Stock Analysis with Stream Processing

Alan Lau
Rupesh Basnet
Bohan Chen

Final Project for Programming Language Paradigm CSC33500
Spring 2018

The City College of New York

Contents

Introduction

Implementations

Results

Conclusions

References

I. INTRODUCTION

Lazy evaluation is an important concept in computer programming, its purpose is to minimize the require computation for computer program. In contrast, the other type of evaluation is eager evaluation which is used by mostly other programming languages. Lazy evaluation is especially useful in functional programming languages. When using deferred evaluation, the expression is not evaluated immediately after it is bound to the variable. Instead, the expression is evaluated when the value is taken. That is, the statement like x:=expression(The result of the expression is assigned to a variable). An explicit call to the expression is evaluated and the result is placed in x, but first regardless of what is actually in x until there is a reference to x in the following expression. When its value is required, the evaluation of the following expression itself can also be delayed, and finally this fast-growing dependency tree is calculated in order to generate a certain symbol for the outside world to see.

A stream represents a sequence of data elements, each of which is calculated only when required. This allows large or even infinite sequences of data to be represented and manipulated with familiar operations like “car”, “cdr”, “map” or “fold”. In such manipulations only as much as needed is actually held in memory at any one time. Stream are implemented using promises, or delay evaluation, which is how the underlying calculation of values is made only when needed, and the values then retained so the calculation is not repeated. In such way of function, stream became very useful tool when dealing with infinite sequences of data. Stream processing is important because it can analyze large volumes of data immediately as it is being produced. Continuous data streams are ubiquitous: they arise in telecommunications, health care, financial trading, and transportation, among other domains.

With the help of stream function, such as stream-cons, stream-car, and stream-cdr, our application can process with infinite number. When we design a program, almost every program requires one or more inputs. Sometimes we need to consider whether the program’s input has to be limited because when we don’t know what and how many the input should be, the input will affect program’s run time and space complexity. In the way of stream implementation, we don’t need to worry about the input of the program is finite or infinite. it can analyze large volumes of data immediately as it is being produced. Stream plays an important role when processing user inputs and process as many inputs as needed and keep the rest input as a compromise.

One of an important development about stream was that the algorithmic idea behind the generation of prime numbers. It was originally conceived by Eratosthenes, a Greek mathematician, astronomer, and geographer who devised a map of the world and estimated the circumference of the earth and the distance to the moon and the sun. The input of the function sieve is the natural numbers starting from 2. The first element in the input is taken to be a prime number. Let us say the first such number is p. No number p*n, where n is a natural number

greater than one, can then be a prime number. Recursively, the first number which comes out of the actual chain of sieves is a prime number, and it is used set up a new filter. This is due to the simple fact that the recursive calls of sieve function itself. The Sieve of Eratosthenes is a more sophisticated example of the use of streams. Its process produces the stream of all prime numbers.

After many paper researches and considerations, we finally decided to do the second topic about streams. Within the topic of streams, we were intrigued by one of the Stream implementation exercises of the book, *Structure and Interpretation of Computer Programs* under the topic "Streams as signals." This topic describes how we can model streams as computational analogs of "signals" in signal-processing systems. Pairing the values of signals in terms of the time intervals provided some unique exercises in the book. Stock analysis, could be one such real life signals that we can process by using the stream implementation in Scheme.

Our idea is to design a stock analysis program. This project idea was inspired by "IBM Streams Processing Language: Analyzing Big Data in motion". The IBM Streams Processing Language is the programming language for IBM InfoSphere Streams, a platform for analyzing Big Data in motion. The "Big Data in motion" means continuous data stream at high data-transfer rates. The research paper provides an overview of the IBM Streams Processing Language and different types and usages of streams. It also describes that a continuous stream is an infinite sequence of data items and a stream graph is a directed graph in which each edge is a stream and each vertex is an operator instance. Our project uses the way how IBM stream processing language handling big data at massive scales.

