revenue-producing applications that will become an increasingly important part of their business or non-revenue producing applications that make their supplier and customer interaction more efficient. And these applications require the type of performance and scalability that NoSQL provides. These applications are complex and may store session information, user generated data, products pricing data, user likes/dislikes, market data, sensor data, telematics data, etc. and increasingly require the agility and capabilities of NoSQL.

2.3 The Major Differences Between NoSQL and Relational Database Management System (RDBMS)

Fundamentally speaking, both NoSQL databases and relational databases are an organized collection of data organized to model relevant aspects of reality, in a way that supports processes utilizing this information. They are designed to work with large quantities of information including inputting, storing, retrieving, and managing that information and are set up so that one set of software programs provides all users with access to all the data. The main concept relating to Database Management System (DBMS) is that it is an organized way of managing, retrieving and storing data and a suite of computer software providing the interface between users and a database or databases.

NoSQL database and relational database have the same functions as all databases should have, but they also have some unique features. These differences make these two kinds of database useful for different fields.
When the set of data items are organized with the help of formally described tables, this kind of the database formed is known as a relational database. It can easily be accessed, created and extended. Because of this, it is the predominant choice for storing data using a hierarchical model or network model. (Nishtha Jatana et al, 2012) For the past 40 years, relational databases have been the most popular in the data world. Relational models first appeared in the early 1970s. Early versions of SQL-like languages were also developed in the early 70s, with modern relational databases appearing in the late 1970s, and becoming popular by the mid-1980s. (Jeff Cogswell, 2012) It was a good fit for the architectures of the time, and it soon became a standard interface to different database packages. To be sure, compatibility has never been great - but it’s been an issue of adapting to different dialects, rather than rewriting the entire application. A classic relational database (RDB) can be described as a table-based data system, but there is minimal data duplication and sets of data can be accessed through a series of relational operators like joins and unions. The problem with such relations is that complex operations with large data sets quickly become prohibitively resource intense, although generally the benefits are reaped at the application level where database code need not be convoluted.

When traditional tables are not used to store data, it is known as a non-relational database. These databases may store the data in the form of key-value stores, XML format, multidimensional databases and so on. A non-relational database is commonly known as NoSQL. As the data rapidly grows, NoSQL databases become more and more widely used as a non-relational database. Non-relational data stores could provide for web-scale data storage and retrieval, because it views the data more closely to how web
apps view data, i.e. a key/value hash in the sky.

The major difference between these two types of database is that most NoSQL databases are simple “key-value stores” instead of having tables with columns and rows as you would find in a traditional RDBMS. Each piece of data that goes into the database is given a key, and you use the key to get it back when you want to retrieve the data. This simplicity is beneficial, because it helps busy sites achieve extremely low latency, even under high load, when paired with a large number of servers and a fast network. The simplicity of the key-value model also simplifies development. (Daniel Bartholomew, 2010)

Here is a simple comparison between NoSQL and relational Databases:

Relational Database

- Pre-defined Schema
- Standard definition and interface language
- Tight consistency
- Well-defined semantics.

NoSQL Database:

- No predefined Schema
- Per-product definition.
- Interface language
- Getting an answer quickly is more important than getting a correct answer.

3. Selection
In this section I will discuss my choice of financial data, the selection of database category and specific NoSQL databases. Big data infrastructures tend to demand high processing, namely input/output operations per second (IOPS) performance and very large capacity. Speed, reliability, scalability, cost-effectiveness, efficiency, latency and uptime considerations are major selection criteria when it comes to selecting the right database. The financial data selection is based on easy access to view and download real-time data, popularity, and abundant attributes to allow retrieval of complex financial data. The selection of database category is based on its features and potential performance to store and query chosen data, as well as its cost, which includes price, equipment expense, and storage cost. The selection of a specific NoSQL database is based on its features and potential performance to store and query chosen data, as well as its cost including initial price, setup cost and labor cost.

3.1 Financial Data Selection

There isn’t a clear definition of financial data. For a corporation or other large entity, the term "financial data" refers to information regarding performance in terms of income, expenses, profits, revenues, operating income, etc. usually over the course of a full fiscal year, when accompanied by discussion of significant events that have affected performance. For an individual or small business, the term "financial data" refers to bank account information, debts, assets, and credit ratings. An example is a state agency that must collect financial data for administering programs such as Medicaid.

However, I used the term ‘financial data’ here referring to all kinds of data related to financial companies, financial markets, and financial performance, which could include
but is not limited to regional economic data, foreign exchange data, equity and equity indices, fixed income, options and implied volatility, futures, and commodities like precious metals. Also included are company financials such as the documentation of the charges for services, costs of providing services, revenues generated from services and revenues from other sources etc.

In a world of exploding financial information, we consume ever-increasing volumes of data, and use it in increasingly complex ways. In the mid-1990s, the NYSE produced half a million trades and quotes per day; today this is typically half a billion per day with peaks of over two billion records per day. As more and more transactions are conducted electronically the volumes of data are increasing and frequently amount to billions of records per day. Today’s financial institutions need to be able to collect, analyze and store vast quantities of data. Institutions must be able to handle high ongoing data volumes, high-volatility market events and trillions of historical records while satisfying increasingly tight compliance and regulatory requirements.

