DEBS Grand Challenge: Continuous Analytics on Geospatial Data Streams with WSO2 Complex Event Processor

Sachini Jayasekara, Srinath Perera, Miyuru Dayarathna, Sriskandarajah Suhothayan
WSO2 Inc.
Mountain View, CA, USA
{sachinij, srinath, miyurud, suho}@wso2.com

ABSTRACT

DEBS Grand Challenge is a yearly, real-life data based event processing challenge posted by Distributed Event Based Systems conference. The 2015 challenge uses a taxi trip data set from New York city that includes 173 million events collected over the year 2013. This paper presents how we used WSO2 CEP, an open source, commercially available Complex Event Processing Engine, to solve the problem. With a 8-core commodity physical machine, our solution did about 350,000 events/second throughput with mean latency less than one millisecond for both queries. With a VirtualBox VM, our solution did about 50,000 events/second throughput with mean latency of 1 and 6 milliseconds for the first and second queries respectively. The paper will outline the solution, present results, and discuss how we optimized the solution for maximum performance.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed Applications; H.2.4 [**Systems**]: Concurrency

General Terms

Performance, Experimentation, Measurement, Design

Keywords

Complex Event Processing, Events, Disruptor pattern, Pipeline, Event Processing Languages, Geospatial data streams

1. INTRODUCTION

The DEBS Grand Challenge is an annual event where various different event-based systems compete to solve a real-world complex event processing problem scenario. The 2015 challenge is a problem related to conducting realtime streaming analytics on high volume geo-spatial data streams in the context of transportation systems [6]. The data set is based on a taxi trip report stream obtained from New York city in the year 2013. The challenge poses two problems: detection of the top ten most frequent routes in a sliding time window of 30 minutes, and identification of top ten most profitable regions of the city. The goal of the solution is to process the data using the two queries with maximum throughput and minimal latency.

One could build a solution from ground up or use an event processing technology. Among the candidate technologies are Complex Event Processing [2] technologies like WSO2 CEP, Esper, and StreamBase, Stream Processing technologies like Apache Storm and Spark Streaming [13], and Rule Engines like JBoss Drools.

Many of these models have been tried out in previous DEBS challenges. For example, Rabl *et al.* [9] and Jerzak *et al.* [5] used custom solutions, Geesen *et al.* [3] used stream processing, Aders *et al.* [1] used both rules and stream processing, and Koliousis *et al.* [7] introduced a new automata in a custom solution.

Based on our experience, Apache Storm and Rule engines are too slow because they only process events in the range of 10,000s of events per second. However, successful solutions in Grand Challenges have achieved more than 100,000s events per second. Complex Event Processing too can reach the required performance. For example, in the last year Grand Challenge, we reported 400,000 events/second using WSO2 CEP [8].

Our solution for 2015 challenge is built on top of an open source Complex Event Processor called WSO2 CEP. We use Siddhi Query Language (SiddhiQL), which is a query language associated with WSO2 CEP, to formulate our solution. For performance optimizations, we have introduced several extensions and also done some changes to the WSO2 CEP code. Our solution achieved more than 0.3 million events per second on 8-core commodity hardware with less than one millisecond latency.

The rest of this paper is structured as follows. Section 2 provides background details about WSO2 CEP. Then we outline the problem and the basic solution for Grand Challenge in Section 3. We describe the optimizations made to our solution in Section 4. The performance results obtained from our solution are presented in Section 5. Finally, Section 6 concludes the paper.

2. OVERVIEW OF WSO2 CEP

WSO2 Complex Event Processor (WSO2 CEP) is a light weight, easy-to-use, Complex Event Processor (CEP) which

DEBS'15 June 29 - July 3, 2015, OSLO, Norway.

DOI: http://dx.doi.org/10.1145/2675743.2772585.

is available as an open source software under Apache Software License v2.0. WSO2 CEP lets users provide queries using an SQL like query language where it will listen to incoming data streams and generate new events when the conditions given in those queries are met.

WSO2 CEP supports following types of queries.

1. *Filters* trigger when a given simple filter condition (e.g., total_amount <10) is true.

2. *Windows* collect and maintain events for a time window or a length window with sliding or batch modes and let users carry out aggregation functions on top of those windows.

3. *Joins* match up two data streams and create a new data stream based on a joining condition

4. *Patterns* accept a temporal query pattern written as a regular expression and trigger when a given condition has met. For example, the expression "Trip[total_amou nt <10] -> Trip[total_amount >50]" would match when two events occur such that the first have a total amount less than $10 and the second has a total amount more than $50. Sequences on the other hand are similar to Patterns. However, Sequences must exactly match the sequence of events without any other events in-between.