Our project implementation was followed by the research paper "Lazy Evaluation Methods for Detecting Complex Events" by Ilya Kolchinsky, Izchak Sharfman, and Assaf Schuster. This research paper is about using complex event processing system to detect complex patterns over a stream of primitive events. The complex event processing system employ Non-deterministic Finite Automata to detect the sequences. However, because the sequences are very frequent at the beginning of the event, this research paper introduced the lazy evaluation method to solve this problem by process events in descending order. The authors evaluate their mechanism on real-world stock trading data and demonstrating a performance gain of two orders of magnitude. We also use some of the ideas from "Efficient Pattern Matching over Event Streams" by Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman where we compare each stocks' data in event streams.

Stock invest becomes one of the most popular investment choice in this modern world. Everyone is able to invest in the stock market. Whether you are rich or poor, there're high price stocks and low price stocks for people to purchase. However, stock investment does have some risks if investors have lack of stock analyzing skills. So, investors are better to have some knowledge and tools to help them in order to gain money. In our project, we take this chance of design a program using stream, we made a stock analyzing application. Our stock analyze

program will help the user to determine which of the stocks user purchased or about to purchase are the most profit one.

II. IMPLEMENTATION

(The codes is attached for easier referencing)

The main idea for our implementation is to simulate reading the data for the stocks and then based on the data find out the certain occurrence of some events. Instead of reading the data, the way we did it is through using random function to generate the data since we don't know how to use R5RS to get the input. The random function is implemented using the pseudo randomness formula and referencing to the website "stack overflow" (the link is attached in the source code). Moreover we also use the 'randint' function to generate a random integer in certain range.

```
(define random
  (let ((a 69069) (c 1) (m (expt 2 32)) (seed 19380110))
    (lambda new-seed
      (if (pair? new-seed)
          (set! seed (car new-seed))
          (set! seed (* 1.0 (modulo (+ (* seed a) c) m))))
      (/ seed m))))
```

The initial seed that we used is Knuth's birthday, since the seed remain the same everytime we re-run the code, therefore the result will be the same. By multiplying by '1.0', we make the random function to generate number in floating point instead of fraction.

```
(define (randint . args)
  (cond ((= (length args) 1)
         (floor (* (random) (car args))))
        ((= (length args) 2)
         (+ (car args) (floor (* (random) (- (cadr args) (car args))))))
        (else (error 'randint "usage: (randint [lo] hi)"))))
```

In addition, we implemented the 'cons-stream', 'stream-car', 'stream-cdr' based on the textbook.

```
(define (cons-stream a b)
  (cons a (delay b)))

(define (stream-car stream)
  (car stream))

(define (stream-cdr stream)
  (force (cdr stream)))
```

As we are developing our code, we realize that when we want to show the data, the data is in long floating point which looks quite messy. Therefore we decide to implement our own rounding function.

```
(define (my-round number precision)
  (let ((a (round (* number (expt 10 precision)))))
    (* a (expt 10 (* -1 precision)))))
```

The function ‘my-round’ taking two parameters--number and precision. ‘Number’ is the number that we want to round and ‘precision’ is the number of decimal digit that we want to have.

Moreover, we define our variables at the global scope. Although it do pollute the global environment, but it makes it easier for us to ‘tune’ the variables as we are working on it. First we initialize the prices for three stocks. Then we initialize the probabilities for the stocks to either rise or drop and the probabilities for the prices to rise or drop for a certain percentage. The way we did it is to make stock ‘a’ has higher chance to drop, stock ‘b’ has higher chance to rise and stock ‘c’ has the higher chance than stock ‘b’ to rise. Stock ‘a’ and ‘b’ start at the same price, while stock ‘c’ start at lower price. This way we ensure that the event that we want to find will certainly occur and will occur after some reasonable iteration.

