DEBS Grand Challenge: Predicting Energy Consumption with StreamMine3G

André Martin*, Rodolfo Marinho*, Andrey Brito*, Christof Fetzer*
* TU Dresden - Dresden, Germany
* Universidade Federal de Campina Grande - Campina Grande, PB, Brazil
andre.martin@tu-dresden.de, rodolfo@grupomarinho.com.br, andrey@dsc.ufcg.edu.br, christof.fetzer@tu-dresden.de

ABSTRACT
In this paper, we present our approach on solving the DEBS Grand Challenge using StreamMine3G, a distributed, highly scalable, elastic and fault tolerant ESP system. We will provide an overview about the system architecture of StreamMine3G and implementation details of an application aimed at consumption prediction and outlier detection. Using our elastic approach, we can provide an accurate prediction as we can keep a practically unbounded history able to deal with high volume, highly fluctuating workloads. Our system also provides techniques for dealing with incomplete data in the source stream, which is a common problem when processing data from a large number of sources. Finally, we provide performance measurements showing that we are able to process the dataset given as part of the 2014 DEBS Challenge (135 GB) at a throughput of up to 40 kEvents/s.

Categories and Subject Descriptors
C.2.4 [Distributed Systems]: Distributed applications

General Terms
Algorithms, Design, Reliability

Keywords
Complex Event Processing, CEP, Event Stream Processing, ESP, Scalability, Migration, State Management, Fault Tolerance

1. INTRODUCTION
Smart grids is a general term to describe the usage of information technologies to increase efficiency and robustness of the power grid. The fourth edition of the DEBS Grand Challenge calls for applications centered around sensor data recordings originating from the so-called smart plugs in private households. A smart plug continuously monitors power consumption of a power outlet. Applications are provided with a continuous stream of smart plug recordings with the objective of deriving new streams that provide (1) a short term prediction of the energy consumption and (2) detect outliers [9]. A major obstacle in the course of solving this challenge is the incompleteness of the provided data stream as well as the amount of data provided, which exceeds the processing capacity of any single-node solution, requiring a distributed data processing system.

As the application is provided with a data set in the form of a continuous stream of events, and results must be produced as quickly as possible, well established approaches such as MapReduce [7] cannot be applied. Instead, the challenge calls for ESP applications implemented on top of ESP systems such as Storm [4] and Apache S4 [8] originated from the open source domain, or commercial ones such as Esper [2].

Since the challenge requires estimating future energy consumption, which can only be accomplished using historical data as reference, the ESP system to power the solution must support stateful operators. However, open source systems such as Storm [4] and Apache S4 [8] provide no explicit state management [6], hence, users are forced to use other forms of storage, such as databases or key-values stores, for managing application state. Nevertheless, using external state management limits throughput, scalability and elasticity.

Commercial CEP systems such as Esper [2] and SAP Sybase ESP provide implicit state management (at the cost of throughput) and a convenient abstraction for performing queries on a continues stream of data using a high level query language (CQL – Continuous Query Language). However, this solution is not well suited for solving the challenge as lookups and updates of historical data for deriving a future energy consumption are brittle to express through those CQL based languages. Instead, the challenge calls for an ESP solution supporting explicit state management, for efficiency, as well as a MapReduce-like interface, for scalability.

In this paper, we present our DEBS challenge solution based on StreamMine3G, a scalable, elastic and fault tolerant ESP system with a convenient MapReduce-like interface. Our solution enables sophisticated applications to consist in only of a few lines of code.

The rest of the paper is structured as follows: In Section 2, we introduce the reader to the general architecture of StreamMine3G, the programming model and general concepts. In Section 3, we present our solution and give details on various implementation aspects such as on how to derive the prediction and how to improve accuracy by deriving missing load measurements in incomplete data streams. We
then present an evaluation of our solution in Section 4, and conclude the paper with Section 5.

2. STREAMMINE3G

SYSTEM ARCHITECTURE

Our solution for the DEBS challenge is based on StreamMine3G [5], a distributed ESP System with a MapReduce-like programming paradigm. Such programming approach, if compared to approaches based on SQL-based languages, gives the most freedom to the user as it simplifies the implementation of custom operators. However, high level abstractions such as the ones provided by SQL-based languages using standard CEP operators can also be run on StreamMine3G through an abstraction layer.