5. *Event Tables* map a database or in-memory table into a collection of events so that can be joined against data streams.

WSO2 CEP also supports data partitions where users can specify how to partition the events in a stream to multiple partitions. Execution of different partitions happen in parallel.

WSO2 CEP uses a SQL like Event Query language to describe queries. For example, the following query detects the number of taxis dropped-off in each cell in the last 15 minutes.

```
from Trip#window.time(15 min)
select count(medallion) as count
group by cellId
insert into OutputStream
```

Listing 1: Example SiddhiQL query.

Furthermore, WSO2 CEP provides many extension points where a user can augment its processing through their own custom filters, windows, and event processors. In this solution, we have used those features to do custom implementations for most performance sensitive parts of the code.

When WSO2 CEP receives a query, it builds a graph of processors to represent the query where each processor is an operator like filter, join, pattern, etc. Input events are injected to the graph, where they propagate through the graph and generate results at the leaf nodes. Processing can be done using a single thread or using multiple threads, where in the latter case we use LMAX Disruptor [11] to exchange events between threads. More details of the WSO2 CEP is available from [10].

3. BASIC SOLUTION

2015 Grand Challenge data set is a log of taxi activities collected from New York City over the year 2013. The data set has been released under the FOIL (The Freedom of Information Law) by Chris Whong which is accessible from [12]. The data set is 33.3GB in size and it comprises of 173 million taxi trip records. We found that the entire data set contains trip reports corresponding to 14,144 taxis. The following description outlines the solution using WSO2 CEP operators. The next section describes any non-trivial optimization to the basic solution.

These queries operate with external times provided with the streams. Hence we only know the time has elapsed when we have received an event with that time stamp. For example, we will know to expire the events in the 30 minutes window only when we have received an event having the corresponding event data.

3.1 Query 1: Frequent Routes

The goal of the first query is to continuously count and output changes to the most frequent routes over a 30 minutes sliding window. Since the query is continuous, the updates must be sent whenever the output has changed. Moreover, when there are multiple routes with the same count, we need to emit the most recent 10 routes.

Implementing the first query involves a 30 minutes sliding window and finding most frequent ten by counting the occurrences of each route. Figure 1 shows the high level block diagram of the first query.

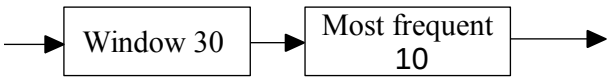

Figure 1: Query 1 block diagram.

This can be further expressed using the following WSO2 CEP query.

```
from countStream#window.time(30 min)
    #frequentK(startCellNo,
        endCellNo, 10, iij_timestamp)
select pickup_datetime_org, ..
insert into q1outputStream
```

Listing 2: CEP Query for Query 1.

In the query, the 30 minutes sliding window notifies the frequentK transform function when a new event has been added or removed. The transform function tracks the counts and outputs the most frequent K routes if that routes list has been updated due to this event.

3.2 Query 2: Profitable Areas

The goal of the second use case is to continuously find the cells that are most profitable for taxi drivers at the given moment. Profitability is defined as the median fare plus the tip for last 15 minutes divided by the number of taxi drivers who have dropped-off and have not taken a new trip in the last 30 minutes.

Implementing the second use case involves two parts: finding the taxis waiting in the given cell and finding the median

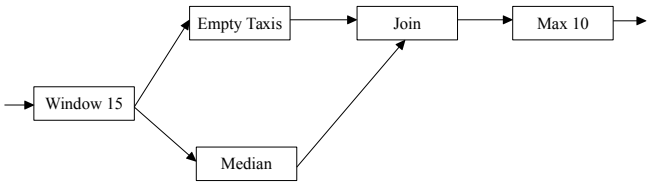

Figure 2: Query 2 block diagram.

profit of that cell. Then those two can be trivially combined to find the profitability and be used with a MaxK function to find the most profitable cells (See Figure 2).

For calculating the median, we need to calculate the median over a sliding window of 15 minutes. This can be done using a window with the median aggregate function as follows.

```
from cell_based_taxi_trips
        #window.time(15 min)
select debs:median(farePlusTip)
as profit, ... group by startCellNo
insert all events into profitStream ;
```

Listing 3: Query to find median profit.

To find the number of taxis waiting, we count the number of drop-offs and reduce the taxis that have either timed out or have done another drop-off. We can find when to reduce the count using a temporal event pattern that looks like following.

```
from d = TaxiTrip
 -> ( TaxiTrip[d.medallion == medallion]
            or TaxiTrip[ts - d.ts > 30 min])
```

Listing 4: Query to find taxis to remove.

