**TRADING ANALYTICS**

**INTERNSHIP PROJECT**

of the

**UNDERGRADUATE PROGRAM**

in

**INFORMATION TECHNOLOGY**

(B.Tech in IT)

***Submitted by:***

*Diksha IIT2013026*

**IIIT-Allahabad**



**INDIAN INSTITUTE OF INFORMATION TECHNOLOGY ALLAHABAD**

***Date: 28.04.2017 Mr. Gaurav Shah***

***(Supervisor)***

**CANDIDATE’S DECLARATION**

*I hereby declare that the work presented in this project entitled “Trading Analytics”, submitted in fulfillment of the eighth semester of Bachelor of Technology (B.Tech) course, at Indian Institute of Information Technology, Allahabad, in Information Technology is a reliable record of my original work carried out under the guidance of Mr. Varun Paliwal, Mr. Gaurav Shah and Mr. Shounak Sen, due acknowledgments have been made in the text of the project to all other material used. This work was done in full consent with the fundamentals and constraints of the recommended curriculum.*

**Place: Mumbai**

**Date: 25.07.2017**

**Name of the student**

*Diksha IIT2013026*

**ACKNOWLEDGEMENT**

*The author would like to express her sincere gratitude to Mr. Varun Paliwal without whose able guidance the on-going proceedings of the project would have been a mammoth task. Thanks to Shounak Sen for providing the financial guidance required for the project.*

**TABLE OF CONTENT**

Abstract ……………………………………………………………………….. 8

1. Introduction and Motivation ………………………………………. 8 - 12
2. Problem Definition…………………………………………………. 12 - 14
3. Literature Survey ………………………………………………… 14 - 16
4. Proposed Methodology ………………………………………… 16 - 26

4.1. Feature Extraction …………………………………………… 16 - 20

4.1.1. SIFT …………………………………………………... 17 - 19

4.1.1.1. Scale Space Extrema Detection …………………... 17

4.1.1.2. Keypoint Localization …………………………….. 18

4.1.1.3. Orientation Assignment ………………………….. 19

4.1.1.4. Keypoint Descriptor …………………………….... 19

4.1.2. Advantages of SIFT ……………………………………... 19

4.2. Vocabulary or Codebook Generation ……………………………... 20

4.2.1. KMeans Clustering Algorithm …………………………….. 20

4.3. Training Classifier …………………………………………….. 21 - 22

4.3.1. SVM ………………………………………………………. 22

4.4. Pseudocodes ………………………………………………… 23 - 25

4.5. Algorithms used for comparison ……………………………… 25 - 26

4.5.1. HOG ……………………………………………………… 26

4.5.2. Standard SIFT + Brute Force Matcher ……………… 26

5. Experiments and Results …………………………………………….. 26 - 33

5.1. Experiment 1 …………………………………………………... 26 - 30

5.1.1. Accuracy Vs. Vocabulary size …………………………….. 29

5.1.2. Computational time Vs. Vocabulary size …………………. 30

5.2. Experiment 2 ………………………………………………… 30 - 33

6. References ……………………………………………………………. 33 - 36

**ABSTRACT**

*At Global Markets, we have several business requirements that require very quick turnaround. Thus having a single framework for Rapid Application Development is a foundation that we believe will make us ready for any future tasks. This project is divided into the following sub projects:*

*Analysing the patterns from historical data and run the EPL (Event Processing Language) queries and implement the trading strategies which will eventually run on real time data. The queries are run on the historical data to find the trading opportunities and the data is replayed on the given set of queries to evaluate those patterns. After evaluating the selected queries will be run on real time data which will make business in millions.*

*Feature Extraction, I've worked with the quant researchers for finding the features which yield maximum returns.*

*Designed and implemented a generic API which will fetch historical data from Onetick (a time series database) according to strategy lookback requirements.*

*In finance, a trading strategy is a fixed plan that is designed to achieve a profitable return by going long or short in markets. I've coded three trading strategies using Historical data API, Multithreading, and Java Messaging Service(JMS). These strategies were getting feed from Analytic Engine (feature feed).*

*Designed and implemented a common library for communication between strategy and Analytic Engine using JMS, singleton design patterns and threading.*

*I've also worked on Orderbook building project, i.e. maintaining the real time orderbook by managing the order information.*

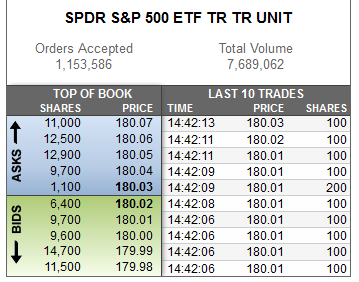
1. **Introduction and Motivation**

In Edelweiss, I worked in designing and implementation of Day Trading strategies. Buying and selling a security within a single trading day defines Day Trading. The trader maintains positions such as long or short in the market. By buying a product such as a stock, commodity, derivative or currency with the expectation that its price will go up, the trader takes the long position and vice versa by selling the mentioned products the trader takes the short position. In day trading strategy the positions are squared of at the end of the day. That if you’re going long in the market, you need to go short at the end of the day to make your position neutral.