Then we implement the function for the stocks to rise or drop.

```
(define s-rising
  (lambda (stock)
    (cond ((eq? stock 'a) (begin
                            (set! temp (stream-car a))
                            (cond ((< (randint 0 10) p-a-rise3%) (cons-stream (* temp 1.03) (cons-stream temp a)))
                                  ((>= (randint 0 10) p-a-rise1%) (cons-stream (* temp 1.01) (cons-stream temp a)))
                                  (else (cons-stream (* temp 1.02) (cons-stream temp a))))))
          ((eq? stock 'b) (begin
                            (set! temp (stream-car b))
                            (cond ((< (randint 0 10) p-b-rise3%) (cons-stream (* temp 1.03) (cons-stream temp b)))
                                  ((>= (randint 0 10) p-b-rise1%) (cons-stream (* temp 1.01) (cons-stream temp b)))
                                  (else (cons-stream (* temp 1.02) (cons-stream temp b))))))
          (else (begin
                  (set! temp (stream-car c))
                  (cond ((< (randint 0 10) p-c-rise3%) (cons-stream (* temp 1.03) (cons-stream temp c)))
                        ((>= (randint 0 10) p-c-rise1%) (cons-stream (* temp 1.01) (cons-stream temp c)))
                        (else (cons-stream (* temp 1.02) (cons-stream temp b)))))))))
```

This function ‘s-rising’ is the function for the stock to rise. Taking one parameter ‘stock’ to determine which stock’s price we are going to adjust.

```
(define s-dropping
  (lambda (stock)
    (cond ((eq? stock 'a) (begin
                            (set! temp (stream-car a))
                            (cond ((< (randint 0 10) p-a-drop3%) (cons-stream (* temp 0.97) (cons-stream temp a)))
                                  ((>= (randint 0 10) p-a-drop1%) (cons-stream (* temp 0.99) (cons-stream temp a)))
                                  (else (cons-stream (* temp 0.98) (cons-stream temp a))))))
          ((eq? stock 'b) (begin
                            (set! temp (stream-car b))
                            (cond ((< (randint 0 10) p-b-drop3%) (cons-stream (* temp 0.97) (cons-stream temp b)))
                                  ((>= (randint 0 10) p-b-drop1%) (cons-stream (* temp 0.99) (cons-stream temp b)))
                                  (else (cons-stream (* temp 0.98) (cons-stream temp b))))))
          (else (begin
                  (set! temp (stream-car c))
                  (cond ((< (randint 0 10) p-c-drop3%) (cons-stream (* temp 0.97) (cons-stream temp c)))
                        ((>= (randint 0 10) p-c-drop1%) (cons-stream (* temp 0.99) (cons-stream temp c)))
                        (else (cons-stream (* temp 0.98) (cons-stream temp c)))))))))
```

This function ‘s-dropping’ is the function for the stock to drop. Taking one parameter ‘stock’ to determine which stock’s price we are going to adjust.

As we have mentioned, we need a function to act as reading the data from outside source. Therefore we implemented this function called ‘reading-data’ which will adjust the stock’s price based on the probability that we have defined. Each time we are reading the data, the variable ‘count’ is incremented by one to keep track how many iteration for us to find such event to happen.

```
(define reading-data
  (lambda arg
    (begin
      (set! count (+ count 1)) ;; each call will increment the count by 1
      (set! a (if (< (randint 0 10) p-for-a-to-rise)
                  ; if a is rising, determine how much it will rise
                  (s-rising 'a)
                  (s-dropping 'a)))
      (set! b (if (<  (randint 0 10) p-for-b-to-rise)
                  (s-rising 'b)
                  (s-dropping 'b)))
      (set! c (if (<  (randint 0 10) p-for-c-to-rise)
                  (s-rising 'c)
                  (s-dropping 'c)))
      )))
```

As we are reading the data, we need to check whether the event that we want to find have happened or not. therefore, we need another function to find such event.

```
(define find-event
  (lambda arg
    (if (< (stream-car a) (stream-car b) (stream-car c))
        count
        (begin
          (reading-data)
          (find-event)))))
```

If the event of where stock c's price is greater than stock b's price and stock a's price is the smallest among all is found, the 'find-event' function will be terminated and stop calling the 'reading-data'.

Lastly, we also implemented a function for us to view the price of stocks.

```
(define (run n) ; this will show the price of stocks
  (define (iter n)
    (if (= n 0) (display "stop")
        (begin
          (reading-data)
          (display count)
          (newline)
          (display (my-round (stream-car a) 2))
          (display "                    ")
          (display (my-round (stream-car b) 2))
          (display "                    ")
          (display (my-round (stream-car c) 2))
          (newline)
          (iter (- n 1)))))
  (iter n))
```

In conclusion, instead of getting all the price data for these three stocks, our system makes use the advantage of lazy evaluation which will only get the data as it's needed. What we have presented is the simulation of reading the stock's data and find certain event.

