DEBS Grand Challenge: In-Memory, High Speed Stream Processing

Rohit Gupta
Indian Institute of Technology
Bombay, India
rohitg@cse.iitb.ac.in

Rinku Shah
Indian Institute of Technology
Bombay, India
rinku@cse.iitb.ac.in

Apurva Mhetre
Indian Institute of Technology
Bombay, India
apurva@cse.iitb.ac.in

ABSTRACT

To enable load prediction we require three ingredients; Data acquisition, Accurate prediction algorithms and timely prediction. This work is focused on the third ingredient i.e. Timely prediction. Timely prediction is required for proper functioning of smart grid. Towards this we offer a scalable, distributed and incremental solution.

We present a solution that utilise multiple techniques to evade performance degradation maintaining timing requirements. These techniques include:(1) Splitting the nested processing into stages to maintain high throughput. (2) Customised median finding algorithms to generate timely output from high volume data. (3) Pre-fetching to eliminate disk access time for accessing the historical data. Implementing these together, our technique is able to perform *11K* predictions per sec, and *2.5K* outlier computations per sec.

Categories and Subject Descriptors

[**DEBS Grand Challenge**]: 2014; [**DEBS**]: Metrics—*complexity measures, performance measures*

General Terms

DEBS Grand Challenge

Keywords

demand forecast, distributed, scalable, outlier

1. INTRODUCTION

Our problem statement comes from 'DEBS Grand Challenge 2014' [4] and falls under the smart grid domain. It addresses the analysis of energy consumption, and targets on the following problems: (1) Load-forecasting and (2) Outlier detection

Power demand is continuously fluctuating, however when power is demanded it should be made available. To tackle this uncertainty, we need to use power-generating resources effectively. Currently to meet this fluctuating demand power generating stations have to maintain huge standby production capacity. Accurate load and generation forecasting would solve the above problem.

There are twelve queries required to be processed[4]. Ten queries are used to provide load prediction for houses , and individual plugs. Prediction is computed over slice sizes of 1min, 5min, 15min, 60min and 120min. Prediction output is updated every 30 seconds. Two queries are used to detect and report outliers. This query has a sliding window of 1 hour and 24 hours. Outlier result is computed for every incoming/outgoing event, and updated whenever there is change in the percentage of anomalous plugs, for that house.

Challenges

Challenges handled in this paper include:

1. The data set is collected in real-world environment which implies the possibility of malformed data as well as missing measurements. Using the data-set as it is, may lead to inaccurate predictions and outliers.

2. Simultaneous processing for thousands of plugs, within strict deadline.

3. Median computation on dynamic data-set containing millions of records, within strict deadline.

4. Efficient data storage and retrieval, without causing system performance loss.

Main Contribution

Main contribution of this paper include:

1. We implemented two new techniques for median computation: (a) Hash based (b) 2 Binary search tree method. These techniques reduce the median computation complexity to O(1) and is in the range of few milliseconds. These techniques have trade-off between space-complexity and record-insertion-time complexity. *Hash based* technique fits best for Global Median computation, and *2 Binary Search tree method* [3] technique fits best for House-level Median computation.

2. We improved data access performance by implementing: (1)Smart reuse of in-memory data, that eventually reduces the number of disk access cycles, (2)Prediction of next data-block to support pre-fetching.

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

http://dx.doi.org/10.1145/2611286.2611332.

This paper is organised into following sections: Section 2 points out the method used to handle data quality issues. Section 3 describes the System Architecture. Section 4 discusses system evaluation. Section 5 concludes the work.

2. DATA QUALITY ISSUES

As data given by 'DEBS Grand Challenge 2014' [4] is collected from uncontrolled and real environment it has some missing and duplicate measurements. Incoming data goes through a preprocessor, that resolves *duplication* and *missing data* issues. Duplicates are eliminated. Missing data is regenerated by obtaining missing power values, using accumulated energy values.

3. SYSTEM ARCHITECTURE

Our system is developed in C++ using standard libraries. It consists of two parallel, multi threaded subsystems; for forecasting and outlier detection. It has an independent listener that reads the input, and replicates it for both systems. Inter-thread communication is supported through custom queues; that hold the data structure, and are also capable of handling concurrency issues. Data structures implemented are capable of adjusting with new plug data received, therefore system should not be restarted to add new plug, household or house. System also stores its current and historical data in a database, to enable continuation of its execution from its previous state. Our system is divided into following parallel subsystems viz. 1) Listener, 2) Forecasting System, 3) Outlier Detection System.