The strategies I worked on were Orderbook Structural non-price based strategies. There are different kind of orders that are placed in market such New, Cancel, Modify, the order which gets traded is called as Market order and the rest are placed in the orderbook as shown in figure 1 and are referred as limit orders.

The limit order book has two sides Bid side and Ask side, the bid side is maintained in the decreasing order of prices and the ask side in order of increasing order of prices.

Figure 1:



In order to generate long, short signals in the strategy I have used the features of the limit order book which were yielding maximum returns. The technologies used in building these strategies include Onetick (time series database, Esper (Complex Event Processing Engine), Java Messaging Service, design patterns and the strategies were coded in Java.

1. **Problem Definition**

1. Optimizing esper queries for computing the analytics required for the strategies. The normal esper query took 8 to 12 hours to run for a month's data (historical) which is obviously not suitable. Optimizing the queries to compute analytics using various esper tools.

2. There are three kind of feeds which are used in trading strategies

i. Fast Feed: Orderbook with 5 level depth

ii. TBT Feed: Orderbook with 20 level depth

iii. Analytic Feed: The feed which is calculated using Market feed (exchange feed) or fast feed or tbt feed.

In my case, I required Orderbook of infinite depth, and for securities like NIFTY future the depth can go up to 1000 level, and there around 200 securities whose depth has to be stored. And not only storing the orderbook of infinite depth but also constructing it was very time consuming.

3. Implementing day trading strategies using feature feed from Analytic Engine, scheduling the signal generation of the strategies, and making them scalable to run by loading multiple portfolios.

4. In trading strategies, we have portfolios comprising of securities to be traded on and the strategy specific parameters. For all those securities, the strategy requires a connection to the Analytic engine, and a single JMS connection creates a thread pool, so if we have say 500 securities, than we will end up with 500 thread pools and that will be a huge wastage of resource, also that can cause the other strategies on the server to crash.

5. Maintaining the real time order book to construct features in the analytic engine which are then sent to multiple strategies.

**3. Proposed Methodology**

Esper: The Esper engine works a bit like a database turned upside-down. Instead of storing the data and running queries against stored data, the Esper engine allows applications to store queries and run the data through. Response from the Esper engine is real-time when conditions occur that match queries. The execution model is thus continuous rather than only when a query is submitted. Esper has been highly optimized to handle very high throughput streams with very little latency between event receipt and output result posting. Multi threading is recommended to send events to esper. It has many in built features such as time windows, contexts, etc. which makes the processing optimal on a large scale and improve latency. The queries are run on the historical data to identify patterns to find the trading opportunities and the data is replayed on the given set of queries to evaluate those patterns. After figuring out the patterns, we ran the queries or the patterns on the live data and we will have the live running strategies at the end which will make business in millions. In this project, we need to drill-down to analyze exact market conditions and replay consolidated order books around the time of trade.

An event bean object represents a record (event) in the perpetual query's result set. Each event bean object has an associated event type object providing event metadata. To run an EPL query , a message listener is registered to Complex Event Processing(CEP) engine. The EPL is created by passing the statement to CEP engine.

The event type is needed to be configured to CEP engine and the events are sent to esper. For computing the analytics an Update Listener is implemented which receives one or more eventbeans events with each invocation.

The results of the query can be polled or read out through the iterator method of the EP statements. The statement iterators will return the instances of eventbeans.

The events are needed to be processed by esper engine and this achieved by using EPruntime interface. This interface is also used for setting and getting the variable values and execute the on-demand queries.

Java object can be sent to the cep engine as shown in the below code snippet. The sendEvent method is required to be overloaded as events can be represented as different classes in Java, therefore the sendEvent method can take the required parameters

The below code snippet shows how to send a Java object event to the engine. Note that the sendEvent method is overloaded. As events can take on different representation classes in Java, the sendEvent takes parameters to reflect the different types of events that can be send into the engine.

