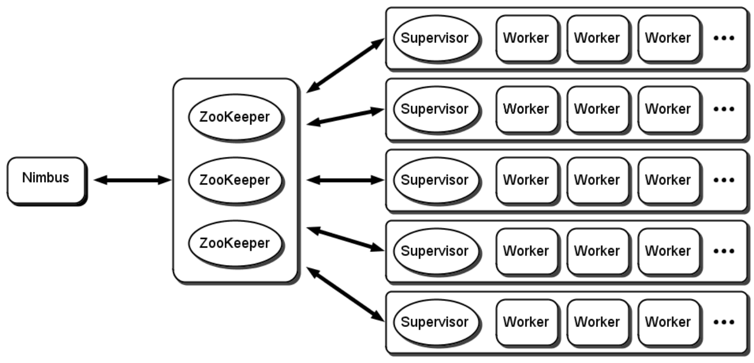

# Streaming-based data validation in Advanced Metering Infrastructures

*Master's Thesis in Computer Science - Algorithms, Languages and Logic*

JOHAN SWETZÉN

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, October 17, 2015

Streaming-based data validation in Advanced Metering Infrastructures

JOHAN SWETZÉN

©JOHAN SWETZÉN, October 17, 2015

Examiner: MARINA PAPATRIANTAFILOU

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone +46(0)31-772 1000

Cover: Structure of a storm cluster

Department of Computer Science and Engineering
Göteborg, Sweden October 17, 2015

## Abstract

In recent years, the development of smart grids has moved forward both technologically and politically. A law (2012:510) introduced in Sweden in 2012, giving consumers the right to have their electricity charged by the hour, brought demands for a more data-intense metering infrastructure. The automatic measuring, often involving wireless communication, introduces errors both in software and during data transmission. Combined with increased data volumes that drive a further increase in errors, the new demands present a challenge to utilities. Measurement errors cannot be allowed to propagate to the data stored by utilities and, in the end, electricity bills. To handle these errors as well as enabling future real-time services, there is a need for validation of very large quantities of metering data with low-latency guarantees.

The systems responsible for such validation need to handle the unbounded streams of data generated by an increasing number of meters as well as more fine-grained energy consumption readings in the future. Another important aspect is flexibility, as these systems need to account for both current and future errors in the data, adapting to changes in the Advanced Metering Infrastructure (AMI).

This thesis aims to address these issues using the data stream processing paradigm, in which complex analysis is performed on unbounded sequences of data. Since this fits the AMI data well, validation rules are built upon this paradigm and then implemented on an actual stream processing engine to assess performance. Furthermore, patterns of common errors are identified, triggering alerts as a first step towards automatic correction of errors. The prototype system is developed in close cooperation with Swedish utility company Göteborg Energi. Data from their current AMI is used both to identify the commonly occurring errors and assess the performance of the system.

In evaluating the prototype system, the results showed a processing capacity of 1500 measurements per second at a latency of no more than a few seconds. This corresponds to 5 million smart meters reporting hourly readings or 1.35 million meters operating at 15 minute intervals.

Investigation of error frequency based on the validation rules found the single most common meter fault at Göteborg Energi. This error affects one particular meter model and accounts for 46% of all errors encountered in the AMI. The prototype system developed in this thesis is able to identify 61% of the occurrences of this this error in the examined data set. These 61% conform to a specific pattern that clearly marks out this particular error.

This thesis shows that stream processing can be used to validate large volumes of electricity meter data online with low processing latency, identifying common errors as they appear.

## Acknowledgements

I want to thank my supervisor at Göteborg Energi, Joris van Rooij, for patiently explaining how the systems at the utility work and coming with input and feedback on my work. I also want to thank Vincenzo Gulisano for all the advice and support on data streaming and academic writing.

Johan Swetzén, Göteborg October 17, 2015

# Contents

# List of abbreviations

| | |
|---|---|
| **AMI** | Advanced Metering Infrastructure |
| **SPE** | Stream Processing Engine |
| **SPE cluster** | An SPE that is operating using a set of networked computers to distribute the computational load. |
| **DAG** | Directed Acyclic Graph |
| **CSV** | Comma-Separated Values |

# 1

# Introduction

The development of an enhanced electricity grid has moved forward both technologically and politically in recent years. One driving force is an increased measurement rate, as a number of countries introduce mandatory hourly billing. When consumers are given the option to pay for their electricity based on the demand at that particular time of day, they gain a higher degree of control over their energy bills. At the same time, utilities are presented with both opportunities and challenges. Shorter measurement intervals open up new possibilities in load forecasting with the potential of tailoring electricity production to the consumption.

The work on fulfilling these new requirements has lead to the construction of Advanced Metering Infrastructures (AMIs): networks of electricity meters, communication terminals and aggregators that gather data needed by the utility. However, not only the metering infrastructure is affected, but the receiving end also needs to be adjusted to accept roughly 700 times the data input compared to monthly readings. There is even talk within the EU about regulating measurements to every 15 minutes, resulting in 2,800 times the data. Such a sampling period has been evaluated in a research setting and can improve load forecasting accuracy compared to hourly readings [8]. It is therefore reasonable to expect further increases in data rates, meaning that the challenges present today are only going to grow.

One of these challenges pertains to validation of the incoming data. For many different reasons, data collected by AMIs is known to be error prone, far from the clean and complete measurement series required for load forecasting or billing. Measurements can arrive out-of-order, duplicated or not at all, sometimes owing to communication losses and re-transmissions particularly common in wireless infrastructures. Losses can also be a symptom of time synchronisation problems as measurements are assigned to incorrect times. Erroneous measurements can be caused by meters that are broken, meters that are known to fail in certain circumstances or even by malicious users looking to decrease their electricity costs by unlawful means.

Owing to the rates at which the measurement data is collected, a validation system needs to be able to process large amounts of data with ease. Potentially millions of measurement values are to be analysed and validated every hour. The data stream processing paradigm emerged in the early 2000s as an alternative to traditional databases [4] when it comes to processing large amounts of data in real-time. The high data rates makes data streaming a good computing paradigm to apply to this problem.

## 1.1 Problem statement

To address the challenges posed by data validation in AMIs, the aim of this thesis is to construct a validation system based on stream processing. The system should be able to handle large amounts of data, of the order of a few million meter readings per hour. The performance of the system is to be evaluated in terms of latency measured in seconds as well as throughput in tuples per second. Scalability should be achievable through the addition of hardware, i.e. the SPE (Stream Processing Engine) should be able to run on multiple servers to increase processing power.

The purpose of the system is to check for a variety of errors. Some of these errors are currently known but will become obsolete and others will emerge in the future. Once an error has been identified, it would be valuable to also automate the process of error correction as much as possible.

There are several factors that complicate the construction of such a system. One difficulty is that the measurements can be received out-of-order. Some values are delayed, causing gaps in the data. At Göteborg Energi, such a gap may take more than 48 hours to fill. This is complicated further by problems in synchronisation of clocks, so that distinguishing between missing and out-of-order data is not trivial. Handling an unbounded interval of tuples is not feasible due to memory constraints, so it is necessary to limit the time interval considered. Still, it is important to discard as few measurement values as possible.

Finding the errors is not always straightforward, but presents some difficulties as well. An obvious error is a decreasing value of the meter stand, that is the number of kWh that have been measured by the meter since it was installed or the consumption was last reset. A decreasing meter stand value would correspond to a negative energy consumption (not to be confused with a positive energy production), which is not possible. The opposite case, a measurement value that is too high, is not as simple to identify. Using the value of the fuse installed, an indisputable upper value for the consumption can be computed. However, only part of the errors can be caught in this way. It might be more reasonable to consider values that lie significantly above the average. While this approach has the possibility of catching more errors, it also runs the risk of flagging values that are not wrong but simply unusual.

These two basic errors are often a part, or symptom, of another error that needs to be identified. Values that are extremely high can for example cause a decreasing value that corrects for the previous errors. Hence, it is important to look at the big picture and factors as high values, decreasing value and even the model of smart meter when trying to identify the most common errors. Weighing in all these factors makes the problem significantly more complex.

## 1.2 Motivation

This thesis is written in collaboration with Göteborg Energi, a Swedish utility operating in and around Göteborg. They took the initiative for this thesis because they are facing the challenges presented here in their daily operation. The almost 300,000 smart meters that are installed in their advanced metering infrastructure range over several different manufacturers and models. Some meters are damaged by lightning strikes and other natural phenomena, causing them to exhibit strange behaviours. Other meters succumb to aging so that broken components introduce measurement or calculation errors. Not only do the meters cause problem, but bugs in software updates to other parts of the infrastructure can be just as damaging. Although they strive to correct problems permanently, some issues are out of their hands and need to be dealt with on a regular basis.

The problems mentioned above result in human resources wasted on tasks that are better suited for a computer to handle automatically. Göteborg Energi has some rudimentary error checking that identifies errors in up to 100 meters every day. This information then needs to be inspected and corrected manually, which means that a few hours every morning are spent performing a very repetitive task. In addition, this simple kind of validation can only discover a subset of the errors that occur. An automated data streaming approach could not only reduce the workload but also perform more advanced validation, thus improving the quality of the data.

## 1.3 Method

Before the implementation of any validation rules, the data streaming paradigm is researched to establish how it can be applied to the problem. The stream processing engine Apache Storm is used, so the specifics of data stream processing using that system is studied as well.

With the background knowledge in place, electricity consumption data from Göteborg Energi can be received and read into Storm. At this point, the data can be analysed for the simplest error: decreasing consumption values. Moving beyond this first error case, experts at Göteborg Energi are consulted to find the errors that commonly occur in their system.

Storm is written in Java, but the programs it runs can be written in other languages as well. Based on the knowledge gained in communication with system experts, a validation system based on the framework Pyleus is developed using Python. Pyleus also has functionality that allows the finished prototype system to be modified without much programming experience.

Finally, the system is evaluated, looking both at the performance in terms of throughput and latency, as well as the degree to which it is able to identify and correct errors.

A simple comparison with the current system in use at Göteborg Energi is used as one metric to assess the results.

## 1.4 Scope

The resulting validation system is to be seen as an initial prototype. First and foremost, alerts should be generated based on the errors discovered. These alerts could then be integrated into existing systems, or the prototype could be developed further to correct the errors observed. A simple text file output mode is sufficient for this kind of operation, graphical tools could be developed later on.

Measurements may not be analysed if they are too much out-of-order. Unbounded data is far too complicated to consider, so bounds need to be introduced to make the data predictable. This will cause some measurement that take a very long time to arrive to be discarded.

# 2

# Technical Background

This chapter presents some of the preliminary technical concepts that are necessary for understanding the rest of the thesis. Stream processing is described in detail along with the specific approach and terminology of Apache Storm. Finally, a section is dedicated to out-of-order data, a common challenge in stream processing.

## 2.1 Stream Processing

This section starts by presenting a historical overview of the development of Stream Processing Engines (SPEs) and then goes on to describe the underlying concepts more in detail.