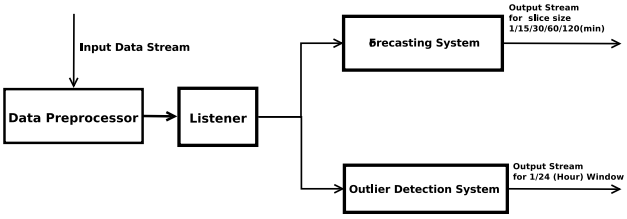

Figure 1: System Architecture

3.1 Listener

Input data is acquired by a Listener module, that generates various input streams of pre-processed data. Listener is parametrized to specify number of copies of input stream to be generated. This helps in distributing and scaling our system. Listener module transmits pre-processed input stream into all the queues connected to it.

3.2 Forecasting System

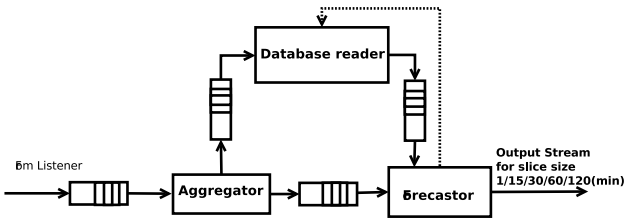

Figure 2: Forecasting System

This module is responsible to forecast the power consumption, for every house, and plug. Forecasting System is divided into three modules viz. a) Aggregator, b) Database Reader, c) Forecaster

3.2.1 Aggregator Module

Power data is aggregated for 30 seconds; and used for further processing. This reduces the disk space required to store the data; and the retrieval is also fast. We call each 30 second interval, a slot. We assigned value of 0 to 2879 for each slot for a the day. Aggregator module is responsible of adding up the power consumed during 30 seconds interval and send it to Forecasting and Database modules.

3.2.2 Database Module

Sqlite is used to store historical data. A separate module takes care of all data read and write operations.

3.2.3 Forecaster Module

This module take recent and historical data to compute predicted power. Historical data is stored on disk, data is fetched in advance to avoid frequent disk access delays.

This module holds per slot aggregate power for the last 2 hour data. It also holds aggregate historical slot data of next 4 hours. Previous data is reused in next prediction to achieve reduction of disk access by 480 to 1.

3.3 Outlier System

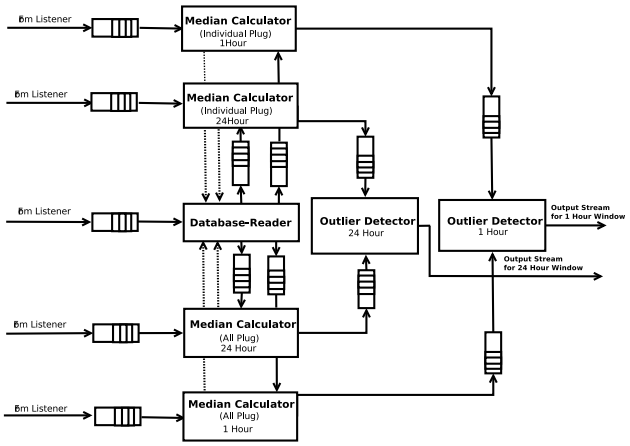

Figure 3: Outlier System

Our system is capable of supporting any window size for outlier detection. Each window size is executed by parallel threads. To slide the oldest power readings are removed, and new readings are added, to the median computation data. We save the old data in disk based or in-memory sqlite database which act as a large queue. Outlier System consist of following modules:

3.3.1 Plug Median module

It uses 2 equal sized binary search tree structure for median computation. This module is expected to hold 3600 power samples per plug for 1 Hour processing, and 86400 samples per plug for 24 Hour processing.

3.3.2 Global Median module

This module computes median for all plug data received. As the distinct values for the power is limited, and significantly less than the number of samples, hash based median computation is used. In this way we accommodate 172.8 Million values in array of size 10 Million. As the number sample increases, the empty space in the array decreases, and the cost of insertion/deletion of keys reduces. This makes the module perform better for higher volume of data.

3.3.3 Outlier module

This module receives the median; and computes outlier by simple comparison in a hierarchical structure.