As shown by the picture, we count each drop-off and reduce the taxis that have either timed out or have started a new trip to calculate the number of free taxis in a cell. Then we apply the last-value window to find the free taxi count and join it with median value stream to calculate the profitability of the cell. The following query depicts the join.

```
from FreeTaxiCountStream#window.lastValue()
    join MedianPaymentStream
insert into ..
```

Listing 5: Join waiting taxis and median.

Finally, we apply a MaxK function, that will track the highest 10 cells that are most profitable. Just like the frequentK function, the MaxK function only outputs a value when the most profitable cells have been changed.

4. OPTIMIZATIONS

In the following discussion, micro-benchmarks are done with a 20 million events subset of the full data set.

4.1 Optimizing frequentK

At a first glance, it looks as if frequentK could potentially have a serious pitfall. In the given use case, there are large number (8.1 billion) of potential routes and counting each route could be very costly in terms of memory. However, a closer look at the use case tells that it is not the case. Data set has only about 13,000 taxis, and a taxi cannot do more than two trips in 30 minutes. If that is the case, the maximum size of events in a window would be less than 26,000. Tracking 26,000 events would only take few kilobytes of memory. This means we do not need to resort to approximation based techniques like Sketching for calculating frequentK (e.g., Homem *et al.* [4])

However, we need to calculate frequentK for every event. Since it is expensive to search all counts for maximum values per every event, we need to keep track of heavy hitters via some index. A trivial implementation would build a reverse index of all counts by placing the count as the key and list of routes for that count as the value in a HashMap.

Following Figure 3 shows the data structures. When the count for a route has changed, we need to first update the CountMap, and then we need to remove the route from the ReverseMap's corresponding routes list and add the route to the new count's route list. Hence we need to do random add, remove operations to the routes lists. Also, since we need to send the most recent routes if there are many routes with the same count, the routes list should be sorted by time the route is seen. To implement the routes lists in ReverseMap, we have used a TreeMap that will keep items in the list sorted and support random add and removals in $O(log(n))$ time when n is the size of the map.

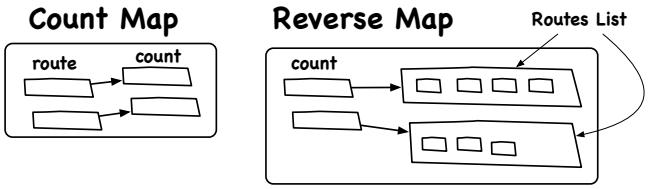

Figure 3: Frequent K data structures

However, since we only need the most frequent K (e.g., 10 in this scenario), we will never need more than 10 items in a route list for a given count. For an example, if the 10^{th} item in a route list has expired, any item older than 10^{th} would be already expired. So we will never need them. Hence we can keep only the most recent K items in a route list against each count and discard the others. This will reduce memory required by ReverseMap to the size of 10X(number of counts). Furthermore, this will improve the add and remove performance of route lists. For example, in one of our micro benchmarks on query 1, we were able to improve the throughput performance by 2.1 times through this optimization.

It is important to note that although implementations are much similar, we cannot use the same method with the MaxK function in the second query. Unlike in the query 2, the profitability values can expire out of turn due to taxis leaving the cell. In such cases, we might need 11^{th} item due to items before 11^{th} has expired.

We can go further by keeping routes only for some K maximum counts in the ReverseMap. However, this can lead to errors. For example, lets say we only tracked count greater than four. However, if all higher counts expired and the max-10 end up including the count of three, our answers would be wrong. Hence, we did not do this optimization.

4.2 Optimizing the Pattern for Counting Taxis

As described in the second query solution, we use a temporal event pattern to count the correction for the number of taxis waiting in a given cell. The pattern works by keeping a state machine for drop-offs by a specific cab as depicted in Figure 4. When the second drop-off happens, it terminates the current state machine and starts a new state machine.

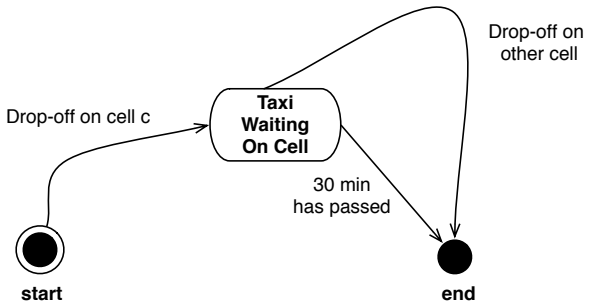

Figure 4: State machine for counting taxis

When a drop-off is detected, the pattern implementation searches for an existing state machine based on the drop-off cell, creates a new state machine if there is none, or progress the state machine if a one is found. Furthermore, it needs to progress all state machines based on the time stamp of the newly received time to detect the 30 minutes condition.