Since the early 2000s, the data streaming paradigm has emerged as an alternative to traditional databases when it comes to processing large amounts of data in real-time. Among the many applications are financial trading and market analysis, as well as network intrusion detection and monitoring of denial of service attacks. Traditional databases are not designed to support this kind of streaming data and in response to these new challenges, Aurora was developed as one of the first SPEs [4]. Aurora was designed to run on a single machine without the possibility of distributing the work. Building upon the functionality of Aurora, among other works, Borealis, an SPE that can scale to several machines, was developed [1]. More recently, systems like StreamCloud [9, 10, 11] have been presented for improved scalability. Others have been released as open source software, like Apache Storm [30] and S4 [24].

For an SPE to function well, there are a few requirements that it should fulfil [29]. Looking at a selection of those requirements gives an insight into the characteristics of an SPE. First of all, keep the data moving. This stands in stark contrast to traditional database models where storing data is central. The most high-performing data streaming systems seldom need to store data, but instead process data messages as they fly by.

Another requirement is to handle stream imperfections, like delayed, missing or out-of-order data. It might be reasonable to wait for some time for a missing message if it only arrives a little late. On the other hand, a computation can never be allowed to block indefinitely but instead time out if the message is actually missing. A third rule is to guarantee data safety and availability. Data must not be lost, despite errors or crashes. In the same way, the SPE should be available at all times and recover from crashes quickly and automatically. Finally, an SPE should process and respond instantaneously. This means a high processing throughput with very low latency, in the range of milliseconds. If the system cannot actually keep up with the incoming data, the whole point of performing continuous computations with stream processing is lost.

The operation of an SPE relies on viewing data as a flow of tuples from beginning to end. The incoming data therefore consists of an unbounded stream of tuples. To extract meaningful information from these tuples, continuous queries in the form of Directed Acyclic Graphs, DAGs, are formed. Edges in the DAG correspond to the flow of data between the operations, represented by vertices. Operations can be for example filters that drop tuples that do not fulfil certain criteria, mappers that transform tuples or union operations that merge several streams into one. All these are examples of stateless operations, they consider only a single tuple when producing a result. Another type of operators are the stateful ones, for example aggregate which performs a computation over several tuples in a single stream and join which combines tuples from several streams. A common way of performing aggregate operations is by using a sliding window, or simply window. To handle the unbounded stream of tuples, only the latest tuples are considered. For example, a time-based window can contain all tuples from the last 15 minutes and a tuple-based window can contain the 10 latest tuples.

The results of a continuous query can be alarms that are triggered or a modified data stream saved to a dedicated database. Another possibility is to keep the results in memory so that they can be queried when needed. An example of a continuous query that finds the highest measured power consumption in a distributed manner can be found in figure 2.1.

## 2.2 Apache Storm

The Storm SPE was first developed at BackType, which was acquired by Twitter in 2011. Twitter continued development and open-sourced it in 2012, this is when it became Apache Storm. This section presents the terminology specific to Storm as well as details on running and scaling a Storm cluster.

### 2.2.1 Topologies, spouts and bolts

A continuous query in the form of a directed graph is referred to as a topology within Storm and it consists of spouts and bolts. Tuples flowing through a topology originate at spouts, vertices with one or more outgoing edges, but no incoming ones. Tuples may arrive at bolts for processing, these are vertices with incoming edges and optional

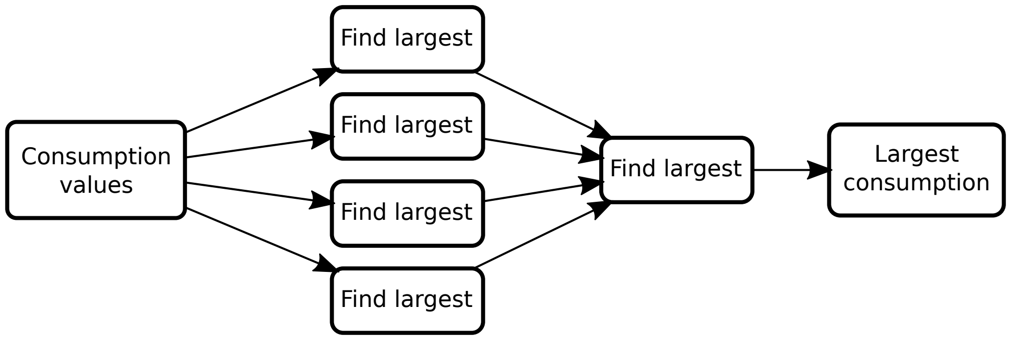

**Figure 2.1:** A topology that finds the highest consumption value, that is the highest number of kWh drawn for an hour. The first group of "Find largest" nodes are running in parallel, possibly on four different machines. The last "Find largest" needs to be a single process to summarise the results.

outgoing edges.

### 2.2.2 Directing the flow of data

A spout or bolt can run on any number of machines in parallel, which calls for a way of deciding what instance of a bolt that handles a certain tuple. For this purpose, a number of partitioning strategies are included in Storm that define how the tuples of a stream are grouped together.

1. **Shuffle** grouping distributes the tuples randomly across the available nodes while still guaranteeing that no bolt processes more tuples than any other.
2. **Fields** grouping partitions the stream by a specific set of fields. For example, a field grouping on "city" ensures that all tuples with the same city value will be processed by the same bolt task.
3. **All** grouping distributes every tuple in the stream to every consumer, thus duplicating it if there are more than one tasks executing the bolt.
4. **Global** grouping sends all tuples of a stream to a single bolt task.
5. **Local or shuffle** grouping shuffles tuples to tasks within the worker process, if possible. The ordinary shuffle grouping is employed in case there is no task for the bolt running in the local process.
6. **None** grouping lets Storm decide how to partition the stream. It is equivalent to shuffle grouping at the time of writing, but it leaves room for future optimisation.

Figure 2.3 can be used to illustrate the most common of the partitioning strategies. Tuples entering the split words bolt would use the shuffle grouping, since it does not matter which task splits a line. Between the split words and count words bolts however,

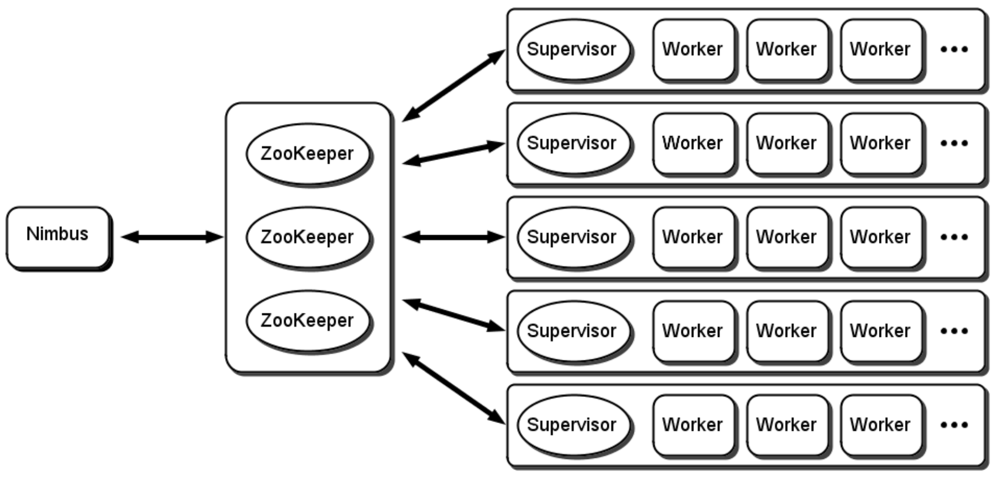

**Figure 2.2:** The anatomy of a Storm cluster. Nimbus, ZooKeeper cluster and worker machines with a Supervisor

a fields grouping on the word is appropriate. This ensures that a particular word always gets sent to the same word bolt task for counting. Finally, the results is logged to a single place, so a global grouping fits well.

### 2.2.3 Anatomy of a Storm cluster

Storm is built so that it is easy to scale it to many machines, while remaining robust and resilient to failure. Locally, a process may stop and it is restarted. Globally, a machine can crash and the task and data that it was working on will be reassigned to another machine. Storm gives the strong guarantee of at-least-once processing of every tuple, so data is never lost in the system.

The high scalability and robustness of Storm is supported by the different components that make up a Storm cluster. Redundancy is supported at whatever level appropriate for any particular use case. As can be seen in figure 2.2, there are three main components that make up a cluster, namely Nimbus, ZooKeeper and Supervisor. Nimbus is the master node, where topologies are submitted. This is run on a single process, so there is no more than one point at which interaction with the cluster can happen. Although this may seem contrary to the reliability features of Storm at first glance, it is no single point of failure. Nimbus is fail-fast and stateless, so all their data is stored either on the local disk(s) or in ZooKeeper. If Nimbus is down for a while, this merely hinders the ability to submit new topologies.

The Storm topologies are stored in ZooKeeper, commonly set up in a cluster with several machines. All communication between Nimbus and the worker machines is done through ZooKeeper, and once a topology is submitted through Nimbus to ZooKeeper, it will keep it running without the involvement of Nimbus. Only in case of machine failure will Nimbus need to redistribute a topology onto new worker machines.

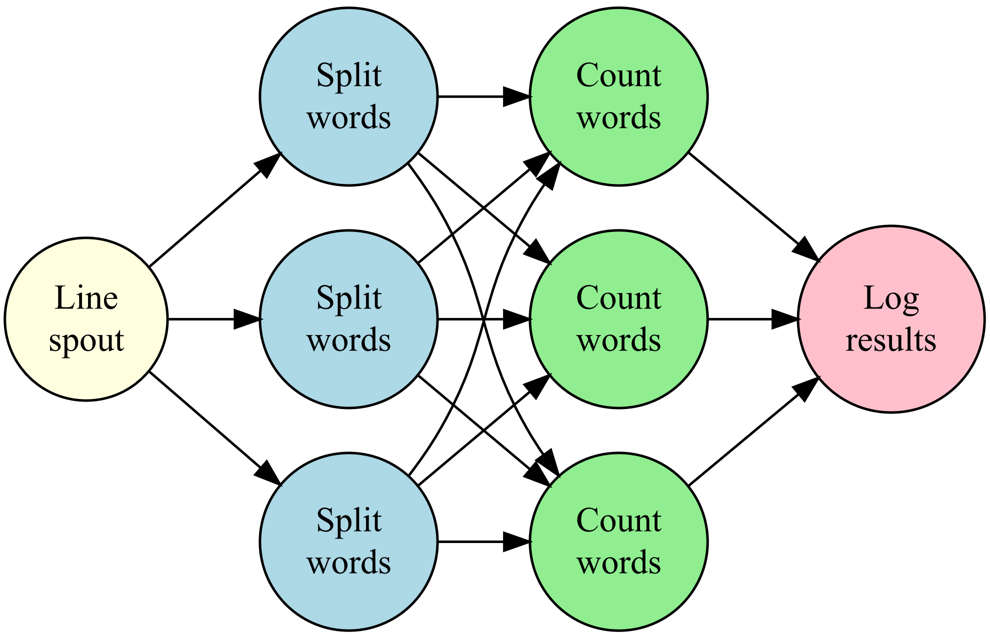

**Figure 2.3:** A topology that counts words in lines. The "split words" and "count words" bolts are running with three worker tasks, parallel bolts, each, potentially distributed on several machines.