Events in StreamMine3G traverse a directed graph of operators, where they can be filtered, transformed, enriched, split, or aggregated, to extract useful information. In order to be scalable, operators are partitioned. Users can either implement their own custom partitioner, such as in Hadoop [3], to route events based on some custom field in the event, or use the default one. An operator partition in StreamMine3G is called a slice and processes a subset of events, determined by the partitioner.

Each slice is equipped with state in order to support stateful operators. State can be conveniently accessed in the process method of the operator without having to implement own bridle locking schemes for consistent state modifications, as state accesses directed at the same slice are serialized in StreamMine3G. However, the hosting of multiple operator slices within one StreamMine3G process still enables harnessing the processing power of nowadays modern multi-core machines.

Regarding fault tolerance and elastic scaling, which required preserving or transferring an operator’s state between machines, users are required to provide appropriate serialization/deserialization code for custom data structures used for storing the operator state. Furthermore, StreamMine3G processes events deterministically. Deterministic processing of events enables the use of active replication in the system for low latency fault tolerance as well as implementing live migration mechanisms by instantiating a second replica and transitioning the processing away from its original location without any interruption or interference in the processing of data stream.

A typical StreamMine3G cluster consists of a set of nodes, where each node can host an arbitrary number of slices. A central component named the manager takes care about the placement of operator slices across physical nodes in a cluster, as well as triggers migration actions when nodes are over- or under-utilized. Hence, a StreamMine3G cluster can expand and contract based on the fluctuation of the incoming data stream or other metrics, such as state size. With respect to the DEBS challenge, the elasticity mechanism of StreamMine3G allows us to keep an infinite history for improving the prediction as operator partitions which are close to saturate the machines memory can be iteratively migrated to newly allocated hosts. Because the migration process is transparent to the operation of the query being processed, StreamMine3G can dynamically adjust its resource pool based on the current demand and without noticeable impact on the application.

3. DEBS CHALLENGE SOLUTION

In the following Section, we will describe the operator graph deployed and running on top of StreamMine3G used to solve the challenge. The challenge defines two queries to be addressed by the system:

1. A load prediction query that provides load forecast based on current and historical load measurements according to the given prediction model, and
2. An outlier detection query to retrieve the ratio of plugs exceeding a certain energy consumption level.

Both queries have in common that they use the same input data set, hence, the resulting operator graph consists of three operators: Source, Prediction and Outliers detection where the latter ones consume the stream produced by the source operator.

We use operator partitioning as provided by StreamMine3G and, as partitioning key, we use the houseId (hId) provided for houses and plugs for different window sizes ranging from 1 to 120 mins.

3.1 Source Operator

The source operator acts as data converter as it consumes events through a file or network stream (binary or text) and converts in messages to flow through StreamMine3G.

As the given input data set is provided as a single 135 GB large file, we partitioned the file to spread it across multiple cluster machines. The sources read the data through memory mapping and parse the data to generate the events. Once a record has been successfully parsed, the corresponding event will be sent downstream to the prediction and outlier detection operators for further processing.

3.2 Prediction Operator

The prediction operator is responsible for providing load forecast based on current and historical measurements. Contrary to the source operator, the prediction operator is a stateful operator keeping history in order to compute such forecasts.

3.2.1 Implementation of the prediction algorithm

As the prediction must be provided at two different levels, house and plug, we consider a data structure for keeping the history such that we can support both levels at no extra memory cost. We achieved this by using the plugs as our most fine granular level in a multidimensional hash map (5-dimensional). The hash map is used to keep the load averages for every house, household, plug and time window. In order to speed up accesses as well as insertions into the hash map, implementations such as boost unordered map [1] can be used as an alternative to the standard STL maps in C++ resulting in constant complexity rather than logarithmic.

As requested by the challenge, a prediction must be provided for houses and plugs for different window sizes ranging from 1 to 120 mins.
In an overview, the prediction operator needs to provide a load prediction for a time window which is two time steps ahead in the future. A load prediction is only generated on the completion of a window and based on the load average of the current window and a median of previous window averages.

