DEBS Grand Challenge: Predicting Power Needs in Smart Grids

Aman Mangal, Arun Mathew, Tanmay Randhavane, Umesh Bellur
Department of Computer Science and Engineering
Indian Institute of Technology- Bombay, Powai, Mumbai-400076
{amanmangal, arunmathew, tanmayr, umesh}@cse.iitb.ac.in

ABSTRACT

Smart grids are becoming ubiquitous today with proliferation of easy to install power generation schemes for Solar and Wind energy. The goal of consuming energy generated locally instead of transmitting it over large distances calls for systems that can process millions of events being generated from smart plugs and power generation sources in near real time. The heart of these systems often is a module that can process dense power consumption event streams and predict the consumption patterns at specific occupational units such as a house or a building. It is also often useful to identify outliers that are consuming power significantly higher than other similar devices in the occupational unit (such as a block or a neighbourhood). In this paper, we present a system that can process over a million events per second from smart plugs and correlate the information to output both accurate predictions as well as identify outliers. Our system is built from the ground up in C++ achieving very high throughput with very low CPU capacity for processing events. Our results show that the throughput of our system is over a million events per second while using under 20% of one core.

Categories and Subject Descriptors
C.2.4 [Computer-Communication Networks]: Distributed Systems

Keywords
distributed system, smart grids, event processing

1. INTRODUCTION

Smart grid is quickly becoming the norm where there are many points of energy generation and the guiding principle is to consume the generated energy as close to the energy source as possible to avoid transmission over long distances. In order to accurately channel energy from the source to the consumption points, it is necessary to rely on a system that can predict the energy requirements at different occupational units in real time, while consuming dense streams of millions of (energy consumption and load pattern) events per second. The DEBS 2014 Grand Challenge[6] focuses on this problem and specified two queries in this space.

While we investigated various open source complex event processing systems such as Esper[4] and Padres[3] as well as different execution environments and queuing products such as Erlang[1] and ZeroMQ [2], we chose instead to build this engine from ground up using C++ and BSD sockets. Previous studies of CEP systems such as Esper [5] led us to believe that the performance of custom solution would be much better. We also did an initial comparison (without any optimizations) of ZeroMQ sockets with BSD sockets and found that latter performs more than 5 times better in terms of rate of processing events. In this paper, we present a simple and elegant solution with very high throughput rate (of over a million events per second) and scalability conceived purely for the purposes of answering the two queries given as the DEBS 2014 Grand Challenge.

The paper is structured as follows: In Section 2, we detail our design philosophy and approach. Sections 3 and 4 describe the system architecture and experimental results for query 1 and 2 respectively. We present future work in Section 5 and conclude the paper with the main contributions in Section 6.

2. DESIGN APPROACH

In line with our philosophy of custom building a solution most efficient for each query, we present different system architectures to handle query 1 and query 2. The requirements of each of these queries dictated the following design choices. Since the load prediction even for a house is dependent only on median values within the house, it is possible to process data for different houses in parallel (possibly on different nodes) in case of query 1. On the other hand, a distributed system in case of query 2, will require a large number of message exchange among the processes to compute the global median before putting out an event. Hence we chose to design the system such that it is distributed by house for query 1 while it is a single process for query 2.

Also, query 2 computes median for a large set of data, potentially of $24hrs \times 3600sec \times 2125plugs \approx 270$ million numbers. Finding exact median of these numbers many times per second will be computationally intensive. We, therefore, compute approximate medians in this case. Whereas in query 1, we compute medians of at most 30 numbers (a month data), which would be computationally feasible.
We have used C++ to implement the solution presented in this paper. We used Arrays and Hashmaps as our primary data structures to store data for quick lookups and quick binary searches. Due to the fact that the number of households in a house and number of plugs in a household are not known apriori, we had to sometimes use hashmaps in place of arrays to store data for different plugs and households.

3. QUERY 1

Query 1 requires us to forecast the average load of each plug and each house in the system for different time slices of length 1m, 5m, 15m, 60m and 120m. The average load forecast is computed over all possible slots of all the time slices. In order to make a forecast for the next to next slot, we use two values - current average load and median load of the corresponding slot of given time slice. We put out a forecast every 30 sec i.e. for each time slice we compute average load whenever we accumulate data for next 30 seconds.

3.1 Architecture

The system consists of a broker process and multiple house processes (one per house). The broker reads the data file and creates an event stream for each house process. The house processes are responsible for forecasting average load of the house and all plugs in the house.

In house processes for each plug, we keep a 30 second accumulator and counter. Whenever we receive an event for a plug, we increment the accumulator with the load\(^1\) value in the received event and increment the counter for computing the average load later. As soon as we receive an event crossing the 30s time window, we compute and output the forecast for all time slices and reinitialize only the 30s accumulator to the load value in the received event and count to 1. We also keep load value accumulators and counters for different time slices (1m, 5m, 15m, 60m and 120m). If any of the time slice boundaries are crossed, corresponding accumulators and counters are reset to 0. We output the forecast as zero if all the data is missing in a slot for a time slice. Note that, the above algorithm is based on the assumption that the timestamp of events coming from the broker never decreases. We, therefore, can make the forecast whenever any event corresponding to any plug in the house, crossing the current time slot for a time slice is received. We do a similar computation for house average load forecast.

