CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

# Streaming Analytics with Provenance in the Advanced Metering Infrastructure

Master's thesis in Computer science and engineering

Johan Taube & William Johnsson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

# Streaming Analytics with Provenance in the Advanced Metering Infrastructure

Johan Taube
William Johnsson

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Streaming Analytics with Provenance in the Advanced Metering Infrastructure



Supervisor: Vincenzo Massimiliano Gulisano, Department of Computer Science and Engineering
Co-supervisor: Dimitris Palyvos-Giannas, Department of Computer Science and Engineering
Advisor: Joris van Rooij, Göteborg Energi
Examiner: Marina Papatriantafilou, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2022

Streaming Analytics with Provenance in the Advanced Metering Infrastructure

Johan Taube
William Johnsson
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

# Abstract

The Advanced Metering Infrastructure (AMI) allows electricity to be monitored and billed with fine-grained resolution through the use of smart meters, whose automated measurement results in vast amounts of generated data. Traditionally, this is handled by temporarily storing the sensed data in databases and then performing some analysis, for example, with the goal of finding faulty equipment. After some time, when the data are no longer deemed useful, it can be discarded to make room for more up-to-date measurements.

The amount of data is expected to increase further in the near future due to trends in the industry and national regulations. This presents a challenge to utility companies as database storage becomes more expensive. Motivated by this challenge, utilities can greatly reduce storage requirements by leveraging the stream processing paradigm to analyze unbounded and continuous streams of meter data as soon as it arrives. However, discarding too much data can be counterproductive, since maintaining the associated source data of an analysis result is beneficial to understanding its cause through further analysis. This is enabled by fine-grained data provenance, which connects each detected event with the source data that contribute to it.

Provenance captures have an intrinsic computational overhead associated with them. This thesis aims to give insight into what type of streaming analysis is feasible in the AMI using a state-of-the-art provenance capture. This is done by implementing and evaluating the performance of a number of streaming queries with and without provenance. The thesis was done in close cooperation with the Swedish utility company Göteborg Energi and therefore the queries use historical data from a real-world AMI.

The results show varied performance on a per-query basis, generally displaying an increase of latency between 24–42% and a decrease in throughput between 22–34% compared to its non-provenance counterparts. However, one query shows a more significant degradation, where the overhead reaches above 200% in latency and memory consumption. Despite the overhead, stream processing with provenance is a viable alternative for analysis in the AMI because it has the potential to greatly reduce storage requirements of meter data.

Keywords: stream processing, data provenance, AMI.

# Acknowledgements

We would first like to thank Joris van Rooij, our supervisor at Göteborg Energi, for welcoming us to his team and always providing timely and valuable feedback. We also want to thank Vincenzo Gulisano and Dimitris Palyvos-Giannas for their commitment and expertise which greatly enhanced the quality of the thesis.

Johan Taube and William Johnsson, Gothenburg, June 2022

# Contents

# List of Figures

# List of Tables

# 1

# Introduction

The modern electricity grid is progressing both thanks to new technology as well as new laws and regulations. One aspect of this is the increasing rate at which consumption measurements are taken, which gives consumers the option to pay for their usage based on the supply and demand at a certain time of day. This is enabled by the Advanced Metering Infrastructure (AMI), which consists of networked metering equipment and aggregators that yield large volumes of power consumption, current, and voltage measurement data. The consumption data are used for both billing purposes, and to analyze patterns and properties of the meter, whereas current and voltage are used only for analysis [1].

Power distribution networks experience technical and non-technical losses of energy [2]. The former is primarily due to natural phenomena such as power dispersion in electrical components. On the other hand, non-technical losses often occur as a result of faulty metering equipment or electricity theft. Data from the metering infrastructure can be used to detect such losses through consumption [3] or frequency analysis [4] of smart meters. However, the use of AMI data is not limited to detecting faults and losses. From a consumer perspective, it can enable one to view fine-grained energy consumption data through a web interface or a mobile application. This personal data may also be intercepted to gain intrusive knowledge about the consumer [5].

Non-technical losses generally result in energy being delivered but not billed. These losses, if left unnoticed, lead to a loss of profits [2], and the cause can potentially be dangerous or expensive to repair [1]. Thus, it is important for utility companies to continuously monitor the meter data in order to identify these losses.

The monitoring process becomes expensive when handling the large amounts of data generated in the AMI. Data storage especially becomes an issue, since storing all the generated data becomes unfeasible due to the immense required storage space. Expenses in the monitoring process can be reduced by only storing data that show unexpected behavior, which means that data should be processed as fast as possible to minimize storage time.

A promising application is to leverage the collective computational capacity of the AMI through a distributed streaming solution [6]. Stream processing allows for

continuous processing of input without the need to store data in databases. It also provides support for designing systems where data is analyzed in a distributed and parallel fashion [7].

In analysis applications that are designed to detect anomalies in sensed data, it is beneficial to save the input data that contributed to a certain output [8]. This allows for understanding the cause of the problems, detecting injections of malicious data [9], replaying the queries, or further analyzing source data using machine learning [10]. However, storing all the incoming data is unfeasible, since the needed storage space would be prohibitively large and expensive. An alternative is to only store data which contributes to any events, making it possible to discard large parts of the incoming data as soon as possible, thus significantly decreasing storage needs. This is enabled by *data provenance*, which in the context of stream processing connects each output event with the input data that caused it. However, provenance is in its essence added meta-data, and thus always brings an additional overhead. In order for provenance to be practically applicable in streaming queries in the AMI, this overhead must not be prohibitively large. Otherwise, the upkeep of provenance might steal significant computational resources, thus limiting the performance of the actual analysis.

A brief explanation of the aim of this work will be provided in the following section. However, to understand a detailed description of the problem, one must first understand the concepts of the AMI, stream processing, and data provenance. Thus, a more detailed problem definition is given in Chapter 3.

## 1.1 Thesis aim

This thesis aims to apply stream processing to the AMI to classify whether smart meter data exhibits anomalous or typical behavior. Furthermore, the goal is to enrich the processing with data provenance collection as a method to reduce storage needs by only storing the data that are considered anomalous. To this end, the aim is to evaluate such a system with customer data obtained from Göteborg Energi, a regional public utility.

## 1.2 Outline

To guide the reader throughout the thesis we present here a short summary of each chapter, which together form a familiar structure:

**Chapter 2, Preliminaries** describe the common operation and components of the AMI and basic stream processing concepts such as tuples, operators, and queries. This is followed by a definition of data provenance including how it applies to stream processing, and finally an assoicated problem of changepoint detection.

**Chapter 3, Problem Definition** first gives a background to the type of system our methods apply, then the problem studied is detailed including the motivation

behind it. We then present a use case for the problem in the industry and lastly, we list the performance metrics for evaluating a solution to the problem.

**Chapter 4, Design and Implementation** describes the data we are working with for evaluation, the design of the streaming analysis targeting different AMI faults, and finally the method of streaming provenance capture.

**Chapter 5, Evaluation** describes the evaluation environment in terms of hardware and software configuration, more specifics about the evaluation data, and the metrics observed for the streaming analysis.

**Chapter 6, Discussion** looks at the results as a whole, and we reason about how the properties of stream processing affect the observed metrics, especially when used in conjunction with provenance capture. We discuss the limitations of the work and where applicable, how this gives opportunities for future work.

**Chapter 7 Related works** introduces previous work dealing with smart meter data through stream processing, provenance capture in streaming, and finally work related to finding sudden changes in energy consumption.

**Chapter 8, Conclusion** completes the thesis by summarizing key findings.

# 2

# Preliminaries

This chapter provides detailed descriptions of various concepts and terminology used in this thesis. Starting with Section 2.1, the data collection system in the power distribution network is explained. The stream processing paradigm, and its components are defined in Section 2.2. Section 2.3 then describes data provenance and why it is interesting to collect such information. Finally, changepoint detection is described in in Section 2.4, which can be used to find faults in the AMI.

## 2.1 Advanced Metering Infrastructure

In order to bill customers for the electric energy they consume, utility companies use meters connected to the customers' facilities that measure their consumption. In Sweden, before 2009 [1], the reading of meters was done by a technician, e.g. yearly, but as technology advanced, the possibility of deploying computerized *smart meters* emerged [11]. These meters are part of a networked computer system that measures, collects, and analyzes data from the electricity distribution network. This system is called *Advanced Metering Infrastructure* (AMI), a wide-spread term used not only by Göteborg Energi [12]. Although the AMI may include other utilities such as gas, water, or district heating, this thesis focuses only on data related to electricity network.

Starting from January 2025, all smart meters in Sweden must be able to produce an update every 15 minutes due to new legal requirements [13]. Compared to previous regulations, which only required hourly reports, this gives a four-fold increase in the amount and rate of data that need to be processed. Furthermore, as meters get replaced they are often not only upgraded in terms of their updating interval but also increase the number of parameters they can measure.

Modern smart meters not only assess energy consumption, but also a variety of other measurements. These include both the production and consumption of active and reactive power, as well as voltage, current, and phase angle. These additional measurements are used for equipment maintenance rather than billing purposes [1].

### 2.1.1 AMI Hierarchy

The AMIs considered in this thesis can be seen as separated into a hierarchy of three distinct layers of components as seen in Figure 2.1. The lowest layer consists of individual smart meters. These do not perform any resource-intensive computations. Instead, their main purpose is to collect and transmit measurement data to the second layer consisting of *Collector units*. These request smart meter measurements at regular intervals over a wireless radio network or power line communication [6]. The final layer consists of the *Central servers* located in the utility or in some third-party cloud provider. These servers can perform heavy calculations and collect measurements from the entire network [12].

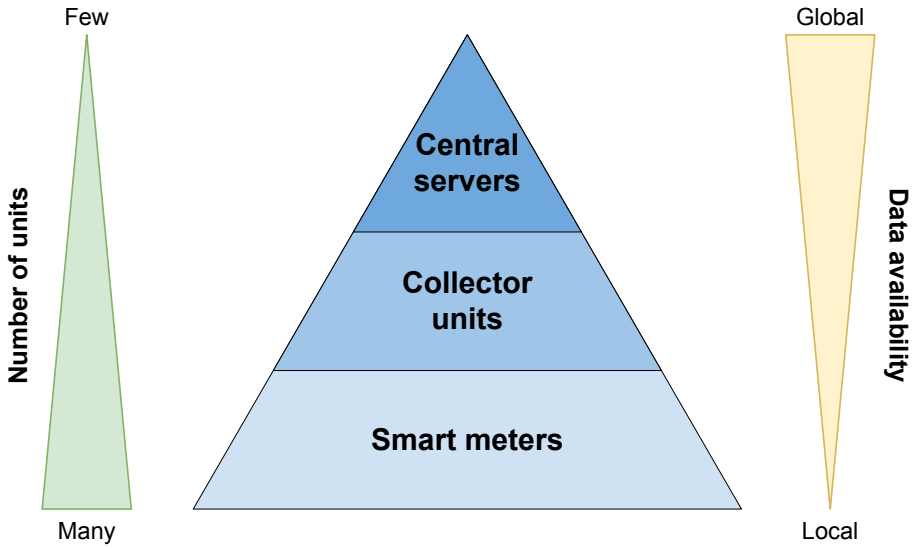



**Figure 2.1:** The hierarchy of networked components in the type of AMI studied in this thesis. Each smart meter communicates with a collector unit, which in turn communicates with central servers.

