Grand Challenge: High Performance Stream Queries in Scala

Dantong Song, Kaiwen Zhang, Tilmann Rabl, Prashanth Menon, Hans-Arno Jacobsen
Middleware Systems Research Group (MSRG), University of Toronto

ABSTRACT

Traffic monitoring is an important stream processing application, which is highly dynamic and requires aggregation of spatially collocated data. Inspired by this, the DEBS 2015 Grand Challenge uses publicly available taxi transportation information to compute online the most frequent routes and most profitable areas. We describe our solution to the DEBS 2015 Grand Challenge, which can process events at a 10 ms latency and at a throughput of 114,000 events per second.

1. INTRODUCTION

The Grand Challenge 2015 revolves around monitoring a feed of taxi transactions. Each taxi event records the time, start and end points, as well as the fare price. Location-based applications, in particular traffic monitoring, are known to be highly dynamic due to the mobile nature of the users and require efficient aggregation of spatially collocated events [1]. The grand challenge provides added semantics to the typical traffic application due to the structured nature of taxi fleets, as well as the pricing dimension of taxi trips. These characteristics drive the queries of challenge which represent analytics for the taxi industry.

We have created a highly optimized event processing engine (written in Scala) to answer the challenge queries. Our primary contributions are the optimization techniques we employed by leveraging the nature and semantics of the addressed queries as well as the nature of the provided workload. Our solution is highly efficient, being able to process the full workload in 1500 seconds with 10 ms latency.

2. ARCHITECTURE

The high-level architecture of our challenge solution is comprised of four major components. These components were incorporated in our final solution after several design modifications to increase throughput by maximizing CPU usage.

The four major components are the Driver, Parser, Query 1 event system and Query 2 event system. The components are connected using the actor model to improve overall concurrency. Furthermore, each of the components has its own independent design strategy, parallel processing strategy, storage strategy designed to optimize the solution for best throughput. An overview of the architecture can be seen in Figure 1. We will describe each subsystem in more detail below.

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

ACM 978-1-4503-3286-6/15/06 ...$15.00.
http://dx.doi.org/10.1145/2675743.2776761.

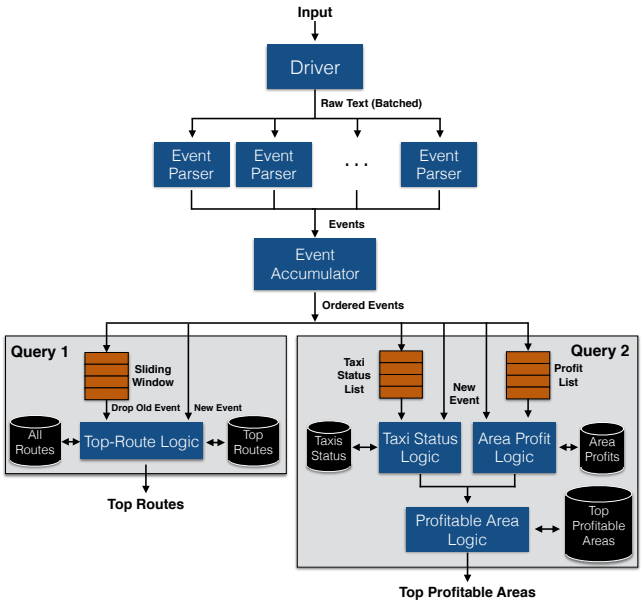

Figure 1: Event system architecture

Note that for the Query 1 and Query 2 event processing components, we have implemented two different versions for each. One version employs Apache Spark to achieve maximum scalability while the other is a custom engine designed to maximize the performance objectives required by the challenge. The remainder of this paper will only describe the custom implementation as it is the one submitted for consideration in the challenge competition.

2.1 Driver

The driver component is responsible for data collection and dispatching. The component continuously reads lines of raw data from the provided input file and dispatch batch of inputs to corresponding parsing component threads. The driver component is single-threaded due to the nature of file I/O. The thread is bottlenecked mainly by the I/O performance of the storage device.

2.2 Parser

In our solution, events are required to be parsed before being sent to the event systems. Through profiling, we identified parsing to be the computational bottleneck of the entire system. Therefore, we decided to use a multi-threaded approach that can efficiently

use all of the remaining CPU power from other components. A new thread will be pulled from a thread pool for each input batch dispatched by the driver component. All threads will parse their respective batches in parallel and send parsed events to the query systems. A lock is used to eliminate any race conditions.

2.3 Query 1 Event System

The DEBS 2015 Grand Challenge requires the first query to output the most favorite routes in the past 30 minute time frame. An overview of Query 1's architecture is displayed in Figure 1. The query is carried out in multiple stages. However, due to the sequential nature of the query requirement, a single thread is used to process all events. When an event enters the system, a distributor process that maintains a time window of previous events first handles it. New incoming events enter together with older events that dropped out from the 30-minute time window are sequentially processed by functions accessing our customized storage model. In order to optimize performance, a separate storage model is used to only keep the top routes. Changes in the top routes are stored and sent to file output.