On each of the worker machines, where the actual computations are performed, the main process is the supervisor. Like Nimbus, it is fail-fast and stateless and runs under supervision so that it is restarted whenever it fails. The supervisor starts one or more worker processes, which in turn run one or more executors. These executors may run several worker tasks, i.e. spouts or bolts. The supervisor takes care of restarting worker processes in case of failure, but if the supervisor itself should crash, the workers are not affected.

The design of Storm allows for the number of worker machines to scale with the ZooKeeper cluster and handle a very large numbers of machines. For example, Twitter is running Storm clusters of several hundred machines [31].

## 2.3 Out-of-order data in AMIs

A problem that arises in almost every stream processing setting is that of out-of-order data. AMIs are no exception, and this problem and its causes are presented in this section.

### 2.3.1 Problem

To keep the processing model simple, it is very convenient to assume that the input stream is ordered. For stateful operations, such as aggregation over a sliding window, this allows for a simple implementation using only a buffer of size $N$, the same as the window size, in the case of a tuple-based window. For a time-based window, a variable-length buffer is sufficient. When the stream is not ordered however, these kinds of implementations cause loss of data that arrives too late.

### 2.3.2 Causes of out-of-order data

Although it is common to model a data stream as an ordered sequence of tuples, real-world stream systems are affected by naturally occurring disorder [19]. Here are some examples:

1. Tuples that have been sent over a network may have taken different paths, consequently they can have different delays.
2. A stream can be a combination of sub-streams originating from different processing nodes. Differences in processing or transmission delays for the nodes can cause disorder in the merged stream.
3. The required ordering may be different from the tuple generation order. For example, network packet records may be generated so that they are ordered on receive time when it is desirable to process the records in the order they were transmitted.
4. The stream processing system itself may introduce disorder in some operations.

The impact of some of these causes for disorder can be minimised, one can for example limit the operations used within the stream processing system to those that retain order. Other issues, like network latency, can never be fully controlled.

In AMIs, network latency is one of the reasons for out-of-order data; two data points sent in order may be reversed if the first piece of data took a slower route through the network. Another cause is dropped measurements that need to be resent, potentially slowing them down to arrive later than some measurements following them. The distributed nature of an AMI makes these kinds of problems unavoidable.

# 3

# Design

Constructing the validation system requires careful consideration of the requirements and overcoming the challenges present. This chapter starts off with the system model and an overview of the system. After that, the main challenges are described, followed by details of the design of each part.

## 3.1 System model

A requirement for the validation system is that the input and output format needs to conform to a given specification. This specification along with an overview of different parts of the system is presented in the following section.

### Data input

The data that is processed by the validation system comes from an AMI of around 270,000 smart meters, each reporting hourly readings. The schema of a reading, presented in figure 3.1, consists of a meter position, which defines the household or location of the meter, a timestamp and a meter stand, i.e. the total energy consumed since the meter was installed.

### Meter consumption values

In order to perform validation of the meter readings, the meter stand needs to be transformed into a consumption value, or delta. A delta is calculated by taking the difference

| **Attribute** | `pos` | `ts` | `ms` |
|---|---|---|---|
| **Description** | meter position | timestamp | meter stand |

**Figure 3.1:** A measurement tuple

| Attribute | pos | ts | ms | cons |
|---|---|---|---|---|
| Description | Position | Timestamp | First meter stand | consumption delta |

**Figure 3.2:** A consumption value tuple

| Attribute | pos | ts | ms | cons | err |
|---|---|---|---|---|---|
| Description | Position | Timestamp | First meter stand | consumption delta | Validation error |

**Figure 3.3:** An error tuple

between two consecutive meter stands, to get the energy consumption during an hour. The schema is shown in figure 3.2.

#### Validation rules

A validation rule is defined as either a stateful or stateless operator that accepts a consumption value as input and passes it to the output in case there were no validation errors. In case the validation fails, an error tuple as seen in figure 3.3 is produced instead. The error tuple is an extension of the consumption value that adds an error identifier in the form of a single letter.

#### Composite error output

Apart from being interesting in themselves, the error tuples produced from a meter are analysed to find composite errors, or patterns of different errors that have the same cause, that have occurred during a day. A composite error is described by a meter position, a sequence of errors and the timestamp of the first error.

## 3.2 Challenges

There are two main challenges that need to be addressed when processing smart meter data. These challenges, incorrect dates and out-of-order tuples, are presented in the following section.

### 3.2.1 Incorrect dates

Some of the tuples received from the smart meters have incorrect timestamp dates; sometimes the time might even be off by as much as a year. This can be caused by a meter that has not yet had its time set after being installed, among other reasons. This problem can be dealt with in several different ways, depending on how much processing is permissible. An advanced way of handling the dates is to compare the consumption values to previously received ones and attempt to correct the time based on where those consumption values would fit in. Another, more crude approach, is to simply discard tuples with dates that are too old. An alert can be triggered in the event of this happening. At Göteborg Energi, the maximum delay for a measurement value is 40 days, so this is a reasonable cut-off point. This crude approach is the one used in this thesis because the incorrect dates are rare enough to be safely ignored in validation. Alerting the utility of these errors is the best way of dealing with it in the long term.

### 3.2.2 Out-of-order tuples

The problem of out-of-order data is a large research issue (see 7.4 Out-of-order data), and highly affects the operation of the prototype system designed in this thesis. The main area affected by disorder is when transforming meter readings into consumption values, since two consecutive readings are required. A solution to this is presented in section 3.3, based on keeping track of intervals that have been processed so that only the values that still need to be handled are stored. This does not require any ordering of tuples, but they will be processed in pairwise order as soon as they arrive.

## 3.3 Pairing operator

The purpose of the pairing operator is to accept potentially unordered tuples and output pairs of consecutive tuples, in the form of consumption value tuples. For most applications, the power consumption is much more interesting than the meter value. For example, to find negative, too high or zero consumption values, the difference between two consecutive meter values is required. This would be simple to take care of had the values arrived in perfect order by the timestamp; however, this is not always the case. A traditional, non-streaming way of dealing with disorder is to store all values received and calculate the consumption once all of them have arrived. Obviously since the data is streaming, it is not possible to store all of it.

A common way of dealing with out-of-order data in streaming systems is using buffering with sliding windows. The basic idea of this implementation of sliding windows is to collect data until a certain time frame is filled with values. For example, a sliding window of 12 hours in this case would collect 12 consecutive hours and do computations on the data once the window is full. As new data arrives, the oldest value is discarded and the calculations are performed on the new window. While a window stores only a limited amount of data, the memory usage can still be a significant if there are many sliding windows; one window for every meter is the case here.

Another option presents itself as one considers that it is most important to consider every pair of consecutive tuples in this case. This could be achieved by saving every tuple received and processing the pairs as soon as they are identified. This approach still leaves the question of when to discard tuples; when a tuple can be considered to be fully processed. To illustrate the problem, look at this simple example of tuples 1, 2, 3 and 4.

Tuple 1 can be part of two possible pairs: (0,1) and (1,2). Therefore, this tuple can only be discarded after tuple 0 and 2 have arrived. If we now assume that tuple 1 and 2 have arrived, one pair can be processed, namely (1,2). Neither tuple can be discarded

| Representation | Validation error |
|---|---|
| `N` | Negative consumption |
| `Z` | Zero consumption |
| `H` | Above fuse level |
| `h` | Above threshold |
| `A` | Above average |

**Table 3.1:** Validation errors and their character representations.

until tuple 3 arrives. As this happens, the pair (2,3) can be processed and thus tuple 2 can be discarded. When discarding a tuple, however, it is still important to keep track of it somehow. When tuple 4 arrives, tuple 3 is fully processed and can be safely discarded. In order to realise that tuple 3 can be discarded, it is necessary to keep some kind of reference either to tuple 2 or the pairs (2,3) and (3,4). Keeping all processed pairs is too costly in storage, as is the case for storing the tuples that are discarded. Instead, the pairing operating is introduced.

The presence of an interval denotes that all tuples inside of the interval have been processed. As tuple 1 arrives, a new interval (1-1), spanning between 1 and 1, is created. At the arrival of tuple 2, this interval is expanded to (1-2). As tuple 3 arrives then, tuple 2 is discarded and the interval is expanded to (1-3). Likewise, tuple 4 updates the interval to (1-4). This approach allows us to store only the tuples that are still waiting to be processed. In case of out-of-order tuples, new intervals can be formed to account for the tuples that are processed. If the tuple 6 arrives and (1-4) is the only interval, a new interval, (6-6), needs to be created. This can later be extended to (6-7), tuple 5 can arrive and join the intervals to (1-6) or a completely new interval can be formed again. Algorithm 1 formalises the process of adding a tuple to intervals.

If all tuples arrive eventually, the pairing operator can be used for continuous streaming data, but missing tuples will cause surrounding data and intervals to be left over. Therefore, it is important to remove both tuples and intervals when they become too old. This is easily solved by a garbage collection task that is performed periodically, twice daily for example.

## 3.4 Validation rules

A basis for further analysis, the validation rules lay the groundwork on which the validation system is built. The five validation rules presented in this section are designed in cooperation with people at Göteborg Energi and inspired by existing rules at the utility. Some of these rules may overlap, but they can be used in such a way that this is no problem.

**Algorithm 1** Adding a tuple to the pairing intervals

```
Require: $intervals$ is a sorted list of intervals
Require: $ts$ is a timestamp
Ensure: $intervals$ contains $ts$
function AddTuple($ts$)
    $tss \leftarrow (ts\text{-}ts)$
    for $i$ in $intervals$ do
        if $tss$ is before i then
            add $tss$ before $i$
            return $intervals$
        if $tss$ attaches before $i$ then
            join $tss$ at start of $i$
            return $intervals$
        if $tss$ attaches after $i$ then
            join $tss$ at end of $i$
            if $tss$ attaches before $i + 1$ then
                join $i$ at start of $i + 1$
            return $intervals$
        if $tss$ does not attach after $i$ or before $i + 1$ then
            add $tss$ after $i$
            return $intervals$
```

### 3.4.1 Negative values

One of the simplest kinds of validation is to alert on negative values for power consumption. For a normal household, this is always a technical error of some sort. The only case where a negative consumption value can be correct is if there is electricity production on site, and the meter is configured to reverse at times where there is a surplus electricity production. For the meters used at Göteborg Energi however, production and consumption is measured separately. So for the AMI studied in this thesis, no negative consumption is allowed.

### 3.4.2 Zero consumption

Although it is not an error in itself, zero consumption can be part of a bigger problem. This does not trigger an alert on its own, but can be a useful building block for finding composite errors. For example, a longer period without energy consumption may be important to look at more closely.

### 3.4.3 Above fuse level

As a maximum value for electricity consumption, the fuse provides a hard limit. For every individual meter, information about the installed fuses is available and has been used to calculate this limit. Although a fuse may temporarily allow values above its rating, it cannot sustain it over time and therefore consumption above this limit triggers an alert. This alert is almost always caused by a technical error, but it can also be caused by a customer who has installed a fuse with higher rating. Whether intentional or not, this needs to be corrected since the network tariffs for the consumer depends on the installed fuse.

