Grand Challenge: A High-Performance Processing System for Monitoring Stock Market Data Stream

ACM DEBS 2022, Copenhagen Denmark

Kevin Li, Daniel Fernandez, David Klingler, Yuhan Gao, Jacob Rivera and Kia Teymourian

June, 2022

The University of Texas at Austin

Table of contents

Introduction

DEBS 2022 - Grand Challenge Overview

- Two queries on Stock Market Stream Data
- The first query is defined to compute the Exponential Moving Average (EMA) with two different smoothing factors of 38 and 100.
- The sequent query is to identify breakout pattern based on the first query.

Exponential Moving Average

Exponential Moving Average equation:

$$EMA_t = \begin{cases} Y_0 & t = 0 \\ \alpha Y_t + (1-\alpha) EMA_{t-1} & t > 0 \end{cases}$$

- The coefficient α represents the degree of weighting decrease, a constant smoothing factor between 0 and 1.
- $\alpha = \frac{2}{1+j}$ where j is a smoothing factor with $j \in \{38, 100\}$.

Example of Stock Price Fluctuations Over Time

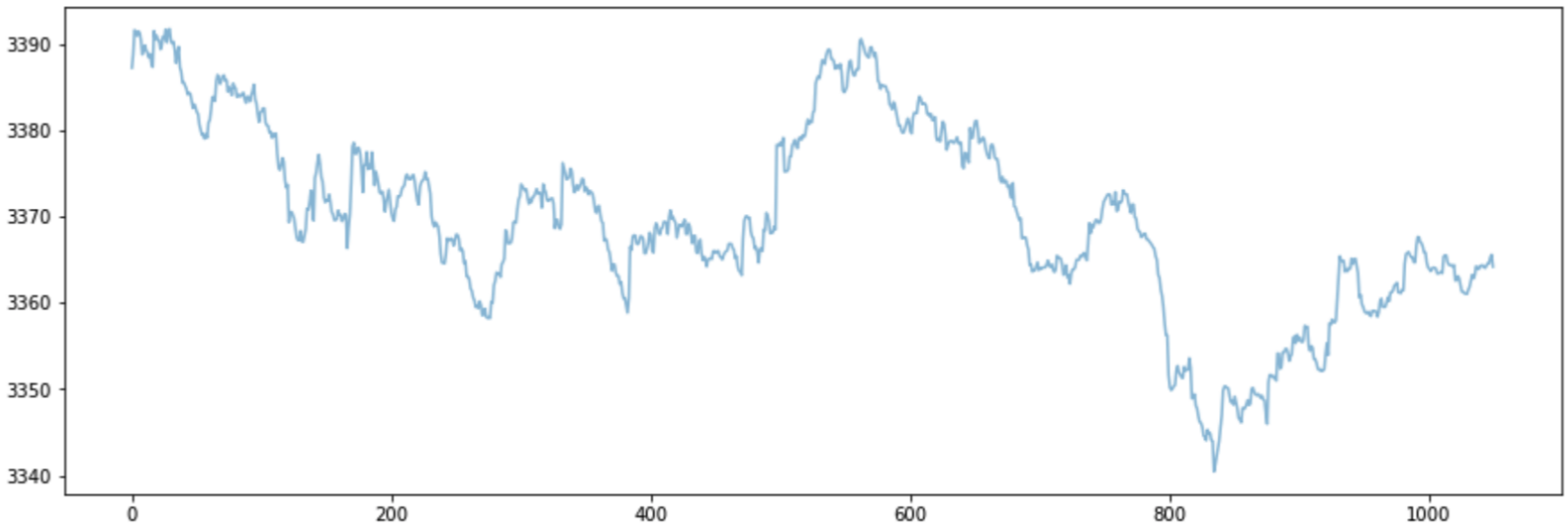


Figure 1: An Example of Stock Price Fluctuations Over Time (Time (s) vs Price)

