

Introduction.....	2
Literature Review.....	3
Design and Methodology	6
Flowchart.....	6
Critical analysis and comparison of simulation	10
Communication	28
Result	39
Conclusion	41
References.....	42

Introduction

The aim of this assignment is to develop a real time system that simulates to a degree of a [real-time stock trading system](#) (Barklam, H. ,2023). The assignment of the trading system has been split into two sides, a stock side and trading side. In brief, the stock side will receive trade order from the trading side and accordingly fluctuate the stock price and return to the trading side, the trading side will be based on the stock price received and make accordingly order placement.

The student of this assignment has taken on the stock side of the trading system to be the main developing system. To note, that in this student scope of the assignment, the trading side was simulated as well, together with a basic front end for visibility of the stock price movements.

Literature Review

Redpanda (rdkafka)

Redpanda is a streaming data platform just like Kafka and RabbitMQ (Alexander Gallego, 2020). Redpanda was selected by the student due to several approaches that it took in an attempt to differ and stand out from the mature brother. Redpanda ease the deployment of a data streaming platform, the supporting features is that a single binary with built in schema registry, http proxy, and message broker capabilities. Another good point that it differs from Kafka, is that Redpanda doesn't rely on external dependencies such as [Apache Zookeeper](#), [KRaft](#), or JVM. Developer experience was in full consideration, that complete with a Redpanda Console, that allows managing of all Kafka ecosystem modules including brokers, topics, consumers, connectors, users and access. Of course, beside just developer experience, there is some big claim by Redpanda that await to be tested by the student, mainly in term of the performance it claimed,

“”

Even with 2-3x additional hardware, Apache Kafka is still 4-20x slower at the tail than Redpanda

“”

([Tristan Stevens](#), 2023)

And the Redpanda API is fully compatible with that of kafka, meaning a drop-in replacement is possible. A key difference in Redpanda from Kafka is the underlying programming language, Kafka is written in Java and Scala, Redpanda is written in C++.

[rdKafka](#) is the rust crate that provides a fully asynchronous, [futures](#)-enabled [Apache Kafka](#) client library for Rust based on [librdkafka](#). It made working with Kafka-alike a lot easier, especially pair with it's high level API that also work excellent in an asynchronous. Pairing with tokio in asynchronous runtime, the provided API become non-blocking and asynchronous, allowing ease of managing with a kafka streaming platform.

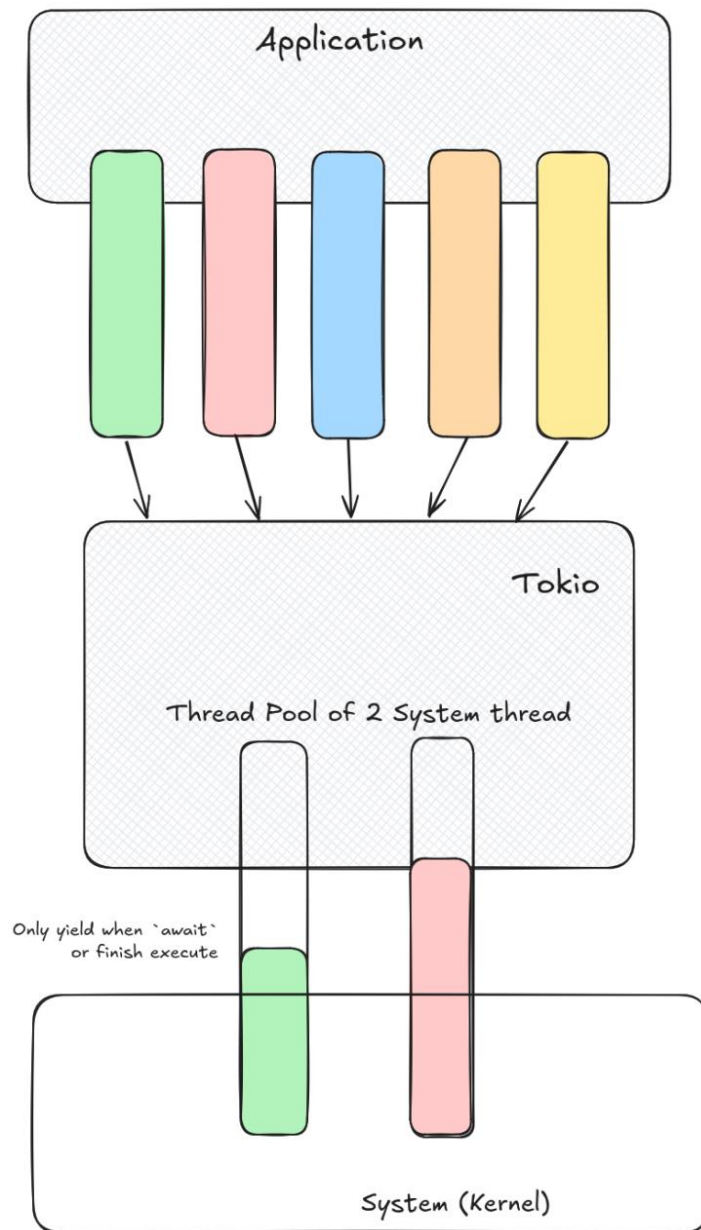
[Redis](#)

Redis is an in-memory data store, it is a high-performance, low-latency key-value database, cache, and message broker. It supports diverse data structures like strings, hashes, lists, sets, and sorted sets, along with advanced types like streams and geospatial indexes. Key thing of Redis is that it enables atomic operations, reducing complexity of avoiding race condition, due to it's nature of single-threaded.

[Tokio](#)

Tokio is an event-driven, non-blocking I/O platform for writing asynchronous I/O backed applications. Tokio is a full-on stack to build production-level applications. However, in this assignment, only the runtime component will be used. So tokio is an asynchronous runtime for Rust application, meaning that the thread that is spawn in the application via `task::spawn` is actually a [green thread](#), or better known as a task in tokio environment. The reason for using tokio in this assignment is to take advantage of this environment revolving around task. Nevertheless, Tokio task, is a light weight, non-blocking unit of execution. They are cheaper to create and manage than OS threads since they are scheduled by the Tokio runtime without requiring OS-level context switches. The tasks are also in cooperatively scheduling, meaning that they keep executed until they explicitly yield (`await`) (Figure below) rather than preemptive scheduling used by OS thread. They are non-blocking in meaning of, when they can't proceed, they will yield control to the runtime to prevent stalling.

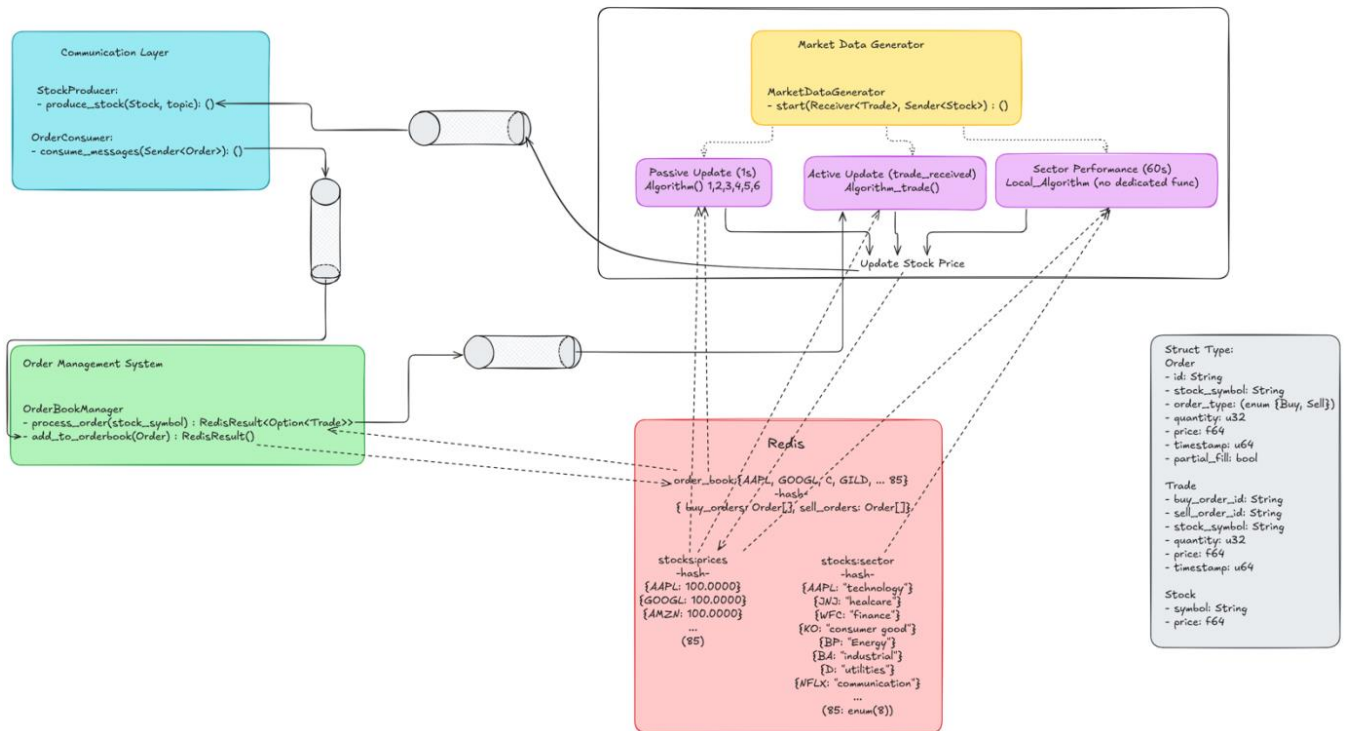
Below is a simple figure explanation of how tokio async runtime work, at the start of the application, with the async runtime macros `#[tokio::main]`, this will create a threadpool with a number of worker thread equal to the number of threads on the system hardware. The application can spawn a task by using `task::spawn` or just `tokio::spawn` (alias), these task will then be scheduled to be executed. A detail understanding of the scheduler can be found here <https://tokio.rs/blog/2019-10-scheduler> (Carl Lerche, 2019).



Design and Methodology

Flowchart

Rust assignment



This is the system design for the stock side of this real-time stock simulation system.

As visible and previously mentioned, this system has an external dependency of Redpanda and Redis. The Redis in this system acts as a database.