III. RESULTS

To better visualize the generated stock data, we decided to make use of the Plot library provided by Racket. In order to plot the stream, we took a window of 100 generated stream of stocks for all three Stock A, Stock B and Stock C. Following are the sample plots of the each stocks over the period of 100 time intervals.

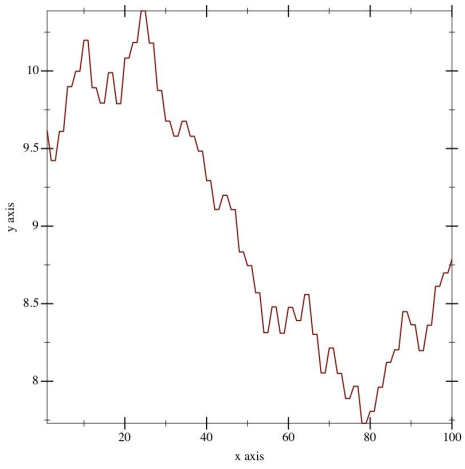


Fig: Stock A

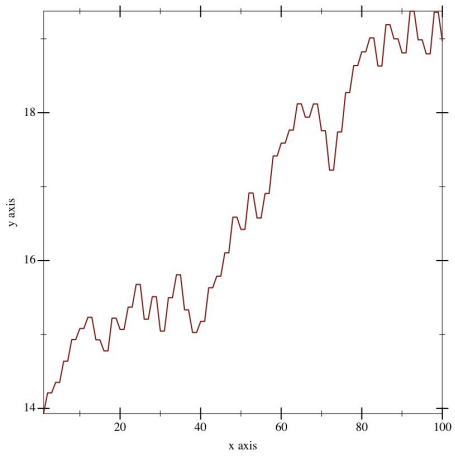


Fig: Stock B

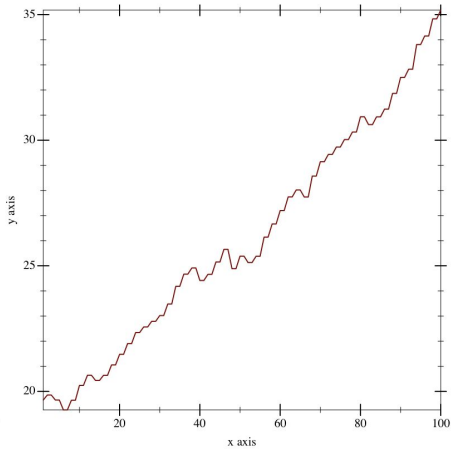


Fig: Stock C

In all the line plots of the stocks, the y-axis is the price level and the x-axis is the time interval. As seen above, the Stock A price tends to decrease over time while the Stock B and Stock C end up increasing as expected. We can observe that the Stock C is increasing at a higher rate compared to the Stock B.

Volume-Weighted Average Price Calculation

In other to test other possible algorithmic trading options, we decided to work create the Volume-Weighted Average Price Calculation (VWAP). As stated in the research paper, *IBM Streams Processing Language: Analyzing Big Data in motion*, a common Algorithmic Stock Calculation for a stream of stocks is the VWAP. These are the prices that reflect the weighted volume, thus taking traded volume of a stock into consideration.

Our approach was to again randomly generate the volume data of all the three stocks and create the ripples of the volume fluctuations. We used the same probability values of the stocks however changed the seed data of the random function to resemble a volume data. The seed data used for each traded volume - Stock A, Stock B and Stock C were 10000, 10000 and 7000 respectively. We created two new procedures *vol-rise* and *vol-dropping,* which were basically

the same procedures as *stock-rising* and *stock-dropping* but with different stream parameters - specifically *vol-A, vol-B* and *vol-c*. Since the defined probabilities are the same for the volume as the stocks probabilities, as the stock price would decrease, the volume would also go down and vice versa. This was done purposely so that we can simulate a similar stock trading environment as in reality.