We designed a Median Container (MC) to compute exact median of average load for a given slot of fixed time slice in case of query 1. The MC provides an interface to insert a new element and retrieve the median of all the elements currently in the container. For this, MC uses a min heap, max heap and a middle variable. The min heap holds the upper half of the inserted values and max heap holds the lower half. If the number of values in the container is odd the middle variable keeps the median, otherwise the median is the average of the top of max heap and min heap. New entries are added while carefully maintaining the above specified property. The asymptotic complexity of insert is \(O(\log N)\) whereas median can be retrieved in constant time.

3.2 Prediction Model

We have used Weighted Average with Adaptive Weights Prediction Model in order to forecast the average load of a house or a plug. The *Predicted Average Load* (PL) for a given time slot of a given time slice is computed as follows:

\[
PL = W \times HM + (1 - W) \times CA
\]

Where HM (History Median) is the median of average loads of previous days for the same time slot of same time slice and W is the weight that we are assigning to HM in comparison with *Current Average* (CA). We make the prediction for next time slot of same time slice. Hence, CA is the average of last time slot. We give equal weights to HM and CA in the beginning and set the initial value of the weight W as 0.5. As the program execution progresses, the value of W is adapted based on the actual value we receive for each output forecast. We define the error in the forecast as follows-

\[
\text{Error} = (\text{AL} - \text{PL})^2 + \lambda \times (W_{\text{new}} - W)^2
\]

\(\lambda\) is a parameter to control the learning over W, AL is *Actual load* and PL (*Predicted Average Load*) is computed using \(W_{\text{new}}\). Our goal is to minimize the above Error. We differentiate Error with respect to \(W_{\text{new}}\) and get the following expression when equated to 0 -

\[
W_{\text{new}} = \frac{W \times \lambda - (\text{AL} - \text{CA}) \times (\text{HM} - \text{CA})}{\lambda + (\text{HM} - \text{CA})^2}
\]

Note that we use different weights for different houses for each time slice. We tried different values of \(\lambda = 0.01, 2, 100\) and achieved minimum error in case of \(\lambda = 2\). \(\lambda\) is kept fixed throughout the experiment.

3.3 Experimental Evaluation

We executed our solution on a cluster of KVM virtual machines. The house and broker processes run on a single core 2.1 GHz virtual machine with 1 GB memory and attached to a 10 Gbit local area network. The virtual machine is running Ubuntu 12.04 on dual processor Intel Xeon E5 2620 base system. We have done experiments running house processes on 1, 2 and 4 virtual machines dividing the number of house processes equally amongst the VMs. For throughput calculations, we redirected the output to the null device. Throughput will simply be the total number of input events divided by the total time taken in processing all the events. We did each experiment three times and took the average of the three throughput values.

![Figure 1: Latency vs No of Houses (Query 1)](image)

As we increase the number of virtual machines, the latency values do not change much as shown in Figure 1 because the broker becomes the bottleneck. But latency increases as the number of houses increases, this is because of additional events between two events of same house. On the other hand, throughput remains nearly constant in all the experiments as shown in Figure 2.
We contend based on utilization of CPU at the broker; house processes consume little of CPU. It is the broker that is currently limiting the scalability of the system. The parsing of the events accounts for most of the time spent by the broker on the CPU. If we were getting events as a stream and can avoid both the disk I/O and conversion of strings to numbers, we can witness the scalability of the system as a whole since the house processes are separate and can be run on different cores.

The Algorithm is inspired from discussions on http://stackoverflow.com/questions/638030

4. QUERY 2

The goal of query 2 is to identify outliers that consume significantly higher amount of energy compared with the average consumption computed globally. A plug is counted as an outlier if median load of the plug is more than the median load of all the plugs in the system for a given sliding window (1 hr or 24 hrs). We need to produce an output event when the percentage of such plugs in a house changes.

4.1 Architecture

To evaluate query 2, we have to compute global median i.e. median of all the data of all the plugs in the system acquired in a given sliding window of length 1 hr or 24 hrs. Then, we compute percentage of outlier plugs every time we receive a new event using plug medians. The frequent comparison of locally computed data with global data enforces a single node solution for all the houses in case of query 2.

There are two processes in the system residing on different nodes. The broker process reads data from a file and sends events to Q2 process. Q2 keeps a linked list of all the events of last 24 hrs in the order they have been received. It also keeps instances of Sliding Median Container (SlidingMc) class in order to compute various medians and S Container (SCont) class for the purpose of finding the number of outliers in every house.

4.2 Median Algorithm (SlidingMc)

In query 2, we have to compute the median of large amount of data frequently. Hence, there exists a trade off between computational complexity and accuracy of the median so derived. The basic idea is to construct a histogram of the data using fixed number of bins. We sort the first M distinct values inserted into the container and use them to build up the (M-1) bins. Every bin is associated with a frequency denoting the frequency of the data greater than equal to the i-th lowest value (including) but less than the (i+1)th lowest values (excluding) among M inserted value into the container. We also store the index of the bin containing the median after an insertion or deletion is performed, cumulative sum of the frequencies upto and including the bin in which median resides and the total number of values inserted into the container. These numbers are used in order to quickly find the median of all values present in the container.