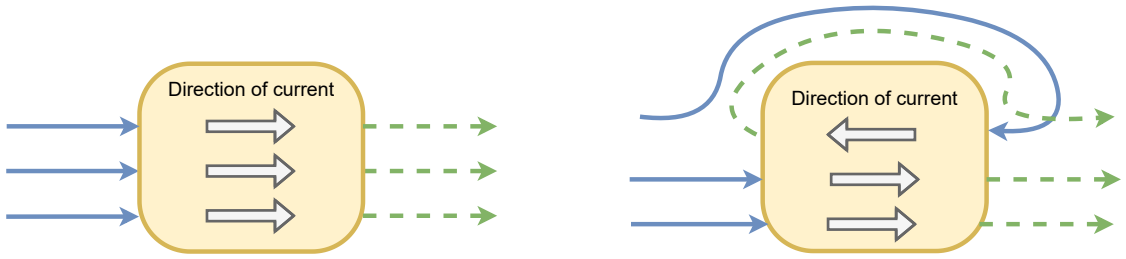
The amount and diversity of data available depend on the layer in which the process is run. In the lowest layer, the smart meters have access only to their own data. In the middle layer, a collector unit has access to data from all the smart meters that report to it. In Göteborg Energi's network, this is usually between 0–300 smart meters. Lastly, the central servers have access to all data and are therefore traditionally where the main analysis processes are run [1, 12].

With the number of components in the AMI, there is a significant risk of faulty hardware. However, some faults are related to the human factor, when technicians are installing or replacing smart meters.

### 2.1.2 Detecting faulty installations in the AMI

When installing a new smart meter or replacing a faulty one, an erroneous installation can cause subsequent readings to have erroneous values. More specifically, there is a risk that some wires are inadvertently reversed when installing the new

meters, as can be seen in Figure 2.2. The phenomenon of reversed wires through electrical meters has been well studied in the field of energy theft, since it makes the energy meter report a lower consumption. If the wires are reversed, the meter will consider the customer's consumption to be production and its production to be consumption. A customer will thus effectively get paid to consume energy. Therefore, some adversaries have resorted to purposefully reversing the wiring through their own meter in order to decrease their utility bill [14].




**(a)** A correctly installed meter. **(b)** An incorrectly installed meter.

**Figure 2.2:** Example of a reversed wire in a common three-phase meter. Whereas (a) is installed correctly, one of the wires in the meter shown in (b) is reversed, which also reverses the current for that phase.

The symptoms of such a wire reversal in a meter are two-fold:

1. Wires are disconnected while the reversal is being made
2. Lower measured energy consumption

Reversed wires can be detected in a number of different ways. First, the previous literature focuses mainly on detecting the first symptom. This is a useful metric when standing against a malicious adversary. However, when the switch is made by a technician who accidentally reversed wires when installing a new meter, we expect to see disconnected wires.

A second simple solution has been to forbid energy production in customer meters. This way, when wires are reversed, the meter will detect the production of electricity instead of consumption and can send an alarm. However, as micro-production of electricity at consumers becomes more widespread (for example, the installation of solar panels), selling energy back to the grid is more common. Therefore, detecting production at a consumer is no longer indicative of a fault.

This thesis will explore a third method that aims to detect the presence of the second symptom: lower energy consumption. This is performed by applying a changepoint detection algorithm, which will be described in Section 2.4. This algorithm can be implemented as continuous analysis of consumption data, enabled by stream processing.

## 2.2 Stream processing

To introduce the concept of stream processing, it is helpful to first define what constitutes a stream. In this context, a stream is a continuous flow of data called *tuples* (or *events*) sharing a common schema of $n$ attributes $a_1, a_2, \ldots a_n$ as well as a timestamp $\tau$. Thus, a tuple $t$ can be concisely denoted as

$$t = \langle \tau, a_2, a_3, \ldots, a_n \rangle. \tag{2.1}$$

An example of a stream is the smart meter readings produced in the AMI where each reading is a tuple that contains a timestamp of when the measurement was taken by the meter and attributes such as the meter ID, energy consumption, or voltage.

The stream processing paradigm is implemented in frameworks known as Stream Processing Engines (SPEs), designed to process streams in parallel and on separate nodes in a distributed system, allowing large volumes of data to be handled with low latency. Aurora [15] is one of the first SPEs developed, and other pioneering works include StreamCloud [16] and Borealis [17] as well as state-of-the-art open-source Apache Storm [18] and Apache Flink [7]. Additionally, commercial platforms have been developed, such as Google Dataflow, Amazon Kinesis, and IBM Infosphere [19] as alternatives to self-hosted SPEs.

SPEs provide programming libraries to aid in writing *streaming queries* (or simply queries) in which streams are processed. The queries are composed of *Sources*, *Operators*, and *Sinks*. The Source produces tuples that are ingested by one or more operators, which in turn produce output tuples. Together, the operators form a Directed Acyclic Graph (DAG) through which tuples flow, are transformed, and are eventually delivered to one or more Sinks. As we later show in Chapter 4, such queries can be constructed to implement analysis of smart meter data.

Stream processing revolves around the idea of unbounded data streams. However, for many applications it is critical to maintain state [7]. To this end, SPEs need to support operators whose output refers to portions of the processed input tuples. Before discussing operators in greater detail we therefore show how this portioning is achieved in stream processing.

### 2.2.1 Windowing

The separation of a stream into finite chunks of tuples is done by a mechanism called *windowing*, whose specific function depends on the *window model*, of which there are many [16], although the discussion that follows is limited to the ones relevant to this thesis. Three models can be defined by a combination of the parameters *Window Size* (WS) and *Window Advance*. The interpretation of the parameters depends on the *type* of window:

- *Count-based* windows are bounded by a defined number of tuples WS, which fit inside them. WA refers to the number of tuples between the start of two

consecutive windows.

- *Time-based* windows are bounded by a defined length of time WS, and WA is the difference in time between the start of two consecutive windows.

From now on, in this thesis, windows will always refer to time-based windows, unless otherwise stated. With this in mind, we say that a tuple $t$ *falls* in a window starting at time $x_1$ and ending at time $x_2$ when $x_1 \leq t.\tau < x_2$ [20]. We also say that a window is started when the first tuple that falls in it arrives and closes when time has advanced enough for no subsequent tuples to fall into it.

Three of the window models commonly seen in SPEs differ in the relation between WS and WA:

- WA = WS. Known as a *Tumbling* window, where a consecutive window is shifted by WS time units, meaning each tuple will only fall into exactly one window.

- WA < WS. Known as a *Sliding* window, where consecutive windows overlap such that a tuple can fall into one or more windows.

- WA > WS. In this case, some tuples may be discarded altogether if they happen to fall in the gaps between windows (gaps of size WA $- WS$). This effectively creates WS-sized samples of the stream.

Finally, *Session* windows are designed to capture and segment different tuples into sessions of activity. An example of this window model is illustrated in Figure 2.4. In contrast to the previously mentioned window models, session windows are not defined by WS and WA. They are instead defined by a *Session Gap* (SG), a predefined amount time that must pass without receiving any new tuples before the window closes. As a result, Session windows do not overlap and do not have a fixed start and end time.

Some operators only apply to bounded amounts of data, therefore a programmer is required to first specify which window model to use and assign values to the parameter(s) of the model. These operators, as well as others that do not utilize the windowing mechanism, are explained in detail in the next section.

### 2.2.2 Operators

Operators process tuples in a query and can be categorized into two groups. The first being *stateless* operators, where tuples are processed independently of preceding tuples. The second being *stateful* operators, where multiple tuples can contribute to the output. Most SPEs support user-defined operators, but also have a number of native operators in both groups. All operators, and their representation in subsequent figures, can be seen in Figure 2.5. Among these are two stateless operators, *Filter*, and *Map*.

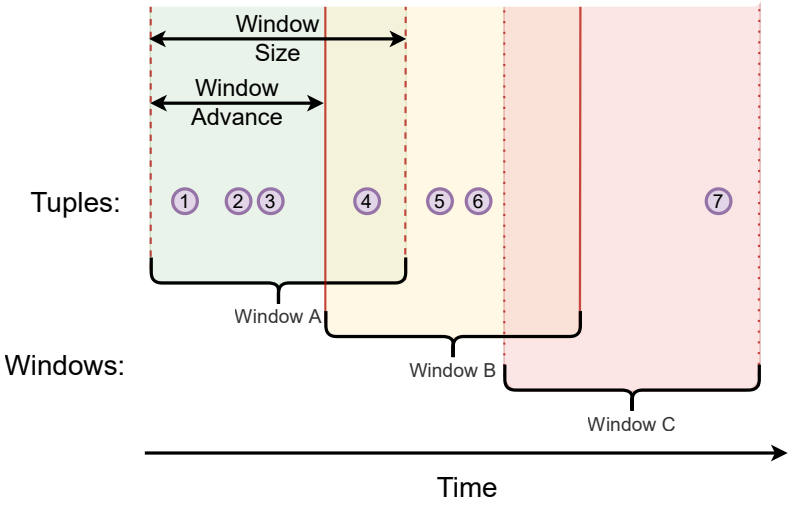



**Figure 2.3:** Example of stream processing using Sliding window aggregation. The windows are defined over the stream by the parameters WS and WA. Tuples 1 through 4 fall in Window A, 4 through 6 in Window B, and event 7 in Window C.

- *Filter* drops a subset of input tuples based on a Boolean condition.
- *Map* produces one or more output tuples based on a user-defined function applied to the attributes of the input tuples. Note that in some SPEs, a differentiation is made between Map, where one input tuple results in exactly one output tuple, and *FlatMap*, the output of which is an arbitrary number of tuples (i.e., zero or more) for each input tuple.

Native operators belonging to the group of stateful operators include *Aggregate* and *Join*.

- *Aggregate* combines all the tuples in the current window into a single output. For example by calculating a running average of a certain attribute of the tuples.
- *Join* produces new tuples by combining tuples from two streams based on a user-defined condition. For example, it is possible to join a stream of data from smart meters with a separate stream of dates when meters are changed. The output can then be defined as a combination of both tuples, holding both measurements from the smart meter, as well as data about the change of that meter.

For the stateful operators, one can optionally define an equivalence class using one or more tuple attributes. In this case, a separate window is maintained for each combination of attribute values found when processing the operators. For example, assuming smart meters can be uniquely identified, an Aggregate operator can be used to get the average consumption reported by each individual meter over a week

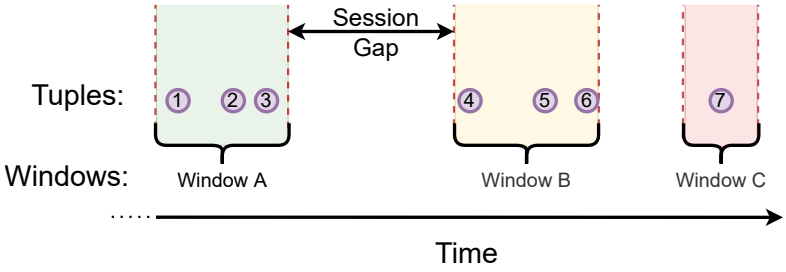



**Figure 2.4:** Examples of Session windows. Once SG time units have passed without the arrival of any new tuple, the window is closed. If any new tuples arrives thereafter, a new window is created. Tuple 1 through 3 fall in window A, 4 through 6 in window B, and tuple 7 falls in Window C.