Therefore the operator performs two steps: In a first step, the history is updated using the new load measurement received (Line 2-7 in Listing 1), while in a second step, a load prediction is provided (Line 9-29) if and only if a window boundary has been crossed (Line 8).

Algorithm 1 Prediction Operator

1: function PROCESS(ms, state)  
2:   slcId ← (ms.ts - initialTime) / sliceLen  
3:   for each wndSize in state do  
4:     plug ← timeslice[ms.hId][ms.hhId][ms.plgId][slcId]  
5:     plug.val ← plug.val + ms.val  
6:     plug.cnt ← plug.cnt + 1  
7:     plug.avg ← plug.val / plug.cnt  
8:   if ms.ts < nextLdPred then return  
9:   for each wndSize in state do  
10:      slcId ← (ms.ts - initialTime) / wndSize.len  
11:      futureSlcId ← initialTime + (slcId + 2) * sliceLen  
12:      k ← dayLength / sliceLen  
13:      n ← (slcId + 2) / k;  
14:     for each house in timeslice do  
15:       for each household in house do  
16:         predHldLd ← 0  
17:       for each plug in household do  
18:         cur ← plug[slcId]  
19:         if n > 0 then  
20:           prevAvgs ← new List  
21:           for i ← 1..n do  
22:             prevAvgs.add(plug[(slcId + 2) - i * k])  
23:             predPlgLd ← (cur.avg + med(prevAvgs))/2  
24:             else  
25:               predPlgLd ← cur.avg  
26:             predHldLd ← predHldLd + predPlgLd  
27:             emit({wndSize.len, futureSlcId, hld, hhId,  
28:                plgId, predPlgLd})  
29:             emit({wndSize.len, hld, predHldLd})  
30:           nextLdPred ← nextLdPred + 30;  

In order to update the history, we first determine the sliceId (slcId) of the current measurement (Line 2). With each arriving measurements, we update the number of measurements received so far for the current time window (Line 6), the accumulated load (Line 5) and the computed average (Line 7) for each individual window size and plug. The first two fields (cnt and val) allow us to continuously update the average upon arrival of a new measurement while the last field saves computational overhead as the precomputed average can be reused multiple times in the second step of the operator. Updating the history is performed through accessing the multidimensional hashmap (Line 4), retrieving a plug data record and updating its values as described previously. Those updates are performed for every window size using a surrounding foreach loop (Line 3).

If the boundary for time window has been crossed (Line 8), the second part of the operator will be executed: In order to provide a prediction for every window size, the computation is performed multiple times using a surrounding foreach loop (Line 3-28). With every iteration, first the sliceId of the current time window as well as the future one used for prediction is retrieved (Line 10 and 11). Furthermore, the number of time windows (n) within a day based on the window size (Line 12) and the number of days (k) since the beginning of the history is determined (Line 13).

The prediction is then computed for every plug (Line 17) within a household (Line 15) and house (Line 14) by first creating a set containing all load averages measured at the same time of day in the past (Line 20-22), and computing the average between the current average and the median of this set (Line 23). If no history is available (Line 19), the prediction will solely be based on the current average (Line 25). The resulting prediction tuples are then emitted (Line 27) to some downstream operator to trigger further actions or simply for visualization purposes (e.g., in a dashboard).

In order to provide a prediction of the load for a house, we summarize individual plug loads (Line 16 and 26) and emit the sum as prediction tuples (Line 27) resulting in another output stream.

As the outgoing tuples contain the window size and an identifier (e.g., hldId), they can be handled either as a single stream or as separate streams depending on the partitioner or message delivery service used in the rest of the application.

3.2.2 Discussion

In order to verify the accuracy of the prediction model provided by the DEBS challenge committee [9], we extended the algorithm given in Listing 1 by adding another field in the multidimensional hashmap to keep both the predictions and the actual consumptions. By keeping this information, we can compare the actual consumption with the previously predicted consumption. This extension allows a modification of the prediction model and a comparison of the modification in terms of accuracy with the prediction model provided by the challenge committee.