### 3.4.4 Above threshold

The fuse level is a very high limit that catches the most extreme technical errors. However, it is often interesting to consider slightly lower values as well. At Göteborg Energi, the decision has been made to trigger alerts on a slightly lower threshold as well. In case consumption is at around 70% of the maximum allowed value for the fuse over the course of an hour, this needs to be examined more closely.

### 3.4.5 Above average

The fixed limit approach to validation of high values can only catch extreme errors. Still, there are erroneous high measurements that are not caught, and there is no guarantee that it will not produce false positives. For this reason it can be more interesting to look at the context of a measurement to decide whether it it too high or not. This idea is captured in another validation rule, based on an exponential moving average (EMA). With this type of average, the weight for each measurement decreases exponentially over time so that the most recent value has the biggest impact. Let $V_t$ be the measurement value at time $t$ and $\alpha$ be a weighting factor between 0 and 1. Then $EMA_t$, the EMA at time $t$, can be calculated as follows:

$$EMA_1 = V_1$$
$$EMA_t = \alpha \cdot V_t + (1 - \alpha) \cdot EMA_{t-1}$$

At the start, the EMA is not very reliable since the initial measurement value is used as a starting EMA value. Once the iterations have converged, the result is more reliable and can be used for finding abnormally high values. In this validation bolt, an $\alpha$ value of 0.1 is used and a consumption that is 100% above the EMA is considered an error.

## 3.5 Pattern matching

The simple errors described in section 3.4 are hard to analyse one by one since there can be up to 24 per meter per day. In fact, there are a few known *composite* errors that can be identified by considering the ordering and kind of errors that has been reported for a smart meter. In order to identify these errors automatically instead of the manual work done at Göteborg Energi today, a formal framework for pattern matching is needed.

One common system for pattern matching is regular expressions, described in appendix A. In order to apply regular expressions to the errors caught by the validation rules, they first need to be transformed into a string that can be matched by a regular expression. There is additional information that needs to be captured apart from the string of errors, for example finding if an error occurred at a specific time of day. If a match is found that also passes through these kinds of validation, it can be considered an instance of a composite error. This section describes how pattern matching is performed in greater detail, as well as how this can be achieved in a streaming fashion.

### 3.5.1 Storing errors

Error tuples received from the validation rules are stored in timestamp order waiting to be processed. Since the data is streaming, a limit to the number of tuples stored is necessary. Because of the requirement at Göteborg Energi to store data for 40 days, $40$ days $* 24$ hours $= 960$ hours of data, 960 tuples, are stored for pattern matching. Older values are removed to ensure that the stream processing can run continuously.

### 3.5.2 Matching errors

Matching is performed at the arrival of every error. Composite errors have a maximum length which is used to only look for matches within a partition of the stored errors. Matching an error with length $l$ at timestamp $ts$ will consider values with a timestamp of $(ts - (l-1)\ hours) - (id + (l+1)\ hours)$ to ensure that every possible interval of length $l$ containing the timestamp $ts$ is processed.

#### Regular expression

In order to match the errors with a regular expression (see Appendix A), representations of the errors within the matching interval are concatenated into an error string, where empty positions are represented as "-". Matches to the regular expression are translated into the start and end times for the match, so that additional properties may be tested for.

#### Additional properties

There are a few properties that cannot be captured by the regular expression:

- The exact number of occurrences of an error within a pattern. An example of this is finding that error 'X' has occurred twice within 10 hours of error 'Y'. This can almost be matched by "X.{1,9}Y", with the addition that one of the matched '.' was an 'X'.
- An expression (or sub expression) match started or ended at a specific time. Error 'Z' occurred at midnight is an example of this.

### Excluding duplicate errors

Every error should only be part of one composite error. Therefore, only the first regular expression match that also passes the additional properties is considered. Due to the fact that partitions of the stored errors that are matched by regular expressions can overlap, there is a risk of finding some composite errors several times. For this reason, the start and end of any matched composite error is stored. In case a match is completely covered by a previously matched error, that match is not considered.

# 4

# Implementation

WHEN GOING FROM DESIGN TO IMPLEMENTATION, some key technology choices as well as limitations are important to describe. This chapter starts by presenting implementation details of the data input, validation rules and pattern matching, followed by a description of two common errors and the patterns used to find them. Finally, the construction of the validation system as a Storm topology is presented. A high-level overview of the system can be seen in figure 4.1.

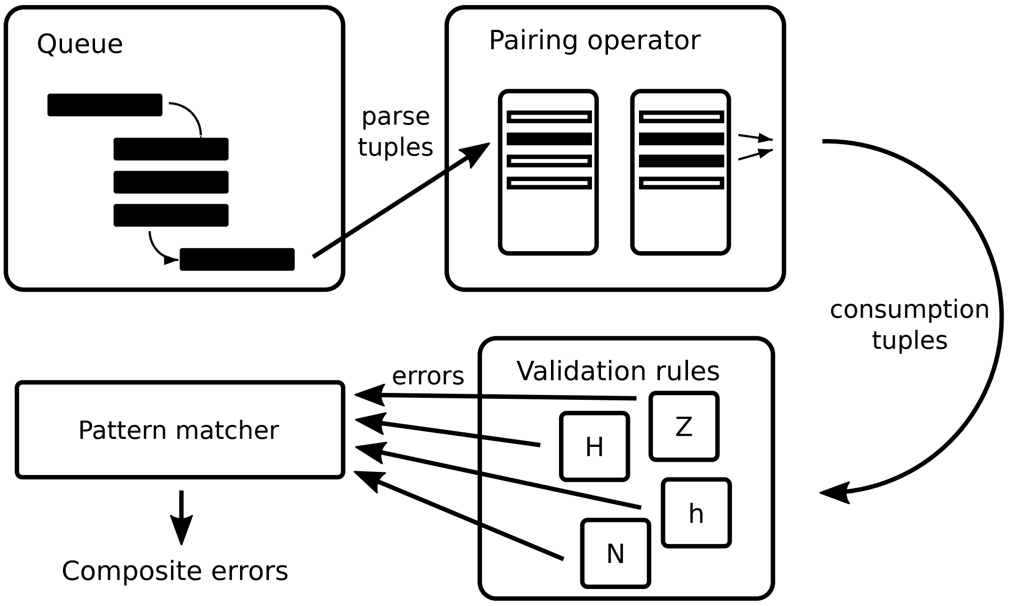

**Figure 4.1:** An overview of the validation system

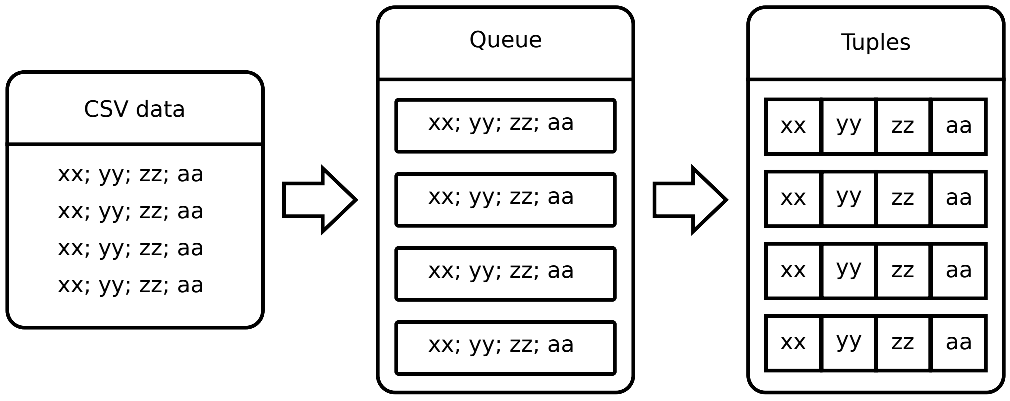

**Figure 4.2:** Data input for the system

<meter position>;<meter position>;<timestamp>;xx;<meter stand>

**Figure 4.3:** An example of an input line

## 4.1 Data input

A fundamental requirement for stream processing is that the data to be processed is presented in the form of a stream, which sets this paradigm apart from batch processing methods like MapReduce. This section presents the design of the data input to achieve stream processing of the data collected from the AMI. An overview can be seen in figure 4.2.

The smart meter data at Göteborg Energi is collected twice daily from each meter. Because there are so many meters, data arrives almost constantly. As this happens, data is aggregated from meters that are located close to each other and compiled in CSV (Comma-Separated Values) text files. This can be used as input into an SPE by reading the files in order and splitting on each line. Files that have been processed can be moved away or simply removed.

In order to make the stream processing system flexible, it is useful to separate the details of the input from the actual processing; this can be accomplished using a message queue. CSV data lines are put on hold in a queue until the SPE is able to process them. For this thesis, Apache Kafka [17], developed at LinkedIn, is used for its great performance and the fact that it works well together with Apache Storm. Because performance of several hundred thousand messages per second is normal for Kafka, it is unlikely that this will become a bottleneck of the system even if it were to scale to handle more meters in the future.

The final part of the input is to parse the CSV lines into tuples so that actual processing of the data can be performed. The format of a line can be seen in figure 4.3, and the relevant parts, namely meter position, timestamp and meter stand, needs to be extracted to form a measurement tuple. The format is shown in figure 3.1 in the Design chapter.

## 4.2 Validation rules

Validation rules are implemented so that they inherit from a Python class which takes care of the Storm details. Only two things are left to define: a single-character representation of the error, and a `validate` function. `validate` gets as input a ConsumptionDiff tuple and should return `true` if the validation passes and `false` if the error was caught. ConsumptionDiff contains position, timestamp, the current meter stand, the previous meter stand and what error character was found previously ('X' in case the tuple came directly from the intervals); all the information needed for validation is contained in this tuple. This setup makes new rules simple to implement.

## 4.3 Pattern matching

The design of the pattern matching operator has been discussed in section 3.5, but this section presents the implementation details necessary to fully grasp its operation.

### 4.3.1 Storing errors

As previously discussed, the pattern matcher receives error tuples from the validation rules. The newest errors need to be stored, while old errors must be purged to enable streaming operation. This is accomplished by a new data structure called a *sliding list*, described below. The error tuples are stored in a sliding list by timestamp. The timestamp is transformed into an index by taking the UNIX timestamp, which is a representation using seconds, and dividing it by 3600, thus receiving a number incremented by one every hour. This method can be used for any regular interval down to one second. A single-character representation of an error is stored at the hour index in the sliding list.

#### Sliding list data structure

The purpose of the sliding list is to 1) allow for assignment of arbitrarily large indices while conserving memory; and 2) limit the number of elements by removing old values, while keeping a familiar list interface. To achieve such a data structure in an efficient manner, a regular list is used as a base. Large indices are mapped to smaller indices so that assigning to a very large index in an empty list still results in a list of size one. The sliding list has a maximum length to conserve memory and elements that no longer fit in the list are removed. The necessary list operations are implemented as follows:

**List creation**
A list is created with a limit, starting index, fill value and starting list, any of which may be omitted. The simplest usage is with only a limit, starting index will change as indices are assigned to. Fill value defaults to "-".