2.4 Query 2 Event System

The second query of the challenge concerns monitoring the change in the most profitable areas. To optimize concurrency, the Query 2 event system is separately implemented and independent from the first query. The architecture of the second query is similar to Query 1 with minor differences. The events follow the same stages with the distributor, storage handling functions, query requirement check, and output. The difference is that Query 2 performs profitability calculation: the system tracks profit and empty taxis in separate time-based event distributors and storage models.

Optimizations – Each storage component has a different performance characteristic, and so we were very careful in our design to use the appropriate data structures where necessary. Our solution expended extra memory to boost the performance in our storage model design. Using large arrays is not always the most efficient storage option, but it can considerably reduce access time to each item in the storage. Additionally, arrays can be very cache friendly, more so than their linked-list of hash-table counterparts. We increase the size of the main array to occupy a large amount of memory to a point where increasing memory usage no longer results in improved performance.

3. EVALUATION

We experimentally evaluated the performance of the solution in terms of throughput and latency (i.e., the time it takes from reading the event to when corresponding output is sent to system I/O). The experiments were conducted using multiple machines hitting different bottlenecks. The best performance is achieved with our workstation with i7 4770K CPU, 8GB of memory and solid-state disk drive.

With our multi-threaded solution, we measured a minimum time of 1523 seconds to finish processing the entire 173 million events with an average throughput at 8.8 microsecond/event and an average latency of 14 ms and 13 ms in Query 1 and Query2, respectively. The queries require events to be processed again when leaving the 30 and 15 minutes time frame. Therefore, the system actually handled input lines with 2x event count in Query 1 and 4x event count in Query 2, effectively having an average throughput of 4.4 microsecond/event and 2.2 microsecond/event in Query 1 and Query 2.

By adopting the actor model we effectively increased the throughput, but suffered a non-negligible latency setback. With our single-threaded experiment, we were only able to reduce the latency of Query 1 and 2 to 3 ms and 14 ms, respectively, which is still more than 1000x slower than our optimized solution.

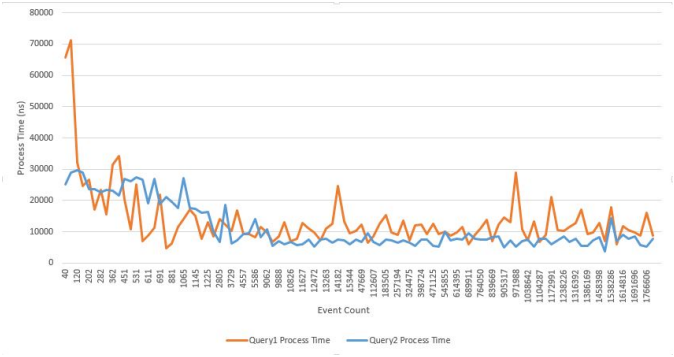

Figure 2: Query Processing Time

3.1 Performance Bottlenecks

In workstations that do not use a solid-state disk drive, file I/O could be the bottleneck. Since the file is read in lines instead of memory blocks, traditional hard disks will be the bottleneck in this case as our event system processes events at a rate of < 10 microsecond/event.

In workstations that have fewer than four CPU cores or do not adopt hyper-threading technology, event parsing will be the bottleneck of the entire system. With a traditional quad-core system that only execute four threads; the three major components will each occupy a single core. The most calculation-intensive parsing threads will only be able to obtain one core, slowing down the entire system. With hyper-threading, which allows two running threads on each core, the problem can be effectively eliminated. The following table compares the time spent in each query relative to the overall time.

	Total	**Query 1**	**Query 2**
Hyper-threading	1523 s	1414 s	1320 s
No hyper-threading	3456 s	1596 s	1480 s

3.2 Storage system training

Event throughput is largely dependent on the performance of the storage structure. Our storage system requires proper training to reach its maximum throughput. Graph 2 shows the event processing time for Query 1 and Query 2 over an 2 million event range. After approximately 2000 events, the processing time of both query stabilizes at 10 microsecond/event for meaningful events over the output.

4. CONCLUSIONS

As a solution to the DEBS 2015 Grand Challenge we designed a highly specialized system that is able to process events in microseconds and can achieve a throughput of more than 100,000 events per second on standard hardware. To achieve this performance, we individually profiled every part of the processing system and optimized each of them using parallelization, cache-conscious data organization, and reduction of overhead, such as value formatting.

5. REFERENCES

[1] A. Hinze, K. Sachs, and A. Buchmann. Event-based applications and enabling technologies. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 1:1–1:15, 2009.