To adapt the histogram with newly coming data, the bin configuration is changed dynamically. A bin is split into two bins if it becomes too full and deleted if it becomes empty. If number of bins are higher than the maximum allowed bins, we merge 2 bins. The insert and delete operation takes O(M) time in the worst case scenario whereas getMedian is a constant order operation.

We did 1000 experiments and computed the relative error in median of 50000 random values. The CDF of relative error is plotted in the Figure 4. It can be observed that the approximate median computed using SlidingMc container shows low relative error. We also did several tuning experiments and found that bin size greater than 1200 gives lowest relative error. Hence, we used 1200 as bin size while implementing query 2.

4.3 S Container

S Container is used to efficiently compute number of plugs in a house having higher median load than the global median.

Figure 2: Throughput vs No of Houses (Query 1)

Figure 3: Linked List in Q2 process

Let’s assume an event E is received by the Q2 process from the broker process with timestamp $T_E$. Let’s also assume the window size to be $W$ which can be either 1 hr or 24 hrs. $T_W$ is the minimum timestamp which exists in the window of length $W$ terminating with the event E as shown in Figure 3. Note that there may be multiple events with the timestamp $T_W$ in the window. In such a case, we choose the left most event with the same timestamp $T_W$ in the linked list. Also, if the data corresponding to the timestamp $\tau = (T_E - W + 1)$ is missing, then $T_W$ will be the first timestamp bigger than $\tau$ for which event for any plug in the system is present. Now, the algorithm proceeds as follows. We shift the left end of the original window (window before the input event was received) one event, keeping the right end fixed. We delete the oldest event from the median container of the corresponding plug and the global median container. Now, we compute new median of the plug and insert the new median into the S container of the house. S container is, then, queried to find out whether the percentage of outliers for a given house has changed after the shift. We output an event if any of the percentage for any house has changed.

We stop when the leftmost event in the window has the timestamp $T_W$. In the last slide, before computing any medians, we also insert the newly received event E into the global median container and the median container of the plug of the event E. Rest of the computation remains same.

To avoid redundant storage of the data, we keep the linked list only for 24 hrs sliding window, and keep a pointer to the first event of the 1 hr window in the same linked list.

3The Algorithm is inspired from discussions on http://stackoverflow.com/questions/638030
load (getNumOfLargeNum() method). The challenge in implementing S Container class is that we don’t know number of plugs in a house before hand. Hence, we had to use a map (key-value storage) in order to store the plug medians which leads to O(K) complexity of the getNumOfLargeNum method where K is the total number of plugs in the house. We solved the problem by using a sorted array of plug medians. When a new plug median is inserted, we require the old plug median to search its location in the array. We replace the old median with the new median and move the newly inserted median such that the array remains sorted.

The asymptotic complexity of this algorithm is also O(K) but in the given scenario, a momentous change in median values is unlikely. Small movement of data, therefore, will be required while sorting the array while an insertion is performed. On the other hand, getNumOfLargeNum method will always be cheaper than log(K).

4.4 Experimental Evaluation

We used the same setup as of query 1 for experimental evaluation of query 2. The broker and Q2 processes are executed on different virtual machine in the same cluster. We are able to achieve a constant average throughput of 0.48 million events per sec as shown in Figure 5. Because there is only one process computing all the percentage values, throughput doesn’t change significantly in terms of input events/sec as the number of house increases. The CPU utilization level in case of Q2 process is always close to 100%.

For query 2, latency of an output event is defined as amount of time taken to output the event after the corresponding input event is received. Note that one input event may result in more than one output event with increasing latency values for the events put out later. Latency graph for query 2 is shown in Figure 6. Due to the fact that the first 10 houses have disproportionately large number of events compared to rest of the houses, the 90th percentile for 10 houses is extraordinarily high.

5. FUTURE WORK

In the presented work, we have not used work values. We could use them in query 1 implementation while computing total load for a given time window. Though, we will only compute average load for longer time slices such as {60 min, 120 min} using work data. This would lead to more precise values of average load for a time slot of given time slice.

6. CONCLUSIONS

In summary, the main contributions of this work is as follows. First, a custom design and implementation of a smart grid plug load predictor and outlier identifier system. It is our belief that the custom design led to the significant throughput we observed with the given data set. In fact, the throughput of the system is much higher than what is reported because the bottleneck at broker process is due to parsing which is really not part of the predictor system. Second, a scalable architecture to process dense smart plug load data streams. Every house process can be given its own core to run on and hence, the scalability is limited to the number of house processes we want to process the data for. Third, a fast and lightweight C++ implementation of the median based prediction heuristic to forecast the load at smart plugs for different time windows. Fourth, a method and code for efficiently sliding two different timeslice windows over a continuous stream of events.

7. REFERENCES