**Assignment within existing indices**
: The index is recalculated using the starting index and then assignment is done in the underlying list.

**Assignment outside of existing indices**
: The underlying list needs to be extended and possibly slide. The extension (positive or negative) of the list is calculated, the fill value is inserted to pad up to that index and the assignment is performed. Finally, the list shrinks from the start or end if needed and the new starting index is calculated.

**Indexing**
: When retrieving a value, the index is recalculated using the starting index. *index* in the sliding list corresponds to $index - starting\ value$ in the underlying list.

**Slicing**
: A range of values can be extracted from the list. In this case the indices are recalculated and the slice of the list is returned. Any index of the wanted slice that is not part of the list is ignored. For example, requesting elements 1–10 when the list contains only 5–10 will result in the whole list.

### 4.3.2 Finding errors

Using a sliding list with a limit of 960 (24 hours for 40 days), algorithm 2 stores the errors and finds pattern matches. Matches that are found produces alerts in the form of a text file that can be accessed by the people responsible for error correction.

## 4.4 Common errors

As described in section 3.5 and 4.3, composite errors can be detected by pattern matching of validation errors. In this section, two such errors that are commonly found in actual validation at Göteborg Energi are presented, together with the pattern matching rules that can be used to identify them.

### 4.4.1 Rushing and reversing

A certain type of meter has a frequently occurring error, usually exhibited by between 10 and 30 but sometimes up to 100 meters per day. The error has no apparent reason, but when it appears the meter needs to be replaced. The problem is that the meter sends extremely high, seemingly random consumption values instead of the actual consumption. This can happen between zero and 23 times during a day. Every day at midnight, the absolute meter position, not just the consumption between two hours, is transmitted which reveals how much was actually measured during the day.

When analysing readings like these, the validation rules alert on one or more extremely high consumption values during the day, and a negative value at midnight,

**Algorithm 2** Add an error and find an error pattern for a meter.

```
Require: $err$ is a single-character representation of an error
Require: $ts$ is a timestamp
Require: $errors$ is a sliding list of errors
Require: $patLen$ is the maximum length a pattern can have
Require: $regex$ is the regular expression to match
 1: function AddAndMatchError
 2:     $hour \leftarrow$ (unix timestamp for $ts$) / 3600
 3:     $errors[hour] \leftarrow err$
 4:     $errorstring \leftarrow errors[hour - patLen : hour + patLen]$ as string
 5:     $matches \leftarrow$ matches to $regex$ within $errorstring$
 6:     for $match$ in $matches$ do
 7:         $(start, end) \leftarrow (match.start, match.end)$ transformed into hour
 8:         if $end$ is at midnight and NotMatched$(start, end)$ then
 9:             mark $(start, end)$ as matched
10:             return $start$ and matched string
11: function NotMatched$(starthour, endhour)$
12:     for $(start, end)$ in matched errors do
13:         if $starthour >= startandendhour <= end$ then
14:             return false
15:         else
16:             return true
```

correcting the errors. Therefore, the following regular expression can be used as a basis for finding this error:

```
[Hh][Hh-]{0,22}N
```

Here, “H”, “h” and “N” signify a value higher than the fuse allows, higher than the threshold that is acceptable, and lower than zero respectively. It will match one high value (above fuse or above threshold), between 0 and 22 values that are not negative (hence “Hh-” and not “Hh-N” or “.”), and finally a negative value. This needs to be extended to ensure that the negative consumption error occurred at midnight, in other words that the end of the pattern match happened at 00:00. In summary, this corresponds to one or more high values during the day and one single negative value at midnight.

### 4.4.2 Incorrect readings after a power outage

An error that is common after power to the meter has been out is that two consecutive meter readings are put together into one. This means that the first hour has zero consumption and the following hour has a potentially double consumption. For most customers this is not a problem, but those who are paying a different price depending on the hour can be affected. A pattern for finding some of these errors can be defined as follows:

```
ZA
```

Here, “Z” and “A” represent a zero value and a value higher than the average respectively.

## 4.5 Streaming validation with Storm

The validation system is implemented as a Storm topology, shown in figure 4.4. Data input from Kafka is performed by a spout and the rest of the topology is made up of a number of bolts. A line reader bolt parses the CSV lines and outputs tuples to a date filter bolt. The date filter bolt discards tuples timestamped with a future date or a date that is more than 40 days old, triggering an alert for these tuples as well. These bolts both use the shuffle grouping of Storm, while the others are grouped by the meter position field. An intervals bolt pairs tuples as described in section 3.3, and tuple pairs are emitted to several validation bolts. Garbage collection in the intervals bolt is implemented using tick tuples emitted daily.

Another important consideration for the intervals bolt is persistence. In the current implementation, all data is held in memory, but this can be moved to an external cache or database that is persisted to disk. In this way, the bolt would not be vulnerable to data loss in case of a crash.

There are several ways to structure the validation bolts, but there is good reason for the layout shown in the figure. The Zero consumption bolt catches more errors than the other validation bolts, so filtering out those errors first decreases the load on the rest of them. Above threshold catches high consumption values, as does above fuse, but since the threshold is lower more errors are caught here. All the errors found by above threshold need to be compared to the fuse value as well, but values that are not above the threshold are definitely not above the fuse value. Values that are not too high according to the threshold still need to be validated by the negative bolt.

When errors are found in the validation bolts, these are emitted to the pattern match bolt which finds the common errors present here. Common errors that are identified by the pattern match bolt will trigger alerts stored in a text file.

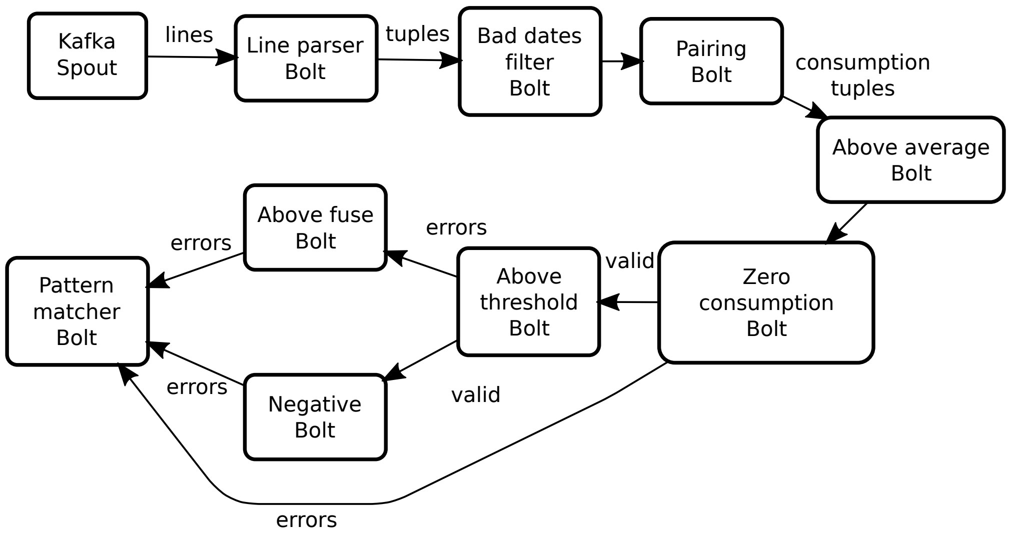

**Figure 4.4:** The spout and bolts of the storm topology

# 5 Results and Evaluation

This thesis presents a prototype validation system consisting of validation rules and pattern matching. These rules and patterns have been evaluated with regards to accuracy, prevalence and performance and the results are presented in this chapter. First, the system setup is presented, followed by a description of the tests performed. Finally, the results of the tests are reported.

## 5.1 Evaluation setup

The prototype validation system is run with data from the AMI of utility Göteborg Energi, consisting of roughly 270,000 smart meters. The prototype has been implemented on a virtual server with two AMD Opteron processors at 2.6 GHz and 4GB of RAM. This rather modest system configuration is used to run Kafka and Storm as well as read data from log files into the Kafka queue.

Two different tests were performed to evaluate the system. One was run with a month's worth of meter data and all validation rules and patterns enabled and another one was run during 12 of these days with fewer rules to test rushing and reversing.

In each of the two test cases, the number of errors matched were assessed as well as the number of pattern matches. Performance in terms of latency and throughput were also measured. In the second test, pattern matching were compared to the composite errors found manually.

## 5.2 Full system evaluation

The validation system has been evaluated using the data produced by the 270,000 smart meters during one month. All the validation rules were enabled: zero and negative consumption, above average, above threshold and above fuse. Three of these rules are currently used, in a non-streaming fashion, at Göteborg Energi, but zero consumption

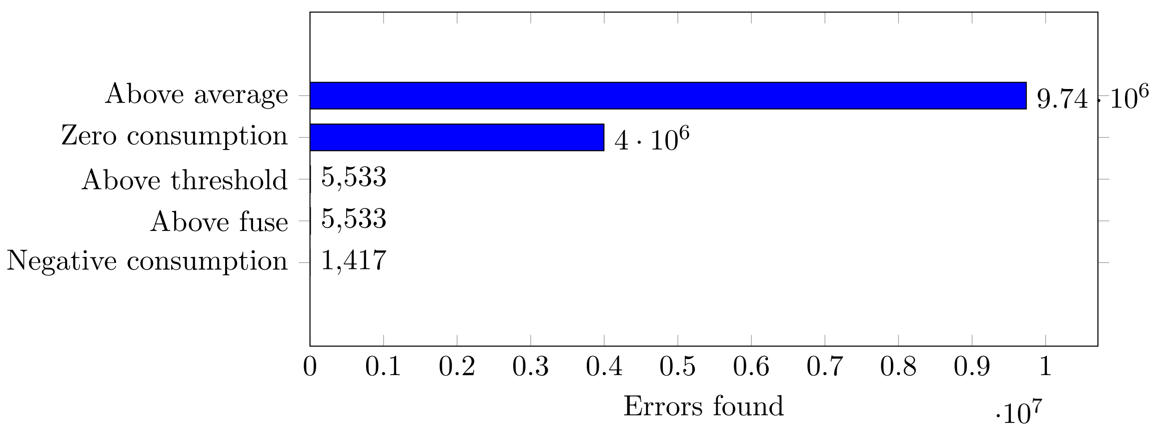

**Figure 5.1:** Number of errors found over a period of one month.

and above average are new. The new rules are used in an attempt to formulate patterns for composite errors that can easily be missed with current validation.

### 5.2.1 Validation rule outcome

The number of errors found for each validation rule can be seen in figure 5.1 and show the following findings:

- When comparing the number of matches for different rules, above average greatly outnumbers the rest. This cannot be because there are that many errors, but because the number of false positives is very high. This is discussed more in detail below.
- Zero consumption is the second most common error. This is to be expected, since it is the only "error" that is completely valid in many cases.
- All the consumption values that above the threshold were also above the fuse, so the threshold has not been useful at all during this month.
- The least common error is the negative consumption.