Start with the struct type that are used throughout and in consensus with the trading side of this system as well:

<i>Order</i>	Orders are to be received from the trading side through the Redpanda via <i>communication layer</i> . They represent the trade order placement by trader, it has information of id, stock symbol, order type, quantity, price, timestamp, and partial fill.
<i>Trade</i>	A trade is created when there is a buy order and sell order of a stock able to reach agreement for the trade to be formed. The agreement come from the domain understanding of trade, where given a stock, a buy order's price must be higher than the sell order's price (thought this is just basic logic without going into too deep). The Trade can be created by the order management system, in the order book, process order function. A trade contains information of the buy order id, sell order id, stock symbol, price, and timestamp.
<i>Stock</i>	The stock type only consists of 2 information, symbol and prices, it is meant to be sent to the trading side after new price of the stock has been tinkered. It is used as the database type for hash stocks:prices, it is produced by Market Data Generator, Update stock price.

The Flow

Start off from Communication Layer, when a new order is consumer from the trading side via Redpanda, then it will be sent to a channel that picks up by the Order Management System, the order management system will then add this order to the order book in the database, the order book look something like this, the figure below is a hash database order book for the hash key AAPL stock, in each stock, the order book has two fields: buy orders and sell orders, and the value in each of the field, is a list of Order type (mentioned above). To note, that when adding the order-to-order book, the application will add to buy order in descending order of price; as opposed to sell order in ascending order of price.

Field ↕	Value ↕
buy_orders	[{"id":"1da3d28b-c4f6-412f-8e59-469432fc241b","stock_symbol":"AAPL","order_type":"Buy","quantity":129,"price":68.079475,"timestamp":17...
sell_orders	[{"id":"91d349d9-413b-4543-bf48-c02336003ab8","stock_symbol":"AAPL","order_type":"Sell","quantity":88,"price":76.88976000000001,"ti...


After adding an order to the order book, the process order function of the order book manager will be called, this process order will immediately compare the first order of both buy orders and sell orders in the order book, this is because of the agreement that state the price of buy order must be greater than the price of sell order, only then the trade can be satisfied for both parties. Getting the first order of buy order (the largest price), and first order of sell order (the lowest price), will easily tell if the trade can be satisfied, if it can be satisfied, then according to the quantity of the order, it will be reduce as accordingly, meaning if there are 100 quantity of the buy order, and only 70 quantity of the sell order, only 70 of the quantity of both order will be satisfied, leaving the remaining 30 quantity of buy order in the order book. However, the trader can disable this option by signing their order with the flag 'partial order' found in Order type (mentioned above).


If a trade is created, then it will send to a channel into Market Data Generator, in the market data generator, there are main three task thread that create the live of market data generator, let's continue with the story of the trade first, it will get received by the Active Update task, and upon receiving the trade, an algorithm named algorithm_trade will get executed, that perform calculations on the received trade (quantity and price) together with the current live stock price of that stock, and after some calculation, a result impacted new stock price will get updated to the database and send to trader via update stock price function. In the Market Data Generator, when it is not receiving trade from channel, there are 2 other task that are always active and impacting the stock price as well, the first is the Passive Update task, this task will get triggered every second, and upon triggered, it will get the orders from the order book of all 85 stocks along with the current stock price, and these intelligence will get pass to a series of algorithms, naming 6 algorithms that is according to the stock trading domain (more info on recoding XD). After the series of algorithms calculation on the order book against stock price, a new stock price will get updated to the database and sent to the trading side via update stock price function. The similar story goes for the third task of sector performance, which get executed every 60 seconds, a local algorithm that will get the stock price and stock sector from the database and simulate impact of a sector of stock


moving in same direction, and then new price of all the stock in each sector get sent back to the database and the trader side via update stock price function.


So that is about how the simulated stock side of the real time stock trading system goes. Now for some figures to relate you to the story:

Real-Time Stock Prices	
Symbol [⌵]	Price [⌵]
AAPL	36.1043
ABT	101.4707
ADBE	195.2955
AMGN	167.7489
AMZN	172.3264
AVGO	172.3481
AXP	96.4143
BA	71.7762
BAC	91.5732
BIIB	243.3706
BKR	78.5121
BLK	94.3349
BMJ	101.2866
BP	143.4475
BRK.B	96.3834
C	16.3403
CAT	90.8514
CEG	85.8346
CI	97.0313

Symbol
C 

Buy/Sell
Buy 

Quantity
100 

Price
200 

Place Order

This is the stock and the stock price visible from the trading side, where the price of each stock are continuously updated, all from the previous mentioned function – update stock price, that send the latest stock to the stock producer via a channel, that go through the Redpanda and to the trading side.


```

127.0.0.1:6379> hgetall stocks:prices {
{
  "AAPL": "100.0",
  "GOOGL": "100.0",
  "MSFT": "100.0",
  "AMZN": "100.0",
  "NVDA": "100.0",
  "META": "100.0",
  "TSLA": "100.0",
  "CRM": "100.0",
  "ORCL": "100.0",
  "IBM": "100.0",
  "CSCO": "100.0",
  "INTC": "100.0",
  "ADBE": "100.0",
  "QCOM": "100.0",
  "AVGO": "100.0",
  "JNJ": "100.0",
  "PFE": "100.0",
  "MRK": "100.0",
  "ABT": "100.0",
  "MRNA": "100.0",
  "LLY": "100.0",
  "GILD": "100.0",
  "BMY": "100.0",
  "AMGN": "100.0",
  "UNH": "100.0",
  "CI": "100.0",
  "MDT": "100.0",
  "TMO": "100.0",
  "BIIB": "100.0",
  "REGN": "100.0",
  "JPM": "100.0",
  "BAC": "100.0",
  "WFC": "100.0",
  "C": "100.0",
  "AAPL": "Technology",
  "GOOGL": "Technology",
  "MSFT": "Technology",
  "AMZN": "Technology",
  "NVDA": "Technology",
  "META": "Technology",
  "TSLA": "Technology",
  "CRM": "Technology",
  "ORCL": "Technology",
  "IBM": "Technology",
  "CSCO": "Technology",
  "INTC": "Technology",
  "ADBE": "Technology",
  "QCOM": "Technology",
  "AVGO": "Technology",
  "JNJ": "Healthcare",
  "PFE": "Healthcare",
  "MRK": "Healthcare",
  "ABT": "Healthcare",
  "MRNA": "Healthcare",
  "LLY": "Healthcare",
  "GILD": "Healthcare",
  "BMY": "Healthcare",
  "AMGN": "Healthcare",
  "UNH": "Healthcare",
  "CI": "Healthcare",
  "MDT": "Healthcare",
  "TMO": "Healthcare",
  "BIIB": "Healthcare",
  "REGN": "Healthcare",
  "JPM": "Finance",
  "BAC": "Finance",
  "WFC": "Finance",
  "C": "Finance"
}

```

These are the stock prices hash and stock sector hash stored in the database, because this database is redis. There are total of 85 stocks, and total of 8 unique sectors.

```

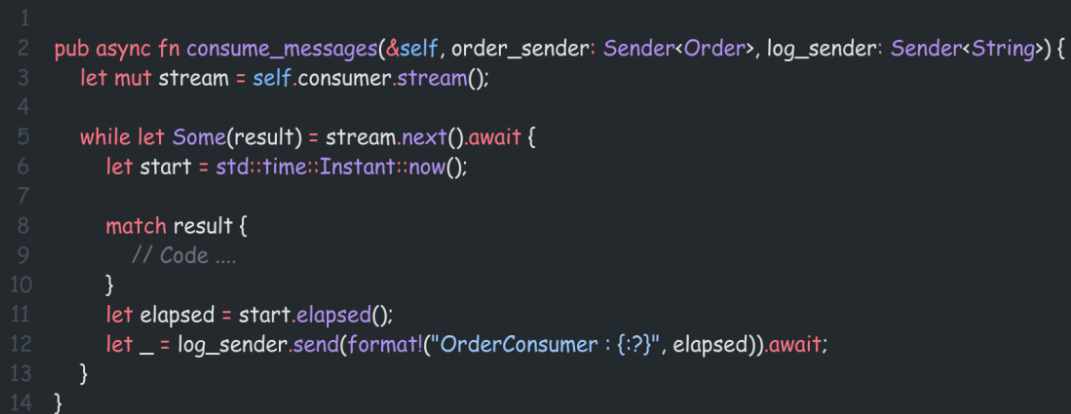
1 let (oms_sender, mut oms_receiver): (Sender<Order>, Receiver<Order>) = channel(100);
2 let (mdg_sender, mdg_receiver): (Sender<Trade>, Receiver<Trade>) = channel(100);
3 let (stock_sender, mut stock_receiver): (Sender<Stock>, Receiver<Stock>) = channel(100);

```

These are the main 3 channels that facilitate the communication between threads in the system. Though there is a fourth channel `let (log_sender, mut log_receiver): (Sender<String>, Receiver<String>) = channel(100);`, it will be mentioned in the later action for logging purpose.