Buy and Sell Advice Based on Breakout Patterns of EMA 38 and 100

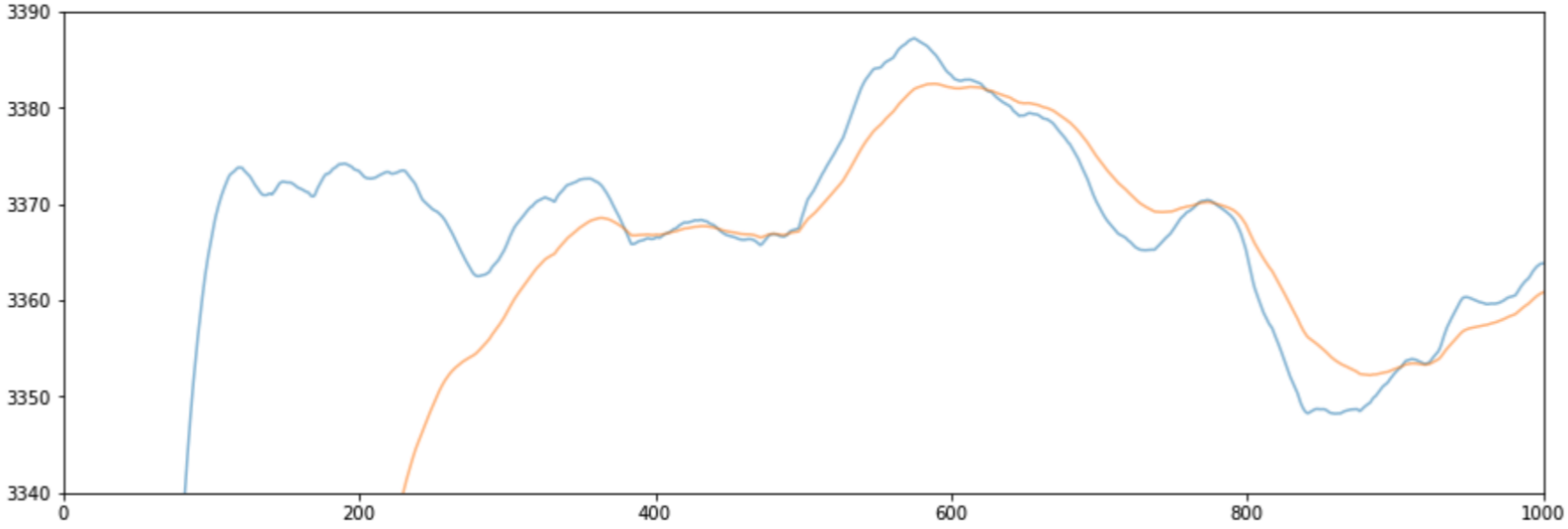


Figure 2: Example of Query 2 (Time (s) vs Price) - Buy and Sell advice based on Breakout Patterns of EMA 38 (orange) and 100 (blue)

Use one of the OSS or implement from scratch?

Should we use one of the Open Source Stream processing systems like:

- **Apache Spark - Streaming**
- **Apache Flink - Streaming**
- **Apache Storm**
- **Apache Kafka for streaming pipelines**
- **Esper Stream processing**
- **WindFlow**[1] in C++
- ...

We decided to implement the DEBS Grand Challenge from scratch.

Why?

[1] WindFlow data stream processing https://paragroup.github.io/WindFlow/

Why implementing from scratch?

- The Challenge queries are simple, therefore we can optimize the processing in our implementation much better.
- Our stream throughput is not very high. Maximum a Batch of 10k events, as defined by the Challenge.
- Systems like Spark or Flink include internal data processes; to name a few:
  - Communications between the master and worker processes
  - Internal caching and pipeline generations including batch processing, blocking data exchanges,
  - Checkpoint barriers, watermarks signaling or iteration barriers to provide guarantee of a FIFO order of and snapshots of the data stream
- Lots of configuration parameters which modify the performance/latency.

Our goal was correctness with highest possible throughput and low latency.

