DEBS Grand Challenge: Real time Load Prediction and Outliers Detection using STORM *

Abhinav Sunderrajan
TUM CREATE Limited
1 CREATE Way
#10-02 CREATE Tower
Singapore 138602
abhinav.sunderrajan@tum-create.edu.sg

Heiko Aydt
TUM CREATE Limited
1 CREATE Way
#10-02 CREATE Tower
Singapore 138602
heiko.aydt@tum-create.edu.sg

Alois Knoll
TUM CREATE Limited
1 CREATE Way
#10-02 CREATE Tower
Singapore 138602
knoll@in.tum.de

ABSTRACT

In this work we present our solution towards the DEBS 2014 Grand challenge. We also discuss the set of novel and generic techniques used to enhance the performance of our STORM [4] based stream-processing platform while implementing the challenge queries.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: General

Keywords

Stream processing engine, Complex event processing, Stream archive queries

1. INTRODUCTION

The DEBS Grand Challenge 2014 provided recordings from smart plugs measuring power consumption related values which are deployed across a number of households [9]. The goal of the challenge is to develop a stream-processing system for computing short term load-predictions and detecting outliers over this high velocity data stream. We made use of an open-source distributed stream processing system called STORM for horizontally scaling our solution. We employ generic techniques such as archive data streams, hybrid queries and a novel architecture which combines a high-level CEP system with STORM for better performance.

The organization of the paper is as follows. In Section 2 we discuss the two queries comprising the challenge. In Section 3 we discuss the architecture and elaborate upon the optimizations incorporated to enhance the performance of our stream-processing system. Section 5 evaluates the system while executing the challenge queries. The paper concludes in Section 6.

*This work was financially supported by the Singapore National Research Foundation under its Campus for Research Excellence And Technological Enterprise (CREATE) programme.

DEBS'14, May 26-29, 2014, MUMBAI, India.

http://dx.doi.org/10.1145/2611286.2611327.

2. QUERIES

2.1 Query 1

Query 1 involves predicting short term load over time intervals ranging from 1 minute to 60 minutes at the resolutions of individual houses and plugs. The prediction model though conceptually simple is more of a query processing problem. The predicted load is the average of current load (averaged over a time interval) and the median load for the past 'N' days at the time slice corresponding to the prediction interval. We employ *proactive archive streams* discussed in Section 3.2.2 for efficient retrieval of specific data from a massive archive consisting of historic load values.

2.2 Query 2

Query 2 involves estimating the outliers in terms of load consumption by computing the percentage of plugs in each house having a median load greater than the global median load for all plugs over a specified time interval. The primary issue involved in implementing a solution for this query is the process of computing the global median load over the past 1 hour and the past 24 hours by using a sliding window with a slide equal to 1 second. The naive approach of storing the load values for each reading over a high volume data stream over long periods of time is infeasible. The operation of computing global median for all plugs is further complicated by the fact that the operation cannot parallelized making it non-scalable.

3. ARCHITECTURE & OPTIMIZATIONS

In this section we describe the architecture of our data-stream processing platform which is implemented in Java. We also discuss the optimizations incorporated to increase the performance and scalability of our system.

3.1 Core components

3.1.1 STORM

The data stream processing platform has been built using *STORM* as the core framework. STORM is an open-source distributed real time computation engine which enables a

developer to neatly package the stream-processing logic to processing units called *bolts*. Each bolt can be parallelized by configuring the number of threads it needs to execute. STORM also defines an abstraction called the *spout* which is the source of input data-streams.

A STORM *topology* represents a network of spouts and bolts with each bolt subscribing to the output of a spout or a preceding bolt. The entire topology can be distributed across a cluster of slave nodes making it scalable. Further, STORM uses a lock-free ring buffer called the *LMAX disruptor* [5] for efficient data transfer between the bolts. We have leveraged the power of LMAX disruptor in intra bolt threads for efficiency and to mitigate the effects of *stop-the-world generational garbage collection* [2].

3.1.2 Redis

The need for a centralized in-memory data-structure was necessitated due to the distributed nature of the stream-processing system where the processing bolts could reside in any node in the cluster. *Redis* [3] can be described an in-memory key-value database. Redis enables the storage of data-structures such as lists, Maps and Sets in memory allowing extremely fast retrieval of a value from these data-structures using an associated key.

3.2 Optimizations

In this subsection we discuss the optimizations made for enabling our stream-processing system to address the queries posed in the challenge. The techniques discussed are largely generic and can be adopted to enhance the performance and scalability for purposes other than this challenge.