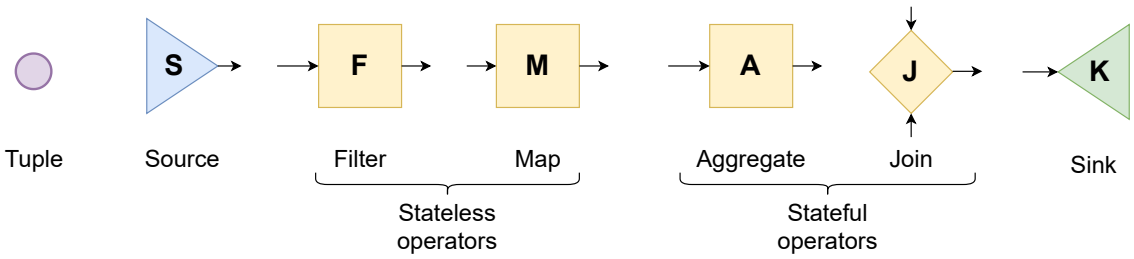



**Figure 2.5:** Tuples, Sources, Sinks, and standard Operators, as they are represented in other figures throughout the thesis.

of readings. In Apache Flink, for example, the equivalence class is referred to as *key* and is similar to the group-by statement in relational databases. Therefore, we use the term *key-by* (or just KB) throughout this thesis.

**An example query.** To give more intuition for the concepts covered in this chapter so far, we present an example streaming query that could be applied in an AMI. The query is illustrated in Figure 2.6.

The smart meters in this example simply report a time-stamped energy consumption (in kWh) value each hour and can be uniquely identified by an integer `id`. This is used in the first Filter that drops tuples $t$ not within a specified range $t.id \notin (x, y)$. Imagine that this ensures that only tuples from private households are processed by the rest of the query. An Aggregate groups by `id` to calculate the mean consumption per hour with a Sliding window using a window size of seven days and an advance of one day. Note how the attributes of the tuples before and after the Aggregate changes, the usage is replaced by the average. Only households with a high consumption are interesting; therefore, a Filter only forwards tuples that average above 10 kWh per hour for a week. Finally, a Map transforms the unit from kWh into Sink tuples with MWh before its delivered to the Sink.

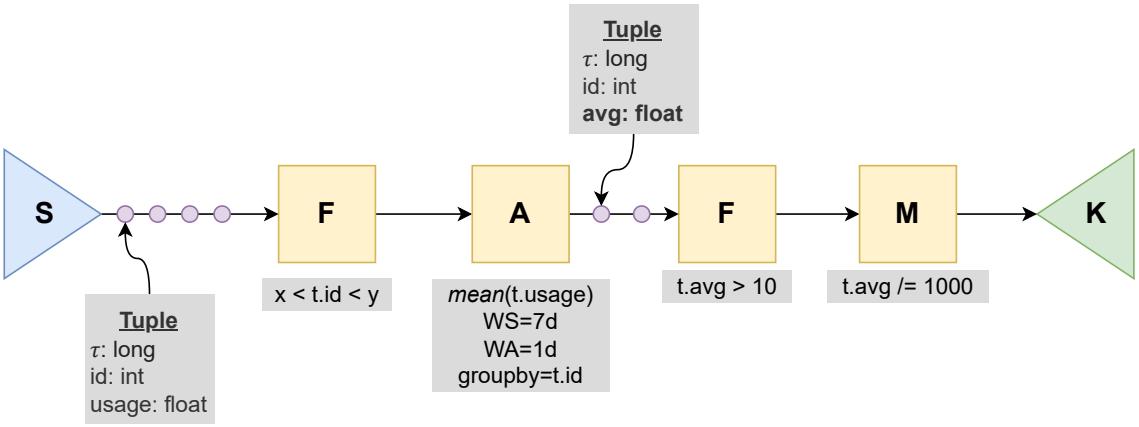



**Figure 2.6:** DAG of an example AMI streaming query. The attributes of source tuples and tuples after the Aggregate are highlighted to show how the state of a tuple may change at intermediate points in a query.

### 2.2.3 Time in stream processing

How tuples are handled in windows depends on their timestamp. There are generally two different ways of assessing time in stream processing. An example of how they differ can be seen in Figure 2.7.

- *Event* time is the moment when the event occurred. This time is extracted from an embedded timestamp in the data. In the case of the AMI, this is predominantly when a value was sensed by a smart meter.
- *Processing* time is the time at which a calculation is done. This time is the system time of the server where the query is running.

Note that this means that event time and processing are rarely synchronized. While processing time always advances as the internal clock of the server progresses, event time can jump almost arbitrarily according to when the data was handled. When data arrives out of order, their timestamps will also be out-of-order. Furthermore, replaying old data, or receiving delayed data, can generate tuples with timestamps far from the past.

The distinction is necessary not only to be able to ensure reproducibility regardless of the current system clock, but also because of the time that passes while the data is in transit to the central servers. In this thesis, a timestamp will always refer to event time, unless otherwise stated.

## 2.3 Data provenance

Provenance is essentially meta-data that describe a production process of an end product. Provenance collection (also known as capture) and processing is critical in a variety of situations, such as assessing quality, ensuring repeatability, or reinforcing trust in the end product [21]. To narrow down the definition, provenance capture can

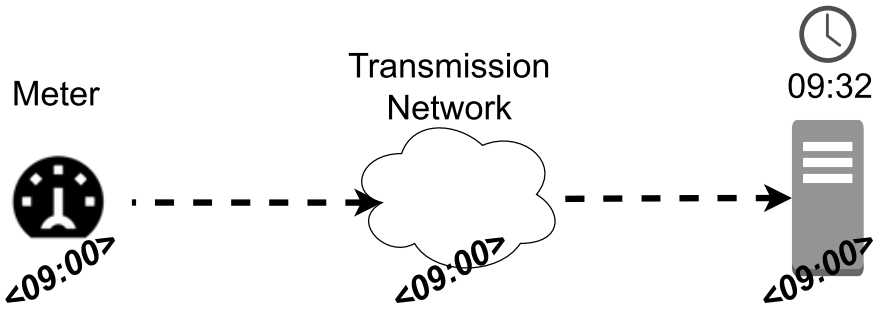



**Figure 2.7:** An example of how *event* and *processing* time differs. An event is created when the local clock of the meter is *09:00*. When it arrives the local clock of the server is *09:32*. Therefore, the *event time* would be *09:00*, while the *processing time* would be *09:32*.

be classified into a hierarchy of categories. In one of such classifications, presented by [21], the most specific class of provenance capture is *data provenance* where individual data items make up the provenance itself, e.g. the source tuples in a stream. This level of provenance spans more than three decades of research [22]. More recently, it has been studied in the context of stream processing [17, 8, 20] where provenance is in the form of source tuples.

A more precise definition of provenance in stream processing is given in [8], called *fine-grained data provenance*. It is based on the transitive contributes-relation between source and sink tuples, which is defined below for the operators introduced previously in Section 2.2.2.

**Definition 1.** We say that the input tuple $t_{IN}$ to an *operator OP contributes* to an output tuple $t_{OUT}$ if:

i $OP$ is a Filter and $t_{OUT} = t_{IN}$

ii $OP$ is a Map and $t_{OUT}$ is created upon the processing of $t_{IN}$

iii $OP$ is an Aggregate and if $t_{IN}$ is in the window of tuples whose aggregation results in the creation of $t_{OUT}$

iv $OP$ is a Join and $t_{OUT}$ is a tuple from a pair of tuples within time distance WS.

For each sink tuple $t_{SINK}$ produced by a query, a solution to fine-grained data provenance must associate all the source tuples that contribute to $t_{SINK}$ [8].

Provenance capture is by nature a computationally heavy operation, and a challenge is to make a solution to fine-grained data provenance lightweight enough to run on large amounts of data without causing too much overhead for distributed deployments [8]. How this can be achieved in SPEs is described in Appendix A.1.

## 2.4 Changepoint detection

To detect the faults introduced by faulty meter installations described in Section 2.1.2, studying the problem through the lens of changepoint detection is compelling. The reason being that changepoint detection involves searching for one or more points in time where a process generating time series data transitions between different states. This is especially useful in industrial systems where data is collected and continuously monitored with sensors [23].

More formally, changepoint detection can be applied to time series whose characteristics can change at unknown instants of time $t_1 < t_2 < \cdots < t_K$, where the goal is to estimate these instants. Depending on the domain, the number of $K$ changes could be unknown, in which case $K$ needs to be estimated as well. For the problem of defective meters studied in this thesis, $K$ is known. In fact, we have $K = 1$, corresponding to a single meter change. With this in mind, changepoint detection can be defined as an optimization problem, where the goal is to choose the best segmentation of a data stream that minimizes a sum of costs for all $K$ segments that define the segmentation. Therefore there are two choices to be made for a changepoint detection algorithm depending on preliminary knowledge of the task at hand, firstly the *cost function* and secondly the *search function*.

### 2.4.1 Cost function

Each segment is associated with a cost, which is computed by a *cost function* $\mathcal{C}$, which indicates the homogeneity of a segment. A good cost function for a given domain should mean that the lower the cost, the greater the indication that the segment is homogeneous. On the other hand, a high cost should indicate that the segment is heterogeneous.

As is illustrated in Figure 2.8, by placing a changepoint at a favorable location in the segment, the total cost can be decreased. The reason being that the sum of the cost of the two subsegments can be smaller than the cost of the original segment. The decreased total cost, also called *gain*, is denoted by $G$.

The gain of placing a changepoint at time $\tau$ in a segment spanning from $a$ to $b$ can be determined by evaluating Equation 2.2 [23].

$$G = \mathcal{C}(a, b) - (\mathcal{C}(a, \tau) + \mathcal{C}(\tau, b)) \tag{2.2}$$

Simply put, it is the gain achieved from splitting a large costly segment into two smaller, cheaper segments.

By evaluating the gain for all $\tau \in (a, b)$, a curve of gain values is obtained, as can be seen in Figure 2.10b. Choosing $\tau$ according to what gives the largest gain in the segment gives the best possible changepoint.

**(a)** Data points and their distances to their mean.

**(b)** A changepoint has been placed in the middle of the series.

**Figure 2.8:** An example of how a time series can be divided into two distinct parts, in order to lower their total *cost*, which in this case is the distance to the mean. By placing a changepoint in the middle of the series, the sum of distances to their respective means are minimized.

The cost function used in this thesis is the least-squared-error model, commonly referred to as *L2*. It is designed to detect mean-shifts in a time series. This makes it valuable for use in detecting the faults that can be induced at a meter change, where one phase is wired incorrectly and therefore erroneously record a lower consumption. The L2 cost function $\mathcal{C}_{L2}$ of the segment from $a$ to $b$, is defined as:

$$\mathcal{C}_{L2}(a, b) = \sum_{t=a+1}^{b} |y_t - \bar{y}_{a...b}|^2 \tag{2.3}$$

where $\bar{y}_{a...b}$ is the mean of the segment from $a$ to $b$. In other words, the cost is the sum of the squared distance from the mean, for all data points in the segment.

### 2.4.2 Search function

A *search function* is applied to search for an arbitrary amount of changepoints. There are two sub-groups of search functions [23]:

- *Optimal* search functions always returns exact solutions, however, they usually have a large computational complexity.
- *Approximate* search functions on the other hand, only produce an estimate, but instead boast a lower complexity making them more lightweight.