Architectural Design

Parallel Processing

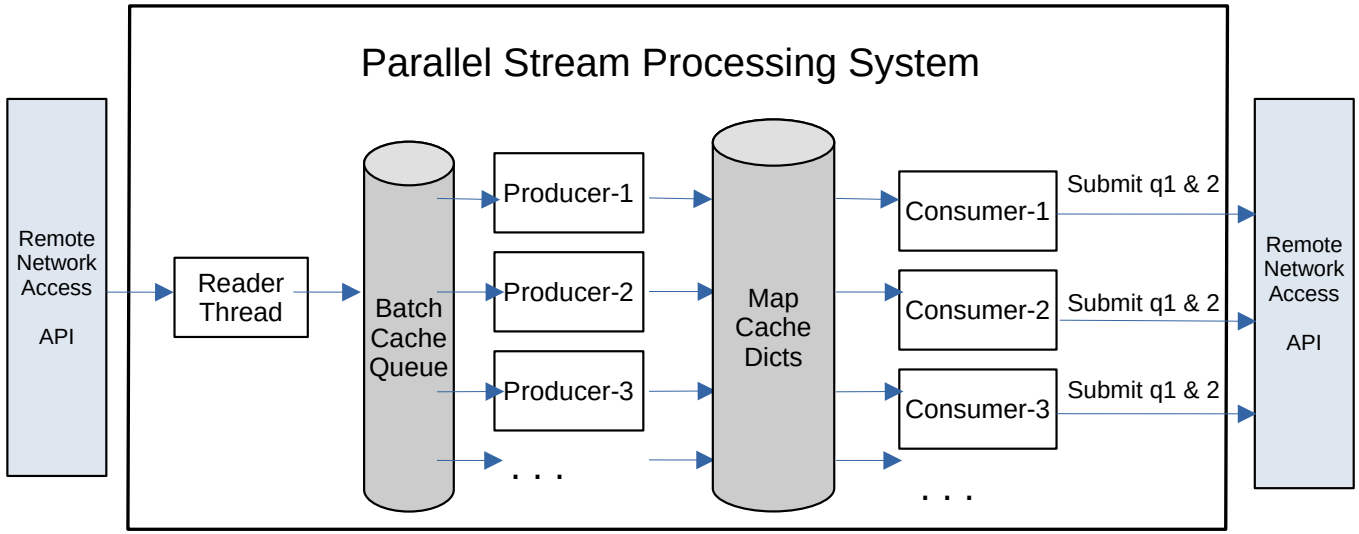


Figure 3: Parallel Processing of Events By a Set of Producers and Consumers without a Data Dispatcher.

- Reading Batches and Caching them
- The Reader reads from the DEBS Challenge API
- The Submitter had to be a single submitter