After researching the topic, I found there was limited access to big real financial data one could download at no cost to use for experimental purposes. Most online resources only provide a summary of data, or small data samples. Data can be searched and viewed but not downloaded for free as one large file. Yahoo! Finance provides Exchange Traded Fund (ETF), mutual fund, and stock data to search and view, and download in real time, as well as historical stock data in CSV format. Historical end of day (EOD) data could be downloaded by entering the Stock Symbols, but only with 7 columns which are too simple to model using either RDBMS or NoSQL. In contrast, real
time stock data can be downloaded with multiple attributes to choose from (See Table 1 below)

<table>
<thead>
<tr>
<th>a</th>
<th>Ask</th>
<th>a2</th>
<th>Average Daily Volume</th>
<th>a5</th>
<th>Ask Size</th>
<th>b</th>
<th>Bid</th>
</tr>
</thead>
<tbody>
<tr>
<td>b2</td>
<td>Ask (Real-time)</td>
<td>b3</td>
<td>Bid (Real-time)</td>
<td>b4</td>
<td>Book Value</td>
<td>b6</td>
<td>Bid Size</td>
</tr>
<tr>
<td>c</td>
<td>Change &amp; Percent Change</td>
<td>c1</td>
<td>Change</td>
<td>c3</td>
<td>Commission</td>
<td>c6</td>
<td>Change (Real-time)</td>
</tr>
<tr>
<td>c8</td>
<td>After Hours Change Real-time</td>
<td>d</td>
<td>Dividend/Share</td>
<td>d1</td>
<td>Last Trade Date</td>
<td>d2</td>
<td>Trade Date</td>
</tr>
<tr>
<td>e</td>
<td>Trade Date</td>
<td>e1</td>
<td>Error Indication (returned for symbol changed invalid)</td>
<td>e7</td>
<td>EPS Estimate Current Year</td>
<td>e8</td>
<td>EPS Estimate Next Year</td>
</tr>
<tr>
<td>e9</td>
<td>EPS Estimate Next Quarter</td>
<td>f6</td>
<td>Float Shares</td>
<td>g</td>
<td>Day's Low</td>
<td>h</td>
<td>Day's High</td>
</tr>
<tr>
<td>j</td>
<td>52-week Low</td>
<td>k</td>
<td>52-week High</td>
<td>g1</td>
<td>Holdings Gain Percent</td>
<td>g3</td>
<td>Annualized Gain</td>
</tr>
<tr>
<td>g4</td>
<td>Holdings Gain</td>
<td>g5</td>
<td>Holdings Gain Percent (Real-time)</td>
<td>g6</td>
<td>Holdings Gain (Real-time)</td>
<td>i</td>
<td>More Info</td>
</tr>
<tr>
<td>i5</td>
<td>Order Book (Real-time)</td>
<td>j1</td>
<td>Market Capitalization</td>
<td>j3</td>
<td>Market Cap (Real-time)</td>
<td>j4</td>
<td>EBITDA</td>
</tr>
<tr>
<td>j5</td>
<td>Change From 52-week Low</td>
<td>j6</td>
<td>Percent Change From 52-week Low</td>
<td>k1</td>
<td>Last Trade (Real-time) With Time</td>
<td>k2</td>
<td>Change Percent (Real-time)</td>
</tr>
<tr>
<td>k3</td>
<td>Last Trade Size</td>
<td>k4</td>
<td>Change From 52-week High</td>
<td>k5</td>
<td>Percent Change From 52-week High</td>
<td>l</td>
<td>Last Trade (With Time)</td>
</tr>
<tr>
<td>l1</td>
<td>Last Trade (Price Only)</td>
<td>l2</td>
<td>High Limit</td>
<td>l3</td>
<td>Low Limit</td>
<td>m</td>
<td>Day's Range</td>
</tr>
<tr>
<td>m2</td>
<td>Day's Range (Real-time)</td>
<td>m</td>
<td>50-day Moving Average</td>
<td>m</td>
<td>200-day Moving Average</td>
<td>m5</td>
<td>Change From 200-day Moving Average</td>
</tr>
<tr>
<td>m6</td>
<td>Percent Change From 200-day Moving Average</td>
<td>m</td>
<td>Change From 50-day Moving Average</td>
<td>m</td>
<td>Percent Change From 50-day Moving Average</td>
<td>n</td>
<td>Name</td>
</tr>
<tr>
<td>n4</td>
<td>Notes</td>
<td>o</td>
<td>Open</td>
<td>p</td>
<td>Previous Close</td>
<td>p1</td>
<td>Price Paid</td>
</tr>
<tr>
<td>p2</td>
<td>Change in Percent</td>
<td>p5</td>
<td>Price/Sales</td>
<td>p6</td>
<td>Price/Book</td>
<td>q</td>
<td>Ex-Dividend Date</td>
</tr>
<tr>
<td>r</td>
<td>P/E Ratio</td>
<td>r1</td>
<td>Dividend Pay Date</td>
<td>r2</td>
<td>P/E Ratio (Real-time)</td>
<td>r5</td>
<td>PEG Ratio</td>
</tr>
<tr>
<td>r6</td>
<td>Price/Price Estimate Current Year</td>
<td>r7</td>
<td>Price/EPS Estimate Next Year</td>
<td>s</td>
<td>Symbol</td>
<td>s1</td>
<td>Shares Owned</td>
</tr>
<tr>
<td>s7</td>
<td>Short Ratio</td>
<td>t1</td>
<td>Last Trade Time</td>
<td>t6</td>
<td>Trade Links</td>
<td>t7</td>
<td>Ticker Trend</td>
</tr>
<tr>
<td>t8</td>
<td>1 yr Target Price</td>
<td>v</td>
<td>Volume</td>
<td>v1</td>
<td>Holdings Value</td>
<td>v7</td>
<td>Holdings Value (Real-time)</td>
</tr>
<tr>
<td>w</td>
<td>52-week Range</td>
<td>w1</td>
<td>Day's Value Change</td>
<td>w4</td>
<td>Day's Value Change (Real-time)</td>
<td>x</td>
<td>Stock Exchange</td>
</tr>
<tr>
<td>y</td>
<td>Dividend Yield</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 1.** Special tags for real time stock data attributes (retrieved from http://www.gummy-stuff.org/Yahoo-data.htm)

In conclusion, in order to discuss database selection and run a sample test, real time stock data would be the most appropriate choice for now.