Initial implementation was slow because for each event, the WSO2 CEP pattern implementation searched all state machines to enforce the 30 minutes timeout. This was improved by keeping an index from the last drop-off event time for each event to the state machine. Furthermore, as we will discuss under string value optimizations in Section 4.5, we later converted the state machine lookup to an array index lookup.

4.3 Optimizing Median

As discussed before, the number of drop-off events in 15 minutes window would be less than 25,000 events most of the time. For the last year's (2014) DEBS Grand Challenge, we have explored a number of approaches to calculate the median.

1. Keep an unsorted list and sort on demand - sorting events for each event is slow.

2. Min-Max heap - maintains a heap of values so that we can find the middle value by shifting values from the heap to left or right based on the current inserted value. This has about $O(log(n))$ complexity given n values.

3. Bucket approach - divide the range to a set of buckets and count how many values fall into each bucket. To have cent level accuracy assuming maximum fare + tip value of $100 would only need 10,000 buckets which will take less than 1MB of memory. We can track fares that are higher than $100 just as a count and correct the median accordingly. The $100 limit we picked based on the understanding of the domain. We always keep track of the current median and update the median when a new value is added or removed, which is only a left or right shift from the current median. As a result, this operation can be done with $O(1)$ complexity.

4. Reservoir Sampling keeps a random sample of the data in the window. When we have filled all slots, either we replace expired items if such items are available or else replace a random item. Then we calculate the median using the sample. A primary advantage of this approach is that we only need to keep a smaller sample in memory.

We performed a micro benchmark to compare above approaches using 1 million values. In average Min-Max Heap took 1s, Reservoir Sampling took about 4ms, and bucketing took less than 1ms. We have chosen bucketing as it provides both accuracy and speed. Furthermore, in one of our micro-benchmark implementations on query 2, we observed 14 times average throughput performance improvement (from 4,947 events/second to 69,744 events/second in three consecutive runs) after switching from Min-Max Heap to the bucketing based median calculation.

4.4 Avoiding Joins

Joining multiple event streams is expensive. In the Query 2 processing, a significant cost was spent on joining median values and the empty Taxi count. Even with an index, joins cost about $O(log(n))$ times per each event where n is the number of cells in the system.

However, when we calculate the empty taxis count in the query 2, we can have the median data already calculated. Then, by copying the median into the input events that we sent into the empty taxis counts calculation, we were able to calculate the profitability of a cell as a simple expression instead of having to join two streams.

4.5 Improving String based Lookups

In the implementation, we will lookup data using medallion and cell IDs at different places. Those we can optimize as follows.

We can represent cells as an integer between 0 and 359,999 by using $600*y+x$ in the 600×600 grid. Furthermore, we can give each medallion an integer between $0-15,000$ by keeping a reverse lookup from the medallion to integer. Then, we can replace several HashMaps in our implementation with an array access via the index using these integers as listed below.

1. When we track Max10 of profits for cells, we can keep an array to keep track of the profit for each cell instead of a HashMap.

2. In Empty Taxi count pattern, we can replace cell to state machine mapping and medallion to that taxi's metadata maps with arrays using the above indexes.

This will convert the cost of those lookups from $O(log(n))$ to $O(1)$.

4.6 Parallelism: Fully using the Computer

So far we discussed processing using a single thread of execution. However, computers have multiple cores (e.g., DEBS VM has 4 cores) and to get the best numbers we have to fully utilize all the cores.

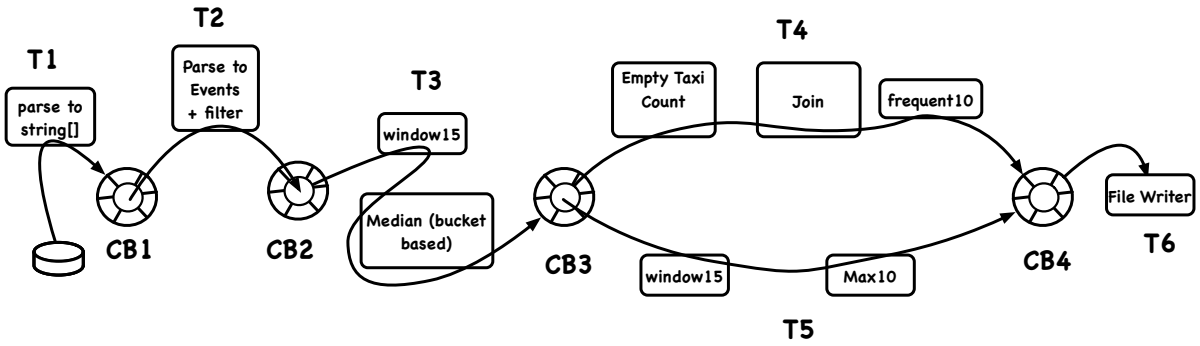

Figure 5: Full parallel solution

4.6.1 Data Partitioning

A common approach used for parallel execution is partitioning data. For example, we can partition the events using the hash of the route specified in each event. Then each partition will calculate frequentK with its data and then we can merge those results and calculate the final value. Since these calculations can finish at different times, the final merge for a given event needs to wait for all the partitions to finish processing that event.