Event Dispatcher and Submitter Architecture

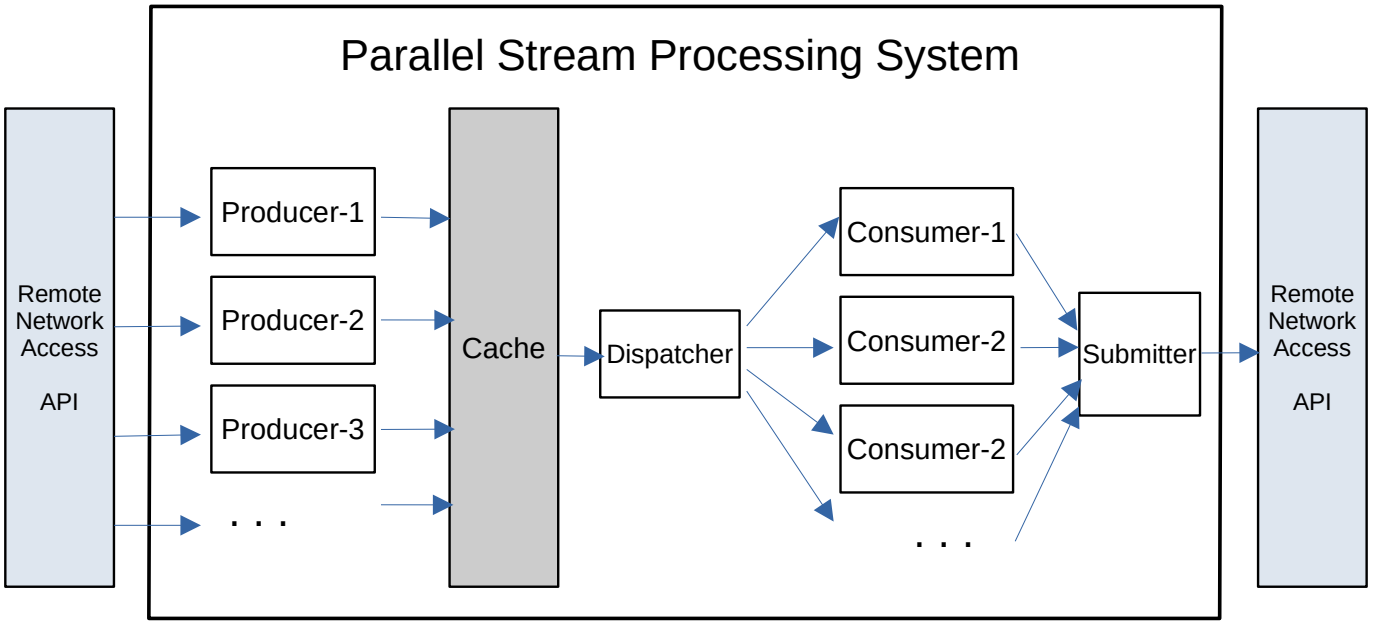


Figure 4: Event Dispatcher and Submitter Architecture

- Event Batch Dispatcher distributes the batches
- Producers read the Event from API, create internal data windows
- Event Windows are cached similar to Figure 3

Event Data Dispatcher

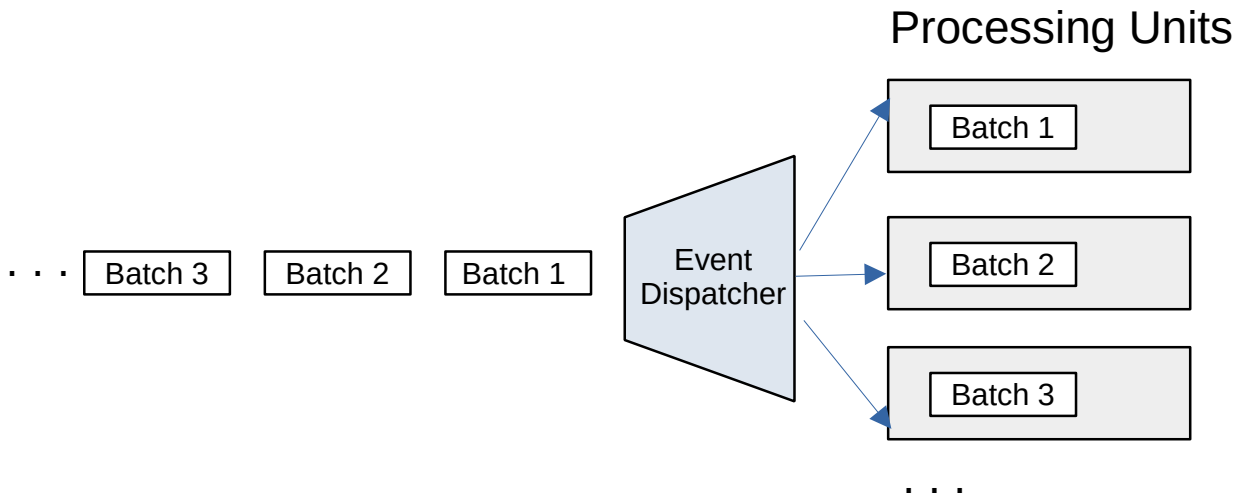


Figure 5: An Event Data Dispatcher

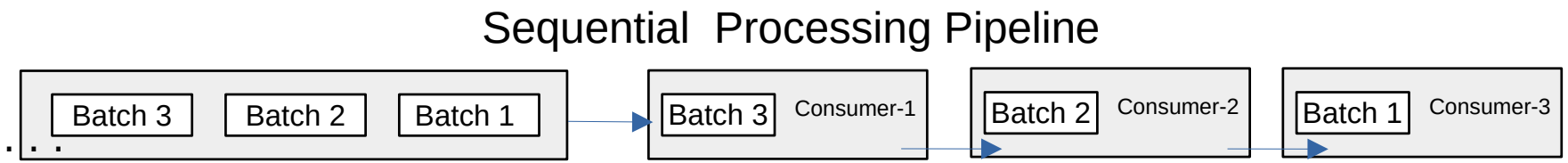


Figure 6: A Sequential Processing Pipeline. Each consumer reads a batch and passes to the next consumer.