The search method used in this thesis is *binary segmentation*, an approximate function first proposed by [24]. It works by recursively slicing the stream at a detected changepoint, and then analyzing the two subsegments until all $K$ changepoints has been found, or when the maximum gain does not meed a required threshold. An example of how the method works is illustrated in Figure 2.9. Binary segmentation

is one of the fastest search methods available, with a complexity of $O(n \cdot log(n))$, where $n$ is the number of tuples in a subset of the data stream.

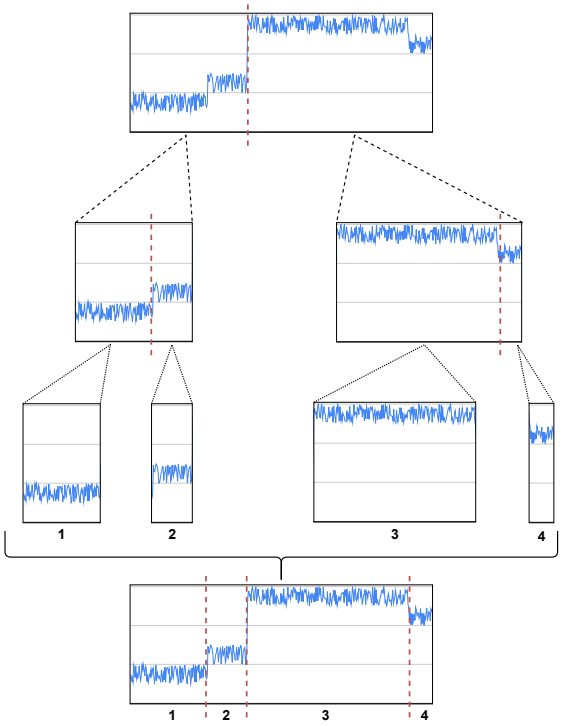
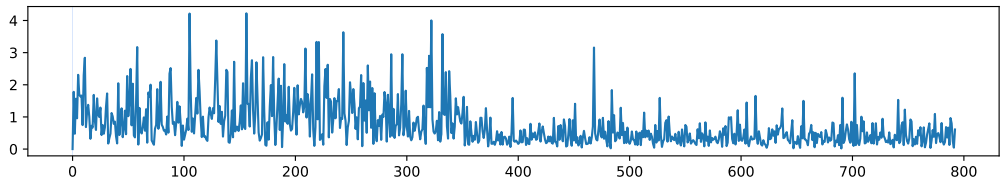


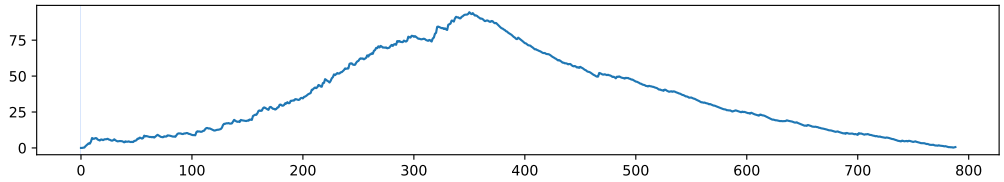

**Figure 2.9:** Two iterations of Binary Segmentation dividing a stream into four segments.

### Changepoint detection – an example

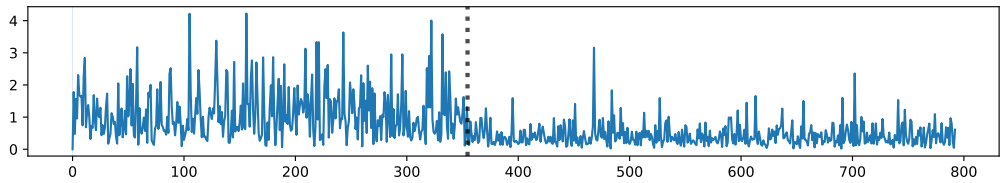
An example of how the changepoint algorithm works can be seen in Figure 2.10. Figure 2.10a shows the input to the algorithm. When processed by the cost function, a gain is calculated for each $t$, which can be seen in Figure 2.10b. In Figure 2.10c a changepoint is then placed at the point where the gain is maximized.




**(a)** The stream where the changepoint algorithm is applied. In this case it is generated time series data, drawn from an F-distribution with an induced changepoint.




**(b)** Gain for each time $t$ in (a). The maximum value occurs at $t = 352$.




**(c)** Changepoint marked with a dashed line in the input data at $t = 352$, where the gain function has its maximum value

**Figure 2.10:** The process of determining the placement of a changepoint.

# 3

# Problem definition

Now that the reader is familiar with the relevant preliminary concepts in the previous chapter, we will move on to discuss the problem in more detail. This means first introducing the the type of system to which our methods apply, the problem it aims to solve, the challenges faced, as well as the metrics on which performance will be assessed by.

## 3.1 System overview

The system considered in this thesis is an instantiation of an AMI of the type introduced in Section 2.1. It consists of data collection from smart meters and Collector Units which are fed to a time series database with limited storage, as can be seen in Figure 3.1. Due to its limited storage, the time series database is cleared of data at regular intervals. An SPE, in our case Flink, connects to this database to perform analysis and connects Sinks to another database intended for long-term storage.

This thesis focuses on the SPE part of the system, where algorithms need to be designed to find patterns in the data that require further analysis or action. For example, recurring faults reported by the analysis for one meter could mean that a technician needs to visit the customer to manually measure currents and voltages or replace the meter.

## 3.2 Streaming analytics with provenance

With increasing data volumes from smart meters, relying on a temporary time series database for analytics becomes increasingly cumbersome and expensive. Its storage requirements can be initially be mitigated by reducing the interval at which old data is discarded. However, this does not scale. One can bypass the database entirely and instead process data continuously, only keeping what is most important. The problem then is how to define which input data are important enough to validate long-term storage. One solution is provenance, where the analysis itself expresses which data to discard. In addition, the system must be expressive and performant enough for the type of analysis that is useful in an AMI. In this situation, stream

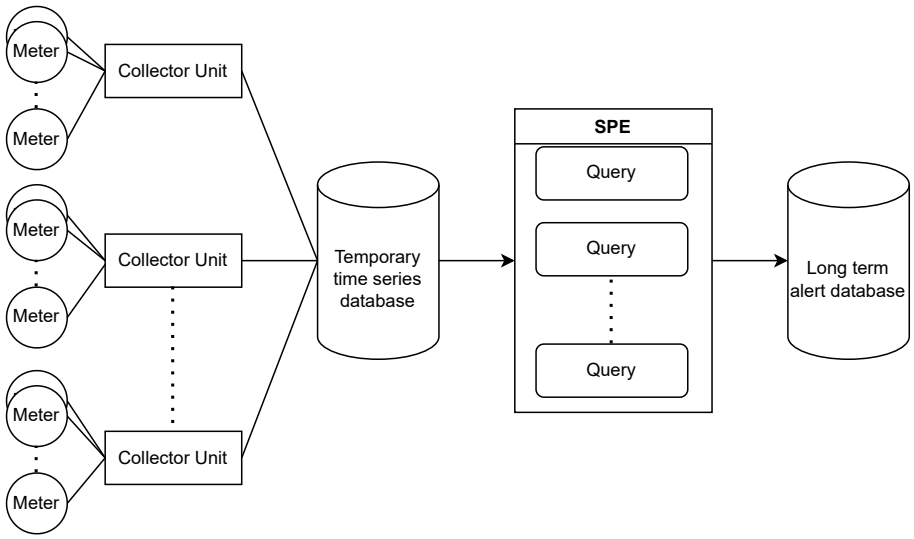



**Figure 3.1:** The system architecture studied in this thesis, including the data collection by smart meters and the stream processing engine responsible for analysis.

processing is advantageous, in part due to its support for continuous processing and to distribute the analysis over multiple machines.

Current state-of-the art provenance capture in stream processing shows that it has an acceptable overhead for a number of cyberphysical systems [8, 20]. However, its application in the industry has been limited. In order to determine whether provenance capture can be used in tandem with stream processing in industry, the overhead needs to be accurately measured.

### 3.2.1 Use case

As a use case to evaluate provenance in the system described in Section 3.1, this thesis is done in collaboration with the regional public utility company Göteborg Energi (GE). When the new laws [13] come into effect, GE will be compelled to replace all of its old smart meters. The new smart meters are capable of providing a reading every 15 minutes [25], increasing the rate of readings by a factor of four compared to the current system of hourly readings. Additionally, according to an expert at GE, the largest increase in data comes from the new properties that can be measured with smart meters. This combined increase of both the data rate and the amount of measured properties means that the total volume of data processed has the potential to increase by a factor of 100.

Due to the increased data volume caused by the new meters, GE is interested in examining the possibilities of using stream processing instead of the store-then-process paradigm currently used in their analysis. Further, in order to find the root cause of faults, GE also wishes to add provenance to their queries. However, the provenance collection must not incur too large overheads, otherwise it will hinder analysis altogether by consuming too much computational resources.

In order to have a baseline for evaluating provenance in the AMI, existing analysis queries obtained from a previous trial project at GE using stream processing need to be adapted and enriched with provenance collection. In addition to these queries, for a more thorough study, a more complex query that can detect faultily installed smart meters is to be implemented. Although this is a use case, the queries are applicable regardless of the specific architecture of the underlying power network, which makes them viable in a wide range of deployments.

## 3.3 Challenges

Applying streaming provenance to a real-world scenario with data from real customers introduces a number of challenges:

- *Fault source ambiguity*: Implementing streaming queries to detect faulty smart meters is a challenging task. Despite having access to hourly meter readings, one cannot easily distinguish if the source of anomalous consumption readings are due to customer behaviour or the result of a faulty smart meter. This means the memory overhead is likely to grow with provenance capture, since many source tuples need to be maintained by stateful operators to yield accurate results. Without provenance capture however, state can sometimes be reduced by incrementally updating an accumulator, such as maintaining a running average.

- *Limited SPE toolbox*: Not all operators in a modern SPE can be instrumented while still ensuring the solution to fine-grained data provenance is correct. As such, enriching queries with provenance collection means certain trade-offs must be made – queries may not always be implemented in the most straightforward way for the specific SPE.

- *Performance*: Collecting provenance involves an intrinsic overhead. Streaming queries must be carefully implemented with this in mind, to keep the overhead at a reasonable level. This facilitates the potential to run queries in a distributed fashion on hardware typically found in a cyber-physical system. A query with stateful operators that use windows involving hundreds of thousands of source tuples will quickly overburden main memory. On the other hand, the provenance of strictly stateless queries are of little use.

## 3.4 Performance metrics

To evaluate whether provenance is practically applicable in the AMI for analysis queries, the induced overhead will be evaluated. The performance of six queries will be evaluated, comparing the performance with and without provenance. This will be done by recording the following metrics:

- **Throughput** (or rate): the number of source tuples ingested by a query per unit of time. This metric is measured in tuples/second.

- **Latency**: the wall-clock time that passed between the production of a sink tuple after all contributing source tuples have arrived at a query.
- **CPU usage**: the percentage of the maximum available performance being utilized.
- **Memory consumption**: how much RAM is utilized by the SPE while running a query, measured in MB.

Streaming analytics with provenance is to be considered a feasible solution to the storage problem if the evaluation of these metrics show that there is no need to invest heavily in new hardware to support provenance capture for the analysis.

