Grand Challenge: Scalable Stateful Stream Processing for Smart Grids

Raul Castro Fernandez, Matthias Weidlich, Peter Pietzuch
Imperial College London
{rc3011, m.weidlich, prp}@imperial.ac.uk

Avigdor Gal
Technion - Israel Institute of Technology
avigal@ie.technion.ac.il

ABSTRACT

We describe a solution to the ACM DEBS Grand Challenge 2014, which evaluates event-based systems for smart grid analytics. Our solution follows the paradigm of *stateful data stream processing* and is implemented on top of the *SEEP* stream processing platform. It achieves high scalability by massive data-parallel processing and the option of performing semantic load-shedding. In addition, our solution is fault-tolerant—the large processing state of stream operators is not lost after failure.

Our experimental results show that our solution processes 1 month worth of data for 40 houses in 4 hours. When we scale out the system, the time reduces linearly to 30 minutes before the system bottlenecks at the data source. We then apply semantic load-shedding, maintaining a low median prediction error and reducing the time further to 17 minutes. The system achieves these results with median latencies below 30 ms and a 90th percentile below 50 ms.

1. INTRODUCTION

The goal of the ACM DEBS Grand Challenge is to conduct a comparative evaluation of event-based systems by offering real-life event data and requirements for event queries. The 2014 edition of the challenge [15] focuses on smart grid analytics. The challenge is based on measurements of energy consumption, expressed as "instantaneous load" and "cumulative work", at the level of individual electricity plugs. These measurements are synthesised from real-world profiles derived from a set of smart home installations. The event queries focus on two types of analytics: (i) short-term load forecasting and (ii) load statistics for real-time demand management.

We observe three main characteristics of the 2014 challenge:

High data volume. The data includes load and work events of individual plugs at a rate of approximately one measurement per second. For the considered 40 houses with roughly 2000 plugs, this yields a volume of more than 4 billion events for one month. Given future predicted growth, applications of smart grid analytics can be expected to process data of hundreds to thousands of houses, increasing the volume by one or two orders of magnitude.

Unbounded and global state. The event queries require sophisticated handling of processing state. Short-term load forecasting relies on historical data, which requires aggregates to be maintained for an unbounded time window. This is challenging due to the ever-growing state to be stored. The performance of computation on this state is likely to degrade over time. Moreover, load statistics for real-time demand management require a global aggregation over all houses and plugs, which limits the options to distribute processing. Unbounded and global state also affect fault-tolerance mechanisms because it becomes expensive or even impossible to recreate all state by reprocessing events.

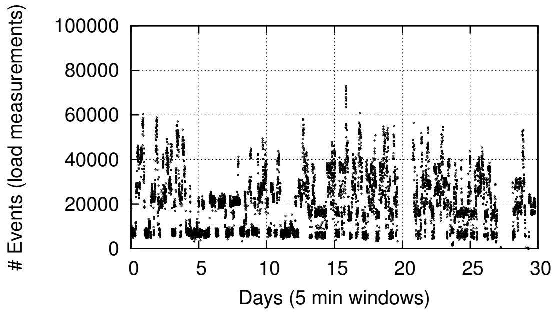

Figure 1: Load measurements (five min window) indicating change of more than five watts.

Large measurement variability. For the load and work values in the dataset, we observe a large variability in the frequency with which the values change over time. Some plugs report close to constant load for long periods of time, and significant changes in load are correlated between plugs, following global energy consumption patterns. This effect is shown in Figure 1, which depicts the number of load measurements per five-minute window when ignoring changes of less than five watts. Considering only the events that indicate larger changes in load, the amount of events that need to be processed varies by up to a factor of seven over time.

Given these characteristics, we present a solution to the challenge that is based on the *SEEP stream processing platform* [7]. SEEP executes event processing queries as stateful distributed dataflow graphs. It supports the definition of stream processing operators as user-defined functions (UDFs). The main features of our solution using SEEP are as follows:

1. **Data-parallel processing.** To handle the high volume of events, our solution scales the processing of events in a data-parallel fashion on a cluster of nodes.
2. **Optimised stateful operators.** Given the complex state of the event queries, our solution exploits stream operators with efficient state handling specific to a given query, e.g. through indexed in-memory data structures.
3. **Filtering and elasticity.** We exploit the long periods of relatively constant load measurements in the dataset by performing semantic load-shedding, thus reducing the total events to process downstream. To support resource-efficient deployments when the input event rate varies over time, our solution can dynamically provision processing resources on-demand.
4. **Fault tolerance.** Our solution supports fault-tolerant processing, which is crucial for any continuously running data analytics application on a cluster of nodes. Instead of reprocessing all events after failure, operator state is recovered from periodic state checkpoints with low overhead.