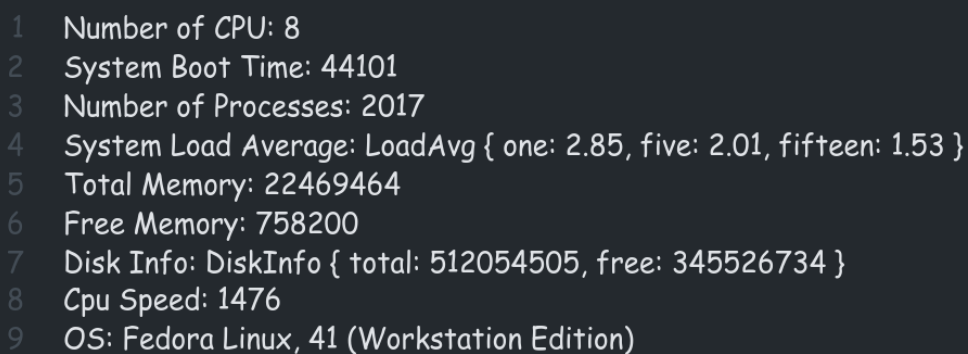
Critical analysis and comparison of simulation

The log channel was used to record the execution time of each task. For example, the following figure shows the code necessary to record the execution time of each message consume, and log it to a log channel to record in a txt file.



```
1
2 pub async fn consume_messages(&self, order_sender: Sender<Order>, log_sender: Sender<String>) {
3     let mut stream = self.consumer.stream();
4
5     while let Some(result) = stream.next().await {
6         let start = std::time::Instant::now();
7
8         match result {
9             // Code ....
10        }
11        let elapsed = start.elapsed();
12        let _ = log_sender.send(format!("OrderConsumer : {:?}", elapsed)).await;
13    }
14 }
```

To start off, this is the system information of the student computer hardware, nothing important.



```
1  Number of CPU: 8
2  System Boot Time: 44101
3  Number of Processes: 2017
4  System Load Average: LoadAvg { one: 2.85, five: 2.01, fifteen: 1.53 }
5  Total Memory: 22469464
6  Free Memory: 758200
7  Disk Info: DiskInfo { total: 512054505, free: 345526734 }
8  Cpu Speed: 1476
9  OS: Fedora Linux, 41 (Workstation Edition)
```

Another advantage of using tokio asynchronous runtime, is that because it is a runtime middleman between application and hardware, it can be monitored with a highly detail crate - “console-subscriber”.

```
views: t = tasks, r = resources
controls: select column (sort) = -- or h, l, scroll = ↑↓ or k, j, view details = ⇐ invert sort (highest/lowest) = 1, scroll to top = gg, scroll to bottom = G, toggle pause = space,
quit = q
```

Warnings

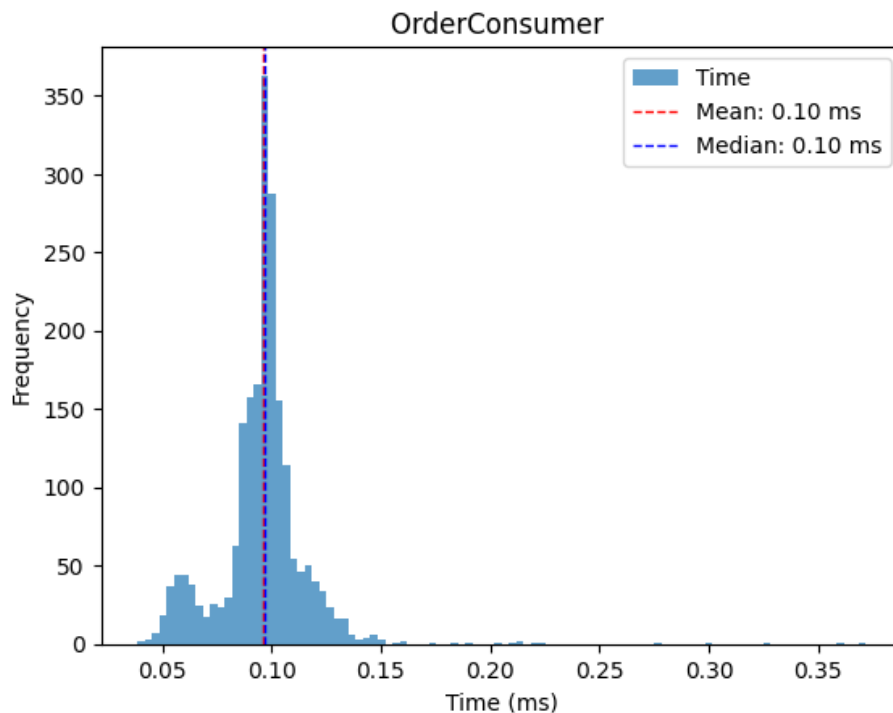
1 tasks are 1024 bytes or larger

Warn	ID	State	Name	Total	Busy	Sched	Idle	Polls	Kind	Location	Fields
	13	II		13s	739µs	0ms	13s	1	task	<cargo>/rdkafka-0.36.2/src/util.rs:474:9	size.bytes-312 target-tok
	14	II		13s	118µs	0ms	13s	1	task	src/main.rs:66:27	size.bytes-800 target-tok
	16	II		13s	72µs	0ms	13s	1	task	<cargo>/redis-0.27.5/src/aio/tokio.rs:178:27	size.bytes-320 target-tok
	17	II		13s	26µs	0ms	13s	1	task	src/main.rs:109:50	size.bytes-880 target-tok
	22	II		13s	283ms	618ms	12s	944	task	src/main.rs:166:27	size.bytes-880 target-tok
	23	II		13s	6ms	4ms	13s	18	task	src/main.rs:177:22	size.bytes-304 target-tok
	26	II		13s	282ms	116ms	13s	878	task	<cargo>/redis-0.27.5/src/aio/tokio.rs:178:27	size.bytes-320 target-tok
	28	II		13s	42µs	0ms	13s	1	task	<cargo>/redis-0.27.5/src/aio/tokio.rs:178:27	size.bytes-320 target-tok
	30	II		13s	142ms	44ms	13s	343	task	<cargo>/redis-0.27.5/src/aio/tokio.rs:178:27	size.bytes-320 target-tok
	32	II		13s	101ms	46ms	13s	511	task	<cargo>/redis-0.27.5/src/aio/tokio.rs:178:27	size.bytes-320 target-tok
	33	II		13s	205ms	33ms	13s	258	task	/home/shine/Documents/Rust/Rust_Asgn/stock_side/market_data_generator/src/price_updater.rs:61:13	size.bytes-416 target-tok
	34	II		13s	503ms	16ms	12s	78	task	/home/shine/Documents/Rust/Rust_Asgn/stock_side/market_data_generator/src/price_updater.rs:82:9	size.bytes-936 target-tok
	35	II		13s	38µs	0ms	13s	1	task	/home/shine/Documents/Rust/Rust_Asgn/stock_side/market_data_generator/src/price_updater.rs:106:9	size.bytes-800 target-tok
1	36	II		13s	161ms	31ms	13s	172	task	/home/shine/Documents/Rust/Rust_Asgn/stock_side/market_data_generator/src/price_updater.rs:124:9	size.bytes-1032 target-to

Using "console-subscriber", monitoring of the runtime can be achieved, this figure told the story of an average of 14 tasks are always active to keep the application running, it is visible there is 1 Kafka task, that keep consume message, and 5 Redis thread (assumption made for this are internal load of the Redis library).



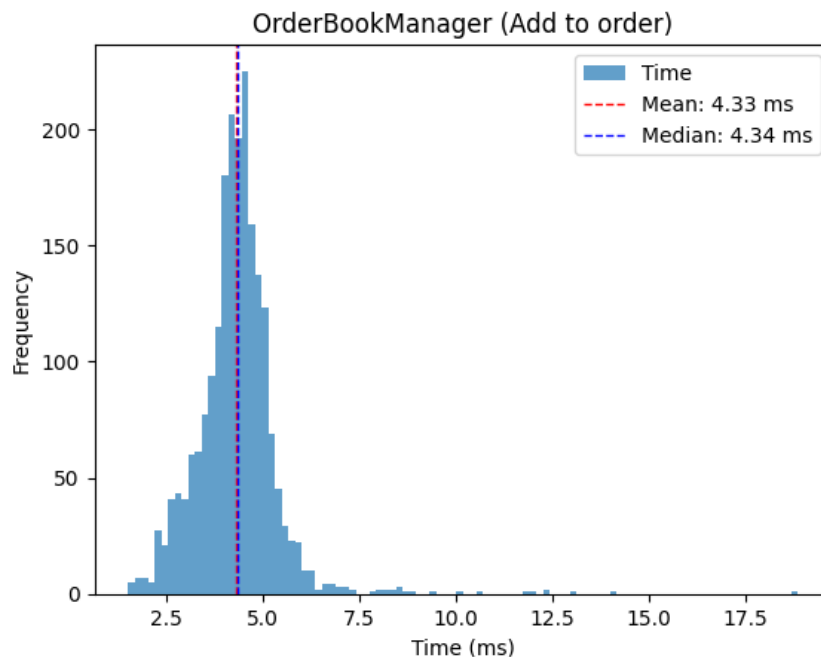
The detail of Consumer.consumer_message task, this is the task that stream consume message from trading side for Order, can be seen that even at the worst, p99, it still manages to keep its poll time at only 2.18ms.



Further analysis on the performance of this thread shows that it manages to keep a mean execution time of 0.10ms.



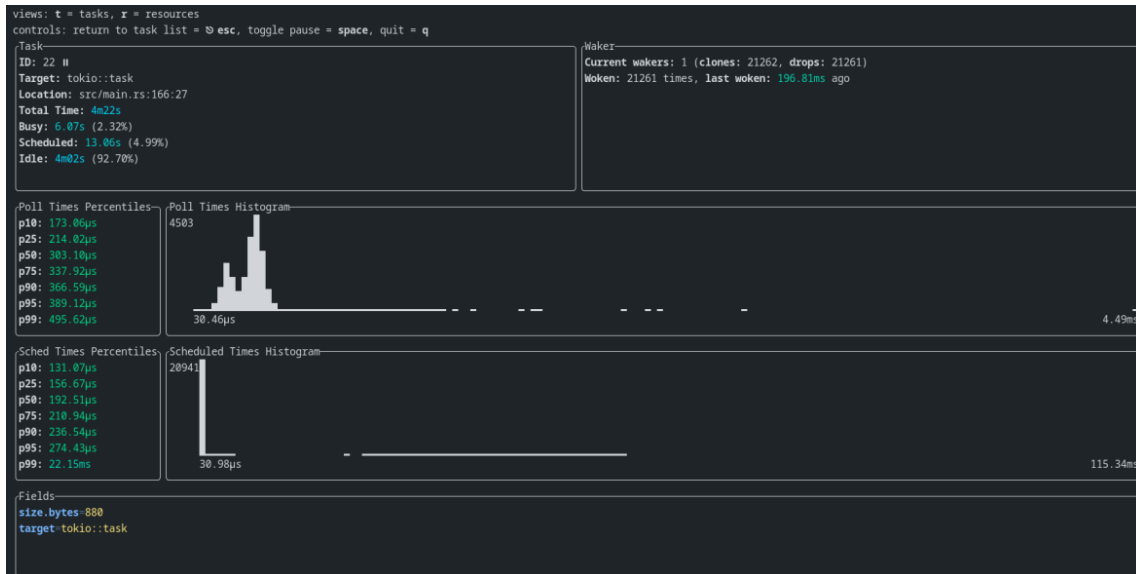
This is the thread of Order book manager, where it manages the add to database as well as process order to check for potential trade, even with massive task load and that of heavy dependency on Redis, it still manages to keep the poll time extremely low and steady.