The unique identifier for stock data is the stock symbol which is assigned to each security traded on a particular market. Every symbol corresponds to a unique company’s
full name. For example, AAPL is for Apple Inc.; OODH is for ORION DHC, Inc., and HD is for Home Depot, Inc. But a stock symbol isn’t fixed forever. For example, it could be changed when there is a merger of two companies, and a new third symbol is selected for the combined entity. One solution is to keep track of every symbol’s effective date and expiration date. This will result in extra column spaces but make more sense to users. Another solution is to update the ticker symbol after the change is approved by the stock exchange. This will save the extra space, but users may not be able to find the historical stock if they don’t know the new symbol.

For real time stock data, time will be another critical attribute. Users may search for the price change over certain time periods to grasp the trend of a certain stock. This would include date and exact time when the data are produced. An issue could arise with respect to the format used to store the date and time. Some databases store time formats as the number of seconds since an arbitrary date. For Linux systems this is 00:00:00 Coordinated Universal Time (UTC), Thursday, 1 January 1970. Storing data from before 1970 would cause problems as a result. If a negative time were given then the lexicographic order would sort these times backwards. It is much easier and makes more sense to store date in a simple string format as YYYYMMDD (Year, Month, Day), time as HHMMSS (Hour, Minute, Second) in 24-hour clock. Problems can still arise with data that are produced on a daily basis instead of several times a day like the EOD data. Rather than storing them with the time of the EOD, these readings like summary reports could be put in separate tables that only encode date. What’s more, these summary readings could be a result of calculations of all the records during a day such as highest, lowest and total volume. A user’s sample needs to include stock data analyzing the
current trend or perform a range query on price, volume, or any other important factor to identify a stock to purchase. Consider a fundamental factor in analyzing NASDAQ Dozen, the earnings per share (EPS) as an example. EPS represents how much money the company is making in profits per every outstanding share of stock. The higher the EPS is, the more money your shares of stock will be worth because investors are willing to pay more for higher profits. So users will need to check if the EPS are increasing from a certain date to now. Clearly, most financial attributes are numerical.

3.2 Database Category Selection

3.2.1 Relational Database

Relational database has taken over the major information storage and retrieval market ever since it was invented in 1970 due to its strong functionality. The relational model is powerful in terms of reliability, flexibility, robustness, and scalability requirements. Most of the information is stored in the database rather than the application, so the database is self-documenting. It is easy to add, update or delete data. It has the benefit of convenient data summarization, retrieval and reporting. The database is structured in a tabular form with highly related tables; the nature of the database is predictable. Also, it is quite simple to make changes to the schema of the database if necessary. (Nishtha Jatana et al, 2012) Furthermore, relational databases have the advantage of availability and familiarity. SQL Server, MySQL, Oracle, Microsoft Access or some other relational databases are usually already installed at many companies. They offer a common interface, Open Database Connectivity (ODBC), and most IT professionals are familiar with them. It's no surprise that relational databases are being
used for most data processing, particularly when the requests for data come from management personnel familiar with this kind of database.

However a relational database may not be the ideal choice for big financial data especially when they are non-relational, or users demand faster speed, less cost and huge volume. With respect to cost, many of the features provided by relational database are not used, hence simply adding to the cost as well as the complexity of the database. In addition, the structure of the relational database where data is stored in form of tables can give rise to high complexity in cases where data cannot be easily encapsulated in a table. Finally, relational database does not support high scalability; until a certain point better hardware can be employed but beyond that point the database must be distributed. When the amount of data turns huge the database has to be partitioned across multiple servers, and this partitioning can pose several problems because joining tables in distributed servers is not an easy task. (Nishtha Jatana et al, 2012)

Despite the performance advantages of other databases, some companies opt for using relational databases for storing and processing financial data. This is completely understandable, since those are the tools that company IT staff and upper management are most familiar with for now. But software with complex logic or business rules and high transaction volume for time series data are not practical with relational database management systems. Queries for historical data, replete with time ranges and roll ups and arbitrary time zone conversions are difficult in a relational database. Compositions of those rules are even more difficult. This is a problem compounded by the free nature of relational systems themselves. Many relational systems are often not modeled correctly with respect to time series data.
Thus, there was a compelling need to store and access huge data in a better way, giving rise to a newer form of data storage known as NoSQL.

3.2.2 NoSQL Database

NoSQL database is an approach to data management and database design useful for very large sets of distributed data. NoSQL, which encompasses a wide range of technologies and architectures, seeks to solve the scalability and big data performance issues that relational databases weren’t designed to address. NoSQL is especially useful when an enterprise needs to access and analyze massive amounts of unstructured data or data that's stored remotely on multiple virtual servers in the cloud.

In contrast to relational database, NoSQL systems generally have six key features (Rick Cattell, 2010)

- The ability to horizontally scale “simple operation” throughput over many servers
- The ability to replicate and to distribute/partition data over many servers
- A simple call level interface or protocol (in contrast to a SQL binding)
- A weaker concurrency model than the ACID transactions of most relational (SQL) database systems
- Efficient use of distributed indexes and RAM for data storage
- The ability to dynamically add new attributes to data records

All these six features make NoSQL stands out over relational database in terms of handling big financial data. I will expand the discussion of specific databases in detail later in Section 3.3 NoSQL database selection.
3.2.3 Real Time Database

Strictly speaking, real-time database is only one of a number of specialized databases one could consider. I discuss it separately here because it appears to provide the best fit for addressing real time stock data.