We implemented this idea, but with more parallel executions, the solutions took longer than the single thread version. In further analysis, we made following observations.

In the solution, the frequentK operator is $O(log(n))$ operator with n size of the window. Due to the log complexity, gains we get at each partition (p) due to smaller data size (n/p) of the partition is very small. E.g., when n=25,000 (most likely estimate due to earlier discussion), gain with 10 partitions is about 25%, and that gain in most cases would be smaller than the cost of synchronizing the multiple threads of executions to ensure ordering.

Therefore, we cannot get more performance by partitioning data for this use case.

4.6.2 Execution Pipeline

We could also get parallel executions using a pipeline. A pipeline is a network of threads placed one after the other with a queue placed in-between each thread. Inputs are placed in the first queue, and the threads listen to the queue will process the inputs and place results in downstream queues. We can implement queues using a circular buffer as demonstrated by LMAX Disruptor.

We can use more threads by adding more circular buffers in-between. However, the effectiveness depends on hands off cost H. Consider a task which processes an event takes t time. We can break the task into two stages and get two threads by putting a circular buffer in-between. If the circular buffer adds H additional time (per event) to the process, processing an event will take $t + H$ time. Since two threads are processing tasks, processing N events will take $N.(\frac{t}{2}+H)$ time.

Obviously, this will increase the latency for each event, but it can improve the overall throughput as well. If the throughput is high, then the overall processing finishes faster. This means we face a trade-off between latency and throughput. We will discuss this in the results section.

One limitation of the pipeline approach is that it is only useful to speedup up to handful of cores (e.g., 10). This is because each thread needs to tie up with a stage, and in a machine with large number of cores, the problem will run out of independent stages to be placed in a pipeline. However, since VM given in the grand challenge only has 4 cores, this is not a problem.

We built the pipeline depicted in Figure 5 as our first parallel version. Here LMAX circular buffers are provided by WSO2 CEP. The pipeline runs five threads denoted by $T_1, ..., T5$ simultaneously.

As shown in the figure, we first read the data off the disk, parse them to a string array, and place them in circular buffer CB1. Then we parse the strings, remove the events that fall out of the area, and generate the filtered events using T2. The next step (T3) calculates the 15 minutes window, calculates the median, and places the results in CB3.

Thread 5 (T5) picks the events from CB3, calculates another 15 minutes window, and runs Max10 to finish the first query. The first query has 30 minutes window and we calculate it as two 15 minutes windows so that we can reuse the first 15 minutes window between both the queries.

Thread 4 (T4) picks the events from CB3 and calculates taxis waiting on each cell. Then T4 calculates the profitability using the median values calculated in T3 and runs Max10 to finish the second query.

We run the median that is need in the second query as the first step to avoid joins as that is needed for calculating the profitability in T4.

This solution will have 6 threads, but T1 and T6 will be mostly doing I/O. Therefore, the other four threads could utilize the 4 cores available in the VM.

When we did micro-benchmarks with this setup we got a throughput about 125,000 events/second in a 8-core physical machine. We further investigated how threads are being utilized, and Figure 6 shows the summary.

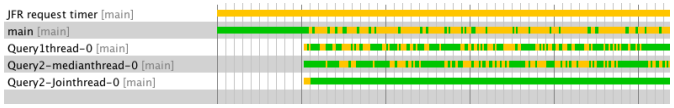

Figure 6: Thread utilization

As shown by Figure 6, T4 that does join and empty Taxi counting was almost fully utilized while others not fully utilized. Then to measure the effect of T4, we removed the

T4 processing and received about 250,000 events/second. This suggests that T4 processing has become the bottleneck. When T4 is slow, CB3 will put back pressure on T3, which will put back pressure through the pipeline and slow down all the processing.

Then we introduced a circular buffer inside T4, before frequent10, which will take off some of the processing from T4. The micro benchmarks improved by about 15% after this change. However, the latency for Query 2 with this solution was about 50ms, which was high. The next section describes an alternative design for further improvement.

4.6.3 Pipeline with a Single Circular Buffer

So far, we built a pipeline by placing circular buffers between each different stages. Then events will be read from circular buffers, processed, and written to the next circular buffer.