#### EMA errors

The number of errors that are above the exponential moving average is very large, but most of the consumption values are pretty low. 59% of the EMA "errors" are triggered by a consumption lower than 1 kWh, which can be caused by an oven that is used for 40 minutes. 84% of the errors are below 2 kWh, a consumption that is absolutely reasonable if the oven is used for the entire hour, something is cooked on the stove and the television is on.

This examination leads to the conclusion that the exponential moving average is not a reliable validation rule. Further refinement or a different approach would be needed to use a moving average for validation.

### 5.2.2 Pattern matching outcome

Both the “after power outage” and “rushing and reversing” patterns were evaluated to see how many errors they matched.

#### After power outage pattern

The “after power outage” pattern was matched 170,000 times in one month of data, that is over 5,500 times a day. Around 12,000 meters show this pattern and a single meter can exhibit the error several times during a day. Since both zero consumption and above average are very commonly occurring errors, the abundance of matches to this pattern is unsurprising. Since the EMA errors have been found to be unreliable, the results from this pattern can be questioned as well. Until a more reliable “high” error rule is available, the usefulness of this pattern is limited.

#### Rushing and reversing pattern

The “rushing and reversing” pattern was matched 640 times in one month of data, so an average of 21 errors per day. This can be compared with the average number of erroneous meters per day, which was 67. In other words, a bit over 31% of the erroneous meters every day were rushing and reversing.

### 5.2.3 Performance benchmarks

Throughput of the system was found by running the topology with live data and increasing the input rate step by step until the CPU usage was at 100% and the system could not keep up with the data any longer. Then, then speed was lowered so that the system could just keep up with all the data. The throughput measured in this way was 1,000 tuples per second, enough for real-time validation of an AMI with 3,600,000 meters, assuming hourly readings. The highest latency observed was 4 seconds.

## 5.3 Rushing and reversing evaluation

The “rushing and reversing” error is a well known commonly occurring error that has to be corrected daily. This rule has been evaluated separately during one and a half week to compare it to the manual validation currently performed at Göteborg Energi in order to judge the effectiveness of the pattern. The approach and results are presented in the following section.

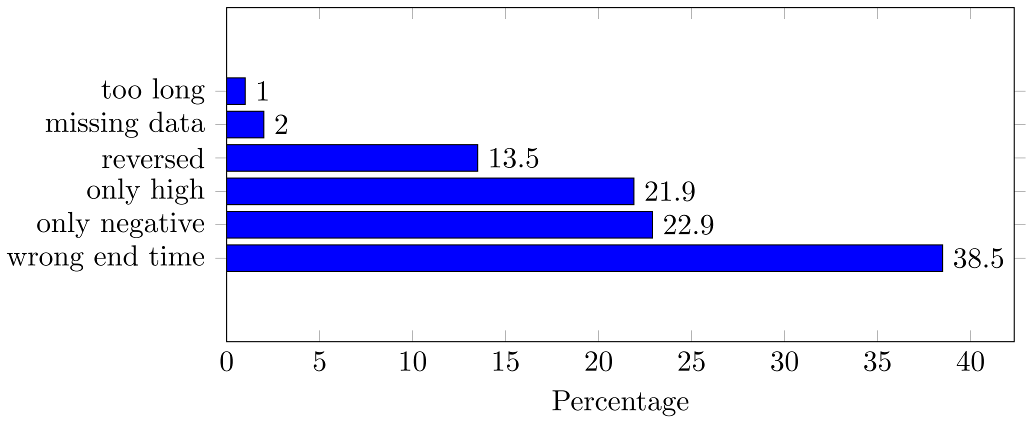

**Figure 5.2:** Reason for rushing and reversing errors not being matched by pattern.

| | Total | Manually matched | Matched by pattern |
|---|---|---|---|
| Meters with errors | 410 | 190 (46.3%) | 116 (28.3%) |
| Days with errors | 1050 | N/A | 411 (39.1%) |

**Table 5.1:** Total errors, negative or above fuse, compared to the number of errors manually and automatically matched as "rushing and reversing". The pattern catches a majority of the manually identified errors, but not all of them.

### 5.3.1 Validation system setup

In the evaluation conducted for this particular rule, three of the validation rules were active: above fuse, above threshold and negative values. Rushing and reversing was the only active pattern.

### 5.3.2 Number of alarms generated

During the testing period, there were 410 meters in total that had some kind of validation error. Out of these, 190 meters were manually identified as being rushing and reversing, the most common error. The prototype system diagnosed 116 of these meters as experiencing this problem during at least one day, so 61% of the rushing and reversing meters were found. This is explored further in the next section.

Many of the 410 meters experienced errors during more than one day, so summing up the days for all the 410 meters yields a result of 1050 days of errors.

In addition to looking at the number of meters, the absolute number of errors can be examined. As mentioned in section 5.2.1, the above threshold and above fuse errors were found to be completely overlapping, so they should not be counted twice. Counting the negative value and above threshold errors then adds up to 4161 errors in total.

### 5.3.3 Pattern matching accuracy

Table 5.1 presents a comparison of the total errors and the errors identified as “rushing and reversing”, both manually and by pattern match. As can be seen in the table, rushing and reversing measurements account for the issues in almost half of the erroneous meters. The pattern match does not identify all of these errors, however. 74 meters, or 38.9% of them, did not match the pattern. As can be seen in figure 5.2, there are several different reasons for this. At 38.5%, the most common one is that the end of the pattern is at the wrong time, i.e. the negative consumption is not recorded at midnight, but at some other hour. After that, we have meters with only negative, reversing, values at 22.9%, followed closely by only high, rushing, values at 21.9%. 13.5% of non-matched meters were found to “reverse and rush” rather than the usual “rush and reverse”, so the negative values came before the high ones. The next error, at 2%, was because of data arriving after the end of the testing period, hence missing for the validation system but matched manually because of prior knowledge. Finally, 1% of errors were caused by the pattern being too long. Normally, high values are negated at midnight but here they were not corrected until the next day.

Although the meter matching rate is only 28.3%, the pattern proves to be more successful when looking at the number of errors instead. At 39.1%, the portion of actual errors identified show that the meters that match the pattern are more prone to reoccurring errors. It would have been interesting to compare that to the portion of errors found from manual identification; however, that data is not available since errors were not reported for each meter each day.

#### False positives

During the week of testing there were a few individual errors found by the system that were not manually identified as matching that particular error. However, further investigation revealed that they did actually exhibit the behaviour expected of a rushing and reversing meter, but they had been ignored because they were scheduled for replacement. The errors were identified before the meter was replaced though, so the patterns were correct. Further discussion on these replaced meters can be found in section 6.1.1

### 5.3.4 Performance benchmarks

The throughput using this validation setup was measured in the same way as for the full system evaluation, and was found to be 1,500 tuples per second. This is more than sufficient for the needs of Göteborg Energi and has the potential for continuously validating the consumption of 5 million hourly measurements, or 1.35 million measurements every 15 minutes. The highest latency observed was 4 seconds; however, due to of out-of-order arrival of measurements, it can take a long time for some values to produce an output. This cannot be mitigated by the system since all necessary data has to arrive to produce some output. Still, partial results are reported if possible.

# 6

# Discussion

In this chapter, the results from evaluating the pattern matching are discussed, as well as implementing the system in actual use and considerations regarding sustainable development and ethic.

## 6.1 Potential and limits of pattern matching

The pattern matching presented in this thesis is limited to a few cases, but still a large portion of the errors reported during each day are actually covered. Although the portion of meters matched by the pattern is not as big, it is interesting to note that the meters that have the most reoccurring errors are identified.

The fact that there were many meters that did not match the pattern although they had the "rushing and reversing" error is an interesting find on its own. This was not the outcome that the system expert at Göteborg Energi had expected, but with this system provides a tool to evaluate the characteristics of different errors. In addition, the log of pattern matches identified provides valuable statistics that can be used when assessing the overall quality of the AMI.

Through the power and flexibility of regular expressions, adding different errors in the future should be straightforward. The case where pattern matching fits especially well is when some easily identifiable error occurs frequently. In the event that most errors are irregular this method is not as useful. In the particular case studied here at Göteborg Energi, pattern matching has been shown to hold great potential.

### 6.1.1 Dealing with false positives

There were no actual false positives in the pattern matching, only meters that were scheduled for replacement and therefore not considered in the manual validation. Such a distinction can be made in order to avoid reporting the same meter twice, but the erroneous data still needs to be validated and corrected. So, although it may be interesting to treat replaced meters in a special way in the future, such an extension is not needed for validation and falls outside the scope of this thesis.

### 6.1.2 System performance

The throughput of the prototype validation system is not impressive if compared to the performance seen in other projects using Storm, but there are two reasons for this. First of all the evaluation hardware is not specifically tailored to stream processing and it not very powerful either. Secondly, using Python greatly decreases the speed because bolt communication then happens through a shell rather than through shared memory in the JVM, as is the case for Java bolts.

## 6.2 Using the system

To use this system in production, two important steps are needed:

**Displaying alerts** The output of this prototype is in the form of text files that are simple to analyse but not suitable for use by the people working with validation. This data should instead be presented in a graphical interface, coupled with error correction.

**Correcting errors** Interpolating the incorrect values between the start and end of an error would be possible using the data present in the system. The interface would need to display the error found, with some context, the date and time of the error, the meter position and feature a button for correcting the error.

On days when there are 50 meters that have had problems, a system like this has the possibility of decreasing the workload significantly. Even with an average of 21 rushing and reversing meters per day, or 640 per month, the time saved can still be significant. An error can take around two minutes to correct manually, so that means $2 * 640/60 = 21.3$ hours can be saved per month, or 256 hours per year. Assuming a hourly cost of 50 €, that is a saving of 12,800 € per year. This is time, and in the end money, that can be spent on quality work rather than repetitive error correction.

## 6.3 Contributions and effects on sustainable development

The potential of the work outlined in this thesis is to be able to automate the fundamental validation of extreme values to free up human resources. As the number of customers increase and possibly the measurement intervals as well, manually correcting the errors becomes unsustainable. In this way, the validation system opens up possibilities for the future. In the short term, a system that deals with the extraordinary values can allow Göteborg Energi to move beyond these obvious errors and find meters that are performing incorrectly, potentially drawing more current than necessary. This can allow them to focus on more subtle errors in the future, enabling more accurate, “green-friendly” billing and giving consumers greater control of their energy consumption. In a future where measurement information is used to make informed choices about energy consumption, it is crucial that this information is accurate.

This thesis also contributes to the body of research surrounding the development of a cyber-physical grid, or smart grid. This development is necessary to be able to handle renewable energy sources that may be intermittent as well as increasingly distributed. Validation of data is as central to this development as measuring electricity consumption, because the usefulness of the data decreases if it is not correct.

## 6.4 Ethical considerations

Analysis of household electricity consumption is necessary to find technical errors as well as malicious consumers trying to steal electricity; however, this presents an ethical challenge, since a lot of information can be extracted from electricity usage. This poses privacy concerns for customers that would not like to be monitored in their own home. Considering that it is possible to identify when people get up in the morning, if they watch TV a lot or when they are away on holiday, the information that can be gained from electricity consumption alone is substantial.