With the first part, add to order performing at execution time of only mean time of 4.33 ms. This function required to read from the database (Redis) and write it back, and sorting the order book.



And the second part, the process order/ check for trade, manages to keep the execution time so low, at mean of 6.78 ms. It can be visible that the graph show two hump/ peak, the analysis on the code found out that the smaller time peak (left) belong to when there is no trade able to made; and the higher time peak (right) belongs to when a trade is found, then a trade has to be made, and calculate the stock quantity deduction, and save it back to both the buy order and sell order.



This tell the story of `Producer.produce_message()`, where it is the thread that produce message back to the trading side, it wait on a channel from the 'update stock price' function, and then send it to the Redpanda, the number are within microseconds, similar to stream consuming message.



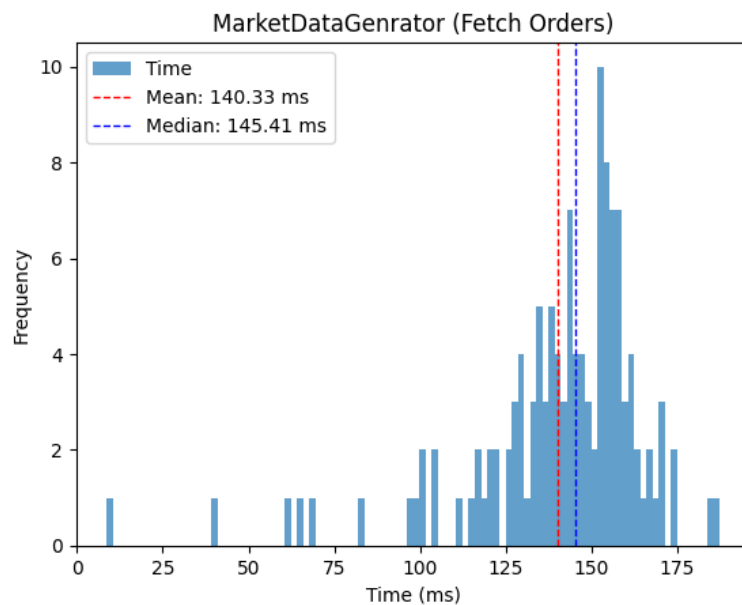
Tasks supporting Task 1: to fetch the orders and update the `shared_orders` every 5 seconds. Explanation below, but performance wise, this thread is stable and polled on average every millisecond.

This thread task is created to support the passive update task (mentioned above), after some critical performance analysis of the system, it is found out that the 1 second interval task – Passive Update, is too resource intensive, due to every second needing a fresh release of the whole copy of order book, hence for a better performance, and acceptance of (5 – 1) seconds offset of latest copy of order book, this was dedicated it's own thread that execute every 5 seconds and update the shared_order for the passive update to use.

```

1 let shared_orders: Arc<AtomicPtr<HashMap<String, (Vec<Order>, Vec<Order>)>>> = Arc::clone(&shared_orders);
2 tokio::spawn({
3     let log_sender = log_sender.clone();
4     async move {
5         let mut interval = interval(Duration::from_secs(5));
6         loop {
7             interval.tick().await; // Wait until the next 5-second interval
8             let start = std::time::Instant::now();
9             let orders = fetch_orders(&mut redis_conn).await.unwrap();
10            let new_orders = Box::new(orders);
11            let old_orders = shared_orders.swap(Box::into_raw(new_orders), Ordering::SeqCst);
12            unsafe {
13                let _ = Box::from_raw(old_orders); // Free the old orders
14            }
15            let elapsed = start.elapsed();
16            let _ = log_sender.send(format!("MarketDataGenrator (Fetch Orders): {:?}", elapsed)).await;
17        }
18    }
19 });

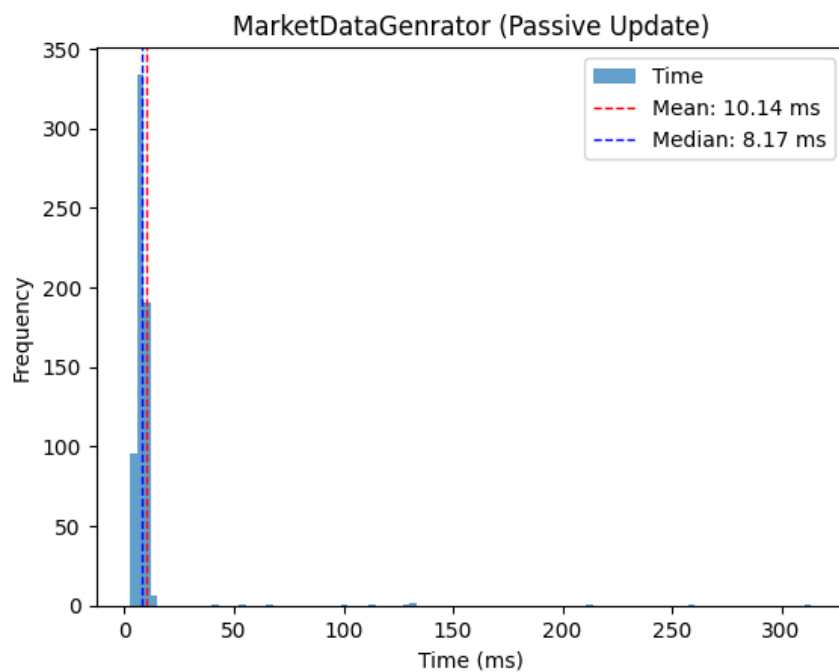
```



The execution time for this intensive task is at 140 ms average, which in compared to other, is quite bad, that is why, it needs to be at its own thread and not impacting the Passive Update thread.



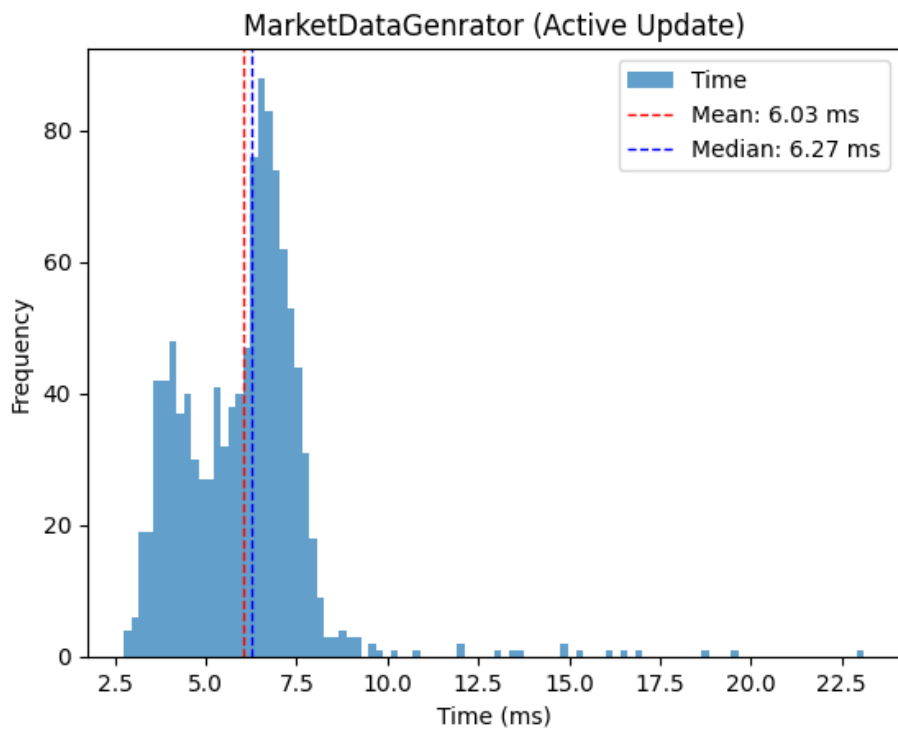
Task 1: passive update



Without fetching order in the logic, this resource intensive thread manages to cut down to only 10 ms average execution time, giving it so much more potential to faster the interval time.



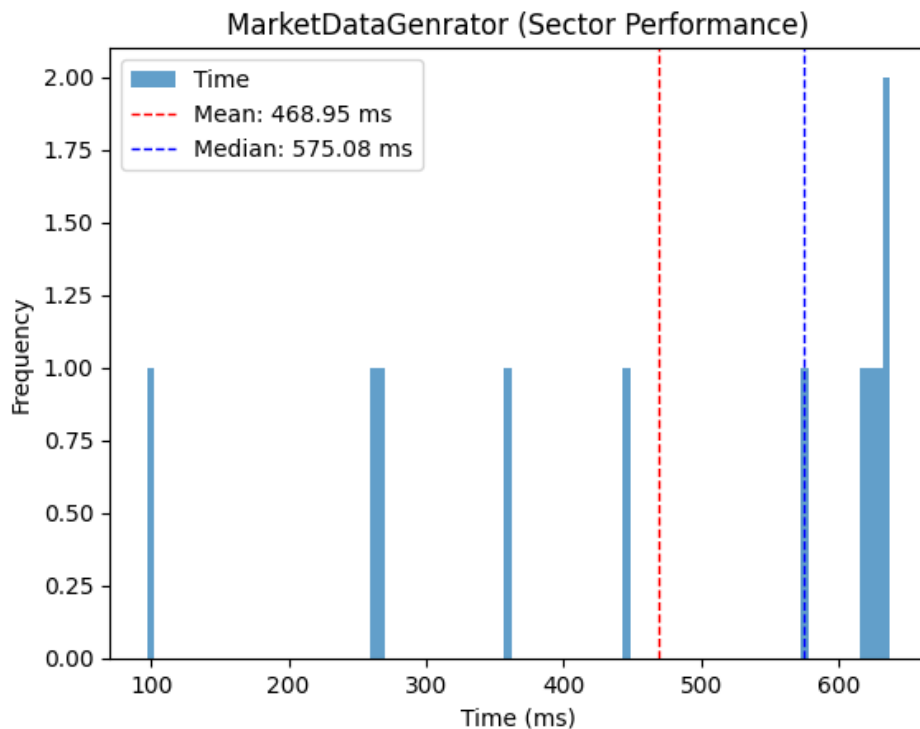
Task 2: active update, poll time on stable of around 1 ms is still acceptable at good value.