As the provided model improves its prediction accuracy by taking into account a constantly growing history, it does not consider weekly or monthly patterns. However, in western countries, work and private life are dominated by a weekly pattern that is also reflected in energy consumption: The energy consumption for an office or factories is likely to be considerably lower during weekends, if compared to weekdays. This applies also to large commercial consumers, for example, bakeries, which usually close on Sundays.

We therefore extended the original prediction operator to compute a prediction on those weekly and monthly patterns and adjust the original prediction using a weighting factor.

3.3 Outliers Detection Operator

The objective of the outliers detection operator is to provide a stream of events for each house with the ratio of plugs which have a median load higher than the median of all plugs of all houses. Similarly to the prediction query, different window sizes must be considered: 1 and 24 hours.

In order to keep the history needed for the medians of the two different window sizes, we are using a multidimensional map for maintaining the measurements at plug level as well as second map maintaining the global measurements. The two maps are instantiated for the two windows for which an output stream must be produced. We are using two maps as we keep the measurements in an ordered sets for fast retrieval of the median value. While this approach consumes more memory as we keep measurements at two levels (plug and global), we save computational resources needed for the sorting of the time measurements needed to determine the medians. To reduce the additional memory usage, we summarize the messages by keeping only the load value and its
timestamp. This approach considerably reduces memory requirements.

Contrary to the prediction operator where an infinite history must be maintained, measurements can be discarded after expiration (i.e., do not belong to the current sliding window). The pseudocode of our implementation of the outliers detection operator is depicted in Listing 2.

Algorithm 2 Outliers Detection Operator

```java
1: function PROCESS(m, state) 2:   for each wndSize in state do 3:      msSet ← wndSize.mDimMap[m.houseId][ms.plugId] 4:      msSet.add(ms) 5:      wndSizewndSizeMap.add(ms) 6:      for each ms in wndSizegloblMs do 7:         if ms has expired then delete ms 8:      for each house in wndSize.multiDimMap do 9:         for each household in household do 10:            if plug in household do 11:               if ms has expired then delete ms 12:               for each house in wndSize.multiDimMap do 13:                  plugs ← 0; 14:                  plugGlobalM ← 0; 15:                  for each household in household do 16:                     plugs ← plug + 1; 17:                     if median(plug) > wndSize globlM then 18:                        plugGlobalM ← plugGlobalM + 1 19:                     ratio ← plugGlobalM / plugs 20:                     if wndSize.ratio[hhId] < ratio then 21:                        emit((timeWindowEnd, hhId, ratio)) 22:                        wndSize.ratio[hhId] ← ratio
```

First, the measurement is included in the current time window and outdated measurements are discarded (Line 3-11). In order to include the measurement in the time window, we perform a lookup by plugId (Line 3) to retrieve an ordered set of measurements and add the measurement to it (Line 4). In addition to the measurement sets on plugId level, we add the measurement to the global set (Line 5) needed for determining the ratio of outliers later on.

Prior to computing the ratio of outliers, outdated measurements are removed from the global set (Line 6-7) and from plugId level (Line 8-11) by iterating over the ordered sets and evaluating the timestamps of each measurement. In order to compute the ratio of outliers, we iterate over the individual plug existing in each household and house (Line 12, 15 and 16), and count the number of total plugs existing in each house (Line 13 and 17) and the number of plugs that are above the global median. In a last step, the ratio is computed for a house (Line 20) and emitted to downstream operators, if it changed from the previous computation (Line 21 and 22).

3.4 Dealing with Incomplete Data

In this section, we describe our approach for dealing with incomplete data. As outlined by the challenge description [9], load and work measurements might be missing, which is a common case in large scale analytics. However, since work measurements are provided in addition to the load measurements where the former one accumulates over time, it is possible to derive missing load values from the accumulated work stream by computing the difference of the current work value and the last work value, when we have also seen a load measurements. In this way, we can complete the load measurement stream and improve data quality. Note, as we use only load measurements for both prediction and outliers detection, we only improve the data quality of load measurements (i.e., completing the load measurement stream).

Listing 3 shows the implementation of our data completion operator. The intuition of the algorithm is as follows.