Calculating Volume-Weighted Average Price Calculation

We followed the program given in the Hirzel, M., et al. paper, in order to compute the VWAP. The first step was to create a pair of streams, which is similar to the "PreVwaps" output stream from the paper but without any trading windows. This output stream of pairs consists of the Volume of the particular stock with the Volume * Price of the stock, as shown the output stream below.

```
STREAM((7267.418777090884 . 63843.79278039775)
(7340.8270475665495 . 63850.17779817757)
(7340.8270475665495 . 63850.17779817757)
(7196.889262320146 . 61978.42923527234)
(7196.889262320146 . 61978.42923527234)
 continued ...)
```

Since, VWAP calculation required us to create sliding window in order to calculate the moving averages, we first created another stream output which was to take the car of the previous output stream of pairs - Volume and the cdr of the stream pairs - Volume * Price of the stock and calculates the moving sums of 10 time interval window. This new stream is also a pair of streams. Finally, we calculated the VWAP by creating a new stream of running sum of :Volume * Price divided by the Volume of that stock at that time interval. We called these streams the running-averages.

```
STREAM(8.784933789924535
8.74122547712328
8.72675349395497
8.69837737528477
8.681239768504229
 continued ...)
```

A

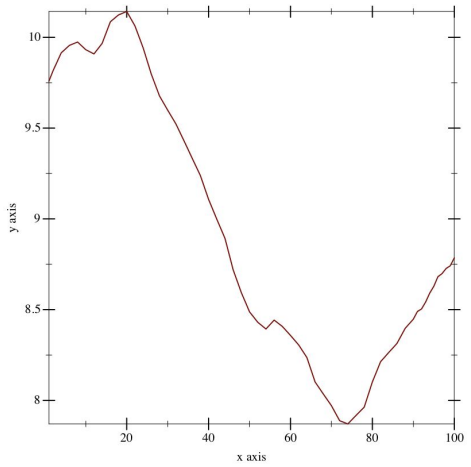
```
STREAM(18.971839898265248
19.16351336662517
19.228250763223556
19.11887471029089
19.053605461508823
 continued ...)
```

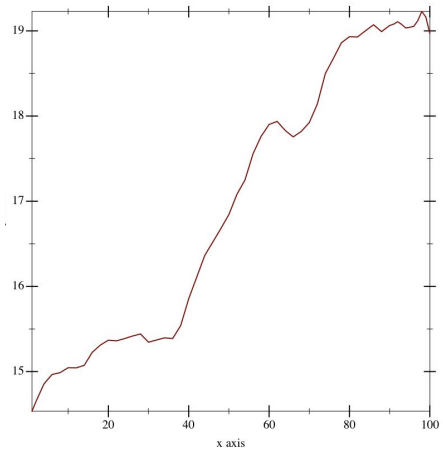
B

```
STREAM(35.17936590615938
35.00778435269814
34.94945795003639
34.74757240888085
34.62693834321754
 continued ...)
```

C

In order to get a better understanding of how these VWAP values, we plotted the stream by converting 100 time interval streams into a list and using the Racket's Plot library. As shown in the graphs below similar to the stock graph, the VWAP are a lot smoother and does represent the volume weighted running average. The y-axis of the graph is the price levels and the x-axis is the time intervals.





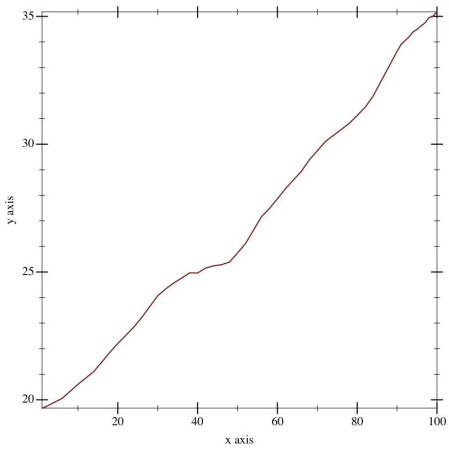
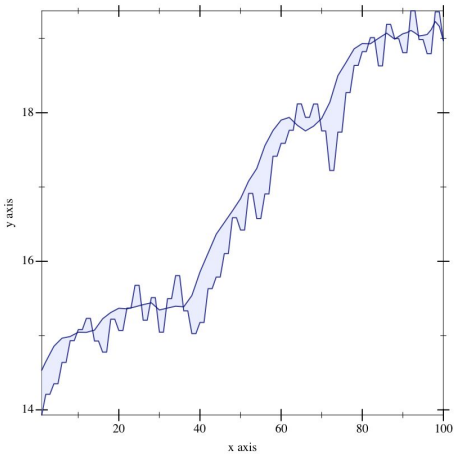