4. EVALUATION

4.1 Analysis of Median computation methods

Median should be computed for prediction on a small historic data (one per day). Median is required to be computed for each plug whenever an event is received for that plug. the data set for this computation will be in the range of 3.6 thousand to 86.4 thousand. We tested different median computation methods to find the best method for each of the above objectives.

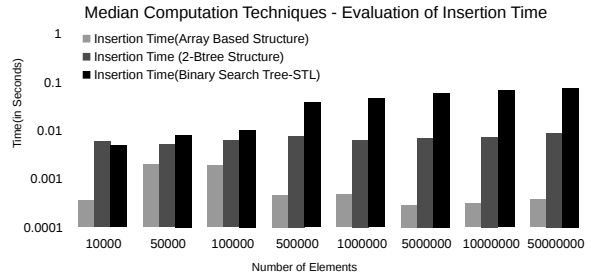

Figure 4: Evaluation of Insertion Time

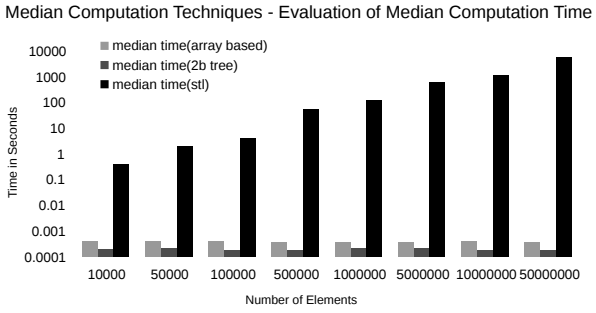

Figure 5: Evaluation of Median Computation Time

Balanced Binary tree (STL)

STL multiset is used to sort the power values. Median is computed by scanning the multiset.

Space complexity is O(n), one key insertion cost is O(log n). Median retrieval cost is O(n) as half of the tree is traversed.

Hash based technique

An array is used to store the number of occurrences of each power value (key). A reference of the key holding the median value is maintained. Whenever a new value of power is added, the occurrence count for that key is incremented. To find new value of median, sequential scan of the array is performed to find the next key with non-zero occurrence.

As the number of distinct values of power is limited, Space complexity is O(k), where k is the size of the array. One key insertion cost is O(1), additionally a cost of O(e) is incurred to find the new position of median. here e is the number of continuous free elements in the array that need to be traversed to move median pointer. Median retrieval cost is O(n). For high data volume the value of e becomes very small therefore this method is suitable.

2 equal sized Binary tree with median stored separately

It holds median in one / two variables and rest of the data is stored in two equal sized threaded binary. When a new power value is inserted, it is compared with the median to identify whether it should be inserted to the left, right tree or it is in between the two median.

If the value is inserted to one of the trees, one of the following action is performed to maintain equal size of two trees. (1) If there are two medians, one of the median is inserted to other tree. (2) If there is one median, first/last element is removed from the tree, in which the new value is inserted.

Space complexity is O(m), where m is the distinct value of keys. insertion cost is O(Log n), additionally a tree operation could be required to make the two three equal size. this cost will also be O(log n). Median retrieval cost is O(1).

We also analysed few other methods to find median like Boost multiset, B tree [1], and B+ tree [2]. but these were not used because of their complexity.

4.2 System Performance

Experimental Setup

We have performed our experiments on a system of following configuration. Ubuntu 12.04.4 LTS, Linux 3.8.0.29 generic (x86-64) kernel running on 8x Intel(R) Core() i7-3770 CPU @ 3.40 Ghz.

Our system reads complete data file, processes the required data (10, 20, 30, or 40 house), and ignores data for remaining houses. It start processing next data immediately after finishing for one data. This enabled us to process entire month data in few days. Time-stamp was logged while reading next time-stamp data from the file. Times-tamp was also logged while generating each output. These time-stamps were used to compute various performance matrices.

Figure 6 show that average throughput increases, with the increase in the number of houses. This is because disk access cost is amortised over more plugs. This effect should be because of accelerated execution where disk access is the main bottleneck .

Figure 7 show that throughput varies within limits. This should be because of the median computation algorithm. If data is more the array is more densely filled, therefore processing time is less. Figure 8 shows that the per query prediction latency (average) increases with the increase in number of houses.