# 4

# Design and Implementation

In the introduction, we stated that data provenance is a useful property to have in the analysis of AMI data. In this chapter, we present the system to which this analysis applies and the input data available to streaming queries. Further, a number of streaming queries applied to the different data streams are described in detail. These queries have different properties that are later used for evaluating the overhead of collecting provenance in the streaming system.

## 4.1 Data streams in Göteborg Energi's AMI

The data processed by the analytics system at GE comes from an AMI consisting of around 282 000 smart meters. Each meter produces readings every hour, and multiple Collector units gather these over a wireless connection with each meter. The Collector units then forward a set of readings from multiple meters to the central servers and databases, as is described in more detail in Section 2.1.1.

The data collection system exposes three data streams summarized in Table 4.1. The stream with the highest rate of tuples is *Energy usage*, which provides hourly reports of energy consumption for each of the utility's customers. On the opposite end in terms of throughput is the *Meter changes* stream, which as the name suggests, contains events that represent changes (such as repairs or installations) performed by technicians.

An assumption about the data is that all tuples arrive in the order in which they were created by the smart meter. This is a reasonable assumption as long as stateful operators are keyed on the unique identifier of a meter. However, if for another use case the data were significantly out-of-order, [26] proposes a method to deal with this given that the data exhibit a bounded delay, which could be applied.

**Table 4.1:** The input streams available from smart meters in Göteborg Energi's energy distribution network. Note the different data rates of the streams.

| **Stream name** | **Interval (per meter)** | **Tuples/hour (entire AMI)** | **Stream description** |
|---|---|---|---|
| Energy usage | 1 hour | 300 000 | Hourly readings of energy consumption per customer |
| Voltage & current | approx. 12 hours | 25 000 | Snapshots of voltage and current for three phases |
| Meter changes | approx. 10-15 years | approx. 0.7 | Timestamps when a meter was put into service and when (if at all) it was retired from use |

**Table 4.2:** Four hours of meter readings from a single meter with example values, as it would look drawn from the Energy usage stream.

| **Facility ID** | **Series** | **Timestamp** | **Value** | **Type** |
|---|---|---|---|---|
| 42140193 | 42140193_E02 | 2021-03-02 09:00 | 0.90 | Measured |
| 42140193 | 42140193_E02 | 2021-03-02 10:00 | 0.43 | Measured |
| 42140193 | 42140193_E02 | 2021-03-02 11:00 | 0.57 | Measured |
| 42140193 | 42140193_E02 | 2021-03-02 12:00 | 1.07 | Measured |

To give an idea on the size of source tuples themselves we present the attributes of the *Energy usage* stream. Several example readings from this stream are shown in Table 4.2. Each customer has one meter and can be uniquely identified by their *Facility ID*. The *Series* attribute specifies how to interpret the number given in the *Value* attribute. In the example readings, we show the Series E02 where the *Value* column represents the customer's energy usage (in kWh) for the past hour. The *Timestamp* gives the date and hour when the reading was produced. Finally, the attribute *Type* specifies how the reading was obtained. Other than simply being the measured value, the reading might, for example, be interpolated between two different readings if measurements in between have been lost.

The attributes of another type of reading, drawn from the *Voltage & current* stream, are shown in Table 4.3. Each customer has a single meter that can be uniquely identified by its *Meter ID*. The *Timestamp* gives the date and time when the reading was produced. The non-metadata of the tuple are voltage ($V$) and current ($C$) for all phases in a three-phase meter (*P1*, *P2*, and *P3*).

**Table 4.3:** Four example tuples from a single meter with voltages V for each phase P as it would appear drawn from the Voltage & current stream.

| Meter ID | Timestamp | VP1 | VP2 | VP3 | CP1 | CP2 | CP3 |
|---|---|---|---|---|---|---|---|
| 5199 | 2021-02-19 03:06 | 232.8 | 233.1 | 234.5 | 0.0 | 0.0 | 0.3 |
| 5199 | 2021-02-19 20:06 | 227.6 | 229.3 | 232.0 | 5.4 | 0.8 | 0.6 |
| 5199 | 2021-02-20 03:06 | 231.0 | 231.5 | 232.4 | 0.0 | 0.0 | 0.3 |
| 5199 | 2021-02-20 20:06 | 233.7 | 233.9 | 234.3 | 0.0 | 0.0 | 0.3 |

## 4.2 AMI analytics queries

This section presents in greater detail the different queries used to evaluate the provenance system. The queries have different properties, in order to evaluate if and how these properties affect performance of provenance.

**Table 4.4:** Summary of queries enriched with provenance.

| Name | Input stream(s) | Analysis summary |
|---|---|---|
| `PeakUsage` | Energy usage | Find sudden peaks in energy usage compared to a running average. |
| `NegativeCurrent` | Voltage & current | Multiple consecutive tuples reported with negative current. |
| `NoVoltage` | Voltage & current | Multiple consecutive tuples reported with zero voltage |
| `BrokenMeter` | Voltage & current | Neither voltage nor current for a specific manufacturer's meter |
| `AverageDifference` | Energy usage, Meter changes | Compares average consumption before and after a meter change. |
| `Changepoint` | Energy usage, Meter changes | Applies changepoint detection algorithm around meter change |

### 4.2.1 PeakUsage

The `PeakUsage` query aims to detect sudden peaks in energy consumption. It starts with a Filter operator, which drops readings marked as invalid by the smart meter. This is followed by an Aggregate with a sliding window of three hours and an advance of one hour, grouping facilities based on their unique identifier. Essentially, the Aggregate computes the average consumption of the first two hours for each customer and adds an attribute for the consumption of the last tuple in the window. Lastly, a Filter forwards tuples that show consumption five times greater than the previous average.

As this phenomenon occurs often in the AMI, the query produces large amounts of sink tuples. It therefore serves as an example where the selectivity remains relatively high, meaning that not many source tuples are filtered out throughout the query.

This is useful for evaluation comparing to other queries to see how provenance is affected by this property. Additionally, the provenance per sink tuple is quite low, with only three contributing source tuples. The DAG of this query can be seen in Figure 4.1.

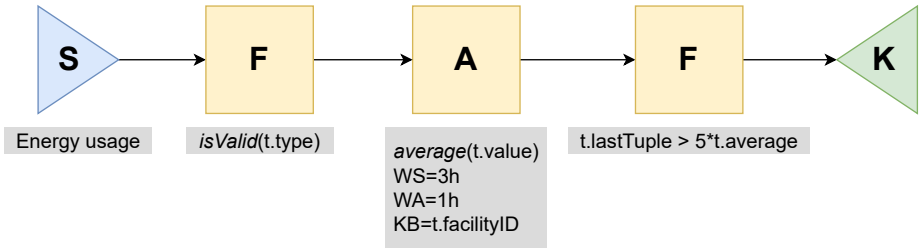



**Figure 4.1:** `PeakUsage` query.

### 4.2.2 NegativeCurrent

The `NegativeCurrent` query (Figure 4.2) is using the Voltage & current stream in order to detect facilities where the current flows consistently in the wrong direction through one of the meters phases. Since a single negative value is not always indicative of a fault, an Aggregate with a sliding window of size 6 days and an advance of 1 day is used. For each new Aggregate window, all phases are initially considered showing faulty behavior (negative current). When a new source tuple is ingested, phases with positive current are marked as non-faulty. If there are any faulty phases remaining when the window closes, an output tuple marked as valid is produced. Finally, a Filter drops invalid tuples (without faulty phases). Taking the window size and rate of the input stream into account, every output tuple will have up to 12 contributing input tuples.

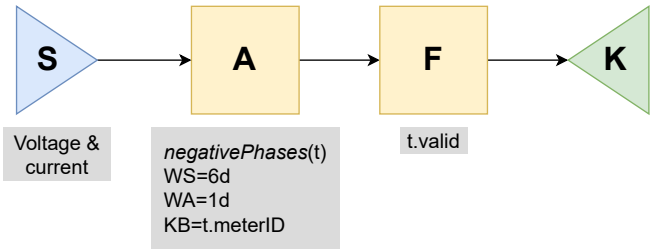



**Figure 4.2:** `NegativeCurrent` query

### 4.2.3 NoVoltage

The `NoVoltage` query uses the Voltage & current stream in order to detect facilities where a current is measured, but not voltage for one phase. This should be physically impossible, since there has to be some voltage present in order for current to exist, and should therefore be investigated further.

Again, a single tuple with this combination of values is not indicative enough of a fault, so an Aggregate with a window size of 6 days is used. This means that every output tuple will have up to 12 contributing input tuples. The DAG for the query can be seen in Figure 4.3.

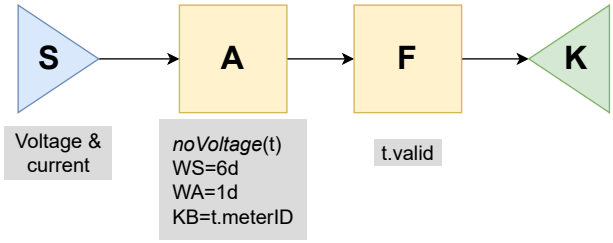



**Figure 4.3:** `NoVoltage` query

### 4.2.4 BrokenMeter

The `BrokenMeter` query differs from the other Voltage & current data queries in that it only applies to a specific brand of meter. For this specific brand, multiple readings of zero voltage and zero current for one phase are indicative of a fault. To detect the fault, a Filter first drops all tuples which are not produced by meters of the specific brand. Similar to the other queries, this is followed by an Aggregate that groups tuples by the unique meter ID. The Aggregate runs an analysis on six days of data at a time, with an advance of one day. A Filter is then applied to drop all invalid tuples which did not exhibit the erroneous behavior. The DAG for the query can be seen in Figure 4.4.

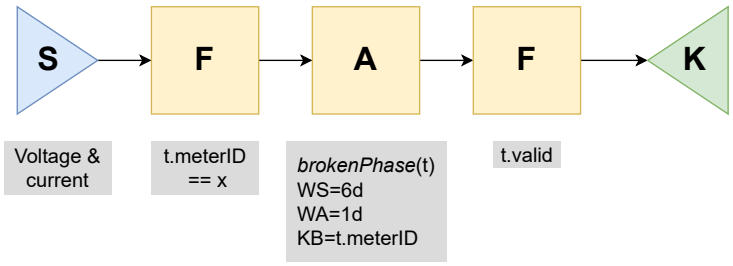



**Figure 4.4:** `BrokenMeter` query.

### 4.2.5 AverageDifference

The `AverageDifference` query aims to analyze the average energy consumption in each facility one week before and after a meter has been replaced. In order to achieve this, both the Energy usage and Meter changes streams are utilized. To lower the memory consumption, the Energy usage stream is first aggregated into daily averages for each facility. The streams are then joined with an *Interval Join*, where each Meter change event is joined with all Energy usage events from the same facility created within seven days. An Aggregate using a session window with a

session gap of one day is then used to calculate the average consumption before and after the meter change event. Collecting hourly measurements for a 14-day period, as well as a meter change event, means that each output tuple has up to 337 contributing input tuples. The DAG for the query can be seen in Figure 4.5.