Our experimental evaluation shows that our implementation processes the challenge dataset with a throughput of 300,000 events per second for the load forecasting and 100,000 events per second for the outlier detection, with median latencies of 17 ms and 136 ms, respectively. The resulting speed-up over real-time processing is 3× (load forecasting) and 2× (outlier detection).

Our solution also scales linearly in the number of cluster nodes used for computation. With 6 (load forecasting) and 7 (outlier detection) nodes, the speed-up over real-time increases up to 13× and 9×, respectively. Moreover, we show that semantic load-shedding leads to a modest median error in the query results, but increases the speed-up by two orders of magnitude. With this set-up, our solution processes one month worth of data for 40 houses in 17 mins.

The remainder of the paper is structured as follows. In Section 2, we give an overview of the SEEP stream processing platform used by our solution. Section 3 gives details on the implemented operators. Section 4 presents our evaluation results. We discuss related work in Section 5, before concluding in Section 6.

2. THE SEEP PLATFORM

Our solution follows the paradigm of **stateful stream processing** because it provides a natural way to implement the proposed queries: it permits the definition of custom state and its manipulation by UDFs. Event processing systems such as Esper [1] and SASE [3] provide high-level query languages and lack the possibility to fine tune the data structures used to maintain the query state. In contrast, stateful stream processing achieves efficient state handling for each specific scenario.

Recently, a new generation of data-parallel stream processing systems based on dataflow graphs have been proposed, including Twitter Storm [9] and Apache S4 [11]. Although these systems allow for massive parallelisation of stream processing operators, they assume that dataflow graphs are static and operators are stateless: they cannot react to varying input rates or efficiently recovery operator state after failure.

In contrast, the SEEP platform [7] implements a stateful stream processing model and can (i) dynamically partition operator state to scale out a query in order to increase processing throughput; and (ii) recover operator state after a failure, while maintaining deterministic execution. As a result, SEEP achieves the following three features:

(1) SEEP is **highly scalable** in order to handle high volume data. For the given challenge, this is an important feature because a realistic set-up for smart grid analytics would require the processing of events emitted by more than 40 houses.

Prior work [7] has shown that SEEP scales to close to a hundred virtual machines (VMs) in the Amazon EC2 public cloud. It achieves this through *data parallelism*—stateless and stateful operators are *scaled out*, i.e. multiple instances of operators are deployed in the cluster. Each instance operates on a subset of the event data. Events are dispatched to instances based on query semantics, e.g. in the challenge dataset the event streams may be hash-partitioned by houses. In addition, SEEP exploits *pipeline parallelism*—chains of operators are deployed on different nodes to reduce latency.

(2) SEEP is **fault tolerant**, which is critical when operator state depends on a large number of past events. The load forecasting in the challenge relies on a model learned from historic data, and outlier detection employs windows with up to 100,000,000 events. Even under the assumption of a reliable event source with access to the full event stream, losing operator state would require the reprocessing all events. Instead, SEEP creates periodic checkpoints of operator state, which are backed up to remote nodes and used to quickly recover state after failure.

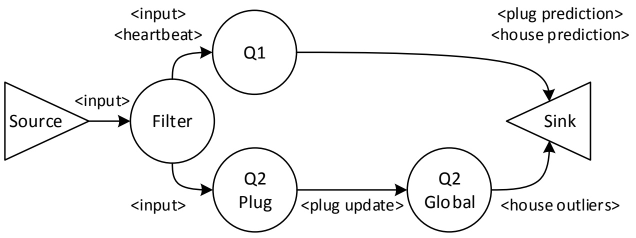

Figure 2: The dataflow graph of our solution.