Fig: VWAP of Stock A Fig: VWAP of Stock B Fig: VWAP of Stock C

Finding Bargains

According to the paper from Hirzel, M., et al, it is possible to find better bargains. Traders and analysts do use VWAP to reduce the noise in stock transactions that occurs throughout the day and to gauge the actual worth price of the stocks that the buyers and sellers are willing to trade on. VWAP gives a support line for the stocks and understanding the VWAP compared against the actual daily stock prices can give some insights. Therefore, we decided to merge the two streams together and visualize it in a graph to see the discrepancies in the prices. As shown in the three diagrams below for each stocks A, B and C respectively, we can observe the differences in the prices more clearly and also notice that the prices of the VWAP can be higher than the actual stock price and vice versa.



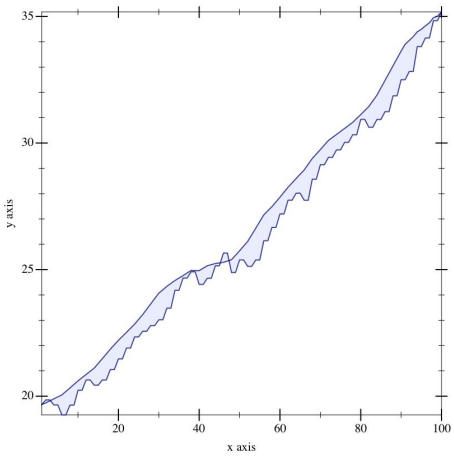


Fig: VWAP & Stock A

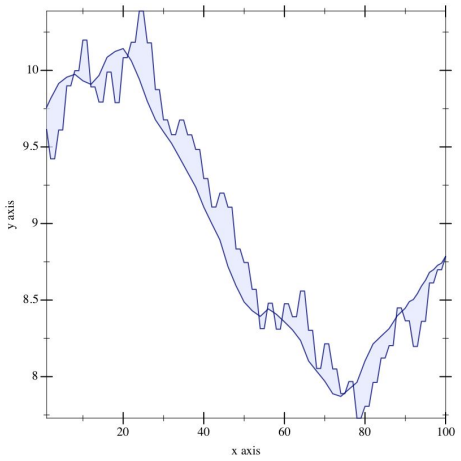


Fig: VWAP & Stock B

Fig: VWAP & Stock C

```
(define (bargain-index stock-stream running-avg-stream)
  (cond
    ((or (stream-null? stock-stream) (stream-null? running-avg-stream)) the-empty-stream)
    ((> (stream-car running-avg-stream) (stream-car stock-stream))
      (cons-stream (* (stream-car stock-stream) (exp (- (stream-car stock-stream) (stream-car running-avg-stream))))
                   (bargain-index (stream-cdr stock-stream) (stream-cdr running-avg-stream))))
    (else (cons-stream 0
                   (bargain-index (stream-cdr stock-stream) (stream-cdr running-avg-stream))))))
```

In order to find a bargain, if the VWAP price of a stock is greater than the stock price itself or the price difference between them is less than the specified threshold, it signifies that the stock can be unvalued. To calculate this new stream of bargain values, we created a *bargain-index* with *stock stream* and the *running average stream*. If the running average stream is greater than stock stream, we *cons-stream* the *stock stream* to the $e^{(stock - running\ avg)}$, else we *cons-stream* 0 into the *bargain-index* stream. According to the paper from Hirzel, M., et al, the higher the bargain index value, the better the bargain as comparing it to the VWAP and the stock prices. To visualize this, we create the *bargain-index* for the Stock B. As we can observe in the graph, the discrete histograms can be correlated with the VWAP & Stock B graph.