EPServiceProvider epService = EPServiceProviderManager.getDefaultProvider();

EPRuntime runtime = epService.getEPRuntime();

// Send an example event containing stock market data

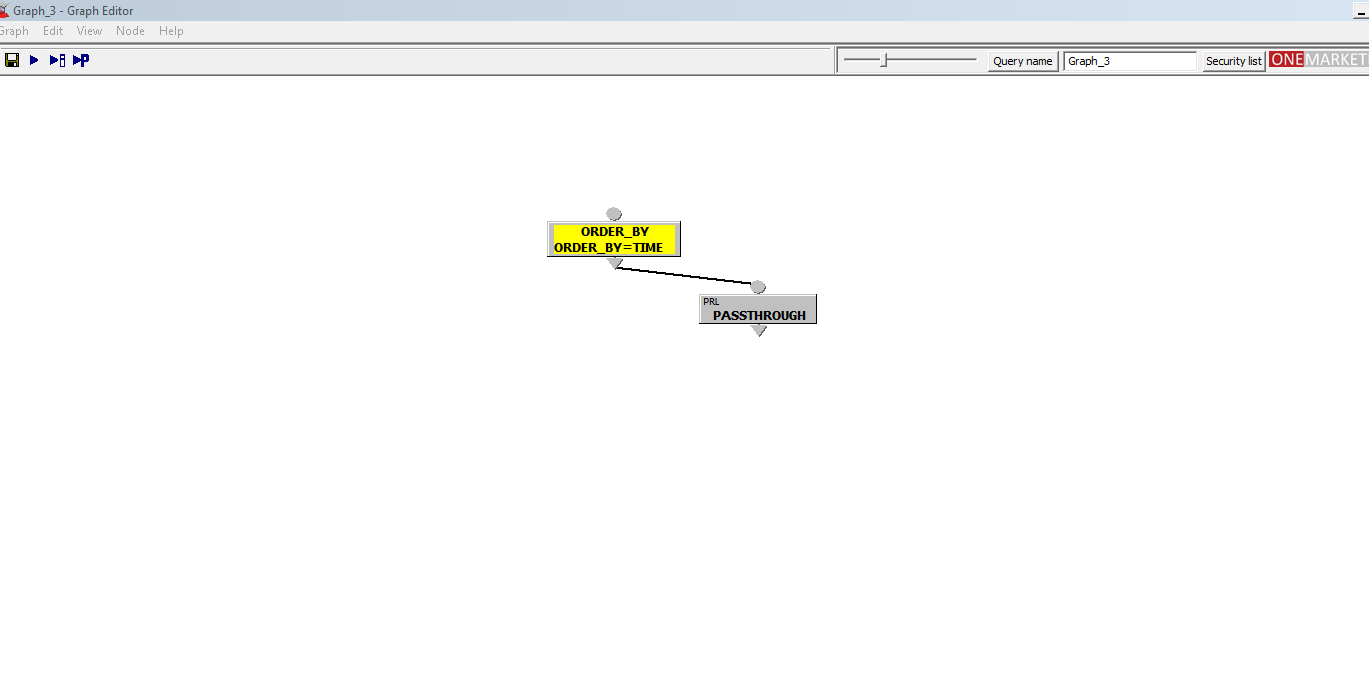
runtime.sendEvent(new MarketDataBean('IBM', 75.0));

## Ti

And in the update listener method we can get the desired result.

Further, the strategies were dependent on historical data for generating signals, and the order book dependent strategies require the orderbook data and that too of infinite depth and to construct orderbook from raw data, for a single security for a single day took around 30 seconds, so we decided to construct the orderbook and store it in a database, in our case Onetick database.

**OneTick** Database is 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 and options, as well as full order book data.



**Figure 1.**

OneTick has powerful analytical tools that enable clients to run historical simulations and back-tests, build transaction-cost models, develop trading and market-making strategies, perform real-time surveillance and answer 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 exchange.

OneTick Database collects and processes millions of ticks per second on a single server and archives billions of ticks per day globally. It comfortably tackles rapidly rising data volumes and real-time data rates, while servicing multiple concurrent user queries on data sets stretching back over years, and does it all with no special hardware required.

OneTick Database includes a wide range of features for the capture, management and analysis of financial data.:

* Content-Aware for Trades, Quotes, Orderbooks, Executions across all asset classes
* Support of both column and row store
* High performance/high precision server-side analytics execution
* Stitch history and real-time as one single data stream
* Comprehensive Authentication/Authorization and data Entitlements

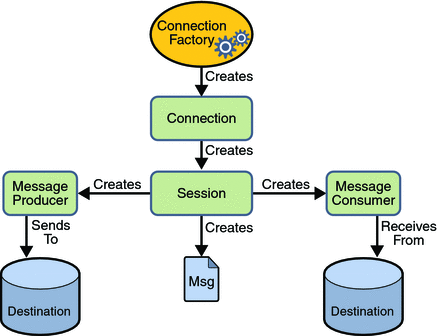
I have designed an API in java for running the onetick query passing through the database containing orderbook. The API across all the languages provides a means to execute queries (stored in OTQ files), access data directly, and execute SQL statements using OneTick’s own dialect. As with all the query tools and mechanisms, the API operates on all the database storage subsystems (archives, in-memory, real-time) providing transparent access across them. The API is callback based, and result sets are self-describing and so it is possible to use exact same client code for historical queries as for continuous / CEP queries. In addition to query execution, the OneTick API can be used to extend the OneTick analytical language and to design custom versions of real-time data collectors, historical or in memory database loaders or CEP adapters.