A real-time database is a processing system designed to handle workloads whose state is constantly changing. (Buchmann et al, 2009) This differs from traditional databases containing persistent data, mostly unaffected by time. For example, a stock market can change very rapidly and is dynamic. The graphs of the different markets appear to be very unstable, and yet a database has to keep track of current values for all of the markets of the New York Stock Exchange. (Kanitkar et al, 1997) Real-time processing means that a transaction is processed fast enough for the result to come back and be acted upon right away. (Carpron and Johnso, 2001) Real-time databases are useful for accounting, banking, law, medical records, multi-media, process control, reservation systems, and scientific data analysis. They use timing constraints that represent a certain range of values for which the data are valid. This range is called temporal validity. A conventional database cannot work under these circumstances because the inconsistencies between the real world objects and the data that represents them are too severe for simple modifications.

These specialized database systems are often used for time series storage, particularly for tick data. For example, Kdb+ is said to the world’s leading market data solution for large financial institutions. It supports analysis and monitoring of both real-time and historical data on a single platform. This means not only the analysis of billions of real-time ticks, but also fast access to terabytes of historical data through the product’s
expressive query language, q. It is a column-oriented database based on the K language. It runs on all industry standard 64-bit server platforms, has a small footprint which makes for simple and straightforward installation, and incorporates a simple API for easy connectivity to external systems and modules. Another advantage is that Kdb+ allows virtually unlimited room to grow. It is currently adopted by Goldman Sachs, Morgan Stanley, Merrill Lynch +, J.P. Morgan, NYSE/Euronext, Deutsche Bank, Commerzbank and some other respected names in finance. Another example would be Onetick. It is said to be the premier enterprise-wide solution for tick data collection, management and research. It captures, compresses, archives and provides uniform access to global historical data, up to and including the latest tick. OneTick has no limitations on data volumes, peak rates or length of stored history, and it collects every tick for all asset class types including equities, fixed income, futures, FX and options, as well as full order book data. OneTick’s powerful analytical tools enable clients to run historical simulations and back-tests, develop trading and market-making strategies, build transaction-cost models, perform real-time surveillance and adhere to regulatory compliance requirements. With its superior features and unmatched functionality, OneTick is being embraced enthusiastically by leading hedge funds, mutual funds, banks, brokerages, market makers, data vendors and exchanges. Goldman Sachs adopts it. Other specialized databases are VoltDB, SciDB, TickBase, Vertica etc. 

All above specialized databases are fast in both storing and retrieving, can work with terabytes of data. They are designed specifically to address big financial data problems according to features of the financial data, but they are commercial products and the commercial offerings tend to be rather expensive.
3.2.4 Combination

Adopting a combination SQL/NoSQL strategy makes it possible to produce a well-functioning product. To have both NoSQL and SQL components in an effective big data management strategy will definitely result in well-tuned data. Some databases are designed to solve a single problem or for a certain kind of data. Some databases are for general use but integrated with as much functionality as possible. The general database loses advantage in terms of a specific problem but has more functionality, while a specialized database can only handle one specific problem very well. To balance the database performance and cost, a general database combined with a specialized database would be a good choice for an institution with special needs. However, this choice won’t be further discussed in my research.

3.3 NoSQL Database Selection

NoSQL represents the next generation databases, being non-relational, distributed, open-source and horizontally scalable. There have been various approaches to classify NoSQL databases, each with different categories and subcategories. Because of the variety of approaches and overlapping requirements regarding the nonfunctional aspects and the feature-set, it could be difficult to acquire an overview of the non-relational database market. According to http://nosql-database.org/, it typically includes the following categories:

- Wide Column Store/Column Families
- Document Store
- Key Value/Tuple Store
• Graph Database
• Multimodel Databases
• Object Database
• Grid & Cloud Database Solutions
• XML Databases
• Multidimensional Databases
• Multivalue Databases
• Event Sourcing
• Some Other NoSQL Related Databases

I will discuss the four most popular categories based on these features in the sections that follow.

3.3.1 Key–Value Store

These systems store values and an index to find them, based on a programmer-defined key. It is the simplest method of data storage which uses a data model similar to the popular memcached distributed in-memory cache, with a single key-value index for all the data. Key–value stores allow the application to store its data in a schema-less way. The main idea here is using a hash table where there is a unique key and a pointer to a particular item of data. It allows the app-developer to store schema-less data. This data consists of a key which is represented by a string and the actual data which is the value in the key-value pair. The data can be a string, an integer, an array, or it can be an object. Thus it loosens the requirement of formatted data for storage, eliminating the need for a
fixed data model. (Nishtha Jatana et al, 2012) However, it is inefficient when you are only interested in querying or updating part of a value, among other disadvantages.

One of the most popular examples of this category is Riak. Some database professionals recommend using Riak for financial application as BankSimple used it. According to its official website, Riak is a distributed database designed for maximum availability: so long as your client can reach one server, it should be able to write data. It features high availability, operational simplicity, scalability and is masterless. Riak's availability focus makes it a good fit whenever downtime is unacceptable. This is critical to financial data management. However, if the data simply cannot be effectively managed as keys and values, Riak will most likely not be the best fit. While Riak offers ways to find values that match certain criteria, if your application demands a high query load by any means other than the keys, Riak will not be as efficient as other databases. Riak does not support indices on any fields except the primary key. The only thing you can do with the non-primary fields is fetch and store them as part of a JSON object; the only lookup you can do is on primary key. This is unacceptable for stock data retrieval.

Key-value stores are generally good solutions if you have a simple application with only one kind of object, and you only need to look up objects up based on one attribute. (Rick Cattell, 2010) But for my selection of data test it is not acceptable, although it is the simplest and easiest to implement using the primary key.