(3) SEEP is **elastic**—it dynamically scales out stateful operators at runtime by partitioning their state. This functionality is particularly useful for event queries with high variability in the input rate. When pre-filtering events in the challenge to ignore minor changes in load, the input rate varies. SEEP can adapt to such workload changes, using cluster resource more efficiently.

3. QUERY IMPLEMENTATION

This section describes how we implemented the event queries from the challenge on the SEEP platform. We first give an overview of the main ideas behind the queries (Section 3.1), before we give details of the operator implementations (Sections 3.2–3.4).

3.1 Overview

The structure of the logical dataflow graph of the queries is shown in Figure 2. An operator *filter* performs semantic load-shedding across all load and work measurements (denoted by `<input>`). This permits, for example, filtering of events that indicate only a minor change in load for a certain plug. The *filter* operator can be scaled out so that different instances realise data parallelism by partitioning the event stream `<input>` per house, household or even plug.

The actual queries are implemented by three operators, namely *Q1*, *Q2 Plug*, and *Q2 Global*. Load forecasting and outlier detection are independent queries—their parallelisation is done separately.

Load forecasting is realised by operator *Q1*, and it is done at two levels of aggregation, i.e., plugs and houses. Hence, the operator can be scaled out by partitioning the respective event stream for the most coarse-grained aggregation, i.e., per house. Data-parallel processing is of particular importance for this operator because the query requires the maintenance of an unbounded time window.

At the same time, the query also requires frequent updates of the result stream, i.e. every 30 seconds as specified by the timestamps of the events. When events are streamed faster than real-time and distributed over a large number of operator instances, however, it becomes impossible to identify the intervals for updating the result stream at a particular instance. To solve this issue, we implement a heartbeat mechanism in the *filter* operator, which emits a signal to operator *Q1* whenever an update is due.

Heartbeat generation is implemented in operator *filter* because the pre-processing of data is less costly than the actual load prediction. Hence, the number of instances of operator *filter* can be expected to be much smaller than the number of instances of operator *Q1*. To cope with data quality issues such as missing values, e.g. as seen around days 20 and 28 in Figure 1, operator *Q1* also features a correction mechanism that is based on the measurements of cumulative work per plug, which will be detailed below.

Outlier detection is split up into two operators. Here, the idea is to separate the part of the query that can be parallelised from the part that requires a global state. Operator *Q2 Plug* thus takes the input stream and maintains the median of the load per plug for each of the time windows. The operator can be scaled out at the level of plugs.

Operator *Q2 Global*, in turn, maintains the global median over all plugs. It also realises the outlier detection and emits the results. Due to its global state, this operator cannot be scaled out. To reduce the amount of computation done at the singleton instance of operator *Q2 Global*, it relies not only on the median values computed by *Q2 Plug*, but also on the information about which measurements entered or left one of the investigated time windows (denoted by `<plug update>` in Figure 2). As a consequence, a large part of the effort to maintain the time windows per plug is performed at operator *Q2 Plug*, which can be scaled out.

3.2 Filter

The *filter* operator realises the following functionality:

Duplicate elimination. To filter duplicate measurements, the operator maintains the timestamps of the last load and work measurements for each plug. Only measurements with a timestamp larger than the last observed (per plug) are forwarded.

Variability-based filtering. To leverage the large variability in the frequency with which load values change over time for optimisation, the *filter* operator can perform semantic load-shedding, ignoring measurements that denote a minor change in load with respect to the last non-filtered measurement. In such a case, measurements of work are only forwarded if the load measurement with the same timestamp has not been removed by the filter procedure. In Section 4, we evaluate the trade-off between this type of filtering and the correctness of the query results with an experimental setup.

Heartbeat generation. The aforementioned heartbeats are generated based on the timestamps of the processed events. Whenever an event with a timestamp larger than the time of the last heartbeat plus the heartbeat interval is received, a new heartbeat is emitted.

3.3 Query 1: Load Forecasting

Operator *Q1* for load forecasting is implemented as follows:

Prediction model. As a baseline, we rely on the prediction model defined in the challenge description, which combines current load measurements with a model over historical data. More specifically, the load prediction for the time window following the next one is based on the average load of the current window and the median of the average loads of windows covering the same time of all past days. The generation of prediction values is triggered by the heartbeats, which are generated by the *filter* operator.