Since any system analysing electricity consumption data has a potential to intrude on privacy, it is important to know what the purpose of the system is and who is allowed to handle the information. Since correct billing is the objective of this thesis, it has been made clear that there is no intention to extract any personal information from the analysis. Furthermore, a system like this, that the utility controls and develops, enables them to perform the necessary validation themselves. Conversely, if the required competence was lacking in the utility and data had to be sent to a third party for analysis, there would be cause for concern. In other words, the data validation system presented in this thesis aids the safe handling of sensitive data rather than posing a privacy threat.

# 7

# Related Work

AMI DATA VALIDATION USING STREAM PROCESSING is an area that has seen only some research. All three areas involved: AMIs, data validation and stream processing, are therefore of great interest when examining the work related to this thesis. The push for utilities towards AMIs started at the turn of the century [15, 26] and today they are commonplace, spawning a wealth of research into how this relates to security [6], consumer behaviour [22] and more. The stream processing paradigm that emerged during the same time has been explored in the context of AMIs since 2010. Several different applications have been found for SPEs in this area, including Intrusion Detection Systems (IDSs) [12], real-time pricing [21], adaptive measurement rates [27] and data validation [13].

A few papers describing these applications are presented here to show that the use of an SPE in the context of this thesis is reasonable. Furthermore, this approach has potential for building a validation system that is scalable as the number of smart meters as well as the frequency of readings increase.

## 7.1 Stream processing of AMI data

A set of requirements for processing smart meter data streams has been formulated by Lohrmann and Kao [21], taking into account scalability, availability, processing latency and data management. Parallel stream processing in clouds is proposed as a means of addressing these requirements and a prototype real-time-pricing application is presented to demonstrate the feasibility of this approach.

The prototype application is capable of handling one million simulated smart meters. Each simulated smart meter has its own TCP/IP connection communicating with a cluster of 19 virtual machines, running the Nephele cloud computing framework [33] on top of the Eucalyptus Private Cloud [25]. This cluster, built using commodity hardware, is able to aggregate the data and present price updates with approximately 10 second intervals.

This thesis neither aims to do real-time pricing nor to run on a cloud computing platform, however the system presented in this thesis is built in such a way that deploying it to a cloud platform is trivial. Hence, the performance results obtained by Lohrmann and Kao can likely be applied to this thesis as well.

## 7.2 Handling data volumes in AMIs

The challenge of handling the massive data volumes produced by AMIs has been handled in different ways. This thesis takes the approach of constructing a system that can scale to handle such large data volumes, but another possibility is to reduce the data rate in different ways. In this section, two approaches to reducing the data are presented.

### Adaptive rate control

Simmhan et al. [27] show that bandwidth usage by power consumption measurements in smart meters can be decreased by 50% using adaptive rate control. Smart meter consumption data is fed into a stream processing pipeline where an alert is triggered on two different conditions. An individual smart meter whose consumption exceeds a certain predefined threshold triggers an alert, as well one that rises at least 25% above the 15 minute running average. The 15 minute running average for both individual smart meters as well as for the utility as a whole is calculated and the latter controls the AMI measurement rate. For throttling, the rate is decided by the difference between the available power capacity at the utility and the current cumulative power usage by consumers. The rate changes in steps and there is a minimum and maximum rate allowed.

IBM InfoSphere Streams [14] is used as the stream processing system and this is deployed on Eucalyptus Private Cloud, which is compatible with and easily transferable to the Amazon Elastic Compute Cloud (EC2) for easy scaling.

Although this thesis will not attempt to decrease bandwidth consumption, the alerts seen here hold some similarities with the validation rules that are to be implemented.

### Random projections

Dieb Martins and Gurjao [7] present another approach for limiting the data volumes received from smart meters, using random projections based on dimensionality reduction. A sketch, or a reduced version, of smart meter data can be produced that is 50% smaller than the original data. This allows for handling of a growing consumer base while achieving a 2% average relative error rate.

The smart meter data streams are analysed offline in this work, so there is no description of how this would be implemented in a real-time system.

## 7.3 Data validation

The works discussed up to this point do not mention data validation, but all require the input data to be correct. This thesis aims to ensure that the data is sanitised and in that way is a prerequisite for all the systems presented this far.

Gulisano et al. [13] have done the most closely related research, showing that stream processing can be used to validate the measurements from thousands of smart meters per second using commodity hardware.

Three continuous queries are presented for data validation: (1) discarding energy consumption values that are negative or exceed a given threshold; (2) discarding energy consumption readings that exceed two times the mean consumption during a given time period; (3) interpolating missing values in the event that two consecutive tuples from the same meter are too far apart in time.

Using different batch sizes, the system presented was able to process messages at rates allowing for continuous processing of data from 5-25 million smart meters, assuming hourly readings. The latency observed was in the range of milliseconds.

This thesis holds many similarities to the work of Gulisano et al. and builds upon the results with other validation rules. One difference is that this thesis looks at commonly occurring error patterns based on the validation rules. Furthermore, this thesis is written in collaboration with a real-world utility and will base the validation rules on the needs of that utility. An analysis of the performance will also be performed using current data from the utility.

## 7.4 Out-of-order data

As has been mentioned already in section 2.3, out-of-order data is a common stream processing problem and one that is addressed in this thesis. There has also been previous research into this area, and two concepts are prevalent: punctuation, and variations thereof, and K-slack. In this section, an overview of these concepts as well as a comparison to this thesis is presented.

### Punctuation and heartbeats

Punctuation semantics were presented as an alternative to sliding or tumbling windows to enable two types of queries on streams that are not normally possible for unbounded streams: stateful operators and blocking operations [32]. Stateful operators applied to unbounded data continue to increase in state until they run out of memory. Blocking operations on the other hand keep processing until all input is read, so with an unbounded stream they will never terminate. To perform these queries, the unbounded stream is divided into an infinite sequence of finite stream partitions by means of punctuations. Instead of considering the unbounded stream, partial results from the individual finite streams can be produced instead. These punctuations that divide a stream are transmitted as normal tuples but carry specific information. A punctuation gives guarantees on the tuples that will later be encountered in a stream. Put simply, a punctuation is a predicate on tuples that will evaluate to `false` for every tuple following it. For example, if tuples contain logical timestamps in the form of increasing integers, a punctuation could indicate that all tuples with a timestamp lower than 10 have arrived.

Punctuation has been explored further and extended into a concept called heartbeats [28], which can be described as punctuations on timestamp. A basic principle for handling out-of-order arrival using heartbeats is the presence of an input manager that buffers out-of-order tuples and outputs them in order of increasing timestamp. The heartbeats are the basis for deciding when the input manager can output a tuple, and they are defined as follows:

> A heartbeat for a set of streams $S_1, S_2, ..., S_n$ at wall-clock time $c$ is defined as the maximum application timestamp $\tau$ such that all tuples arriving on $S_1, S_2, ..., S_n$ after time $c$ must have timestamp $> \tau$.

Heartbeats can be produced externally, but one of the main goals is to deduce heartbeats based on a set of properties that a stream fulfils. Three bounds are required for deducing heartbeats. Assuming that every tuple is timestamped at the source, the presence of clock skew is inevitable. Therefore the first bound on the skew is required for heartbeat deduction. Secondly, out-of-order generation, a bound on how much out-of-order a stream is at emission, also affects ordering of tuples and is necessary to put a bound on. Finally a bound on network latency is required. If the required bounds are known, heartbeats can be deduced so that ordered output from the input manager as well as stream progress can be guaranteed.

Punctuations have also been used for implementing a window query evaluation technique that handles out-of-order tuples without sorting, called Window-ID (WID) [18]. Tuples are processed on arrival by updating each window that a tuple is part of. Although several window aggregation calculations need to be ongoing simultaneously, memory usage is decreased since there is no need to buffer tuples. Punctuations are leveraged in this context to handle the disorder, namely to mark when processing for a window has been completed so that a result can be produced.

Extending the concept even further, an approach to stream processing without ordering constraints, called OOP (out-of-order processing), has been presented by Li et al. [19]. Progress indicators like punctuation and heartbeats are used here to purge state and unblock operators, and they can be generated at the source of the data or at the input of the system. When punctuation is generated by the system itself, the input is assumed to be ordered to make generation easier. This punctuation is then propagated through the system by all operators.

### K-slack

K-slack is a way of formulating flexible ordering constraints on streams. Presented as a part of the k-Mon query processing architecture, these ordered-arrival constraints make up one of several k-constraints identified for the purpose of reducing run-time state for continuous queries [2]. Simply stated, a k-slack constraint specifies that any two out-of-order tuples arrive at most $k$ steps apart. The k-value can be monitored so that the slack is dynamically altered to accommodate all tuples. An upper limit for $k$ is used to discard tuples that are extremely late, this is to avoid memory saturation. K-slack has also been shown to work in distributed environments [23].

### Other solutions

Aside from variations on punctuation and k-slack, the two approaches have been combined to formulate strategies fit for different situations [20]. In streams where out-of-order events are common, a conservative strategy based on punctuation can be used to ensure correctness. On the other hand, when out-of-order events are rare, an aggressive approach using k-slack can fit better. The aggressive strategy produces output quickly, but erroneous result need to be corrected once the out-of-order events have arrived.

Another way of dealing with out-of-order data that has been proposed is to use buffering while processing events speculatively [3]. Here, the simple way of handling disorder with buffers is extended so that time that would have been spent waiting for missing data is instead used for processing the data that has arrived. Some of this processing may be in vain, but through the use of predictors only the most promising events are processed. The end result is both increased throughput and reduced latency.

In a real-time system, more recent values are often more important than older values. This fact is captured in the concept of time-decaying aggregates, which can also be used while dealing with out-of-order data [5]. Values can be weighted by different models, for example exponential decay or polynomially decreasing decay; either way, out-of-order data will be weighted correctly just like the ordered data.

### Intervals

The solution proposed in this thesis, called intervals, differs from the aforementioned works because the requirements are slightly different. Most of the previous works aim to ensure order before processing the data, while that is not necessary in this case. The primary issue in this thesis is ordering pairs of tuples, so total orderedness is never needed. Furthermore, there is a requirement at Göteborg Energi that a measurement must be allowed to arrive up to 40 days late. Such a delay leads to unreasonably large memory usage for ordinary approaches to dealing with out-of-order data.

# 8

# Conclusion

DATA VALIDATION is a challenge for every electricity utility and Swedish utility Göteborg Energi is no exception. A substantial amount of time is spent on manually correcting errors, a repetitive task better suited for a computer, so automating this task would allow for more efficient use of their limited resources. Therefore, a prototype validation system for AMIs based on stream processing has been presented in this thesis. One type of error, called "rushing and reversing" has been found to be the cause of error in roughly 45% of erroneous meters. A pattern matching approach to identifying these errors has been developed that can find almost 30% of all erroneous meters, or 40% of all individual errors. The system is capable of processing data at a rate of 1,000 tuples per second, enough to handle 3.6 million hourly readings or 900,000 readings every 15 minutes. In both cases this is more than the 300,000 meters currently present in the AMI.