Figure 9 shows that the per query outlier latency (aver-

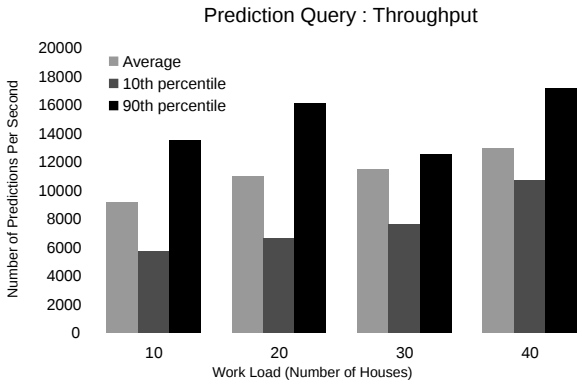

Figure 6: Per Query Throughput Performance: Prediction

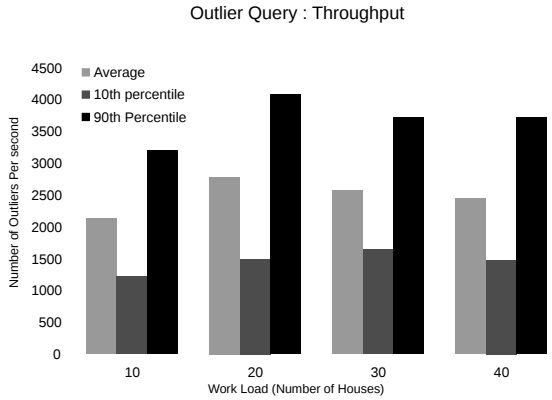

Figure 7: Per Query Throughput Performance: Outlier

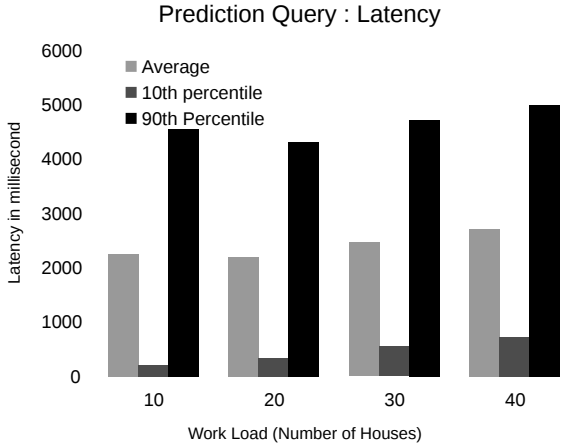

Figure 8: Per Query Latency: Prediction

age) is almost constant. This is because the median computation time complexity of the algorithm used is O(1).

Above experimental results show that our system performance is not affected with the increase in the workload.

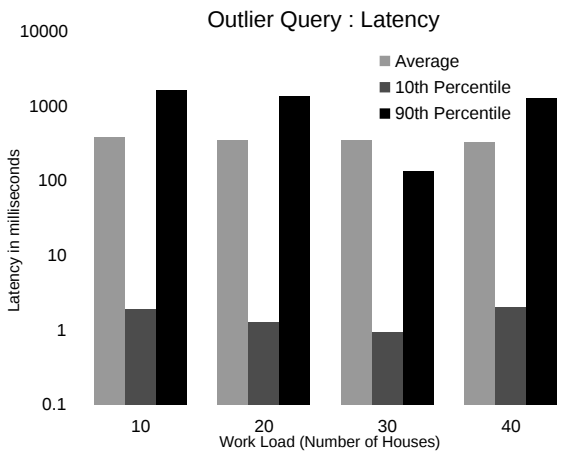

Figure 9: Per Query Latency:Outlier

Average, 10^{th} percentile as well as 90^{th} percentile values for both queries does not vary significantly with workload.

5. CONCLUSION

This paper presents a design to process high frequency plug sensor data. Modular design presented could be easily extended to implement other algorithm. Test results shows that the presented solution could be utilised in processing larger volume of data.

6. ACKNOWLEDGEMENTS

The authors would like to acknowledge the discussions with Prof. Krithivasan Ramamritham and Prof. Umesh Bellur which helped in evolving the solution approach.

7. REFERENCES

[1] Donald Knuth. The art of computer programming, volume 3: Sorting and searching, second edition. addison-wesley, 1998.

[2] Ramez Elmasri Navathe and Shamkant B. Fundamentals of database systems, 6th edition, 2010.

[3] Internet Source. http://www.ardendertat.com/2011/11/03/programming-interview-questions-13-median-of-integer-stream.

[4] Holger Ziekow and Zbigniew 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.