Work-based correction. To address the issues stemming from missing load measurements, our operator exploits measurements of cumulative work per plug. Correction is triggered when the operator receives a work measurement, and the number of recorded load measurements for the preceding window is less than a threshold.

Since work is measured at a coarse resolution (1 kWh), the work values enable us to derive only an approximation of the actual average load. Therefore, the threshold on the number of load measurements allows for tuning how many load values are at least required to avoid the computation of the window average based on work values. A specific value for the threshold is chosen based on the expected rate for load measurements.

If applied, the correction mechanism determines the maximal interval of adjacent windows with insufficient load measurements. The difference between the first and last work measurement for this period is used to conclude on the average load for all the windows.

State handling. Load forecasting relies on the average load per window per plug over the complete history. To cope with the unbounded state of the query, our implementation strives for reducing the size of the state as much as possible.

First, we observe that although results have to be provided for five different window sizes, all of them can be expressed as multiples of the smallest window of one minute. Therefore, our implementation only stores the state for the smallest windows.

Second, since prediction is based on the load average, our operator keeps only a sliding average for the current smallest window and the average load for all historic windows. Load averages are kept in a two-dimensional array (per plug, per window), and an index structure allows for quick access of a global identifier for a plug. The index is implemented as a three-dimensional array over the house, household, and plug identifiers.

For the work-based correction mechanism, additional state needs to be maintained. For each plug and window, the number of load measurements and the first recorded work value is maintained in further two-dimensional arrays.

3.4 Query 2: Outliers

Outlier detection is realised by operators *Q2 Plug* and *Q2 Global*. The former focuses on the calculation of windows and the median load per plug. Its results are then used by operator *Q2 Global* to conduct the actual outlier detection.

Plug windows and median. To maintain the time windows and calculate the median load per plug, operator *Q2 Plug* proceeds as follows. Upon the arrival of load measurement, the value and timestamp is added to either window for the respective plug. The timestamp of the received event is used to remove old events from both windows. Then, the median of the load values for the plug is calculated. If both, the median and the multiset of values of both windows, did not change, no event is forwarded to operator *Q2 Global*. If there has been a change, the new median for the plug as well as the load values added or removed to either window are sent to *Q2 Global* (`<plug update>`).

Outlier detection. To detect outliers, operator *Q2 Global* compares the median values per plug as computed by operator *Q2 Plug* with the global median. To compute the latter, the operator maintains two time windows over all plugs. However, these windows are updated only based on the values provided by the events of the `<plug update>` stream generated by operator *Q2 Plug*.

Receiving an event of the `<plug update>` stream leads to recalculation of the global median for the respective window. If that has not changed, only the house related to the plug for which the update has been received is considered in the outlier detection. If the global median changed, the plugs of all houses are checked. If the percentage of plugs with a median load higher than the global median changes, the result stream is updated.

State handling. To implement the time windows, for each plug, operator *Q2 Plug* maintains two double-ended queues, one containing the timestamps and one containing the load values. Implemented as linked lists, these queues allow to insert new measurements in constant time. Accessing and removing events from the other end of the queue is done in constant time. The queue containing the timestamps is used to determine whether elements of the queue containing the load values should be removed.

To compute the median over the load values, operator *Q2 Plug* maintains an indexable skip-list [12] of these values per plug. Such a skip-list holds an ordered sequence of elements and maintains an additional index structure, a linked hierarchy of sub-sequences that skip certain elements of the original list. We use the probabilistic and indexable version of this data structure—the skip paths are randomly chosen and, for each skip path, we also store the length in terms of the number of skipped elements.

The indexable skip-list allows for inserting, deleting and searching load values as well as accessing the load value at a particular list index in logarithmic time. Calculation of the median is traced back to a list lookup. Since the query requires the lookup only for the median element, and not for an arbitrary index, we also keep a pointer to the current median element of the list, which is updated with every insertion or deletion. Hence, the median is derived in constant time.

Although bounded, handling the state of operator *Q2 Global* is challenging due to the sheer number of measurements that need to be kept (up to 100,000,000 events) and the update frequency. For both windows, our implementation relies on an indexable skip-list and uses a pointer to the median elements of these lists.