3.2.1 Esper-enriched bolts

Despite STORM enabling the overlying stream processing platform to be truly scalable, it does provide semantics to define sliding windows. We have incorporated Esper [1] in our system to create *Esper-enriched* bolts where sliding window and group operations were necessary. Esper is a high-level event-stream processing engine with Java and .NET bindings.

Esper also provides a SQL-like language called *Esper processing language* (EPL) to perform operations such as select, project, filter and join on event streams. EPL defines a variety of sliding and tumbling windows for computing aggregates. Each processing thread of an Esper-enriched bolt is associated with an unique *Esper-engine instance* to process the tuples using an Esper query.

3.2.2 Proactive archive data-streaming

Retrieving historic data from massive archives while processing high-velocity live streams is a challenge. Considering that queries to a massive archives are costly, intelligent techniques should be employed to avoid bottlenecks. For the grand challenge, we ensure that the archive data in appropriately sized chunks (which for Query 1 described in Section 2.1 would equal the defined time slice) are fetched before the arrival of the live event (i.e. pro-actively) to minimize latency to a bare minimum. Considering an example where the archive load aggregates for the past 3 days are required, three concurrent database connections from Monday through Wednesday are used to stream per plug load aggregates between 00:45:00 hours and 01:00:00 hours. It is assumed that the current time is 00:15:00 hours on a Thursday. The per plug load aggregates for the aforementioned time interval are computed and sent to the platform through as many *archive stream spouts* as the concurrent database connections.

The load aggregates per plug (and per house) per time-slice interval are stored in a Redis data-structure. For minimal database-query latency the stream-archive table is indexed on time stamp. With knowledge on the approximate live-stream velocity and time-slice over which prediction is to be computed, the interval at which archive load aggregates are retrieved can be set to ensure that the values are stored in memory before the arrival of live-stream tuples. We refer to this technique as *proactive archive streaming* since only the relevant archive records are streamed and stored in memory for operations such as joining and comparison with the live-stream tuples prior to their arrival. The technique is based on the *porthole scan* approach elaborated in [8].

4. QUERY TOPOLOGIES

4.1 Topology for Query 1

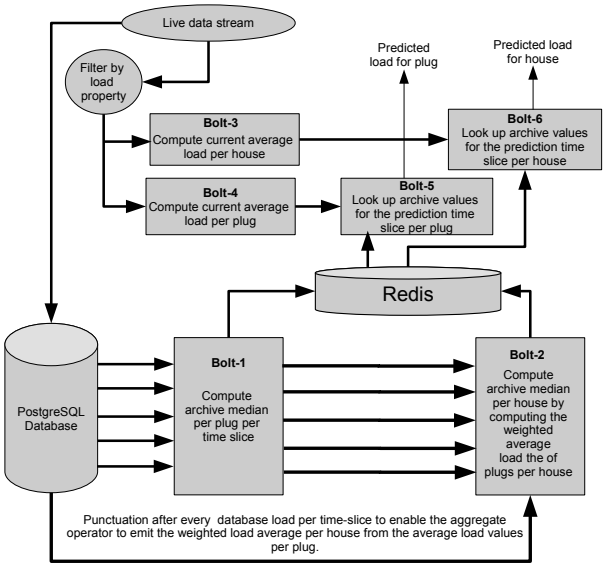

Figure 1: STORM topology for query 1

Figure 1 shows the STORM topology for Query 1. The stream archive is stored in a PostgreSQL database. Concurrent database connections are setup to retrieve historic load aggregates per plug, per time interval through archive streams discussed in Section 3.2. The retrieved values per plug. per time slice, per day are stored in a circular-buffer of size N. The size of the buffer equals the number of days in the archive which are required for load prediction. Storing the archive aggregates in a circular-buffer ensures that irrespective of the number of archive streams we begin with, the count reduces to one after 24 hours since the other N-1 values are already present. Bolt-1 shown in Figure 1 is responsible for computing the per plug archive aggregates before storing the values to the corresponding circular buffer in Redis. Concurrently Bolt-2 connected to Bolt-1 computes the archive aggregates per house, per time slice, per day by

grouping the tuples belonging to a single house and calculating the weighted-average before storing in Redis.

Computing a weighted average is a blocking operation which is resolved by using *punctuations* [7]. Bolt-3 and Bolt-4 group the live stream tuples per house and per plug respectively to compute the current load average over the specified time-interval. Bolt-5 and Bolt-6 look up the Redis data-structure to retrieve the list storing the archive load aggregates for computing the archive-median load values for future load prediction.