3.3.2 Document Store

These systems store documents, as just defined. The documents are indexed and a simple query mechanism is provided. A document-oriented database is a computer
program designed for storing, retrieving, and managing document-oriented information, also known as semi-structured data. In contrast to relational databases and their notions of Relations or Tables, these systems are designed around an abstract notion of a Document. It is similar to key-value store. The model is basically versioned documents that are collections of other key-value collections. The semi-structured documents are stored in formats like JSON. Document databases are essentially the next level of key/value, allowing nested values associated with each key. Document databases support querying more efficiently. The underlying storage structure used in such databases is a ‘document’. Each document store differs in its implementation of data; however each assumes that data is enclosed and encoded in some standard format which may be XML, BSON, PDF or Microsoft office. Each document is represented by a unique key which is a string (URI or path). An API or a query language is provided for fast retrieval of documents on the basis of its content. For instance, a query that retrieves all the documents in which certain fields are set to some particular value. For institutions that need to store all kinds of documents in a database such as company financial reports, it would be the best fit among NoSQL database. But it cannot be the proper choice for stock data since stock data are mostly numeric.

MongoDB is an outstanding example of this category, and it is frequently recommended by professionals who have used it. MongoDB is a general purpose, open-source database. It features: Document data model with dynamic schemas; full, flexible index support and rich queries; auto-sharding for horizontal scalability; built-in replication for high availability; text search; advanced security; Aggregation Framework and MapReduce; large media storage with GridFS. However, the downside is that it is
very unreliable, and no single server durability is available. If something crashes while it's updating 'table-contents', you lose all the data, and reliability critical for most financial data. In addition, its indexes take up a lot of RAM. They are B-tree indexes and if you have many, you can run out of system resources really fast. Thus, while MongoDB's software itself is free, the product is weighed down with management and support costs.

A good example application for a document store would be one with multiple kinds of objects (say, in a Department of Motor Vehicles application, with vehicles and drivers), where you need to look up objects based on multiple fields (say, a driver's name, license number, owned vehicle, or birth date). (Rick Cattell, 2010) So it will be great for financial data including lots of reports or other documents rather than numerical stock data.

3.3.3 Graph Store

A graph database is a schema-less database that uses graph structures with nodes, edges, and properties to represent and store data. Nodes may represent entities like people, businesses, or any other item similar to what objects represent in any programming language. Properties designate any pertinent information related to nodes. On the other hand, edges relate a node to other nodes or a node to some property. One can obtain some meaningful pattern or behavior after studying the interconnection between all three viz. nodes, properties and edges. (Nishtha Jatana et al, 2012) By definition, a graph database is any storage system that provides index-free adjacency. This means that every element contains a direct pointer to its adjacent element and no index lookups are necessary.
General graph databases that can store any graph are distinct from specialized graph
databases such as triplestores and network databases. Instead of tables of rows and
columns and the rigid structure of SQL, a flexible graph model is used which, again, can scale across multiple machines. But it has to traverse the entire graph to achieve a definitive answer. Not easy to cluster. My choice of financial data doesn’t have such complex relationships, so it isn’t a good fit for this test.

3.3.4 Column Store

These systems store extensible records that can be partitioned vertically and horizontally across nodes. A column-oriented DBMS is a database management system (DBMS) that stores data tables as sections of columns of data rather than as rows of data. In comparison, most relational DBMSs store data in rows. It is created to store and process very large amounts of data distributed over many machines. There are still keys but they point to multiple columns. The columns are arranged by column family. It has advantages for data warehouses where aggregates are computed over large numbers of similar data items. The goal of a column store database is to efficiently write and read data to and from hard disk storage in order to speed up the time it takes to return a query. One of the main benefits is that data can be highly compressed. The compression permits columnar operations, like MIN, MAX, SUM, COUNT and AVG, to be performed very rapidly. This is exactly what will frequently be queried for stock data. For example, the calculations of average daily volume, 50-day average volume, and the highest price and lowest price etc. Another benefit is that because a column-based DBMS is self-indexing, it uses less disk space than a RDBMS containing the same data. Compared to row
oriented databases, it’s more efficient when an aggregate needs to be computed over many rows but only for a notably smaller subset of all columns of data, because reading that smaller subset of data can be faster than reading all the data. We can take this advantage to perform range query on certain column(s). It is also more efficient when new values of a column are supplied for all rows at once, because that column data can be written efficiently and replace old column data without touching any other columns for the rows. We can use this advantage to update the highest price or lowest price of the real time stock data frequently.

The best examples of this category are Cassandra and HBase/Hadoop. HBase is an open-source, distributed, versioned, column-oriented store modeled after Google's Bigtable. Just as Bigtable leverages the distributed data storage provided by the Google File System, Apache HBase provides Bigtable-like capabilities on top of Hadoop and HDFS. It is a wonderful choice for accessing random, real-time read/write big data. Cassandra is a massively scalable open source distributed database system, designed by Facebook for storing and managing large amounts of data across multiple servers. It was once described as a BigTable data model running on an Amazon Dynamo-like infrastructure. Cassandra can serve as both a real-time operational data store for online transactional applications and a read-intensive database for large-scale business intelligence (BI) systems. They both provide high availability, scalability and MapReduce technique among multiple data centers. HBase has two kinds of servers, i.e. master server and region server. The master server is responsible for monitoring all region server instances in the cluster, and is the interface for all metadata changes. There could be multi-master servers where all masters compete to run the cluster to ensure
master’s function if any of it shuts down. The region server holds actual data and communicates with clients. In contrast, Cassandra has peer-to-peer symmetric nodes, instead of master or named nodes, to ensure there can never be a single point of failure (SPoF) because very node in the cluster has the same role. Data is distributed across the cluster so each node contains different data, but there is no master as every node can service any request. Hbase has consistency while Cassandra’s consistency is tunable based on user’s need. Hbase provides better performance of range scan, but it only allows range query on the row key and only allows one range per query. Cassandra allows all of the range queries that are necessary to effectively analyze stock data by creating a secondary index. Besides, based on my experience, Cassandra is much easier to set up than Hbase even without solid computer skills. According to other professional’s experience, Cassandra is a simpler implementation and much easier to hack. HBase by comparison is much more complicated and harder to debug and hack. So adopting Hbase would require more efforts on setup or administration and more professional IT staff familiar with Hbase, Hadoop and Hadoop Distributed File System (HDFS).