However, LMAX Disruptor lets us create a pipeline using a single circular buffer. The idea is that multiple threads would work on different regions of the same circular buffer. We place a sequence barrier that stops the second thread from overtaking or working in the same cell as the first thread. With this method, multiple threads can work in sequence without having to copy events between circular buffers. For an example, consider a circular buffer of 1024 cells. We can start three threads T1, T2, and T3, and setup sequence barriers to stop T2 from entering cell being processed by T1 and stop T3 from entering cells being processed by T2. Threads could also edit data and those changes can be seen by downstream threads.

Current WSO2 CEP implementation does not use this feature. We have changed the WSO2 CEP execution pipeline implementation to use this method to create a pipeline. The Figure 7 depicts the new setup.

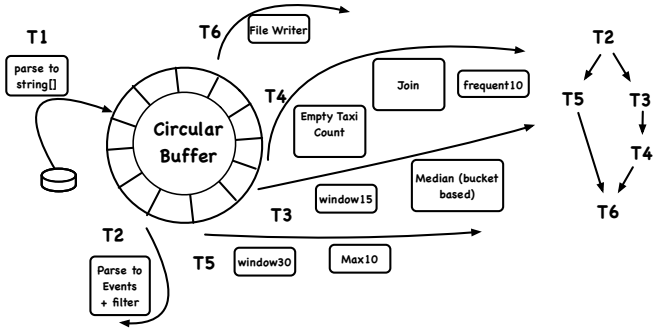

Figure 7: Single Disruptor solution

As the picture shows, multiple threads work on a single circular buffer and sequence barriers are setup in such a way that dependency flows depicted in the right hand side of the picture is established. We have used the same thread names we used in Figure 5.

This change improved the throughput about two times and improved latency very close to zero when running on top of physical hardware. The single circular buffer saves the cost of moving data between multiple circular buffers. Furthermore, when one thread accesses a memory location, it will be used by another thread shortly afterwords. Hence this solution is more friendly to caching as well.

However, when we tried the new version in a VirtualBox based virtual machine, performance was less than one 10^{th} of the physical machine and it was even smaller than the multiple circular buffer solution running on the same VM. We saw the same behavior with both MaxOS and Linux as host machines. We tried out the same code in an Amazon `3.xlarge` image (4-cores), and it did not exhibit the kind of drastic slowdown as with the VirtualBox VM.

Then we redid the single circular buffer version with four threads instead of six threads. With this change the slowdown happens on a VirtualBox VM was reduced by about half. Final results we present in the results section.

Single circular buffer based solution improved latency on both hardware as well as on a VM. However, throughput improvement was effective only on hardware.

4.7 Optimizations from WSO2 CEP

WSO2 CEP also has the following two optimizations available out of the box.

4.7.1 Object pooling

An event processing engine would deal with similar objects over and over again as each event in a stream will have a similar structure. Pooling them could remove both the overhead of garbage collection and the cost of creating them. WSO2 CEP has an event pool that recycles the event objects again and again. However, to minimize the contention, WSO2 CEP uses a different pool for each thread, and also the pool size has a maximum limit to avoid potential memory leaks.

4.7.2 Only keeping required attributes

An event processing engine needs to keep events. For example, windows, joins, and patterns will collect events and produce results later. Often, events have attributes that would never be used again in the downstream processing. WSO2 CEP do static analysis on the outputs generated by each query and only keeps attributes of events that will be used to produce the output from each query. For example, if the events have ten attributes and only two of those are used in the output, the WSO2 CEP window implementation will only keep required two attributes while keeping the events.

5. RESULTS

We tested our solution in a physical machine with 8-core 16GB memory running Ubuntu (version 14.04.1) Linux (kernel 3.13.0-37) and on a 4-core 8GB VirtualBox virtual machine running on the same host machine. We ran the solution against the full data set on Oracle JDK1.7. The following table summarizes the results. Each reading is taken as an average of three runs.

The total execution time is inclusive of the I/O time to read the data and throughout is calculated as the total number of messages divided by the total execution time. However, the delay does not include the I/O time. For the delay, a counter was started once each input data line was read from the file and stopped just before the formatted final output is generated.

We calculated the delay by taking a timestamp once the data line has been read, sending that timestamp with each event we send to the system, and propagating that value through each derived event. The delay was measured using the timestamp just before the results are written to the disk.

Our solution achieved more than 0.3 million events per second on commodity hardware with less than one millisec-

Table 1: Full Dataset Results

Scenario	Total Time (s)	Throughput (events/second)	Q1 Delay (ms)	Q2 Delay (ms)
8-core physical	492	352,002	0	0
4-core 8G VM	3,629	47,722	0	6

ond latency and about 50,000 events per second with 6ms latency on a VirtualBox VM. The VirtualBox solution was about one seventh times slower. However, as we discussed below, on a more optimized virtual environment like Amazon web services, this slowdown was about half of the physical machine's performance.