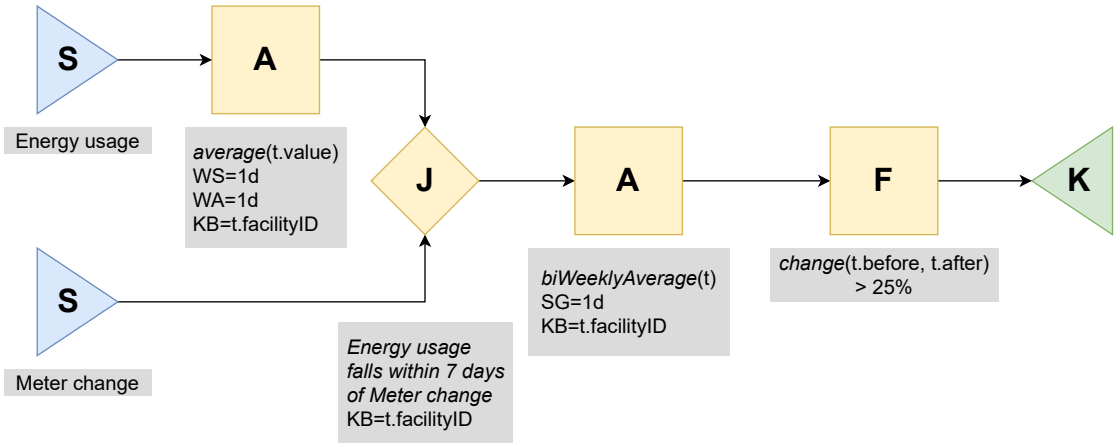



**Figure 4.5:** `AverageDifference` query

### 4.2.6 Changepoint

In addition to the queries described above, we implement a novel method to detect incorrectly installed smart meters. This query is valuable to GE since they have to replace all meters by 2025 due to the new legislation [13]. It is probable that a number of these replacements will be installed incorrectly, and inhibit the faults discussed in Section 2.1.2. Our query attempts to detect these faults by applying a changepoint detection algorithm to find mean shifts in the energy consumption, a symptom of a faulty meter installation where at least one phase is wired backward through the meter. It does this by collecting the data from a week before and after a power meter has been replaced, as seen in Figure 4.6, and thereafter applying a changepoint detection algorithm.

Since the applied changepoint detection algorithm consumes significant resources in terms of execution time and memory, this query also serves as an example of how a query with high computational complexity is affected by provenance.

It is possible to apply this changepoint detection to all 14-day windows of energy consumption data to find sudden changes. However, this would most likely result in many false positives when the energy consumption pattern changes for natural reasons, such as apartment vacancies, vacations, or changes in outdoor temperatures. As such, widespread monitoring would not only consume massive computing resources, but would also require extensive human intervention to double check the detected errors. Therefore, changepoints are only searched for in segments of energy usage close to meter changes (in time).

Our implementation of changepoint detection is strongly inspired by [23] and the

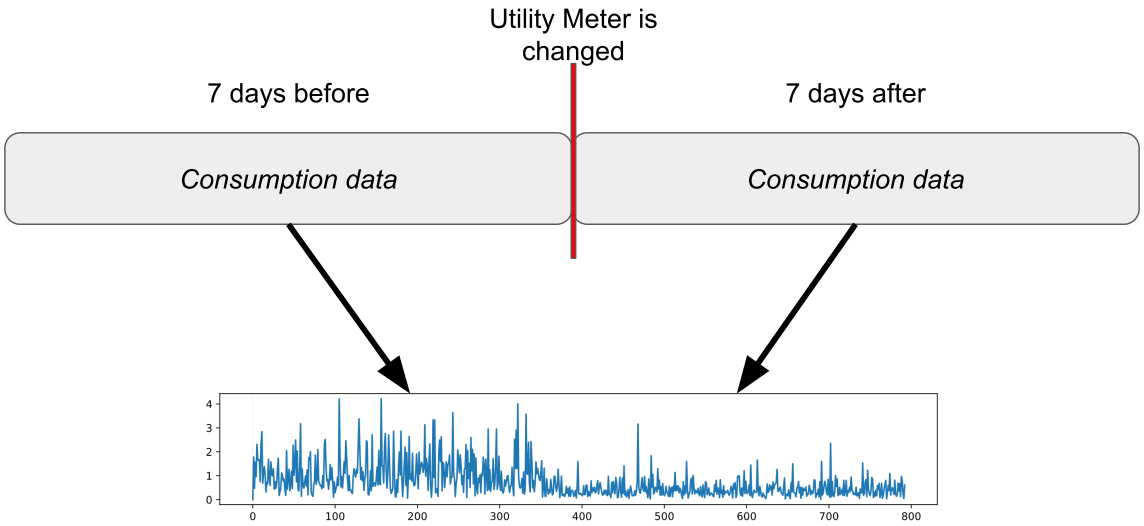



**Figure 4.6:** Monitoring interval for a meter change. To capture enough data to conclude if a changepoint has occurred, while still limiting the size of the computational state, a 14-day interval around the meter change is monitored.

related Python package `ruptures`. It is designed to apply changepoint detection with various search and cost functions. To avoid the overhead related to breaking out from Flink's Java environment in order to utilize the Python package, relevant parts of `ruptures` were ported to Java. The changepoint detection is then utilized in an Aggregate, where a histogram for the current time period is analyzed.

A baseline is observed by monitoring usage 7 days before a meter change. A period of 7 days after the change is also observed, to be used for comparison with the baseline, resulting in a total of 14 days of measurements per meter, as can be seen in Figure 4.6. A changepoint is considered suspicious if within 24 hours of a meter change and there is a 20% change in mean consumption (similar to the `AverageDifference` query). This means a human analyst should be alerted to decide whether the changed behavior necessitates an on-site inspection by a technician. See Figure 4.7 for the DAG of the `Changepoint` query.

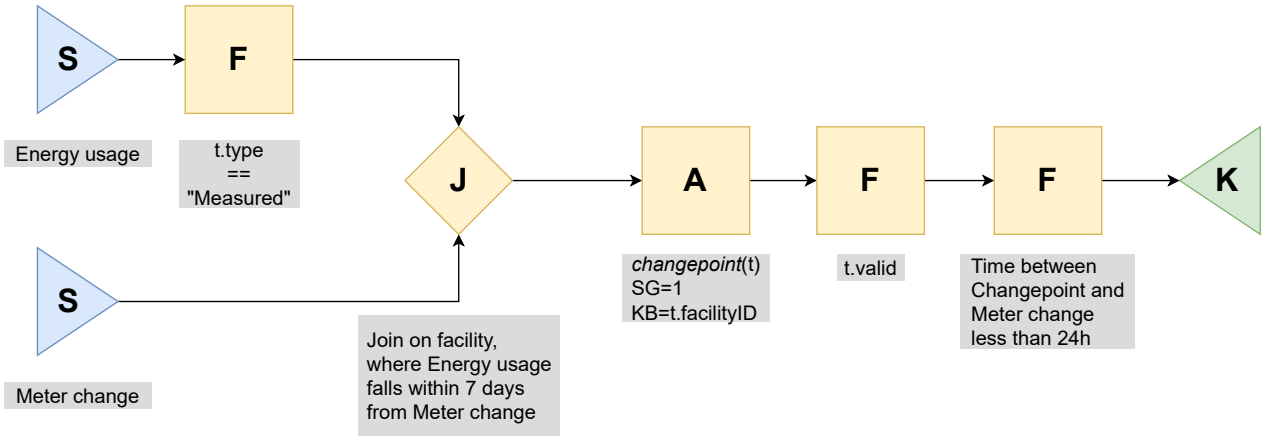



**Figure 4.7:** `Changepoint` query

#### 4.2.6.1 Considerations regarding the changepoint detection algorithm

As mentioned in Section 2.4, a changepoint detection algorithm is built using a cost function and a search method. Below follows some considerations and trade-offs regarding the choices made when selecting the cost function and search method.

**Cost function.** The used cost function L2, assumes that the received values are independent and indentically distributed (i.i.d.). Although power measurements don't follow this rule (few power spikes during the night, most consumption is overall cyclic per 24 hours, etc), over the span of a week or two, which is the scope relevant in this thesis, these hourly regularities are smothered out by the irregularities between days. Furthermore, alternative cost functions which tolerate dependent values generally incur a significantly higher cost, something that is not acceptable when handling the vast amount of data. Therefore the compromise was made to use a non-optimal model in order to get acceptable performance. In addition to this, we found that while more advanced methods greatly affected the calculation times, they didn't improve the hit rate of changepoints to any significant degree.

**Search method.** As mentioned in Section 2.4, the approximate search method binary segmentation is used. Despite being approximate, it (together with our chosen cost-function) most often generated a changepoint within 24 samples from an actual changepoint when running on test data. Altogether, Binary Segmentation was chosen as the search function because it was found that the method gave a good hit rate with a low performance cost. Our implementation supports the search for an arbitrary amount of changepoints. However, for our purposes, we are only interested in the case where there is a single changepoint.

## 4.3 Enriching streaming queries with provenance

To get provenance from a stream processing query, the query must first be *enriched* with provenance. A way to do this is to first implement the query using native op-

erators and then enrich the implementation such that both the result of the analysis and provenance is obtained as output. To achieve this we turn to GeneaLog [8]: a state-of-the-art provenance framework for provenance in deterministic streaming applications.

More specifically we use the version of GeneaLog which is extended with the work done for the evaluation of Ananke [20], a streaming framework for live forward provenance. For a brief description of live forward provenance and the inner workings of GeneaLog and Ananke, see Appendix A.1. Essentially, the extended GeneaLog allows parallelism by handling tuples that arrive out-of-order in stateful operators. More importantly for this thesis, it provides a *transparent* implementation based on encapsulation, which means the native Flink operators are clearly visible in the code. An example of encapsulating Flink code with provenance capture can be seen in Listing 4.1.

The transparent version may increase data serialization overhead [20]. Therefore whether or not to use encapsulation is a matter of performance versus code maintainability and readability. Despite the additional overhead, encapsulation was chosen to more easily conform to existing efforts of stream processing at Göteborg Energi.

**Listing 4.1:** An example of how a query can be enriched with provenance using the extended GeneaLog.

**(a)** A simple example query.

```
1     sourceStream
2       .keyBy(EnergyUsage::getKey)
3       .window(SlidingEventTimeWindows.of(WS, WA))
4       .aggregate(new FindEnergyPeak())
5       .filter(t -> t.isValid);
```

**(b)** Transparently instrumenting the query in Listing 4.1a using the extended version of GeneaLog. The added syntax is highlighted in orange.

```
1  GL.source(sourceStream)
2    .keyBy(GL.key(EnergyUsage::getKey))
3    .window(SlidingEventTimeWindows.of(WS, WA))
4    .aggregate(GL.aggregate(new FindEnergyPeak()))
5    .filter(GL.filter(t -> t.isValid));
```

# 5

# Evaluation

In this chapter, we present our evaluation of the queries described in detail in Chapter 4. Section 5.1 describes the environment in which the evaluations were performed. Section 5.2 presents the performance of the queries and how the performance differs depending on whether the provenance is implemented or not. Lastly, to see whether stream processing with provenance was expressive enough for more complex analysis, we evaluate the accuracy of the changepoint detection query in Section 5.3.

## 5.1 Evaluation environment

The evaluation of a streaming system is based on a multitude of parameters. In order to make this transparent, we here present in detail the properties of the data used, the hardware setup, and the configuration for Apache Flink.

### 5.1.1 Real-world test data

For evaluation, two months of historic data from GE smart meters were used, recorded from the streams presented in Table 4.1. Since the streams are of different rates, their recorded sizes also differ greatly, as can be seen in Table 5.1. This data is then fed into the queries as text-file sources. These sources feed queries as fast as possible to facilitate evaluating multiple queries in a limited time frame. In a real-world system, the tuples would be ingested every hour, perhaps in batches, when the data is retreived from the Collector Units.