4. EVALUATION

Our evaluation assess the performance of our system along three dimensions by investigating:

- whether it scales. Does the system support more houses?
- whether it can cope with the current load with headroom. Can the system process faster than real time?
- how fast we can incorporate predictions. Does the system achieve low latency, even when it is distributed?

We deploy our solution in a private cluster composed by 10 Intel(R) Xeon(R) CPU E3-1220 V2 @ 3.10GHz nodes with 8 GB of RAM, running SEEP on a Linux kernel 3.2.0 with Java 7. We run SEEP with the fault tolerant mechanism enabled.

Scalability. To measure scalability we report *relative throughput*, where we normalize the throughput of the system for the baseline case, and show how it increases as we add cluster nodes. Also, we explain the bottlenecks observed when conducting the experiments.

Throughput. After analysing the available datasets, we found that we need a system capable of processing 377 events/s, 696 events/s and 1565 events/s on average for the 10, 20 and 40 houses dataset to process the incoming input rate over a month. SEEP processes three orders of magnitude faster than this. For this reason, we report *speedup over RT (real time)* as the number of times the system process faster than required to run the queries. As an example, consider a speedup over RT of 2×, which would allow for processing one month worth of data in 15 days.

Latency. We measure the end to end latency of those events that close windows in both queries. To measure latency accurately we place both source and sink of our system on the same cluster node, so that both operators have access to a common clock.

For the given event queries, the processing cost per event is close to constant regardless the dataset size. Dataset sizes, however, have an impact on the total memory required to run the queries. We exploit the stateful capabilities of our system to provide an efficient implementation which expresses the state efficiently, avoiding any kind of performance hit. Note that under this scenario, larger datasets do not impact the throughput of our system, but only the speedup, as there are more events to process.

4.1 Query 1: Load Forecasting

Our implementation of query 1 consists of two operators, a *filter* and *Q1* (see Figure 2). For the baseline system, each of the operators is deployed on a single node of the cluster. For our distributed deployment we scale out from 2 to 6 nodes.

Baseline system. Figure 3 shows the 10th, 50th, and 90th percentile of throughput as requested in the challenge description. As expected, this is constant across the different workload sizes but the speedup over RT decreases as there are simply more events to process. With a speedup over RT of around 9×, the system can process one month worth of data from 10 houses in about one hour, while it will take around four hours to do the same for 40 houses.

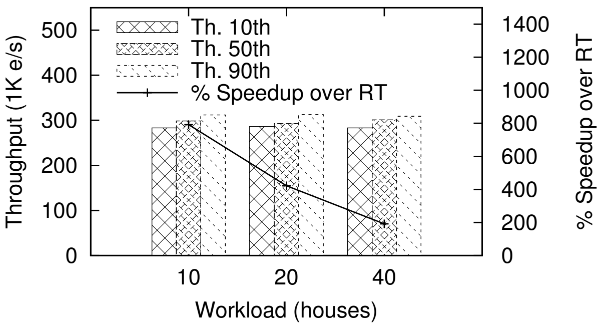

Figure 3: Throughput and speedup as a function of the number of houses. With a constant throughput, growing the size of the dataset implies less speedup.

	Q1	Dist. Q1	Q2	Dist. Q2
10th	4	3	118	40
50th	17	12	136	150
90th	31	21	160	186

Table 1: Latencies in ms for both queries with baseline and distributed deployment.

Distributed system. Ideally, we want the system to scale to support the data coming from more houses, which in our system is equivalent to keep the speedup over RT constant. We exploit data parallelism to aggregate throughput, thus, keeping constant or even increasing the speedup over RT. Figure 4 shows in the *x* axis the number of cluster nodes used during the experiment. The relative throughput increases linearly from 2 to 3 nodes, then sub-linearly until 5, and then we find a spike when using 6 nodes. The reason for the sub-linear behaviour is due to the sink operator, which has to aggregate the results coming from distributed nodes, becoming an IO bottleneck. To confirm this was the case, we scaled out the sink and run the system with 6 nodes, which shows how the throughput increases again. The speedup over RT across this experiment is always increasing, which confirms that our system can scale to bigger datasets while keeping the throughput. We stopped at 6 nodes as the source became a bottleneck at this point. This would not be an issue in a real scenario with distributed sources, but we did not want to break the order semantics provided in the dataset.