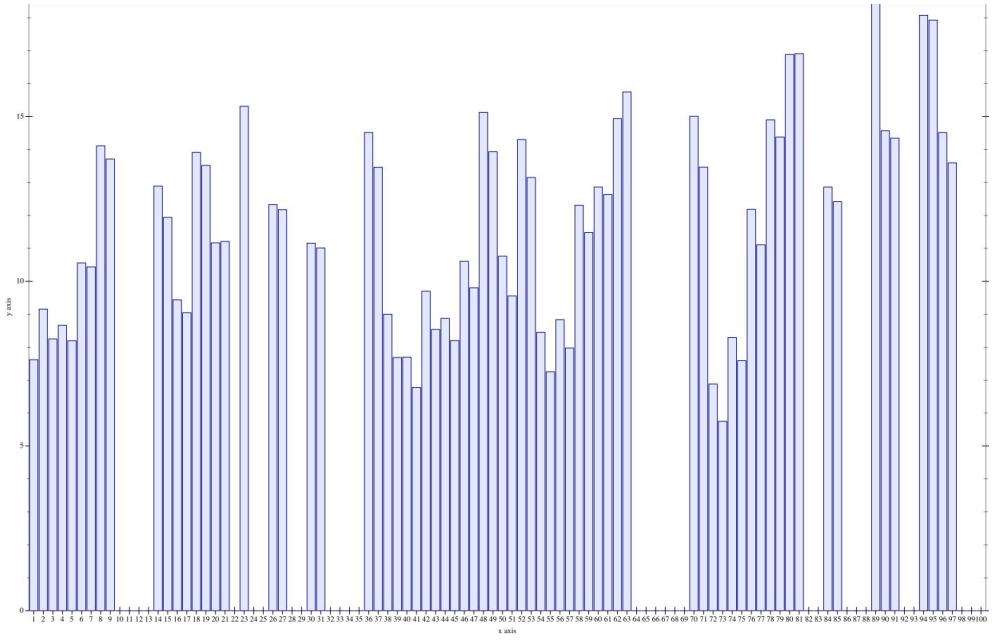


Fig: Bargain-index graph

IV. CONCLUSION

Our stock analysis program with stream processing is able to analyze three different stocks and will show the one that has the highest performance. The program adopts the lazy evaluation method to limit data and process them whenever the requirement needs. We simulated our three sample stocks ‘a’ ‘b’ and ‘c’ by random number generator function and applied the real-world stock price events. These three random generated stocks’ price have different probabilities to rise or drop. We implemented a find-event function to find if there’s a situation where stock a < stock b < stock c exits. We also used the similar random number generator as the stock prices to generate the traded volume for the same time intervals. Using the volume traded quantity and the stock prices we were able to calculate the Volume-Weighted Average Price Calculation, which is typically used in stock streaming analysis to find the trend line, and can be simply referred to the running average. Using the Racket’s Plot library and the Scheme’s Stream implementation we were able to partially recreate the VWAP calculation. VWAP calculation was later used to identify the bargain index at a particular time interval.

Some improvement can be made if we have the actual data for the stock instead of the made up data. One of them is instead of finding the event that have the condition of A < B < C, the user is allowed to specify their own system because we cannot guarantee such event exist, but with the actual data, the stocks are fluctuating in an unpredictable way. Therefore finding such event is possible. The other improvement is that instead of doing this for the price and volume, the system can be improved by using other statistic for the stock, such as comparing the market cap, capital flow, earning and many more.

V. REFERENCES

1) Kolchinsky, Ilya, et al. "Lazy Evaluation Methods for Detecting Complex Events." *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems - DEBS 15*, 2015, doi:10.1145/2675743.2771832.

2) Hirzel, M., et al. "IBM Streams Processing Language: Analyzing Big Data in Motion." *IBM Journal of Research and Development*, vol. 57, no. 3/4, 2013, doi:10.1147/jrd.2013.2243535.

3) Agrawal, Jagrati, et al. "Efficient Pattern Matching over Event Streams." *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data - SIGMOD '08*, 2008, doi:10.1145/1376616.1376634.

4) Abelson, Harold, et al. *Structure and Interpretation of Computer Programs*. MIT Press, 2010.