In addition, we carried out a series of micro-benchmarks to understand the behavior of the solution using a data set that includes the first 20 million events from the full data set. Those results are explained below.

To understand the effect of concurrency, we implemented a sequential version of the algorithm that uses a single thread for all executions. The following chart compares the speedup vs. the number of execution threads for each execution environment. We calculated the speedup as the sequential execution time for each host environment divided by the full execution time.

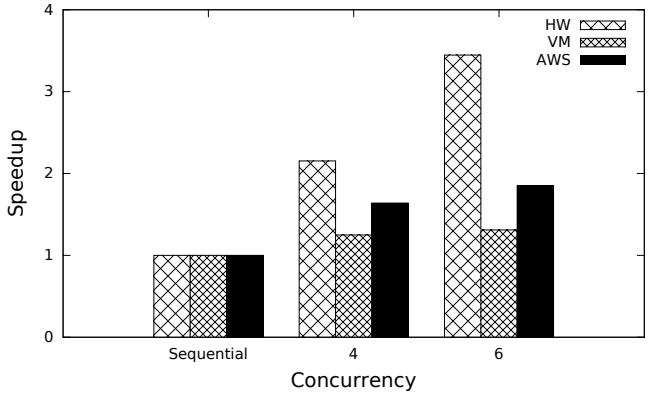

Figure 8: Speedup vs Concurrency

On physical hardware with 8-cores, throughput increased with increasing number of threads. Adding more threads introduced only one and three milliseconds of additional latency for the first and second queries respectively.

With Amazon `m3.xlarge` machine (4-core, 15G RAM) four threads provided speedup of two, but adding further threads did not improve the speedup. Since the machine has only four cores, it is expected that we would not have more speedup with threads more than four.

With VirtualBox VM, speedup was about 0.2 only, and this agrees with the observations we had within the first part of this section.

Figure 9 shows the trade-off between latency and throughput. Each point is represented by the hosting environment (physical machine (HW), VirtualBox VM (VM), Amazon (AWS)), number of threads, and the size of the circular buffer. As expected, the single threaded version provides the best latency. Overall this shows the throughput latency trade off we discussed in Section 4.6.2. Furthermore, higher buffer sizes tend to provide higher latency, and for physical machines, 1024 seems to be the optimal configuration.

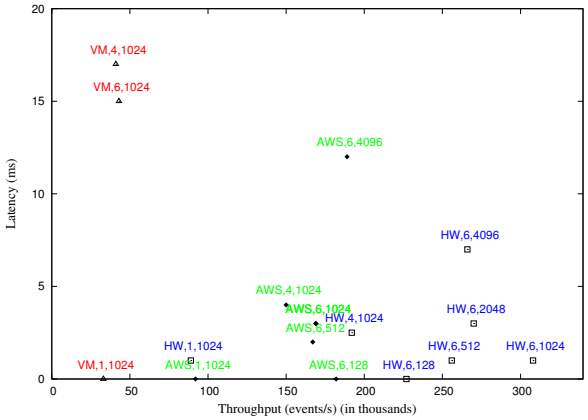

Figure 9: Latency, throughput trade off

The proposed algorithm can run with 512M of memory, hence it is memory efficient. Although we ran final tests with 8G of heap size, even with 512M memory performance impact was less than 10%. Since the algorithm only needs to track data for 30 minutes time window, this is possible. However, with 256M, GC overhead was high and the run failed with "GC overhead limit exceeded" error.

We tested the solution with several garbage collection algorithms. However, the default Parallel GC algorithm turns out to be the best, and algorithms G1 and CMS (Concurrent Mark and Sweep) both reduced the throughput by about 20% and the latency also got worse. When comparing algorithms Parallel GC has least overhead while CMS and G1 can provide smaller GC pauses. However, since our solution works with less than 2G memory, GC pauses are very small. This is the most likely explanation for this behavior.

6. CONCLUSIONS

This paper presents how we used WSO2 CEP, an open source, commercially available Complex Event Processing Engine, to solve the DEBS 2015 Grand Challenge problem. We were able to express both queries as simple SQL like queries in WSO2 CEP query language.

However, for maximum performance, several optimizations are needed. Some of these optimizations are done by improving underline operator implementations and others are done through custom extensions we added to WSO2 CEP through extension points. Finally, to do all processing with a single circular buffer, we did significant changes to WSO2 CEP processing pipeline.

Following are some of the highlights of our solutions to 2015 DEBS grand challenge.

1. frequentK - In the frequentK implementation, we only kept most recent K elements in the route list as others are never used. This also improves the execution time

from $O(log(n))$ to a constant value. Furthermore, this would reduce the memory used and also processing time spent to update the route lists while removing items.