This active update, where it gets the trade from the channel and executes, is able to maintain its execution time at mean of 6 ms.



Task 3: sector performance, mentioned above, this task run only every 60s due to the amount of load it needs to execute, and on assumption of the domain, it is within acceptable that a sector influence algorithm doesn't need to execute at high frequency.



Unsurprisingly, this heavy load task that need to ping the database for the whole price and sector, and perform complex calculations, come at a mean execution time of 469 ms.

Throughout the application logic, there is no use of ARC nor mutex for the redis data except fetch order task (more detail few page later), which is an important tool when it comes to sharing resources in a concurrency environment. However, this is because of the decision made by the student to leverage redis as a database, instead of a local data, which can be hashmap (fastest), or text file.

However, in normal application, when multiple thread trying to access a database, for example Postgres, or MySQL, there still need implementations of mutex lock to ensure one command to the database at a single time to prevent race condition. However, all this is overcome by the use of Redis, this is because Redis is only single-threaded application, which automatically makes it race-safe by default, this allows a simple and fast operation. (<https://www.dragonflydb.io/faq/how-does-redis-handle-concurrency>)

```
for _ in 1..5 {
  let redis_conn = redis_conn.clone();

  let handle = tokio::spawn(async move {
    let mut con = redis_conn.clone();
    for _ in 0..n {
      let field = FIELDS[rand::thread_rng().gen_range(0..3)];
      let value = rand::thread_rng().gen_range(0..1000).to_string();

      let () = con.hset("my_key", field, value).await.unwrap();
    }
  });
}
```

```
1734514528.210128 [0 172.18.0.1:59002] "HSET" "my_key" "field3" "38"
1734514528.309464 [0 172.18.0.1:59002] "HSET" "my_key" "field3" "236"
1734514528.311612 [0 172.18.0.1:59002] "HSET" "my_key" "field3" "421"
1734514528.311632 [0 172.18.0.1:59002] "HSET" "my_key" "field2" "64"
1734514528.410313 [0 172.18.0.1:59002] "HSET" "my_key" "field3" "763"
1734514528.411470 [0 172.18.0.1:59002] "HSET" "my_key" "field2" "619"
1734514528.411487 [0 172.18.0.1:59002] "HSET" "my_key" "field1" "399"
1734514528.509780 [0 172.18.0.1:59002] "HSET" "my_key" "field2" "756"
1734514528.509799 [0 172.18.0.1:59002] "HSET" "my_key" "field2" "834"
1734514528.510894 [0 172.18.0.1:59002] "HSET" "my_key" "field3" "928"
1734514528.610388 [0 172.18.0.1:59002] "HSET" "my_key" "field3" "658"
1734514528.611892 [0 172.18.0.1:59002] "HSET" "my_key" "field3" "834"
1734514528.611912 [0 172.18.0.1:59002] "HSET" "my_key" "field3" "931"
1734514528.709406 [0 172.18.0.1:59002] "HSET" "my_key" "field3" "210"
1734514528.710959 [0 172.18.0.1:59002] "HSET" "my_key" "field2" "42"
1734514528.710982 [0 172.18.0.1:59002] "HSET" "my_key" "field2" "122"
1734514528.810415 [0 172.18.0.1:59002] "HSET" "my_key" "field3" "506"
1734514528.811991 [0 172.18.0.1:59002] "HSET" "my_key" "field1" "851"
1734514528.812022 [0 172.18.0.1:59002] "HSET" "my_key" "field3" "733"
1734514528.909963 [0 172.18.0.1:59002] "HSET" "my_key" "field1" "60"
1734514528.909987 [0 172.18.0.1:59002] "HSET" "my_key" "field2" "977"
1734514528.911357 [0 172.18.0.1:59002] "HSET" "my_key" "field2" "705"
1734514529.009991 [0 172.18.0.1:59002] "HSET" "my_key" "field2" "569"
1734514529.010017 [0 172.18.0.1:59002] "HSET" "my_key" "field3" "201"
1734514529.011307 [0 172.18.0.1:59002] "HSET" "my_key" "field1" "156"
1734514529.109933 [0 172.18.0.1:59002] "HSET" "my_key" "field1" "171"
1734514529.109987 [0 172.18.0.1:59002] "HSET" "my_key" "field2" "976"
1734514529.109997 [0 172.18.0.1:59002] "HSET" "my_key" "field2" "699"
1734514529.209656 [0 172.18.0.1:59002] "HSET" "my_key" "field3" "718"
1734514529.209690 [0 172.18.0.1:59002] "HSET" "my_key" "field1" "527"
1734514529.209696 [0 172.18.0.1:59002] "HSET" "my_key" "field3" "193"
```

The figure on the left show a simple experiment to have 4 threads that parallelly execute the redis command of setting a new value to the redis.

By monitoring the log of Redis command, it can be verified that there is no possibility for race condition to happen in Redis. The figure at the left, show the log of Redis, during the experiment, where all the attempt action of all 4 threads from the experiment are all executed on

the Redis in a single thread – single command manners.

This allow the option to not used ARC or mutex in the application. Although in compare redis to native data storage, such as hashmap with a arc and mutex, redis surely cannot compete with them, but using redis allow for it serve as a database, and easier accessibility to other part of the application.

stage	iters	secs	msecs	iters/s	diff.s
mutex	1_000	0.055	55.338	18_070	+2534.11 %
redis	1_000	1.456	1455.924	686	


```

1 use tokio::time::{sleep, Duration};
2
3 let handler = tokio::spawn(async {
4     for _ in 0..10 {
5         // Sleep for 300ms, simulate some work, don't use tokio::time::sleep
6         sleep(Duration::from_millis(300)).await;
7         // Wait for 1 second
8         sleep(std::time::Duration::from_secs(1)).await;

```

```

I am doing something, TIME: 42:564
I am done, TIME: 42:865

I am doing something, TIME: 43:866
I am done, TIME: 44:167

I am doing something, TIME: 45:168
I am done, TIME: 45:469

I am doing something, TIME: 46:471
I am done, TIME: 46:772

I am doing something, TIME: 47:774
I am done, TIME: 48:075

I am doing something, TIME: 49:077
I am done, TIME: 49:378

I am doing something, TIME: 50:379
I am done, TIME: 50:680

I am doing something, TIME: 51:682
I am done, TIME: 51:983

I am doing something, TIME: 52:985
I am done, TIME: 53:286

I am doing something, TIME: 54:288
I am done, TIME: 54:589

```

In the stock system application code, there are many scheduling task as seen such as that of every second passive update, or every minute sector performance task, the initial scheduling method for them are using tokio sleep, in an easy assumption, of sleeping every seconds, then do the task again.

However, after some experiment, the tokio sleep is not giving the student the intended behavior for the task, as visible in the output at the figure left, using tokio sleep to schedule task, is not accurate, as in this case, the scheduling of this task consider in the time taken for the work to execute, execution time.



```
1 use tokio::time::{interval, Duration};
2
3 let mut interval = interval(Duration::from_secs(1));
4
5 for _ in 0..10 {
6     interval.tick().await; // Wait until the next 1-second interval
```

```
I am starting, TIME: 41:573
I am done, TIME: 42:073

I am starting, TIME: 42:573
I am done, TIME: 43:073

I am starting, TIME: 43:573
I am done, TIME: 44:073

I am starting, TIME: 44:573
I am done, TIME: 45:073

I am starting, TIME: 45:573
I am done, TIME: 46:073

I am starting, TIME: 46:573
I am done, TIME: 47:073

I am starting, TIME: 47:573
I am done, TIME: 48:073

I am starting, TIME: 48:573
I am done, TIME: 49:073

I am starting, TIME: 49:573
I am done, TIME: 50:073

I am starting, TIME: 50:573
I am done, TIME: 51:073
```

So there is a switch into accurate scheduling in tokio environments, that is the tokio interval. Tokio interval will schedule the task in work accurately every intended time, not taking in account of the execution time of the task. However, there is always a concern of whether the task has been execute finish that trigger the execution again, but it is mentioned above, the shortest scheduled task is the passive update at 1 second interval, but mentioned above that this passive update task only require 10 milliseconds to complete execution.

Even if we look at some sporadic task, such as the shortest one, those task that follow the order take in – order consumer, add to orderbook, and process order.

Order Consumer – 0.10ms,
Add to Orderbook – 4.33ms,
Check for trade – 6.78ms,
Total = 11.21ms

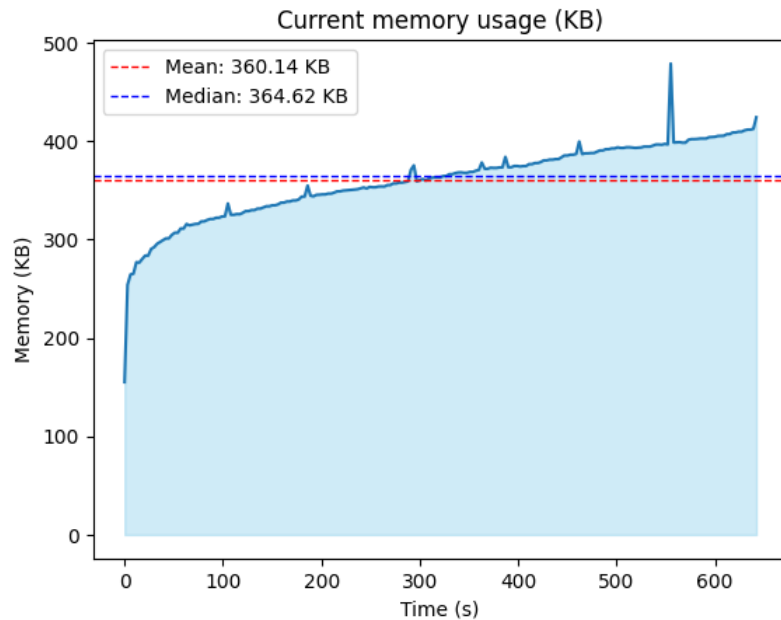
Which is significantly lesser than the average order come in time (mentioned above) of 300 ms.