Algorithm 3 Data Completion Operator

```java
1: function PROCESS(ms, state) 2:   trace ← state[ms.houseId][ms.plugId] 3:   if ms.type == WORK then 4:      if ms.val > trace.lastVal and trace.lastTs <= t then 5:         timeDiff ← ms.ts - trace.lastTs 6:         expectedLoad ← workDiff * 1000 / timeDiff / 3600 7:         loadDiff ← expectedLoad - trace.totalLoad 8:         if loadDiff > 0 then 9:            emit((id, ts, loadDiff, LOAD, plugId, hhId, hId)) 10:         trace.lastTs ← ms.ts 11:         trace.lastVal ← ms.val 12:         trace.totalLoad ← 0 13:      else 14:         trace.totalLoad ← trace.totalLoad + ms.val 15:      if ms.type == LOAD then 16:         emit(ms)
```

We accumulate the load values with every load measurement that arrives (Line 14) since the last time we received a work measurement. Every time we receive a work measurement, we reset the counter (Line 12). Once the next work measurement arrives, we check if the amount of work changed until the last measurement in order to derive missing load measurements if needed (Line 4). If the accumulated work has changed since the last measurement, we can verify if a load measurement was missing and improve data quality the following way: We derive the load measurement by splitting the accumulated work in equal sized load consumption units (Line 6) which is then subtracted by the accumulated load from the load measurements received (Line 7), if any. If the difference is greater than zero (Line 8), then the smart meter has obviously counted more energy consumption than it was reported through the received load measurements. The difference will then be used in order to generate a synthetic measurement that will then be taken into the consideration for both the prediction and outlier detection. Note also that the completion of the data stream is performed at the level of plugs, hence the above algorithm will be applied to every registered plug by accessing a multidimensional map as in the prediction operator (Line 2).

The data completion operator can be installed at two different locations: It can be installed as an successor operator of the source operator for providing a complete data stream to the downstream operators executing the prediction and the outliers detection, or it can be tightly integrated into the prediction and outliers detection component. The former approach comes with the advantage of the reduction of data transferred to the downstream prediction and outliers detection operator while the latter one saves the extra lookup costs in the multidimensional hashmap. We decided for the first design as it results in a cleaner, more modular and more scalable design. Through a co-location of the source operator and the data completion operator on the same physical host, we can furthermore reduce the extra cost in network traffic imposed by this modular design of a dedicated operator.

4. EVALUATION

In this section, we present the results of various experiments we performed in order to evaluate the scalability and
performance of our proposed solution and its implementation.

4.1 Experiment Setup

The experiments were performed on a 40-node cluster where each node is equipped with 2 Intel Xeon E5405 (quad core) CPUs and 8 GB of RAM. All nodes are connected via Gigabit Ethernet (1000BaseT full duplex) and run a Debian Linux 7.4 operating system with kernel 3.2.0. StreamMine3G and the operators needed for the queries are written in C++.

In order to evaluate the scalability regarding the number of nodes used in the system as well as for the given workloads (10, 20 and 40 houses), we partitioned the provided 135 GB large data set into 40 partitions and spread them across our 40 nodes cluster. Each partition contains the whole trace for one house. We chose this partitioning scheme as it prevents the source operator from being a potential bottleneck in the system as data will be read and parsed in a parallel fashion. In a real, live deployment the data would be coming directly from the network and, thus, could be routed appropriately, not suffering from this disk bottleneck.

In our experiments, we measured the event throughput at the worker operators (prediction or outliers detection operator) while the latency represents an end-to-end from the source operator to the worker operator. The latency also includes the overhead of deriving new measurements from incomplete data as described in Section 3.4.

4.2 Workload Experiments

In our first experiment, we measured the per-query throughput and latency as a function of the workload for 10, 20 and 40 houses as depicted in Figure 1 and 2.

In order to generate the required workload, we varied the number of source operators in a way that only the data partitions containing the data for the first 10, 20 or 40 houses are being taken into account while keeping the number of worker operators (i.e., the prediction or outliers detection) constant. Our experiment revealed that for the prediction query, the source operator is the bottleneck, saturating the CPU of the nodes. For the outlier-detection query, the worker operator (the outlier algorithm itself) is the bottleneck.