Based on all above analysis, Cassandra stands out for satisfying all my current needs to store and manage stock data.

3.4 Database Design

I will design a data model for stock data in this section. First, I will design a traditional relational schema for RDBMS to compare with the model design for Cassandra to illustrate the difference between relational store and NoSQL store. Then I will give a full description of the data model designed in Cassandra.
One main concept that is critical to both relational and NoSQL databases is the primary/row key. It is designed to uniquely define the characteristics of each row (also known as record or tuple). The major features are uniqueness and permanence. The key needs to be unique so that a query will only find the relevant information and permanent so that if aspects of your data change over time it will not be split and stored in separate places. For relational schema design, we could simply use the stock symbol as the primary key of basic information of a stock including symbol, exchange name, company full name, description etc. which are relatively fixed information. This is a model for one exchange market; if stocks from multiple exchange markets are stored together, then the primary key would be a combination of stock symbol and exchange name. All the real time attributes will be included in the real time table which will be frequently added or updated. This table contains a composite primary key of symbol, date and time. The other table is EOD data identified by symbol and date. It will be the result of calculations of real time records from the real time table. One symbol will have multiple records of real time data and EOD data. See Figure 1.
Figure 1. Relational Schema Design of Stock Data

For Cassandra, most records will be stored in one big table, although the stock symbol can uniquely identify publicly traded shares of a particular stock on a particular stock market. It will contain multiple records in a big table so it cannot uniquely identify a record. One consideration is whether or not to use a surrogate key for a table. A surrogate key is a generated key (such as a UUID) that uniquely identifies a row, but has no relation to the actual data in the row. It would be a good choice if no other proper row key is available. The disadvantage is that we cannot perform a range query directly on the row key to speed up the query. Another consideration is whether or not to use a timestamp as a row key. It is unique since it is assigned and stored when a piece of data is inserted. The timestamp is mostly used to resolve write conflicts and for garbage collection. We can take advantage of the automatic garbage collection by versions,
setting that the latest version of a cell be kept. For example, keep the latest highest/lowest price for EOD use so we don’t need to compare all data again in the EOD. Another professional once performed an experiment on using the timestamp to store the time dimension, but it was only designed to hold a small number of values. Therefore, it is not a good idea to use it as a row key. Combining the stock symbol, date and time would the best row key I can think of so far. It can uniquely identify a record and we can perform equation query on it if it is Symbol_DateTime; we can even perform range query on the row key if it is DateTime_Symbol using Order Preserving Partitioner (OPP) provided by Cassandra. However, if we adopt OPP, and select DateTime_Symbol as row key, it is very likely to overload some servers because it is ordered by time and will insert all new data to one server at one time. This would also require that the load balancing be watched carefully by the database administrator, thus incurring extra cost. Besides, the columns should be grouped into column families to speed up query based on the data features. The column family is similar to a table in that it is a container for columns and rows. (Apache Cassandra 0.8 Documentation) See Figure 2 for the final data schema design for Cassandra.
Figure 2. Cassandra Schema Design of Stock Data

As mentioned above, OPP will require extra cost and run the risk of overloading one server, so RandomPartitioner (RP) should be adopted for partition. The RP distributes data evenly across the nodes using an MD5 hash value of the row key.

Another feature provided by Cassandra is super column. A super column is a way to group multiple columns based on a common lookup value. The primary use for super columns is to de-normalize multiple rows from other column families into a single row, allowing for materialized view data retrieval. For example, suppose you wanted to create a materialized view of blog entries for the bloggers that a user follows. (Apache Cassandra 0.8 Documentation) Nevertheless, any request for a sub-column deserializes all sub-columns for that super column, so my data model that relies on large numbers of sub-columns isn’t applicable. Furthermore, in order to perform range query, secondary
indexes need to be created on specific columns that need to be queried. Secondary indexes allow for efficient querying by specific values using equality predicates (where column x = value y). Also, queries on indexed values can apply additional filters to perform operations such as range queries. (Apache Cassandra 0.7 Documentation) But this should be approached with caution. Every index created will require extra storage space and slow queries down, so only necessary indexes should be created based on the users’ need.

Another feature worth discussing is MapReduce. Cassandra supports running Hadoop MapReduce jobs against the Cassandra cluster. In a properly configured cluster, MapReduce jobs can retrieve data from Cassandra and then output results either back into Cassandra, or into a file system. For instance, if we want to get the 50-day average volume of all stock, the mapper function is applied in parallel to every pair in the input dataset and extracts a list of all the stock symbols along with its EOD volume within 50 days. After that, the MapReduce framework collects all pairs with the same symbol from the list and groups them together, creating one group for each symbol. The reducer function then applied in parallel to each group produces a collection of the average volume of each group. This feature can be adopted to faster count, compare, or average based on actual user needs.

4. Experiment

In this section, I will describe the data, software and system I used for my experiment, as well as the steps, results and analysis of the experiment.