Another interesting optimization change in the code is the use of threadpool. To keep the passive update task to low at only 10ms (mentioned above), there are many actions considered, one such action is the previous mentioned use of dedicated thread for fetch order that have 4 seconds offset with the passive update. Together with that optimization that bring down passive update from 400ms into just 10ms is the use of ThreadPool to help update the market stock price into the redis, after some analysis, it is found that writing to redis is eating up a lot of resource (execution time), so it is given a thread pool to the update stock redis to use it to execute update stock price so it become a non-blocking task for passive update.

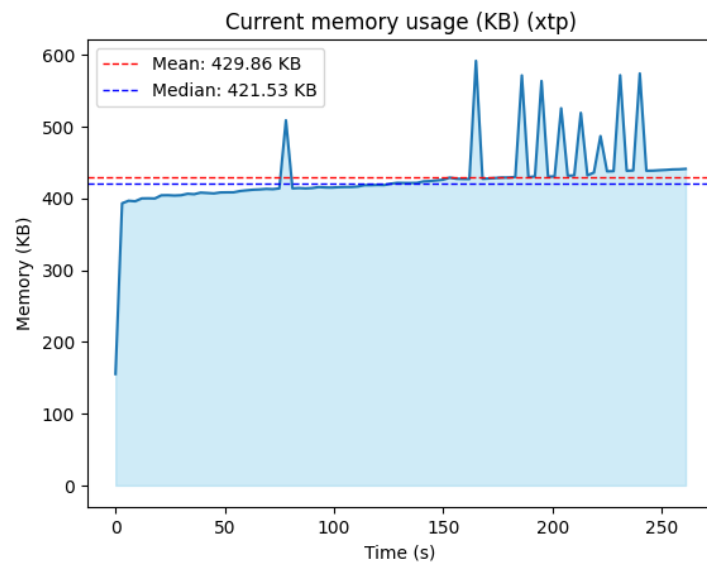
```
1 // Update the stock price in Redis using thread pool
2 if let Some(pool) = pool {
3     let mut redis_conn = redis_conn.clone();
4     let updated_stock = updated_stock.clone();
5     pool.execute(move || {
6         let future = redis_conn.hset("stocks:prices", &updated_stock.symbol, &updated_stock.price);
7         let _result: Result<(), RedisError> = futures::executor::block_on(future);
8     });
9 }
```


There is many experiment and analysis taken to measure the impact of Threadpool on the application beside just bringing down the execution time by ~110ms.

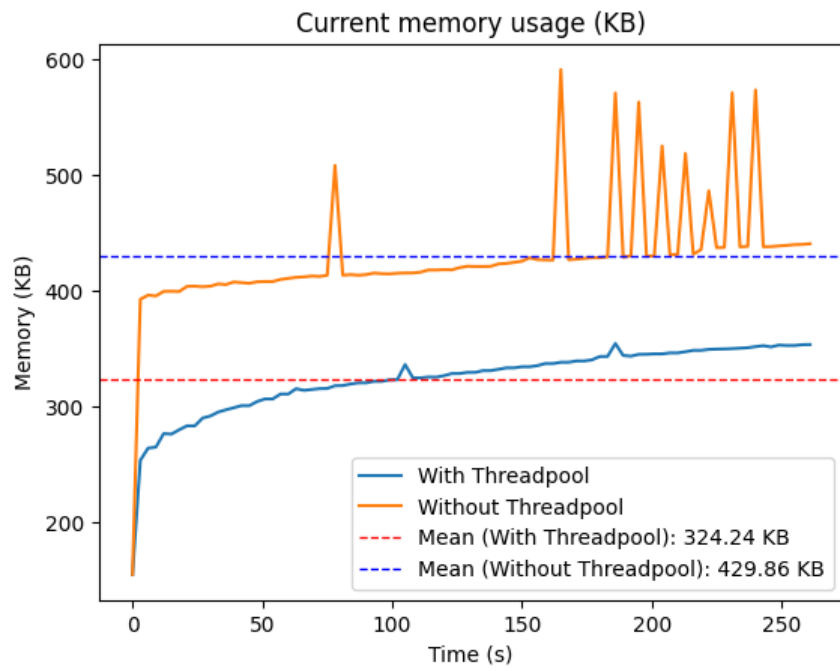
Using thread Pool



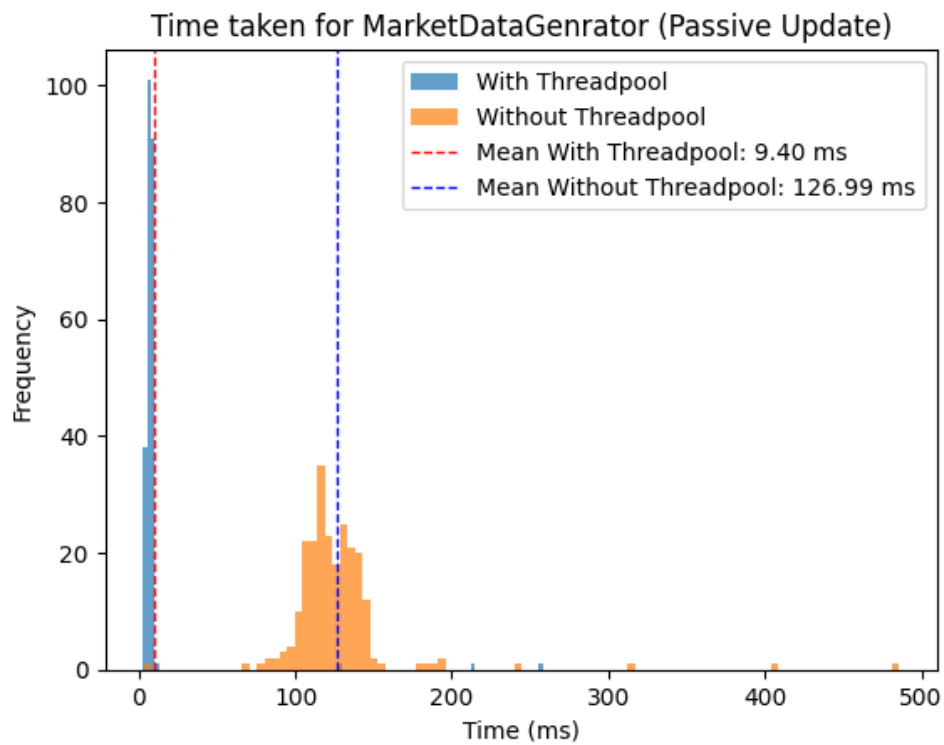
This is the memory measurement of the application through out it's life time of 11 minutes, it averages out at only 360 KB.



And this is the memory measurement for the application for just 5 minutes, the average memory usage is at 430 KB.



This is the comparison of the 2 applications, one with thread pool and another with threadpool for the same first 5 minutes.



This is the execution time of both with threadpool and without threadpool, without threadpool, the execution time average at 125 ms, whereas with threadpool, it fell only to 9.4 ms (the first 5 minutes (slight diff from 10ms (mentioned above) of 11 minutes))

Communication

Redpanda vs Kafka

The following section will start with comparison between Redpanda and Kafka, it will be facilitated with docker, and the contestant are Redpanda, bitnami kafka (known for it's lightweight/ simplicity), and confluentinc kafka (full of feature, heavy).

At the time of record, the bitnami kafka version is at [v3.9.0](#)

All platforms only have 1 cluster and broker. The figure below is the docker compose setups for all three data platforms.

```

1 networks:
2   data_service:
3     driver: bridge
4
5 services:
6   redpanda:
7     container_name: redpanda
8     hostname: redpanda
9     image: redpandadata/redpanda:v24.2.7
10    networks:
11      - data_service
12    ports:
13      - "18081:18081"
14      - "18082:18082"
15      - "19092:19092"
16      - "19644:9644"
17    command:
18      - redpanda
19      - start
20      - --kafka-addr internal://0.0.0.0:9092,external://0.0.0.0:19092
21      - --advertise-kafka-addr internal://redpanda:9092,external://localhost:19092
22      - --pandaproxy-addr internal://0.0.0.0:8082,external://0.0.0.0:18082
23      - --advertise-pandaproxy-addr internal://redpanda:8082,external://localhost:18082
24      - --schema-registry-addr internal://0.0.0.0:8081,external://0.0.0.0:18081
25      - --rpc-addr redpanda:33145
26      - --advertise-rpc-addr data_redpanda:33145
27      - --mode dev-container
28      - --smp 1
29      - --default-log-level=info
30
31    confluentinc-kafka:
32      image: confluentinc/cp-kafka:7.8.0
33      container_name: confluentinc-kafka
34      hostname: confluentinc-kafka
35      networks:
36        - data_service
37      ports:
38        - "9101:9101"
39        - "9092:9092" # Ensure this is included for PLAINTEXT_HOST listener
40      environment:
41        KAFKA_NODE_ID: 1
42        KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: 'CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT'
43        KAFKA_ADVERTISED_LISTENERS: 'PLAINTEXT://confluentinc-kafka:29092,PLAINTEXT_HOST://localhost:9092'
44        KAFKA_JMX_PORT: 9101
45        KAFKA_JMX_HOSTNAME: localhost
46        KAFKA_PROCESS_ROLES: 'broker,controller'
47        KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
48        KAFKA_CONTROLLER_QUORUM_VOTERS: '1@confluentinc-kafka:29093'
49        KAFKA_LISTENERS: 'PLAINTEXT://confluentinc-kafka:29092,CONTROLLER://confluentinc-kafka:29093,PLAINTEXT_HOST://0.0.0.0:9092'
50        KAFKA_INTER_BROKER_LISTENER_NAME: 'PLAINTEXT'
51        KAFKA_CONTROLLER_LISTENER_NAMES: 'CONTROLLER'
52        CLUSTER_ID: 'MkU3OEVBNTcwNTJENDM2Qk'
53
54    bitnami-kafka:
55      image: 'bitnami/kafka:latest'
56      container_name: bitnami-kafka
57      hostname: bitnami-kafka
58      networks:
59        - data_service
60      ports:
61        - '9094:9094'
62      environment:
63        - KAFKA_CFG_NODE_ID=0
64        - KAFKA_CFG_PROCESS_ROLES=controller,broker
65        - KAFKA_CFG_LISTENERS=PLAINTEXT://:9092,CONTROLLER://:9093,EXTERNAL://:9094
66        - KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP=CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT,EXTERNAL:PLAINTEXT
67        - KAFKA_CFG_ADVERTISED_LISTENERS=PLAINTEXT://bitnami-kafka:9092,EXTERNAL://localhost:9094
68        - KAFKA_CFG_CONTROLLER_QUORUM_VOTERS=0@bitnami-kafka:9093
69        - KAFKA_CFG_CONTROLLER_LISTENER_NAMES=CONTROLLER

```

Using Kafka benchmark tool, <https://github.com/fede1024/kafka-benchmark>. The performance benchmarking of the data streaming platform can be measured. Mainly on the throughput. The benchmark test consists of few variants, and also there are 2 types of producer and 2 types of consumer: Base Producer, Future Producer, Base Consumer, and Stream Consumer. The following are the test configuration for all 4 types of API.