In onetick, I’ve written a query and with java api, I am fetching the order book for the required historical lookback. For example, if I am running the strategy say on 22nd July Friday and the look back requirement is of two days, the API will give me the orderbook of 21st JULy and 20th July.

The graph queries consider of nodes as specified in figure 1. The nodes can be connected such that the successive node’s query can be applied over the result of previous nodes.

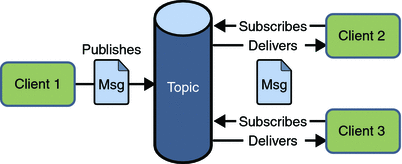
With the help of this orderbook data, features were calculated in an analytic engine and were given as a feature feed to the strategies. The analytic engine is responsible for providing the analytic feed to the strategies, i.e. the feature feed. This communication was taking place via Java Messaging Service. I have designed and implemented an Analytics JMS Publisher Subscriber library for the communication using connection pools and singleton design pattern.

Figure 2:



JMS is an API that provides the medium to send or create or receive the messages from one application to another. Advantage of JMS is that it is asynchronous and reliable. The client can register a message listener with the consumer and whenever JMS provider delivers the message by calling the onMessage method of the listener.

Figure 3:



A typical JMS program goes through the following steps to begin producing and consuming messages:

Looking up for connection factory.

Using the Connection Factory to create a Connection.

Using the Connection to create one or more Sessions.

Using a session and destination to create the required Message Producers and Message Consumers.

And last is to start the connection.

In my case there is a strategy and we need to load portfolios with multiple securities in the strategy, if each security creates a connection and session object of its own, it will lead to huge resource consumption and will create deadlock, in order to avoid that I have created a connection and session pool, and all the securities registering the message Listener to the consumer will get the connection and session from the particular pool only and this will lead to an optimal resource utilization.

With the help of this API we have multiple subscribers subscribing for receiving the messages from Analytic Engine. We could have calculated the features in the strategy also, but the features were being used by multiple strategies and calculating them at a common place and sending to multiple strategies was the optimal approach we followed.

Also I have used singleton design pattern to achieve this, the connection and session pool was created only once, for the first time when a security registered for receiving the signals, once the pool is created, every time another security registers for connection and session, it receives the already created connection-session objects, and thereby creating less number of thread pools.

Now the main part of this project is to implement strategies which generate long short signals based in the feature feed they receive from the Analytic Engine. I have implemented four day trading strategies using the features from the mentioned analytic engine.

The strategies are designed such that they generate long short signals based on the calculations while receiving the feed from the exchange. The signals are generated on a particular bar size and after a particular timestamp. The strategy comprises of two units front end and back end. As shown in figure 4, it is the front end of the strategy. The backend contains the signal generation logic and is scheduled to generate the signal at a particular bar size specified in the portfolio. The portfolio comprises of the securities and the parameters specific to the strategy. After the portfolio gets loaded, we can hit orders: New, Modify or Cancel.

The front end as shown in figure 4. displays any error that comes in the backend, the connection between front end and back end is established via apache Apollo.

[**ActiveMQ Apollo**](http://activemq.apache.org/apollo/) is a faster, more reliable, easier to maintain messaging broker built from the foundations of the original **[ActiveMQ](http://activemq.apache.org/)**. It accomplishes this using a radically different threading and message dispatching [**architecture**](http://activemq.apache.org/apollo/documentation/architecture.html).

However, ActiveMQ is good at what it does and since it’s stable right now, it would be a bad idea to re-write its threading architecture. So that’s where [Apollo](http://activemq.apache.org/apollo/index.html) comes in: right now it’s a subproject of [ActiveMQ](http://activemq.apache.org/) where we can take all of the experiences of the ActiveMQ 5.x line, combine it with a new threading architecture, and develop a messaging broker that will meet demands of systems in the future with millions of destinations, hundreds of thousands of connections, high throughput, and utilize resources as efficiently as possible. Eventually, once backward compatibility is reached, Apollo will become ActiveMQ 6.0.

Now the strategies also need the real time building of orderbook, that took place in the analytic engine. The book builder receives the orders information i.e. if it’s a cancel order or modify or new or trade information. With the help of that information the order book is constructed such that we can calculate the desired features in real time using the order book. There was a lot of synchronization was rinvolved in constructing the order book, sending the messages to multiple strategies, thereby keeping the analytic engine and strategies thread safe.

Results and Conclusions:

Hard