Because the source operator is the bottleneck in the prediction query, we see a slight throughput increase when increasing the number of houses in Figure 1 (left). At the same time, we can see a decrease in latency (left on Figure 2), which is due to the batching behavior in StreamMine3G, where higher volumes of events result in lower latency as events stay shorter periods of time in buffers prior to being sent over the network.

As noted above, in the outliers detection query, the worker operator (outliers detector) achieves a 100% CPU utilization. Due to the extreme overload of this operator, we also experienced a high queuing behavior in StreamMine3G which accounts for a constant increase in latency as events remain longer in an incoming event queue that is constantly growing. As shown in Figure 2 (right), latency can go beyond seconds range and reach up to 23 s (peak) once a node is fully saturated.

Since events are being enqueued in an incoming event queue, we can see an increase of throughput when more houses are added to the workload as depicted in Figure 1 (right). Nevertheless, the net throughput for the outliers detection operator stays constant at around $1kEvents/s$ while the remaining events end up in queues resulting in such high latencies.

In order to prevent memory exhaustion in StreamMine3G when en-queuing events during a temporary or permanent overload, a built-in back-pressure mechanism kicks in once a predefined threshold has been exceeded slowing down upstream operators. The back-pressure mechanism can operate either on a fixed threshold value or dynamically based on metrics such as current memory consumption of the physical node.

4.3 Scalability Experiments

In the last set of experiments, we assessed the scalability of the system by varying the number of worker operators while keeping the number of source operators constant at 40, hence providing the whole data set to the queries. The results of the experiment are shown in Figure 3 and 4.

In our experiment, we saw a throughput increase of roughly 22% when doubling the number of worker nodes for the prediction query while we got only a 2% increase for the outliers detection query which confirms our initial assumption about the limited scalability of the outliers detection query.

Moreover, an increase in the number of worker operators decreases latency as each partition of the worker operator (prediction) maintains only a subset of historic energy consumption profiles resulting in faster processing and lower overall latency. In contrast to the prediction operator, the outliers detection operators needs to maintain a global view of all plugs of all houses, hence, with the current, simple implementation, it cannot be sub-partitioned in a similar fashion as the prediction query.

4.4 Discussion

During our performance evaluation, we identified the outliers detection operator as the least scalable component in the system. This limitation is due to the requirement of the challenge where a median across all plugs must be provided, hence, with an intuitive implementation a single partition has to be used to compute the median and then needs to consume all incoming events. A simple approach to solve this problem is to use an approximation of the median (but this would conflict with the goal of the challenge).

The median computation itself can be performed in two different ways: Through a sorted set which exposes little computation costs, but comes with a high price for the removal of expired entries as an iteration over all entries in the window is required, or, alternatively, through a linked list, which allows an inexpensive removal of expired entries but comes with the cost of sorting each time the median is needed. As the computational cost for sorting is higher than iterating over the sorted set for removing expired entries, the algorithm presented in Listing 2 already depicts the best performing solution.

In order to improve performance, we propose a decoupling of the global median computation across all houses and the per house median by using two operators: One operator which will be deployed as a single partition will be responsible for computing the globals median while a n-partitioned local median operator is in charge for the median computation for each house and correlating its result with the result of the global median operator.
5. CONCLUSION

In this paper, we present our implementation of the DEBS 2014 challenge on top of StreamMine3G, our elastic and fault tolerant ESP system. Besides providing the required load-prediction and outlier-detection output streams, our implementation also handles incomplete data. We presented details of our implementation and discussed alternative approaches for solving the challenge with an analysis regarding performance trade-offs. We finally performed a thorough performance evaluation on our 40-nodes cluster in which we measure event throughput and latency. Through our measurements, we confirmed our assumptions concerning the scalability of each component of the system and achieved a throughput of up to 40 k.Events/s. We also provided a screencast of our working system which can be viewed at: https://streammine3g.inf.tu-dresden.de/debs2014demo

Acknowledgment

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2012-2015) under grant agreement number 318809 (LEADS) and from the CAPES, DAAD and GIZ through the NoPa program (TruEGrid project, 2011-2013).

6. REFERENCES