## 8.1 Future work

Although throughput has been evaluated, the performance testing has only been done on a single machine. One of the advantages of Storm is its scalability, so it would be interesting to confirm that the system as a whole can scale well to more machines when needed.

As mentioned in section 6.2, the next steps in terms of features would be going from alerting to correcting. With the information present in the matched patterns, the distance from this to displaying it in a web interface where a single button push can generate new, interpolated values in place of the identified errors is not far. This has potential to save hours of repetitive error correction work each day, enabling more quality analysis of the data.

It would be useful to analyse errors over time, to see trends in the models, makes or areas that cause the most problems. This information might then be used to avoid errors at the source and could serve as a valuable tool for the long-term operation of the AMI.

# Bibliography

[1] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, and others. The design of the borealis stream processing engine. In *CIDR*, volume 5, pages 277–289, 2005. URL `http://www.cs.harvard.edu/~mdw/course/cs260r/papers/borealis-cidr05.pdf`.

[2] Shivnath Babu, Utkarsh Srivastava, and Jennifer Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Trans. Database Syst.*, 29(3):545–580, September 2004. ISSN 0362-5915. doi: 10.1145/1016028.1016032. URL `http://doi.acm.org.proxy.lib.chalmers.se/10.1145/1016028.1016032`.

[3] Andrey Brito, Christof Fetzer, Heiko Sturzrehm, and Pascal Felber. Speculative out-of-order event processing with software transaction memory. In *Proceedings of the Second International Conference on Distributed Event-based Systems*, DEBS '08, pages 265–275, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-090-6. doi: 10.1145/1385989.1386023. URL `http://doi.acm.org.proxy.lib.chalmers.se/10.1145/1385989.1386023`.

[4] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Nesime Tatbul, Stan Zdonik, and Michael Stonebraker. Chapter 20 - monitoring streams — a new class of data management applications. In Philip A. Bernstein, Yannis E. Ioannidis, Raghu Ramakrishnan, and Dimitris Papadias, editors, *VLDB '02: Proceedings of the 28th International Conference on Very Large Databases*, pages 215–226. Morgan Kaufmann, San Francisco, 2002. ISBN 978-1-55860-869-6. URL `http://www.sciencedirect.com/science/article/pii/B9781558608696500275`.

[5] Graham Cormode, Flip Korn, and Srikanta Tirthapura. Time-decaying aggregates in out-of-order streams. In *Proceedings of the Twenty-seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '08, pages 89–98, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-152-1. doi: 10.1145/

1376916.1376930. URL `http://doi.acm.org.proxy.lib.chalmers.se/10.1145/1376916.1376930`.

[6] M. Costache, V. Tudor, M. Almgren, M. Papatriantafilou, and C. Saunders. Remote control of smart meters: Friend or foe? In *2011 Seventh European Conference on Computer Network Defense (EC2ND)*, pages 49–56, September 2011. doi: 10.1109/EC2ND.2011.14. 00013.

[7] A. Dieb Martins and E.C. Gurjao. Processing of smart meters data based on random projections. In *Innovative Smart Grid Technologies Latin America (ISGT LA), 2013 IEEE PES Conference On*, pages 1–4, April 2013. doi: 10.1109/ISGT-LA.2013.6554421.

[8] M. Ghofrani, M. Hassanzadeh, M. Etezadi-Amoli, and M.S. Fadali. Smart meter based short-term load forecasting for residential customers. In *North American Power Symposium (NAPS), 2011*, pages 1–5, August 2011. doi: 10.1109/NAPS.2011.6025124.

[9] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, and P. Valduriez. Streamcloud: A large scale data streaming system. In *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, pages 126–137, June 2010. doi: 10.1109/ICDCS.2010.72.

[10] V. Gulisano, R. Jimenez-Peris, M. Patiño-Martinez, C. Soriente, and P. Valduriez. Streamcloud: An elastic and scalable data streaming system. *Parallel and Distributed Systems, IEEE Transactions on*, 23(12):2351–2365, Dec 2012. ISSN 1045-9219. doi: 10.1109/TPDS.2012.24.

[11] Vincenzo Gulisano. *StreamCloud: An Elastic Parallel-Distributed Stream Processing Engine*. PhD thesis, Universidad Politécnica de Madrid, 2012.

[12] Vincenzo Gulisano, Magnus Almgren, and Marina Papatriantafilou. METIS: a two-tier intrusion detection system for advanced metering infrastructures. In *Proceedings of the 5th international conference on Future energy systems*, pages 211–212. ACM, 2014. URL `http://dl.acm.org/citation.cfm?id=2602072`.

[13] Vincenzo Gulisano, Magnus Almgren, and Marina Papatriantafilou. Online and scalable data validation in advanced metering infrastructures. In *Innovative Smart Grid Technologies Conference Europe (ISGT-Europe), 2014 IEEE PES*, pages 1–6, October 2014. doi: 10.1109/ISGTEurope.2014.7028740.

[14] IBM. IBM InfoSphere streams: Programming model and language reference, version 1.2.1. Technical report, IBM Corp., 2010.

[15] Chris King and Dan Delurey. Advanced metering: Policy-makers have the ball. *Public Utilities Fortnightly*, 140(17):26–30, September 2002. ISSN 10785892. URL `http://search.proquest.com.proxy.lib.chalmers.se/docview/213205397?pq-origsite=summon`.

[16] Stephen Cole Kleene. Representation of events in nerve nets and finite automata. Technical report, DTIC Document, 1951. URL `http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA596138`.

[17] Jay Kreps, Neha Narkhede, Jun Rao, and others. Kafka: A distributed messaging system for log processing. In *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece*, 2011. URL `http://research.microsoft.com/en-us/UM/people/srikanth/netdb11/netdb11papers/netdb11-final12.pdf`.

[18] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 311–322. ACM, 2005. URL `http://dl.acm.org/citation.cfm?id=1066193`.

[19] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. Out-of-order processing: a new architecture for high-performance stream systems. *Proceedings of the VLDB Endowment*, 1(1):274–288, 2008. URL `http://dl.acm.org/citation.cfm?id=1453890`.

[20] Mo Liu, Ming Li, Denis Golovnya, Elke A. Rundensteiner, and Kajal Claypool. Sequence pattern query processing over out-of-order event streams. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 784–795. IEEE, 2009. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4812454`.

[21] B. Lohrmann and Odej Kao. Processing smart meter data streams in the cloud. In *2011 2nd IEEE PES International Conference and Exhibition on Innovative Smart Grid Technologies (ISGT Europe)*, pages 1–8, December 2011. doi: 10.1109/ISGTEurope.2011.6162747.

[22] T. Mikkola, E. Bunn, P. Hurri, G. Jacucci, M. Lehtonen, M. Fitta, and S. Biza. Near real time energy monitoring for end users: Requirements and sample applications. In *2011 IEEE International Conference on Smart Grid Communications (SmartGridComm)*, pages 451–456, October 2011. doi: 10.1109/SmartGridComm.2011.6102365. 00005.

[23] C. Mutschler and M. Philippsen. Distributed low-latency out-of-order event processing for high data rate sensor streams. In *2013 IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1133–1144, May 2013. doi: 10.1109/IPDPS.2013.29.

[24] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177. IEEE, 2010. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5693297`.

[25] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *9th IEEE/ACM International Symposium on Cluster Computing and the Grid, 2009. CCGRID '09*, pages 124–131, May 2009. doi: 10.1109/CCGRID.2009.93.

[26] Robert W. Richardson. Metering migration. *Energy Markets*, 6(9):64, September 2001. URL `http://search.proquest.com.proxy.lib.chalmers.se/docview/228777826?pq-origsite=summon`.

[27] Yogesh Simmhan, Baohua Cao, Michail Giakkoupis, and Viktor K. Prasanna. Adaptive rate stream processing for smart grid applications on clouds. In *Proceedings of the 2Nd International Workshop on Scientific Cloud Computing*, ScienceCloud '11, pages 33–38, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0699-7. doi: 10.1145/1996109.1996116. URL `http://doi.acm.org/10.1145/1996109.1996116`.

[28] Utkarsh Srivastava and Jennifer Widom. Flexible time management in data stream systems. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 263–274. ACM, 2004. URL `http://dl.acm.org/citation.cfm?id=1055596`.

[29] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 34(4):42–47, December 2005. ISSN 0163-5808. doi: 10.1145/1107499.1107504. URL `http://doi.acm.org/10.1145/1107499.1107504`. 00274.

[30] The Apache Software Foundation. Storm, distributed and fault-tolerant realtime computation, December 2014. URL `http://storm.apache.org/`.

[31] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2376-5. doi: 10.1145/2588555.2595641. URL `http://doi.acm.org.proxy.lib.chalmers.se/10.1145/2588555.2595641`.

[32] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. Exploiting punctuation semantics in continuous data streams. *Knowledge and Data Engineering, IEEE Transactions on*, 15(3):555–568, 2003. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1198390`.

[33] Daniel Warneke and Odej Kao. Nephele: efficient parallel data processing in the cloud. In *Proceedings of the 2nd workshop on many-task computing on grids and supercomputers*, page 8. ACM, 2009. URL `http://dl.acm.org/citation.cfm?id=1646476`.

# A

# Regular expressions

A concept that arose in the 1950s during research into finite automata [16], the term regular expression describes a character string used to define a search pattern. A regular expression applied to a text string results in the sub-strings that were matched by it.

The most simple regular expression is a literal text string. For example, the expression `Regular expressions` will match that exact string. To extend the range of patterns that can be expressed, there are some meta characters that have a special meaning:

| | |
|---|---|
| `.` | A dot matches any single character. Ex. `h.y` matches “hey”, “hay”, “hoy” etc. |
| `[ ]` | Brackets match a single character out of any of the ones found inside. Ranges, like `a-z` can also be used. Ex. `h[ae]y` matches only “hey” and “hay”. |
| `[^ ]` | This is an inverse of the bracket that matches any character not found inside. Ex. `h[^ae]y` matches “hby”, “hcy” etc. but not “hay” or “hey”. |
| `^` | Matches the start of the string. Ex. `^s` matches the ‘s’ in “start”, but not in “A start”. |
| `$` | Matches the end of the string. Ex. `d$` matches the ‘d’ in “end”, but not in “ends”. |
| `?` | Matches the element before zero or one time. Ex. `ha[dy]?` matches “ha”, “had” and “hay”. |
| `*` | Matches the element before zero or more times. Ex. `ha*` matches “h”, “ha”, “haa”, etc. |
| `+` | Matches the element before one or more times. Ex. `ha+` matches “ha”, “haa”, “haaa” etc. but not “h’. |
| `{m,n}` | Matches between *m* and *n* occurrences of the preceding element. Ex. `h{2,4}` matches “hh”, “hhh” and “hhhh”. |
| `( )` | Parenthesis signify subexpressions. Ex. `a(ha)?` matches “aha” or “ahaha”. |
| `\|` | A vertical bar is used to match one of two expressions. Ex. `a\|b` matches “a” or “b”. |
| `\` | The backslash character is used to escape any of the meta characters for use in a literal string. |

Using these characters, very complex patterns can be described by regular expressions.