Implementation

Two Implementations in Python and Java

- Source Code is available here https://github.com/kiat/debs2022/

Two Implementations in Python and Java:

- Our first implementation was a multi-threaded Python
- We have done a subsequent implementation in Java.

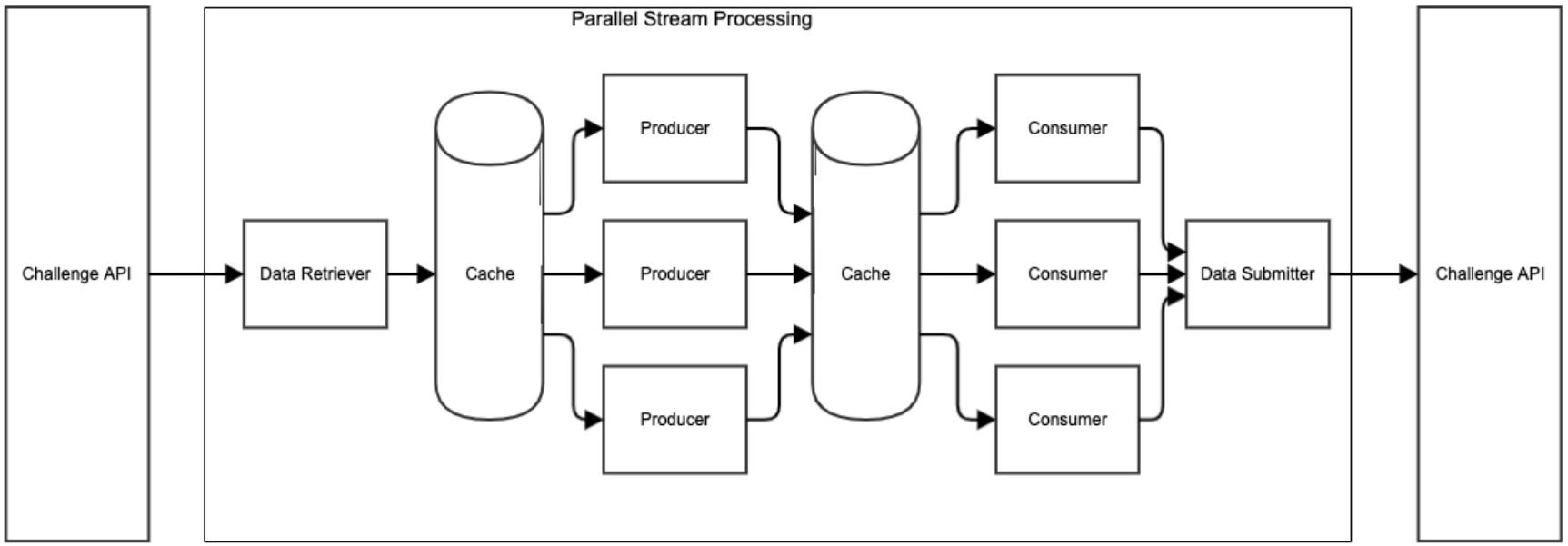
Python Implementation

Summary of the Python implementation:

- Our first, alpha prototype
- Assured correctness of Query Processing
- Allowed for us to test different ideas for processing and submitting batches
  - Multiple producer batch processing
  - Stock market symbol specific Consumers
  - Window pricing based off last recieved event
  - Only previous window values kept, allowing space efficiency
- Included multithreading, giving a baseline for experiments
- Provided context for further improvements, i.e. a Java implementation

Python Implementation

Python Implementation Design:



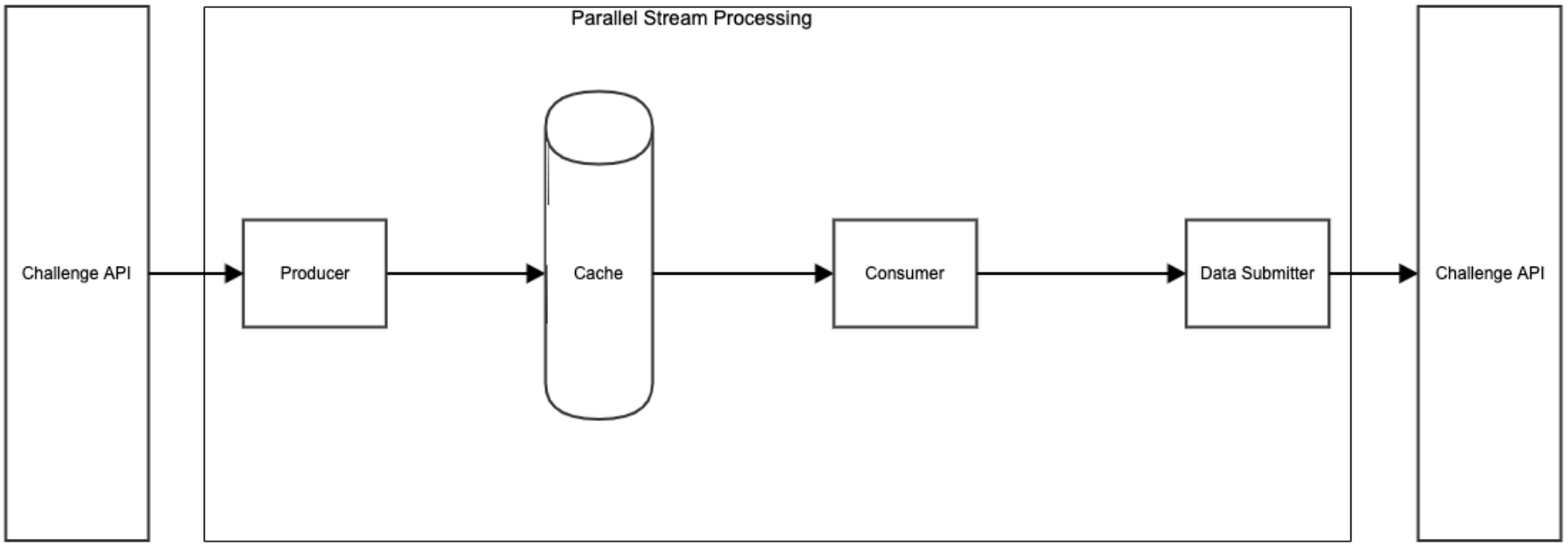
Our Java implementation

Our Java implementation includes two simple threads, one is the event producer (or event batch Reader Thread that communicates over the DEBS Challenge API), and the other is a submitter, that processes cached event windows data for queries 1 and 2.

- Two simple threads:
  - Event producer for main query computations
  - Result submitter
- We use Java lists of batches, and a Java HashMap to cache event windows for each stock symbols.
- Pre-allocate both of these caches in Java ArrayList and HashMap with a specific size.
- Threads are synchronized over a lock mechanism and thread notification.
- The results of both queries are submitted by the consumer thread to the DEBS Challenge API.
- With Batch Size 10k, two Threads was enough.

Java Implementation

Java Implementation Design:



Experiments

Python with Batch Size 1K

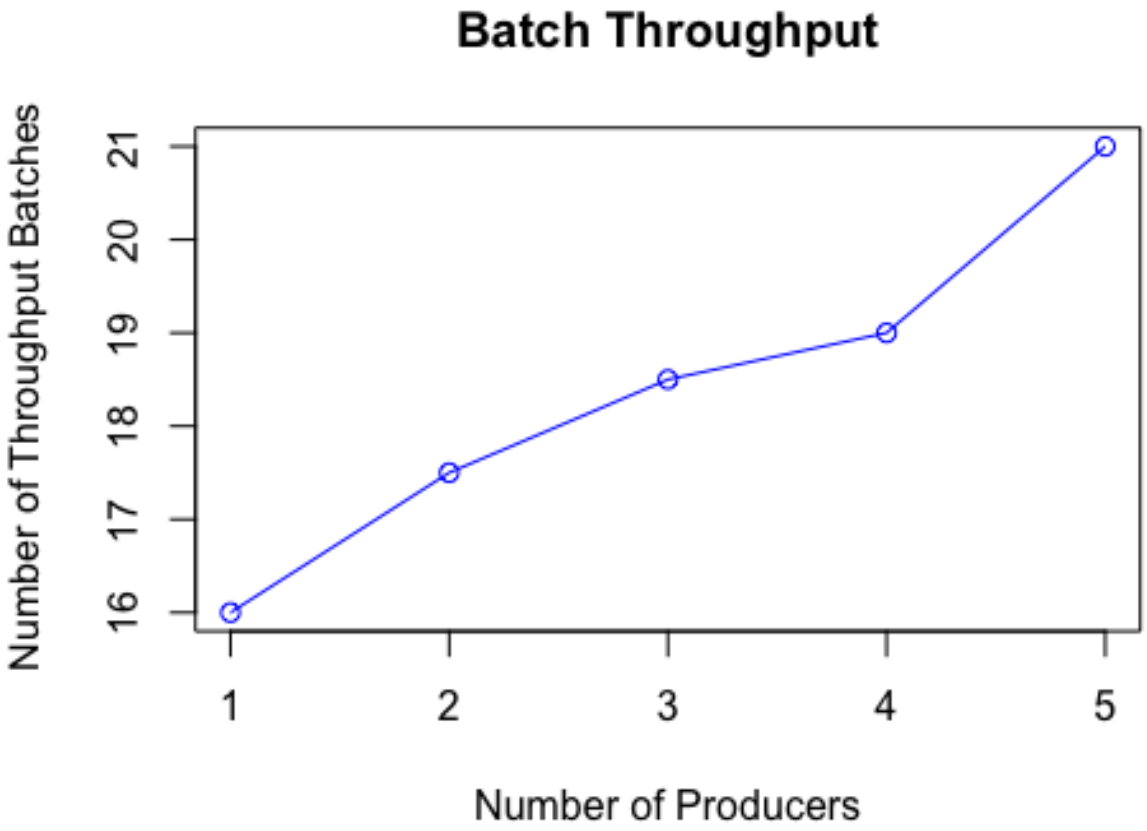


Figure 7: Query 2 throughput for different number of Producers. (Batch Size 1k - Multi-threaded Python implementation)

Java with Batch Size 10K

- Our Java implementation has higher performance because of some language features (Like static variable types and improved iterations over event batches).
- Latency of 19.99 seconds and a 88.21 batches per seconds

Leaderboard

Criteria: both q1 and q2 have results, last active run, benchmark type="Evaluation, benchmark run after 2022-03-17 19:40:04 UTC (switch to batchsize of 10k)"

Latency Ranking - (Sorted by lowest latency ((q1 + q2) /2)

Rank	Team	Latency	Throughput
1	group-17	19.996671	88.2194288519302
2	group-7	25.681919	356.40049439163784
3	group-11	113.37727899999999	45.32088811103257
4	group-12	129.597439	53.10725499209829
5	group-14	399.37638300000003	47.91140119582781
6	group-0	5335.154687	0.1877972121516102
7	group-16	15426.650110999999	0.8652173405621916

Figure 8: Our group, Group 17, Results

Conclusion

Conclusion and Future Work

- Our implementation and evaluation are limited due to the time constraints we had for this work.
- We observed how our from-scratch-implemented system performed compared to some large-scale cluster-based stream processing systems like Apache Storm or Flink (implemented by other groups).
- One potential improvement would be to implement it using a system programming language like C++ or Rust.
- This solution has scalability potential, from larger batch sizes, to more threads, to parallel computing.

Acknowledgements

We would like to thank the Association for Computing Machienary for hosting this conference and challenge.

We would also like to acknowledge our other group members:

- Kevin Li
- David Klingler
- Yuhan Gao
- Dr. Kia Teymourian