**Table 5.1:** Size of the files resulting from recording the streams for two months.

| **Stream** | **Size** | **# of Tuples** (approx.) |
|---|---|---|
| Energy usage | 25.5 GB | 365 million |
| Voltage & current | 1.65 GB | 33 million |
| Meter changes | 63 kB | 1 thousand |

### 5.1.2 Generated test data

Due to the lack of ground-truth data regarding changepoints a data generator is used to provide samples. This generates values in a distribution similar to the distribution of the real-world data, where the generation parameters can easily be changed to simulate faults.

According to GE's experts, the most common fault in a meter is where a single phase (in a three-phase meter) is wired incorrectly, thus not producing any measurements. The symptom of this fault is a roughly one-third decrease in the total measured consumption, causing a changepoint. An artificial changepoint of the same nature can be induced in the generated data by multiplying all data points by a factor, where the factor is 1 before the induced changepoint and 0.66 after it. This reduces total consumption after the changepoint by a third, mimicking the behavior expected when a single phase cannot be measured. An example of a generated data series with an induced changepoint can be seen in Figure 5.1.

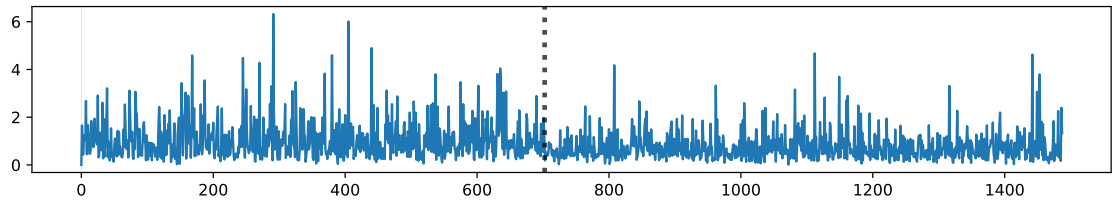



**Figure 5.1:** Generated test data. A changepoint has been induced at the dotted line, after which the generated consumption has been lowered by 33% to simulate a fault where one of thee phases can't be read.

### 5.1.3 System configurations

The evaluations are performed on a virtual Red Hat 7.9 server. The physical server used is a `Intel Xeon CPU E5-2695 v2 @ 2.40GHz` and the virtual machine was configured with 2 vCPUs and 16 GB RAM. The choice to perform the evaluation on this hardware configuration is that it represents the type of machine that Göteborg Energi uses in production. The project is built with OpenJDK 11.0.14.1 and runs on Apache Flink version 1.14.4. Since Ananke [20] was originally built running on Flink version 1.10.0, minor modifications were performed to update the dependency to the latest Flink version. To ensure correctness, this update was made in collaboration with the authors of [20].

Each streaming job is deployed on a local Flink cluster with a heap size of 12 GB. The number of task slots per TaskManager is set to one and, therefore, the maximum parallelism for operators is limited to one. The result of this configuration is that each query is run on a single JVM, processing one (although possibly chained) operator at a time. For a more detailed description of the implications of this set-up, we refer to Appendix A.2 and the original Flink paper [7].

The reason for the relatively large heap size is due to the nature of the data we're working with, as well as the characteristics of the executed queries. Large window sizes with many different keys mean that many different windows have to be kept open simultaneously, thus consuming more memory.

### 5.1.4 Query configurations

For each evaluation, two variants of the same query are executed. One where the query is enriched with provenance and the other variant without provenance. The query enriched with provenance is encapsulated as discussed in Section 4.3. The query without provenance is identical to that with provenance, except for the absence of the encapsulation. In those cases where the original query semantics was modified in order to facilitate encapsulation, the evaluation is made only with the modified queries, and not with the original, thus ensuring the only difference between compared queries being the presence of provenance. This is done in order to ensure fair comparisons.

### 5.1.5 Collecting performance metrics

The metrics presented in Section 3.4 are used to evaluate the performance of queries. They are reported on a per-second basis, where memory consumption and CPU usage statistics are fetched from the Flink cluster through HTTP requests. Throughput is logged by counting the number of lines read per second by the Source. Lastly, each source tuple is enriched with metadata of the wall-clock time it was ingested, which is then compared to the time a Sink tuple is produced to get a latency statistic.

At the beginning and end of a query execution, a significant number of outliers may be introduced due to the warm-up and cool-down mechanics in the execution environment. To avoid this, 5% of the measured values are discarded at the beginning and end of the collected statistics.

Note that the memory consumption of the queries is only an estimate, since accurate measurements of this metric are complicated to perform. This is due to the fact that one can only measure how much memory the JVM occupies, not a "true" measure of how much memory the query itself uses. This is in turn further complicated by Java's Garbage Collector (GC). If we choose to restrict the JVM's total memory, the GC will work harder to free up more memory, artificially decreasing the memory footprint by reusing memory as fast as possible. On the contrary, if we give JVM more memory to play with, the GC will be less active and therefore not free resources as quickly. This increases the occupied memory, even though it is no longer actively used by the query and is just waiting to be collected by GC.

## 5.2 Measured results of queries

In this section, figures showing the performance of the queries described in Section 4.2 are presented. The x-axis of each figure show two variants of the same

query, where *NP* stands for “No Provenance” and *P* stands for “Provenance”.

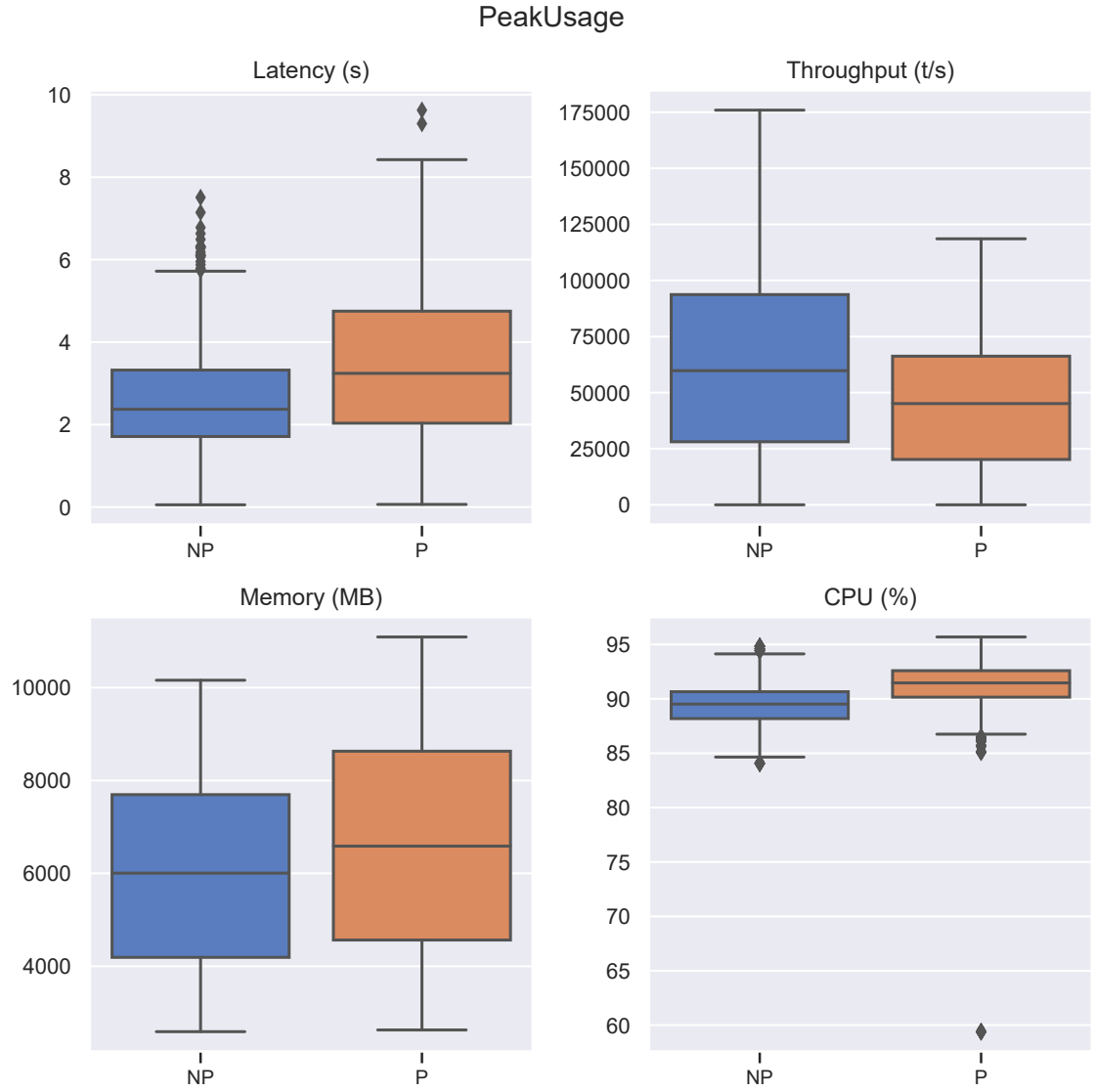



**Figure 5.2:** Performance of the `PeakUsage` query, with and without provenance for all smart meters in the AMI.

In Figure 5.2 the results of the `PeakUsage` query are presented. With a WA of one hour and a WS of three hours, Flink needs to maintain many windows, and as such, there is a significant overhead for window management. This is reflected in the throughput, with a median of about 43 000 tuples per second with provenance, enough to process an hour of tuples (from 25% of all smart meters) roughly every 1.6 seconds. Therefore, with the amount of historic data, the time axis is too wide to perceive any useful information. Consequently, in this case we plot memory as a box plot instead of a line plot unlike later figures. As we shall see, the overhead is nevertheless relatively low compared to other queries. This is expected considering the small window size, i.e. large portions of provenance can often be freed from memory by the garbage collector.