Base Consumer

```
1 ---
2 default:
3   repeat_times: 1
4   repeat_pause: 0
5   consumer_type: BaseConsumer
6   topic: test_topic1
7   consumer_config:
8     bootstrap.servers: localhost:19092 # Redpanda
9     # bootstrap.servers: localhost:9092 # Confluentinc Kafka
10    # bootstrap.servers: localhost:9094 # Bitnami Kafka
11    group.id: 'benchmark_consumer_group'
12    auto.offset.reset: smallest
13    enable.auto.commit: false
14    session.timeout.ms: 6000
15
16 scenarios:
17   10B:
18     topic: benchmark_topic_10B
19     message_limit: 15000000
20
21   1KB:
22     topic: benchmark_topic_1KB
23     message_limit: 375000
24
25   10KB:
26     topic: benchmark_topic_10KB
27     message_limit: 37500
28
```

Base Producer

```
1  ---
2  default:
3    threads: 1
4    # threads: 4
5    producer_type: BaseProducer
6    producer_config:
7      bootstrap.servers: localhost:19092 # Redpanda
8      # bootstrap.servers: localhost:9092 # Confluentinc Kafka
9      # bootstrap.servers: localhost:9094 # Bitnami Kafka
10     queue.buffering.max.ms: 100
11     queue.buffering.max.messages: 1000000
12
13  scenarios:
14    10B_bursts:
15      repeat_times: 5
16      repeat_pause: 10
17      topic: benchmark_topic_10B
18      message_size: 10
19      message_count: 20000000
20
21    1KB_bursts:
22      repeat_times: 3
23      repeat_pause: 20
24      topic: benchmark_topic_1KB
25      message_size: 1024
26      message_count: 500000
27
28    10KB_bursts:
29      repeat_times: 3
30      repeat_pause: 20
31      topic: benchmark_topic_10KB
32      message_size: 10240
33      message_count: 50000
34
```

Stream Consumer

```
1 default:
2   repeat_times: 1
3   repeat_pause: 0
4   consumer_type: StreamConsumer
5   topic: test_topic1
6   consumer_config:
7     bootstrap.servers: localhost:19092 # Redpanda
8     # bootstrap.servers: localhost:9092 # Confluentinc Kafka
9     # bootstrap.servers: localhost:9094 # Bitnami Kafka
10    group.id: 'benchmark_consumer_group'
11    auto.offset.reset: smallest
12    enable.auto.commit: false
13    session.timeout.ms: 6000
14
15 scenarios:
16   10B:
17     topic: benchmark_topic_10B
18     message_limit: 15000000
19
20   35B:
21     topic: benchmark_topic_35B
22     message_limit: 2250000
23
24   210B:
25     topic: benchmark_topic_210B
26     message_limit: 1500000
27
28   1KB:
29     topic: benchmark_topic_1KB
30     message_limit: 375000
31
32   10KB:
33     topic: benchmark_topic_10KB
34     message_limit: 37500
35
```


Future Producer

```

1  ---
2  default:
3    threads: 1
4    # threads: 4
5    producer_type: FutureProducer
6    producer_config:
7      bootstrap.servers: localhost:19092 # Redpanda
8      # bootstrap.servers: localhost:9092 # Confluentinc Kafka
9      # bootstrap.servers: localhost:9094 # Bitnami Kafka
10     queue.buffering.max.ms: 100
11     queue.buffering.max.messages: 1000000
12
13  scenarios:
14    10B_bursts:
15      repeat_times: 5
16      repeat_pause: 10
17      topic: benchmark_topic_10b
18      message_size: 10
19      message_count: 20000000
20
21    35B_bursts:
22      repeat_times: 5
23      repeat_pause: 10
24      topic: benchmark_topic_35B
25      message_size: 35
26      message_count: 3000000
27
28    210B_bursts:
29      repeat_times: 5
30      repeat_pause: 10
31      topic: benchmark_topic_210B
32      message_size: 210
33      message_count: 2000000
34
35    1KB_bursts:
36      repeat_times: 3
37      repeat_pause: 20
38      topic: benchmark_topic_1kb
39      message_size: 1024
40      message_count: 500000
41
42    10KB_bursts:
43      repeat_times: 3
44      repeat_pause: 20
45      topic: benchmark_topic_10kb
46      message_size: 10240
47      message_count: 50000
48

```

It can be seen in both Stream Consumer and Future Producer, there are additional 2 test scenarios: 35B and 210B. These 2 scenarios are specially for this assignment of stock trading system

35B comes from the stock price that is sent from the stock side to the trading side, as you can see, on average, it is only around 10B of data that is being sent to the trading side.

>	12/19/2024, 5:08:36 PM	UL TEXT - 2 B	{"symbol": "UL", "price": 118.6615} JSON - 32 B
>	12/19/2024, 5:08:36 PM	PSX TEXT - 3 B	{"symbol": "PSX", "price": 121.6351} JSON - 33 B
>	12/19/2024, 5:08:36 PM	VZ TEXT - 2 B	{"symbol": "VZ", "price": 89.5621} JSON - 31 B
>	12/19/2024, 5:08:36 PM	AMGN TEXT - 4 B	{"symbol": "AMGN", "price": 118.593} JSON - 33 B

The similar goes for the 210B, it is the average size of the Trade Order place and send by the trading side into the stock side.

>	12/19/2024, 3:50:31 PM	340b7b5a-cd1a-4dc2-98da-183bde54d648 TEXT - 36 B	{"id": "340b7b5a-cd1a-4dc2-98da-183bde54d648", "stock_symbol": "LLY", "order_type": ... JSON - 159 B
>	12/19/2024, 3:50:32 PM	57dc37e5-1398-4d06-a94b-41305340e57c TEXT - 36 B	{"id": "57dc37e5-1398-4d06-a94b-41305340e57c", "stock_symbol": "BP", "order_type": ... JSON - 170 B
>	12/19/2024, 3:50:32 PM	562f905c-6941-4702-807d-d42f0a45f266 TEXT - 36 B	{"id": "562f905c-6941-4702-807d-d42f0a45f266", "stock_symbol": "CVX", "order_type": ... JSON - 169 B
>	12/19/2024, 3:50:32 PM	769f2142-5150-4e6f-9460-c28d61d47122 TEXT - 36 B	{"id": "769f2142-5150-4e6f-9460-c28d61d47122", "stock_symbol": "CVX", "order_type": ... JSON - 170 B

And only Stream Consumer and Future Producer have it is because only this two is in the student application code.

All topic is 1 partition

Consumer	Redpanda		Bitnami kafka		Confluentinc kafka	
	msg/s	MB/s	msg/s	MB/s	msg/s	MB/s
Base consumer - 10B	1319842	12.587	1316829	12.558	1346983	12.846
Base consumer - 1KB	54371	53.097	43993	42.962	42469	41.473
Base consumer - 10KB	5878	57.398	4944	48.278	4874	47.596
Stream Consumer – 10B	178035	1.698	180527	1.722	226518	2.160
Stream Consumer – 35B	604189	20.167	583204	19.467	686604	22.918
Stream Consumer – 210B	195491	39.151	176762	35.400	148544	29.749
Stream Consumer – 1KB	57657	56.305	50261	49.083	41620	40.645
Stream Consumer – 10KB	5524	53.948	5454	53.256	4418	43.143

For the consumer, it is only 1 thread that is consuming, it can be seen that overall, there is not much difference between all 3 data streaming platforms. For detail analysis, some insight can be deduced, on Base Consumer, if the size of the data value is considered quite large, for instance 1KB and more, then there is a slight advantage in Redpanda, though take is with +/-20% performance for all metrics. As for Stream Consumer, some insight can be seen, it is that for the smallest data value of 10B, the throughput drop by a lot, around 5~8 times of that of Base Consumer. And same there is only a slight advantage of Redpanda when the data size is 1KB, however it doesn't hold true for 10KB, as bitnami kafka able to keep up with the performance.

Producer – 1 Thread

Producer	Redpanda		Bitnami kafka		Confluentinc kafka	
1 thread	msg/s	MB/s	msg/s	MB/s	msg/s	MB/s
Base producer – 10B burst	838195	7.994	934248	8.910	968120	9.233
Base producer – 1KB burst	89633	87.532	89966	87.857	99973	97.630
Base producer – 10KB burst	9876	96.447	8543	83.429	8883	86.749
Future producer – 10B burst	470393	4.486	454901	4.338	456871	4.357
Future producer – 35B burst	427058	14.255	431468	14.402	416875	13.915
Future producer – 210B burst	292689	58.617	289193	57.917	294455	58.971
Future producer – 1KB burst	83551	81.593	78972	77.121	81829	79.911
Future producer – 10KB burst	9690	94.628	8904	86.955	8705	85.007

For this test, it is about the producer and using only 1 thread to produce the message, some insight can be seen is that in 10B data size, Redpanda is at a disadvantage compare to kafka, even taking in account of +20% accuracy. And for producer, at the 1KB data size, the advantage seem to be held by the more complex kafka image – confluentinc. At the 10KB data size, Redpanda manage to gain slight advantage, though not much when factor in +/-20%. As for future producer, the story goes about same, all the performance are on par only until 10KB data size, however it just manage to gain slight advantage.