2. Median - For median calculation, we used a bucket implementation that provides full accuracy and performance. This is possible because number of buckets in the payments for a trip is limited. This also improves the execution time from $O(log(n))$ to a constant value.

3. Indexing the Pattern Query - we build an index to locate the state machine in temporal event pattern calculation. This improves the running time from $O(n)$ to $O(log(n))$ where n is the number of state machines in the system. Later by using a monotonically increasing integer to represent the taxi's medallion, we reduce the lookup to an array lookup that has the complexity of $O(1)$.

4. Pipeline - we processed data in parallel by building a pipeline that processes data in different stages where each stage will have its own thread and placing a LMAX Disruptor based circular buffer to exchange events between threads. Then we further improved latency of the processing by using multiple threads working on a single circular buffer with sequence barriers to ensure the order of processing. This further improved latency significantly.

5. Avoiding Joins - Joining multiple event streams is expensive. We avoid cost of joining data by copying the median values to input events sent to free taxi counts calculation. This improves Query 2's profitability calculation, which was a main bottleneck in the system.

Looking back at the solution, we used two types of optimizations. The first type uses various tricks to minimize the amount of processing needed and the second type tries to use the computer fully by running multiple threads of execution.

Sometimes the two types would get each other's way. For example, the first query needs a 15 minutes window and the second needs a 30 minutes window. Although we could save CPU processing by calculating the two windows together with a list events sorted by received time and two cursors, the resulting dependencies between processing slowed down the system more than its gains. Furthermore, the well known method of parallel execution by partitioning data was not feasible due to cost of reordering results. Instead, we used the less common method of building a pipeline with different threads for each stage.

Any contention between threads were expensive and we had to find alternative method for implementing those use cases. For example, instead of joining the median and empty taxis waiting for a cell in the second query, we first calculate the median and then copy it to the input events sent to the empty taxi per cell calculation. We achieved concurrency through running different stages in a pipeline.

Our final solution used about 80% of the CPU, and more concurrency would have introduced significantly high latencies as utilization and latency are often a trade-off. On the final configuration, our solution achieved more than 0.3 million events per second with less than one millisecond latency on a 8-core commodity hardware.

7. REFERENCES

[1] L. Aders, R. Buffat, Z. Chothia, M. Wetter, C. Balkesen, P. M. Fischer, and N. Tatbul. *DEBS'11 Grand Challenge: Streams, Rules, Or a Custom Solution?* ETH, Department of Computer Science, 2011.

[2] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012.

[3] D. Geesen and M. Grawunder. Odysseus as platform to solve grand challenges: Debs grand challenge. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, pages 359–364, New York, NY, USA, 2012. ACM.

[4] N. Homem and J. P. Carvalho. Finding top-k elements in a time-sliding window. *Evolving Systems*, 2(1):51–70, 2011.

[5] Z. Jerzak, T. Heinze, M. Fehr, D. Gröber, R. Hartung, and N. Stojanovic. The debs 2012 grand challenge. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, pages 393–398, New York, NY, USA, 2012. ACM.

[6] Z. Jerzak and H. Ziekow. The DEBS 2015 Grand Challenge. In *DEBS 2015: the 9th ACM International Conference on Distributed Event-Based Systems*, June 2015.

[7] A. Koliousis and J. Sventek. Debs grand challenge: Glasgow automata illustrated. 2012.

[8] S. Perera, S. Sriskandarajah, M. Vivekanandalingam, P. Fremantle, and S. Weerawarana. Solving the grand challenge using an opensource cep engine. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 288–293, New York, NY, USA, 2014. ACM.

[9] T. Rabl, K. Zhang, M. Sadoghi, N. K. Pandey, A. Nigam, C. Wang, and H.-A. Jacobsen. Solving manufacturing equipment monitoring through efficient complex event processing: Debs grand challenge. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 335–340, New York, NY, USA, 2012. ACM.

[10] S. Suhothayan, K. Gajasinghe, I. Loku Narangoda, S. Chaturanga, S. Perera, and V. Nanayakkara. Siddhi: A second look at complex event processing architectures. In *Proceedings of the 2011 ACM Workshop on Gateway Computing Environments*, GCE '11, pages 43–50, New York, NY, USA, 2011. ACM.

[11] M. Thompson, D. Farley, M. Barker, P. Gee, and A. Stewart. High performance alternative to bounded queues for exchanging data between concurrent threads. *technical paper, LMAX Exchange*, 2011.

[12] C. Whong. FOILing NYC's Taxi Trip Data. URL: `http://chriswhong.com/open-data/foil_nyc_taxi/`, 2015.

[13] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.