4.2 Topology for Query 2

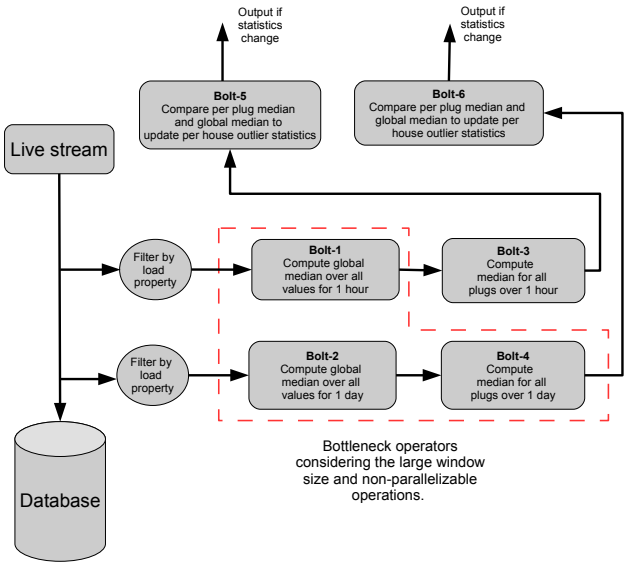

Figure 2: STORM topology for query 2

Figure 2 shows the STORM topology of Query 2. The operators for computing the global median over a sliding time windows of 60 minutes and 24 hours need to be optimized for reasons explained in Section 2.2. As outlined, computing medians over huge sliding windows is challenging if exact results need to be produced. We thus employ stream sampling techniques to produce approximate median load over the data-stream. We thus decided to use the *priority-sample algorithm* [6] which is specifically suitable for sampling over time-stamp based sliding windows. The solution though approximate is quite accurate considering that the individual load values are repetitive. Priority-sampling is also used to compute the global median over an hour and over a day.

Bolt-3 computing medians per plug over a sliding window of an hour computes the exact median while the bolt doing the same operation over one day uses priority-sampling. A Redis based in-memory data structure is updated when ever the percentage of outliers per house changes and the results are sent to the stream subscribers.

5. EVALUATION

In this section we evaluate the performance of our stream processing system under the purview of the challenge guidelines. The experiments were performed on an HPC cluster with CentOS 6.4 operating system. The JVM memory allocated to each worker node and a single *nimbus* i.e. master node in the STORM cluster was restricted to 2 GB. The *parallelism* for all processing bolts (indicating the number of threads per bolt in a node) in the STORM topology was constant throughout all evaluations and thus have no bearing on the relative results.

For workload evaluations, we did not split the input CSV file provided for the challenge into three separate files (as suggested in the challenge guidelines). Rather we added a filter at the live stream spout filtering on the house-id. Finally filtering for load/work and any other property such as house is performed at the stream processing system. Thus the parameter live-stream velocity always implies the number of events sent by the live data-streaming system to the stream-processing platform per second.

5.1 Evaluation for Query 1

For all readings shown below we have restricted the number of archive data streams to three. We have not considered an evaluation which varies the number of archive streams as it is out scope for this challenge. We refer to Query 1a as the query which predicts the load per house and Query 1b as the one which predicts the load per plug. Throughput refers to the number of tuples received by the stream subscriber per second.

Number of slave nodes	Query	Average latency (ms)	Average throughput
1	Query 1a	1071.55	4301.55
	Query 1b	1375.245	4392.25
2	Query 1a	458.92	4907.40
	Query 1b	289.05	4907.40
3	Query 1a	597.43	5038.02
	Query 1b	499.01	5086.04
4	Query 1a	802.65	5156.86
	Query 1b	549.84	5196.07

Table 1: Query 1 parameters on varying the number of processing nodes

Table 1 shows the result of varying the number of processing nodes while maintaining a constant workload of 40 houses and restricting the live stream velocity to 10,000 messages per second. The minimum number of slave nodes required was found to be two. Lack of sufficient memory (fixed at 2 GB per node) and restricted parallelism per node was found to be the main bottleneck while processing the live tuples using a single slave node.

Table 2 shows the results of varying the time slice over which future load was predicted. The workload, live stream velocity and the number of processing nodes were kept constant at 40 houses, 5000 messages per second and two respectively throughout this evaluation. The results indicate that the throughput and latency are time-slice invariant. The results are not surprising considering that the operator computing load averages does not store individual values and instead computes rolling-averages.

Finally Table 3 shows the results of varying the workload while keeping the live-stream velocity (10,000 messages per second) and the number of processing nodes (one) constant. As expected the performance in terms of latency and throughput is best when the number of houses processed is decreased thus reducing the workload.