Producer – 4 Thread

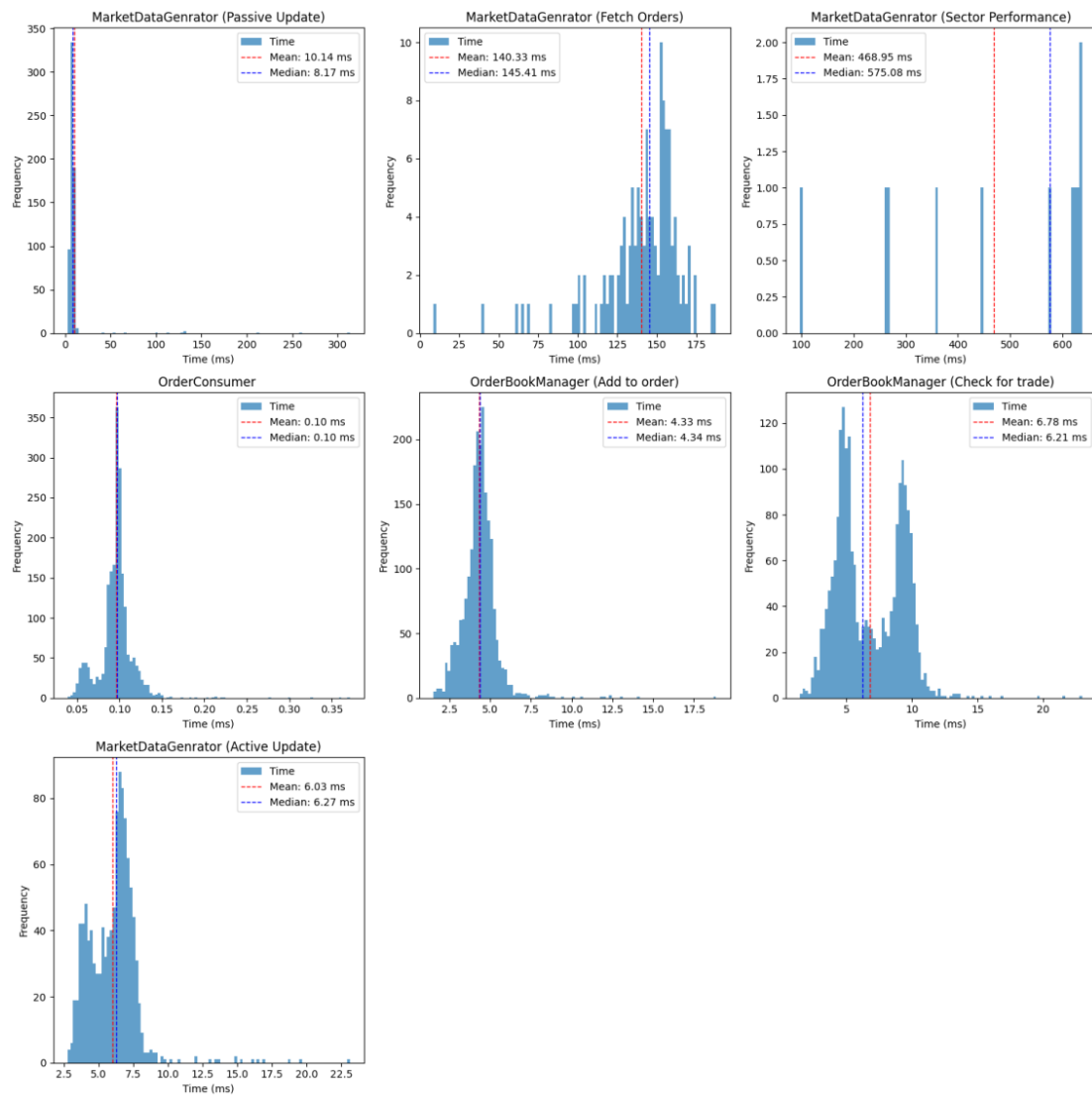
Producer	Redpanda		Bitnami kafka		Confluentinc kafka	
4 threads	msg/s	MB/s	msg/s	MB/s	msg/s	MB/s
Base producer – 10B burst	2327295	22.195	2154940	20.551	2257761	21.532
Base producer – 1KB burst	91191	89.054	93519	91.327	97040	94.766
Base producer – 10KB burst	10549	103.013	9674	94.473	10638	103.890
Future producer – 10B burst	1200576	11.450	1113296	10.617	1180916	11.262
Future producer – 35B burst	937989	31.309	882007	29.440	956023	31.911
Future producer – 210B burst	373459	74.793	390650	78.236	377382	75.579
Future producer – 1KB burst	95147	92.917	80736	78.844	97590	95.302
Future producer – 10KB burst	12206	119.195	9719	94.913	10762	105.097

This is the same producer test, but this time with 4 threads to multiple produce, it can be said the performance of Redpanda is the same as Kafka. However, it is only the confluentinc kafka that match up to the performance, it seem the simpler kafka image – bitnami kafka, is not able to get the full advantage of multiple threads.

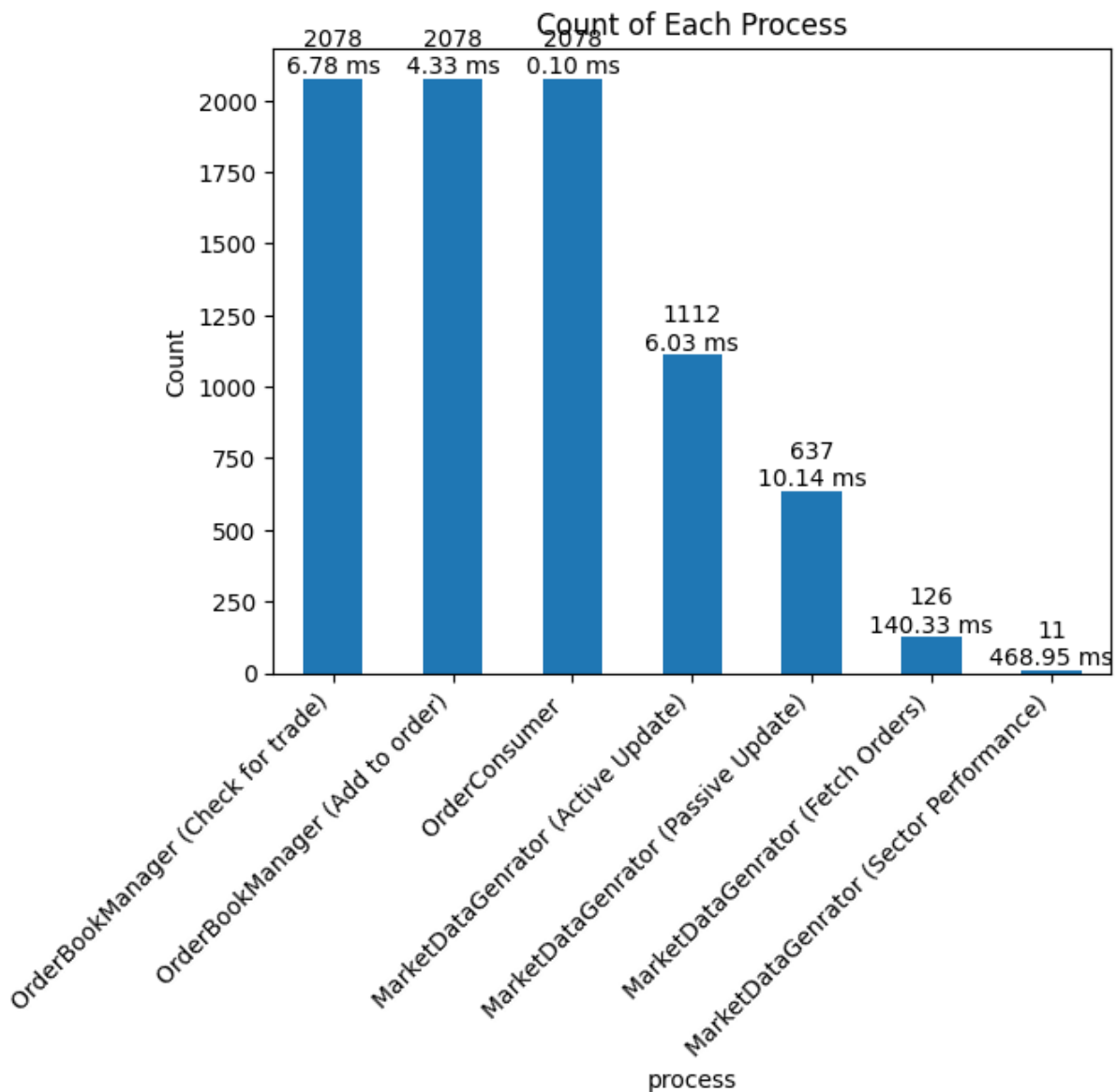
Conclusion

There is not really much performance advantage that Redpanda claimed to have.

Result



This is the super graph for all the execution time of 7 recorded tasks. The major ones together, are within 126 milliseconds, which is lower than that of London Stock Exchange (Wikipedia, 2024).



This is the recorded task execution count in the application time of 11 minutes, all order consume are 100% executed, without dropping request.

It can be seen; this application is designed to optimize with that the high latency task such as fetch order, are smartly designed to execute lesser time, and same goes for sector performance.

Conclusion

There are many careful design ways to optimize this stock trading system, use of Redis has good and bad, as bad in term of the latency it adds, good in term of the accessibility as it acts as a database. There are many critical analyst and experiments that proved the best design and performance of the system.

References

1. Barklam, H. (2023). Real-Time Trading: What it is and How it Works. Investment Guide. Retrieved from <https://www.investmentguide.co.uk/real-time-trading-what-it-is-and-how-it-works>
2. Alexander Gallego. Repanda is now free and Source Available. (2020, December 12). Retrieved from <https://www.redpanda.com/blog/open-source>
3. Tristan Stevens. (2023). Redpanda vs. Kafka with KRaft: Performance update. Retrieved from <https://www.redpanda.com/blog/kafka-kraft-vs-redpanda-performance-2023>
4. Carl Lerche. (Oct, 2019). Making the Tokio scheduler 10x faster. Retrieved from <https://tokio.rs/blog/2019-10-scheduler>
5. Contributors to Wikimedia projects. (2024, April 10). Millennium Exchange - Wikipedia. Retrieved from https://en.wikipedia.org/w/index.php?title=Millennium_Exchange&oldid=1218216585