Table 1 shows the latencies we measured for both the baseline system and the distributed one. The major sources of latency spikes in SEEP are caused by the buffering mechanism used for fault tolerance, and the intricacies of this with the garbage collector under high memory utilisation scenarios. Neither of these happen in the context of query 1. Our latencies are slightly lower than in the non-scaled out case. The reason for lower latencies in the distributed case is the source cannot insert data at higher rates, and thus, events go through the same number of queues and processing elements as in the non-scaled case, but with more headroom.

4.2 Query 2: Outliers

Our implementation of query 2 consists of three operators, *filter*, *Q2 Plug* and *Q2 Global* (see Figure 2). Hence, the baseline deployment comprises 3 nodes of the cluster. For the distributed deployment we scale out from 3 to 7 nodes.

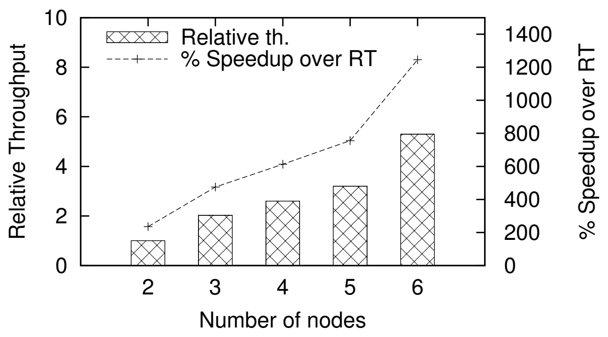

Figure 4: Throughput and speedup over RT as a function of the number of machines. We increase the speedup by scaling out the system to aggregate throughput.

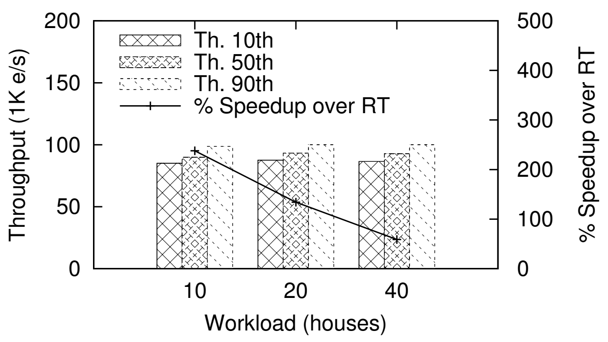

Figure 5: Throughput and speedup as a function of the number of houses, for query 2.

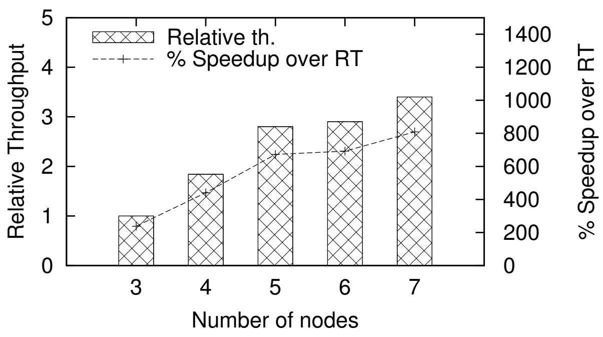

Figure 6: Distributed deployment of query 2. Scaling out the query aggregates throughput and increases the speedup

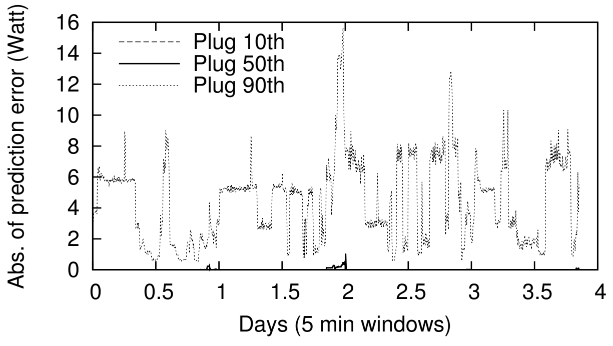

Figure 7: Time series over four days showing a low prediction error for plugs over five minute windows. Note that the 50th overlaps the x axis.