Time-slice (minutes)	Query	Average latency (ms)	Average throughput
1 min	Query 1a	423.24	2520.83
	Query 1b	328.34	2520.83
5 min	Query 1a	521.51	2520.83
	Query 1b	305.57	2520.83
15 min	Query 1a	510.47	2525.21
	Query 1b	293.52	2525.21
60 min	Query 1a	547.60	2511.15
	Query 1b	361.10	2507.97
120 min	Query 1a	510.62	2512.87
	Query 1b	378.78	2512.87

Table 2: Query 1 parameters on varying prediction time-interval.

Number of houses processed	Query	Average latency (ms)	Average throughput
10	Query 1a	64.41	1219.11
	Query 1b	4.69	1215.10
20	Query 1a	67.44	2224.08
	Query 1b	1.89	2220.36
40	Query 1a	1071.55	4301.55
	Query 1b	1375.245	4392.25

Table 3: Query 1 parameters on varying workload

5.2 Evaluation for Query 2

For all evaluations of Query 2, the live-stream velocity was fixed at 10,000 messages per second. We refer to the query computing the outliers over a sliding window of 60 minutes as Query 2a and the query computing outliers over a 24 hour sliding window as Query 2b. The throughput for Query 2 (both a and b) are not considered given that the output tuples are emitted only when the percentage of outliers in a house change.

The results of varying the number of processing nodes while keeping the workload constant (40 houses) are shown in Figure 3. The results indicate that a minimum of three nodes were required for processing the workload. Figure 4 shows the results of varying workload while keeping the number of processing nodes constant at two.

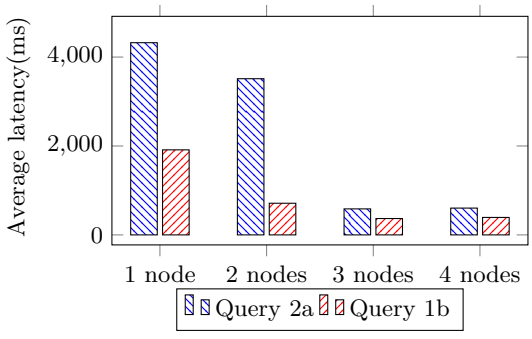

Figure 3: Query 2 parameters on varying the number of processed nodes.

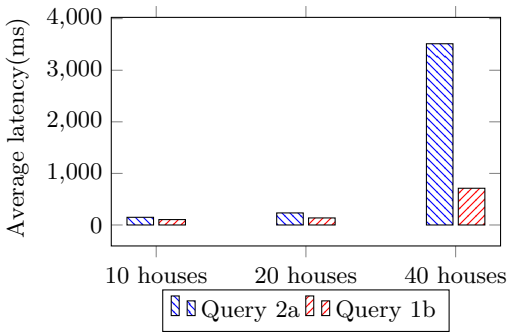

Figure 4: Query 2 parameters on varying workload.

6. CONCLUSIONS

In this paper we presented a stream-processing system to address the queries posed by the DEBS grand challenge 2014. We made use of STORM to enhance the scalability of our platform. To minimize the latency associated with accessing a massive archive, we employed archive streams. Archive streams enabled us to fetch the historic data in chunks of optimal size pro-actively. Sliding window queries and grouping operations over the data-streams were implemented using Esper-enriched bolts. Experiments show that we were able to process data streams at velocities of 10,000 messages per second using just two processing nodes while keeping memory consumption to a reasonable 2GB per node.

7. REFERENCES

[1] Espertech event series intelligence. http://esper.codehaus.org/.

[2] Memory Management in the Java Hotspot Virtual Machine. Sun Microsystems (2006).

[3] Redis data structure server. http://redis.io/.

[4] STORM: Distributed and fault-tolerant real-time computation. http://storm.incubator.apache.org/.

[5] LMAX Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads. http://code.google.com/p/disruptor/, 2011.

[6] B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 633–634. Society for Industrial and Applied Mathematics, 2002.

[7] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *Knowledge and Data Engineering, IEEE Transactions on*, 15(3):555–568, 2003.

[8] K. Tufte, J. Li, D. Maier, V. Papadimos, R. L. Bertini, and J. Rucker. Travel time estimation using niagarast and latte. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1091–1093. ACM, 2007.

[9] H. Ziekow and Z. Jerzak. The DEBS 2014 Grand Challenge. In *Proceedings of the 8th ACM International Conference on Distributed Event-based Systems*, DEBS '14, New York, NY, USA, 2014. ACM.