4.1 Data sample
The five files of sample data used in this experiment were generated from a self-written Java program. All attributes were in accordance with the real time stock data from Yahoo! Finance. For example, I made the price for last trade a random float number with two decimal places between 1.00 and 100000.00, because according to Wikipedia, a US share must be priced at $1 or more to be covered by NASDAQ. If the share price falls below that level the stock is "delisted", and becomes an OTC (over the counter) stock. A stock must have a price of $1 or more for 10 consecutive trading days during each month to remain listed. Furthermore, the highest share prices on the NYSE have been those of Berkshire Hathaway class A, trading at over $100,000/share. The Java program was set to generate five files of real time stock data for 10, 100, 1000, 10000, 100000 companies every five minutes from 10:00:00 to 15:00:00 every day beginning from January 2\textsuperscript{nd}, 2010 and ending on January 11\textsuperscript{th}, 2010, which accumulated to 6,100, 61,000, 610,000, 6,100,000, 61,000,000 lines of records with 28 attributes. For a more detailed look at the specific fields that were used, see Appendix A.

4.2 System

All experiments were performed on a single computer installed with Cassandra. The sample tests were run with the most current stable versions of Cassandra in the beginning of the experiment: Cassandra version 1.2.6. The computer had an Intel Core i5 2.3 GHZ processor, 4 GB of RAM, 320 GB hard drive, and running on Mac OS X Lion 10.7.5. The computer was used as a server. I used Terminal on MacBook Pro to connect to the server installed on the same computer. All tables and keyspaces in Cassandra were
created through the Cassandra CLI, while data transformation and insertion was done with Kettle, a type of open source software also known as Pentaho Data Integration.

4.3 Sample Test

There were five different tests performed to evaluate the performance of Cassandra: data insertion, query by row key only, query by one secondary index, query by two secondary index and query by three secondary indexes. For data insertion, Kettle was used to transform the data from text files and insert it into the keyspaces created in Cassandra. The five files of sample data consisted of 6,100, 61,000, 610,000, 6,100,000, 61,000,000 lines of records individually along with 170,800, 1,708,000, 17,080,000, 170,800,000, 1,708,000,000 data values. For data insertion, each file was inserted three times and averaged. For other tests, queries were run through the Cassandra CLI to access different files of stock data of different size. Each individual query was run ten times and averaged. All query speeds were measured with the Cassandra CLI and the insertion speeds were measured using Kettle software.

4.4 Results and Discussion

4.4.1 Data Insertion

The insertion test results indicated the insertion time had high positive relativity with the amount of records. Cassandra handled all the observations within an acceptable time except for the biggest file with 610,000,000 records. But the extremely slow speed of inserting the biggest data was mainly because the computer system performance was getting worse after long time of continual running. See Table 2 below for specific time cost for different files.
Another thing needs to be mentioned is that one earlier experiment took more than 10 hours to insert around 200,000 records into a standalone version of HBase. Based on his analysis, it was the result of his limited knowledge of tuning HBase. Although this result couldn’t be used to compare the write speed between Cassandra and HBase, it does explain a reason to choose Cassandra over other databases because it’s obviously easier and simpler to achieve an acceptable performance.

### 4.4.2 Querying with Row Key

Certain record was chosen by row key to display all its values. See Table 3 for test results.

<table>
<thead>
<tr>
<th>Amount of Records</th>
<th>6100</th>
<th>61000</th>
<th>610000</th>
<th>6100000</th>
<th>61000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>6.75ms</td>
<td>6.15ms</td>
<td>7.18ms</td>
<td>6.03ms</td>
<td>6.72ms</td>
</tr>
</tbody>
</table>

**Table 2. Data Insertion Time**

<table>
<thead>
<tr>
<th>Amount of Records</th>
<th>6100</th>
<th>61000</th>
<th>610000</th>
<th>6100000</th>
<th>61000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Times</td>
<td>File Input</td>
<td>0.9s</td>
<td>55.0s</td>
<td>11min 1s</td>
<td>1h 49min 24s</td>
</tr>
<tr>
<td></td>
<td>Cassandra</td>
<td>8.6s</td>
<td>1min 5s</td>
<td>11min 11s</td>
<td>1h 49min 35s</td>
</tr>
</tbody>
</table>

There weren’t distinct differences among the results of querying different size of files. We can simply conclude that querying on row key of a table in Cassandra is fast and convenient, regardless of the size of the table.

**Table 3. Query with row key.**
4.4.3 Querying with One Secondary Index

Certain stock was chosen by stock symbol to display all its records. Secondary indexes need to be created on stock symbol for indexing other than row key. See Figure 5 for test results.

<table>
<thead>
<tr>
<th>Amount of Records</th>
<th>6100</th>
<th>61000</th>
<th>610000</th>
<th>6100000</th>
<th>61000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>34.6ms</td>
<td>27.9ms</td>
<td>27.3ms</td>
<td>31.4ms</td>
<td>35.9ms</td>
</tr>
</tbody>
</table>

Table 4. Query with one secondary index.

There weren’t obvious releativity between the speed and the amount of records. With a secondary index, Cassandra ran a little slower than indexing on row key regardless how many records were searched. But it was within an acceptable time frame as it was run on a standalone machine. We can expect it to perform much better on a real distributed system.

4.4.4 Querying with Two Secondary Indexes

Certain stock was chosen to display all its records over a certain periods of time. Secondary indexes need to be created on stock symbol and last trade date. See Figure 6 for test results.

<table>
<thead>
<tr>
<th>Amount of Records</th>
<th>6100</th>
<th>61000</th>
<th>610000</th>
<th>6100000</th>
<th>61000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>136ms</td>
<td>877ms</td>
<td>6834ms</td>
<td>26235ms</td>
<td>411449ms</td>
</tr>
</tbody>
</table>

Table 5. Query with two secondary indexes.
With two secondary indexes, the speeds started to get much slower. While it is notable that Cassandra could perform more complex queries with secondary index, taking over six minutes to return a set of results would not be acceptable in any implementation. Originally the CLI was set to time out after 10 seconds on queries and that limit had to be raised forty times before a query would complete.

One secondary index already slowed down the searching speed. Adding more secondary index would only lead to taking more time for scanning and filtering different columns.