Baseline system.. Figure 6 shows the expected behaviour of decreasing speedup as the dataset grows in events size. This query is computationally more expensive than query 1. In our solution this translates into a total time of 3.2 hours to process one month of data for 10 houses to 13 hours to do the same in the 40 houses case.

Distributed system.. We follow the same strategy of scaling out the system to increase the speedup over the minimum throughput required by the system, reported in Figure 6. When adding more cluster nodes the throughput increases, except between 5 and 6 nodes. The reason for this behaviour is that there were two simultaneous bottlenecks. First a CPU bottleneck that disappears after scaling from 5 to 6 nodes, giving rise to an IO bottleneck. When scaling out the IO bottleneck, we show how the system can keep up increasing the speedup. We stop our query at this point, when the source becomes a bottleneck.

Latencies for query 2 are reported in Table 1. Latencies are higher than in query 1 because the bottleneck in this query was CPU related while in query 1 was IO (serialisation and deserialisation).

4.3 Impact of Semantic Load-Shedding

To investigate the inherent trade-off of result accuracy and computation efficiency implied by semantic load-shedding, we compared the load predictions derived by query one for a sample of four days. We focus on the predictions derived for the smallest time window (one minute). This window represents the most challenging case, since for larger windows, the relative importance of filtered events is smaller and, thus, accuracy is less compromised.

We show in Figures 7 and 8 the absolute error prediction error for plugs and houses, respectively, aggregated for windows of five minutes. For individual plugs, although the 90th percentile shows spikes up to 15 watt, the median error is zero in virtually all cases. For load predictions for houses, in turn, the median error is largely between one and three watts and there is little variability in the results. Based on these results, we conclude that the error is small enough to justify the activation of the mechanism.

Turning to the benefits of load-shedding for processing performance, Figure 9 shows the difference in throughput and speedup over RT when enabling the mechanism. As discussed before, the throughput per node is mostly not affected since processing cost per event is close to constant. However, we observe an improvement of speedup of two orders of magnitude, meaning that one month worth of data for 40 houses is processed in 17 minutes. This drastic speedup together with the low accuracy loss justifies the usage of semantic load-shedding in this scenario. We consider this an important outcome as it allows more headroom to scale out the system to accommodate the load from more houses.

5. RELATED WORK

Our solution is based on the use of distributed stream processing. Various engines for stream processing have been proposed in the literature, such as Discretized Streams [14], Naiad [10], Twitter Storm [9], Apache S4 [11], MOA [5], Apache Kafka [8], and Streams [6]. Discretized Streams support massive parallelisation and state in the form of RDDs (Resilient Distributed Datasets), however, its approach based on micro-batching is not optimal to reduce processing latency. Naiad [10] can scale out across many nodes maintaining low latencies, however the system is not designed to

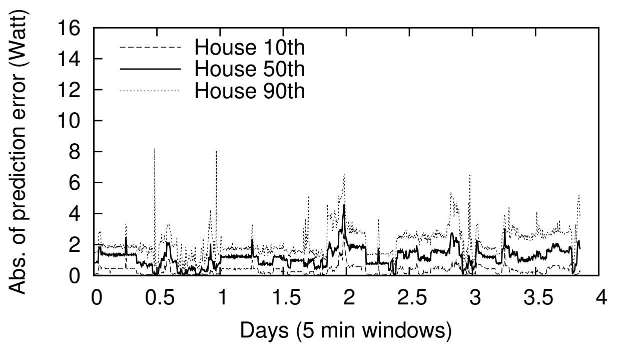

Figure 8: Time series over four days showing a low prediction error for houses over five minute windows.

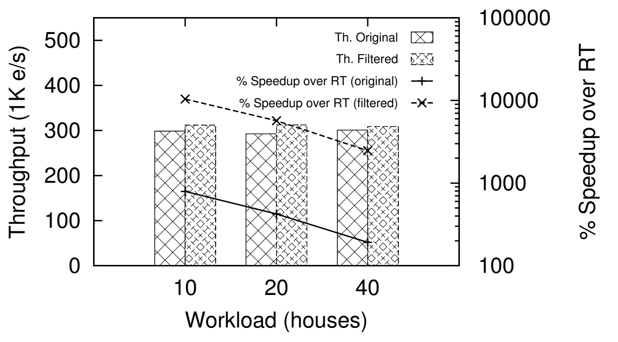

Figure 9: When applying semantic load-shedding, the speedup grows two orders of magnitude (note the logarithmic scale). The system process one month worth of data for 40 houses in 17 minutes.