4.4.5 Querying by Three Secondary Indexes

A certain range of price was set to display all qualified stock records over a certain periods of time. Secondary indexes need to be created on last trade price, last trade date and exist. See Figure 7 for test results.

<table>
<thead>
<tr>
<th>Amount of Records</th>
<th>6100</th>
<th>61000</th>
<th>610000</th>
<th>6100000</th>
<th>61000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>382ms</td>
<td>3317ms</td>
<td>10931ms</td>
<td>41962ms</td>
<td>658098ms</td>
</tr>
</tbody>
</table>

**Table 6.** Query with three secondary indexes.

The performance of three secondary indexes was unacceptable either. It took even more time than querying with two secondary indexes as expected in 4.4.4. This degradation in performance was also a result of how Cassandra handled range queries on columns other than row key. Each query must contain at least one equality (‘=’) operation in order to execute any greater than/ less than (‘>/<’) operations. Cassandra scans through the table creating a list of all records that match the equality operations and filters those
results based on the greater than/less than operations. No part of selecting the range of qualified stock data query could satisfy the equality requirement, so a dummy variable was inserted into each record that was set to ‘OK’. Therefore, the real process of this query was that Cassandra scanned the entire table to create a list of all the records which were actually the whole records and then filtered down based on the price and date constraints. As a result, the more records needed to be scanned, the more time was used to return the results.

5. Limitation and Future Work

5.1 Limitation

There are a couple of major limitations to the sample tests performed in this experiment. First, I did not have access to acquire abundant real stock data with the kinds of necessary attributes for real time data. In addition, I am not an expert in the financial fields. I can only choose some important attributes from real time stock data provided by Yahoo! Finance, and I designed the sample query based on my research and limited knowledge of financial data. Which means, it can’t represent typical usage in all financial institutions. For actual professional users, they may have their own frequently used queries and choice of attributes. They need to run their own test to determine if certain database attributes can satisfy them. Secondly, I didn’t have access to proper equipment to set up a full-scale implementation of Cassandra. Cassandra is designed to distribute and run on multiple servers which are each much more powerful than the single computer that I used for test. Besides, while Cassandra does offer a stand-alone version of software to run on a single server, some of the more powerful functionality may be emulated or
ignored in the stand-alone version so the results could not give a complete or accurate representation of the capabilities of the system. Thirdly, I started this research without any background in NoSQL even though I did learn and use some relational databases. Due to my limited programing experience in Mac OS X environment, I failed to set up more databases on my computer to compare with the performance of Cassandra. It leaves the test results less convincing for comparison.

5.2 Future work

According to all limitation listed above, future study will include acquiring existing financial data to analyze and test; conducting interview or survey with financial professionals to know their real needs for managing financial data, then redesigning the data model and test queries based on their needs; setting up other proper databases on the same machine to compare their performance with each other.

6. Conclusion

Embracing new technologies like Hadoop, MapReduce and NoSQL databases in an effort to gain control over expanding volumes of big data so they can be mined for insights can lead to competitive advantages and other business benefits for financial institutions. Adopting NoSQL database to solve specific problems of big data is a must given current trends. In conclusion, based upon the above analysis, it is advantageous to apply NoSQL technology to store and manage financial data. It has been said that NoSQL database isn’t suitable for financial purposes because of its transactional nature. But based on my assumed usage of stock data, Cassandra challenges this assumption, and appears to be a good fit for stock data requiring real time transaction processing and
interactivity. Whether these decisions apply to the actual usage of stock data or other financial data will require that more analysis and experimentation be conducted before implementing any large part of the system. It should be noted is that if a certain kind of financial data is heavily relational, NoSQL should even be considered. Although NoSQL isn’t widely used to deal with financial data, many financial institutions have an incredible appetite for information and performance improvement which makes them a great candidate for the NoSQL approach. Most financial institutions are still very conservative, moving carefully and slowly forward, but I think a move in this direction is inevitable. Another new feature, Atomic batching, should help organizations that require transactional integrity across business processes, such as an online merchant that needs to make sure orders are captured even when a component such as a hard drive fails right in the middle of a transaction. Previously, developers would have to build processes, such as retry mechanisms, into their code to guarantee transactional integrity.
Appendix A: Data attributes

The data for this experiment consists of 28 fields including row key:

Row key: Symbols_ RealTime (YYYYMMDDHHMMSS)

1. Symbols
2. Company Name
3. Last Trade Date
4. Real Time
5. Last Trade (Price Only):
6. Open
7. Previous Close
8. Dividend Yield
9. P/E ratio
10. Ask (Real-time)
11. Bid (Real-time)
12. Book Value
13. Day’s Low
14. Day’s High
15. 52-week Low
16. 52-week High
17. Average Daily Volume
18. 50-day average volume
19. Volume
20. Ask Size
21. Bid Size
22. Change (Real-time)
23. Earnings/Share
24. EPS Estimate Current Year
25. EPS Estimate Next Year
26. EPS Estimate Next Quarter
27. Exist: OK

These attributes were selected from real time stock data provided by Yahoo! Finance. They were chosen simply because some were real time and some were important factor for stock data. The last attribute ‘Exist’ was for range query to be performed successfully in Cassandra because an equation is needed when performing multiple range queries.
Appendix B: Sample queries.

Test #1: Insertion

create keyspace StockData;

Test #2: Index on Row Key

get StockData[‘A_20100110110500’];

Test #3: One secondary index

get StockData where StockSymbol = ‘A’ ;

Test #4: Two secondary indexes

get StockData where StockSymbol = ‘A’ and LastTradeDate > 20100110;

Test #5: Three secondary indexes

get StockData where Exist = ‘OK’ and LastTrade > 20.00 and LastTradeDate > 20100110;
Reference


Apache Cassandra 0.7 Documentation. Datatax. Retrieved 2013