be fault tolerant when the managed state is large. The other systems support massive parallelisation of stream processing, but lack support for managing stateful operators and for reacting to varying input rates. Both issues are of particular relevance for the grand challenge—the queries feature a large and complex state and the number of events indicating changes in load shows large variability. Therefore, we grounded our solution on SEEP [7], which supports stateful operators, dynamic scale-out, and fault tolerant processing.

To improve the processing performance, we apply semantic load-shedding, dropping input events in a structured way to achieve timely processing. Load-shedding is common technique for optimising event processing applications, in particular for achieving high throughput [13, 4]. In some cases, load shedding for distributed stream processing may in itself become an optimization problem [13]. Our approach exploits domain semantics, i.e., the size of a change of a measurement value, to decide on which events to filter. Our evaluation showed that this approach results in only minor inaccuracies in the load forecast. However, filtering leads to drastic performance improvements. The speedup realised by the system grows by two orders of magnitude.

6. CONCLUSIONS

In this work, we presented a highly scalable solution to the ACM DEBS Grand Challenge 2014. We based our solution on SEEP, a platform for stateful stream processing that supports dynamic scale out of operators and recovery of operator state after a failure. To achieve efficient processing, we presented implementations of the stream processing operators that are geared towards parallelisation and effective state management, e.g., using queues and skiplists. In addition, we exploited the fact that there are long time periods over which measurement values are relatively constant. Further details on our solution including a screencast are available at [2].

The experimental evaluation of our solution indicates that the system can indeed cope with high volume data, processing it with 300,000 events per second for the load forecasting and 100,000 events per second for the outlier detection and median latencies of 17 ms or 136 ms, respectively. We also conclude that the system is capable to handle larger workloads since it scales linearly when adding cluster nodes. Further, we demonstrated that semantic load-shedding can lead to huge performance gains. While filtering leads to approximate query results only, in our case, the resulting bias was rather modest, with a low median error for the prediction. However, the speedup over real time was increased by two orders of magnitude, which allowed us to process the whole dataset in 17 minutes.

Our approach to filter events for more efficient stream processing opens several directions for future work. On the one hand, the generation of filter conditions from a set of queries would allow for automating the approach. On the other hand, filtering may be guided by the amount of tolerated inaccuracies in the query results. Probing unfiltered results for certain time intervals would enable to assess the consequences of filtering and adapt its selectivity dynamically.

7. REFERENCES

[1] `http://esper.codehaus.org/`.

[2] `http://www.doc.ic.ac.uk/~mweidlic/gc14/`.

[3] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *SIGMOD*, pages 147–160. ACM, 2008.

[4] B. Babcock, M. Datar, and R. Motwani. Load shedding in data stream systems. In *Data Streams*, volume 31 of *Advances in Database Systems*, pages 127–147. Springer, 2007.

[5] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer. Moa: Massive online analysis. *The Journal of Machine Learning Research*, 99:1601–1604, 2010.

[6] C. Bockermann and H. Blom. The streams framework. Technical Report 5, TU Dortmund University, 2012.

[7] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD*, pages 725–736. ACM, 2013.

[8] J. Kreps, N. Narkhede, and J. Rao. Kafka: A distributed messaging system for log processing. In *NetDB*, 2011.

[9] N. Marz. Storm - distributed and fault-tolerant realtime computation, 2013.

[10] D. G. Murray, F. McSherry, et al. Naiad: A Timely Dataflow System. In *SOSP*, 2013.

[11] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW)*, pages 170–177. IEEE, 2010.

[12] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[13] N. Tatbul, U. Çetintemel, and S. Zdonik. Staying fit: Efficient load shedding techniques for distributed stream processing. In *VLDB*, pages 159–170. VLDB Endowment, 2007.

[14] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *SOSP*, 2013.

[15] H. Ziekow and Z. Jerzak. The DEBS 2014 Grand Challenge. In DEBS, Mubai, India, July 2014. ACM.