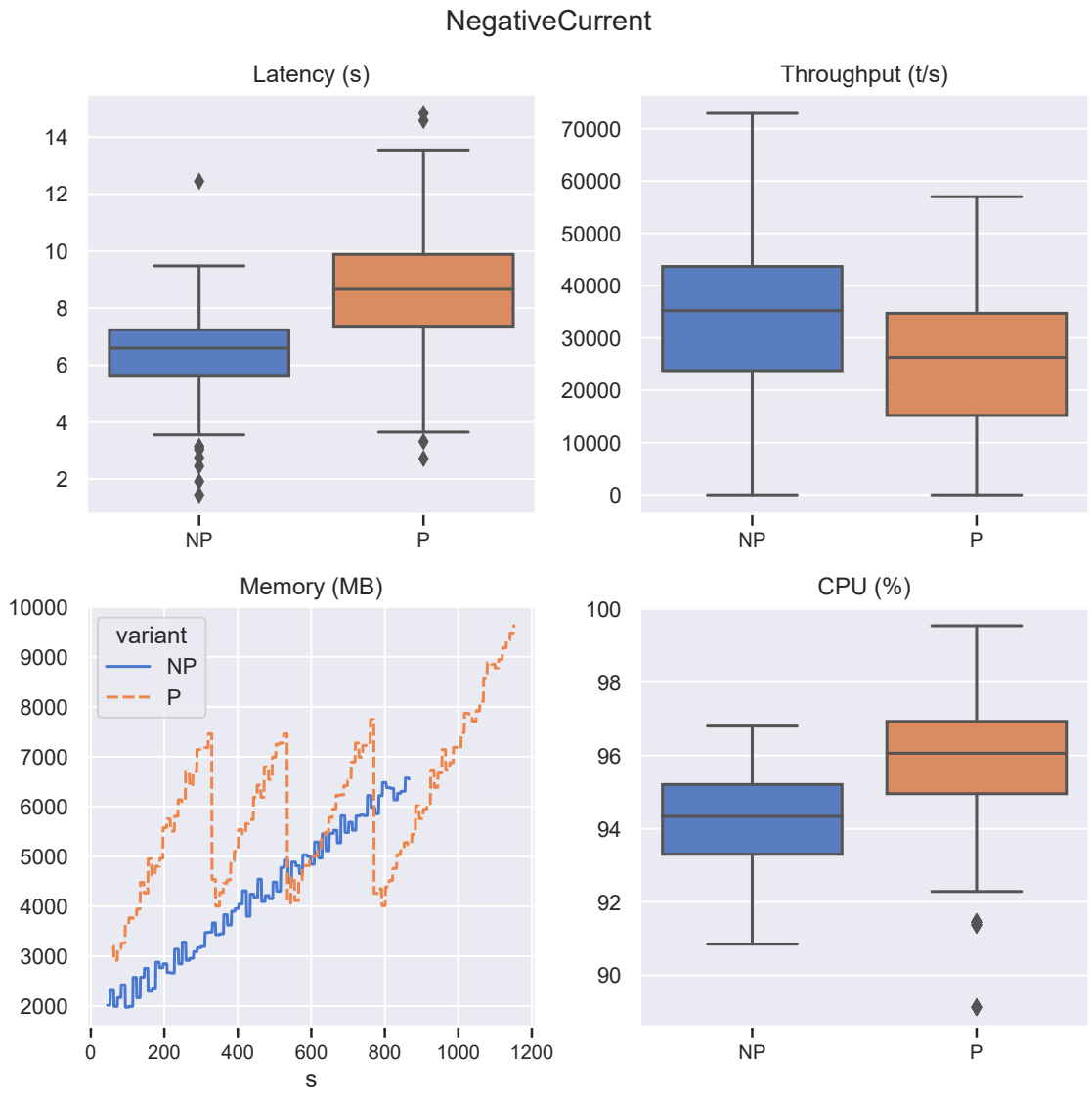



**Figure 5.3:** Performance of `NegativeCurrent` query, with and without provenance for all smart meters in the AMI.

In Figure 5.3 `NegativeCurrent` is presented, which uses the Voltage and current stream. It has a bigger WS (six days) compared to `PeakUsage`, note however that due to the event time rate of this input stream, a sink tuple has at most 12 contributing source tuples. The steepness of the memory graph shows how the garbage collector is needed with provenance to reclaim memory from old objects. CPU usage is close to maximum in both cases.

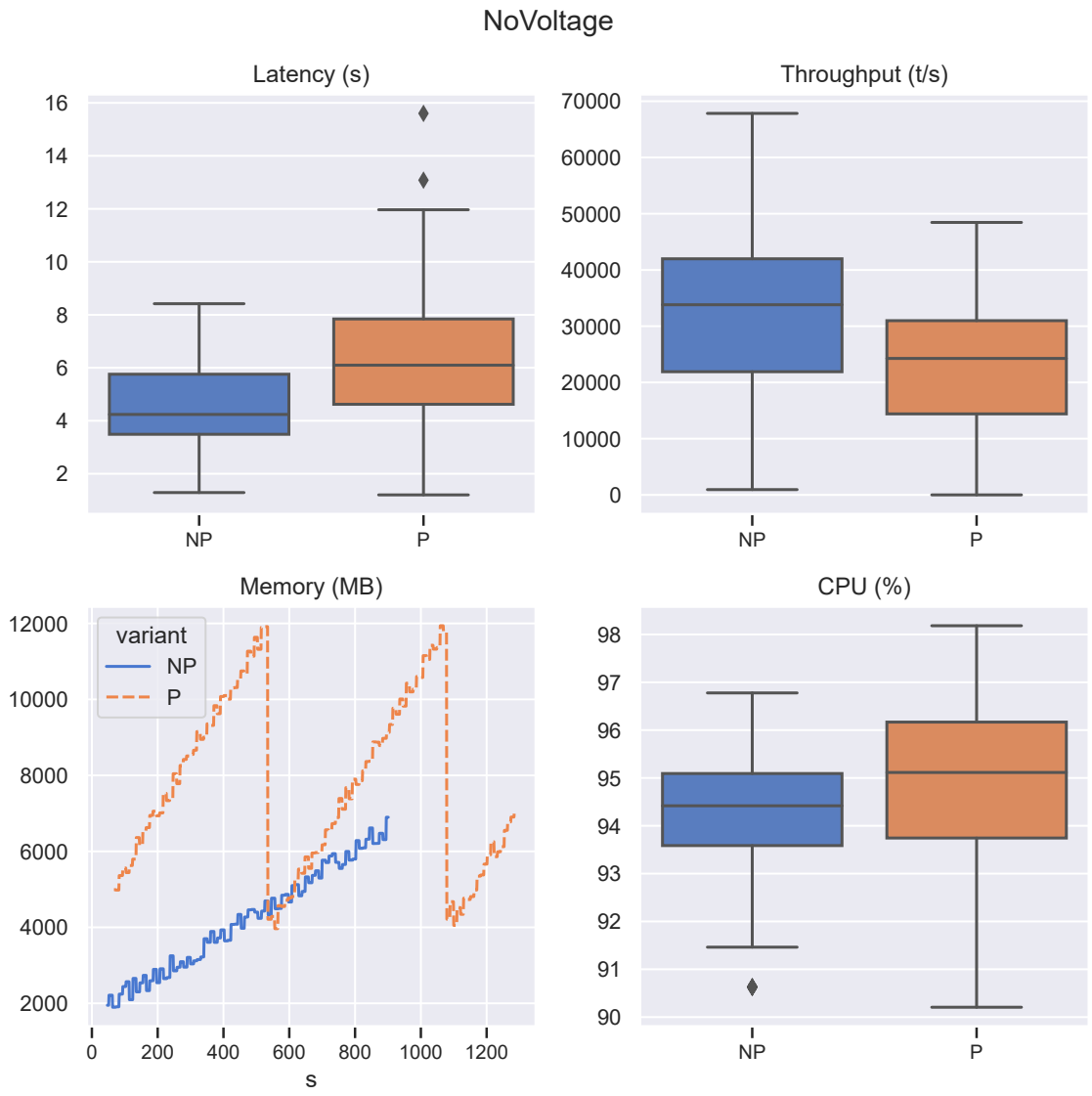



**Figure 5.4:** Performance of `NoVoltage` query, with and without provenance for all smart meters in the AMI.

In Figure 5.4 the results of the `NoVoltage` query are presented. Its performance characteristics are very similar to that of `NegativeCurrent` presented in Figure 5.3. This is not surprising, since their semantics are very similar to each other.

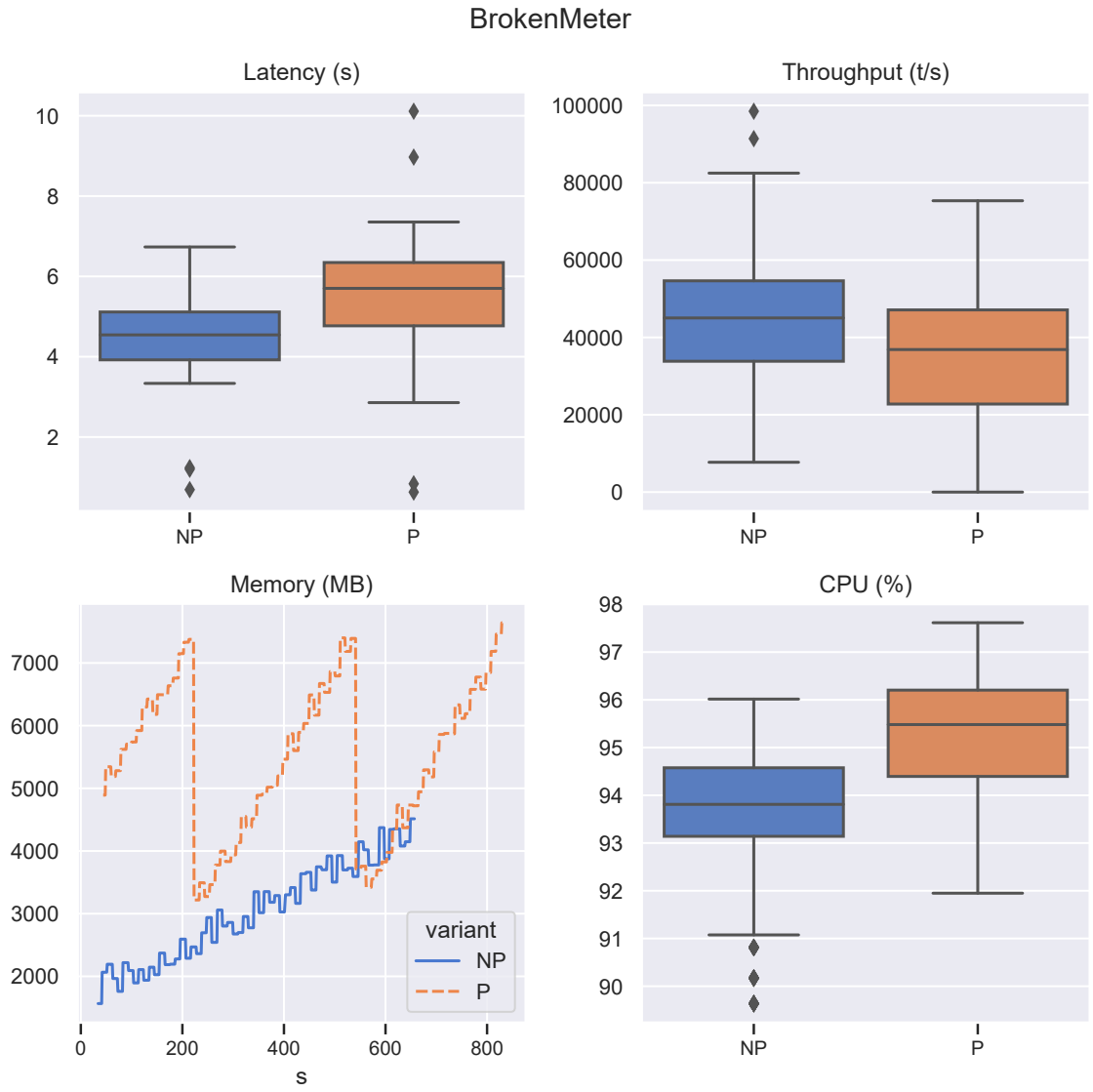



**Figure 5.5:** Performance of `BrokenMeter` query, with and without provenance evaluated on data from all smart meters.

The `BrokenMeter` query (Figure 5.5) differs mainly from `NoVoltage` and `NegativeCurrent` in that it filters to one specific meter brand. However these still constitute roughly 60% of all meters. Therefore the difference in performance visible, although not major. It is a query where provenance capture can be applied without worrying too much about the performance impact.

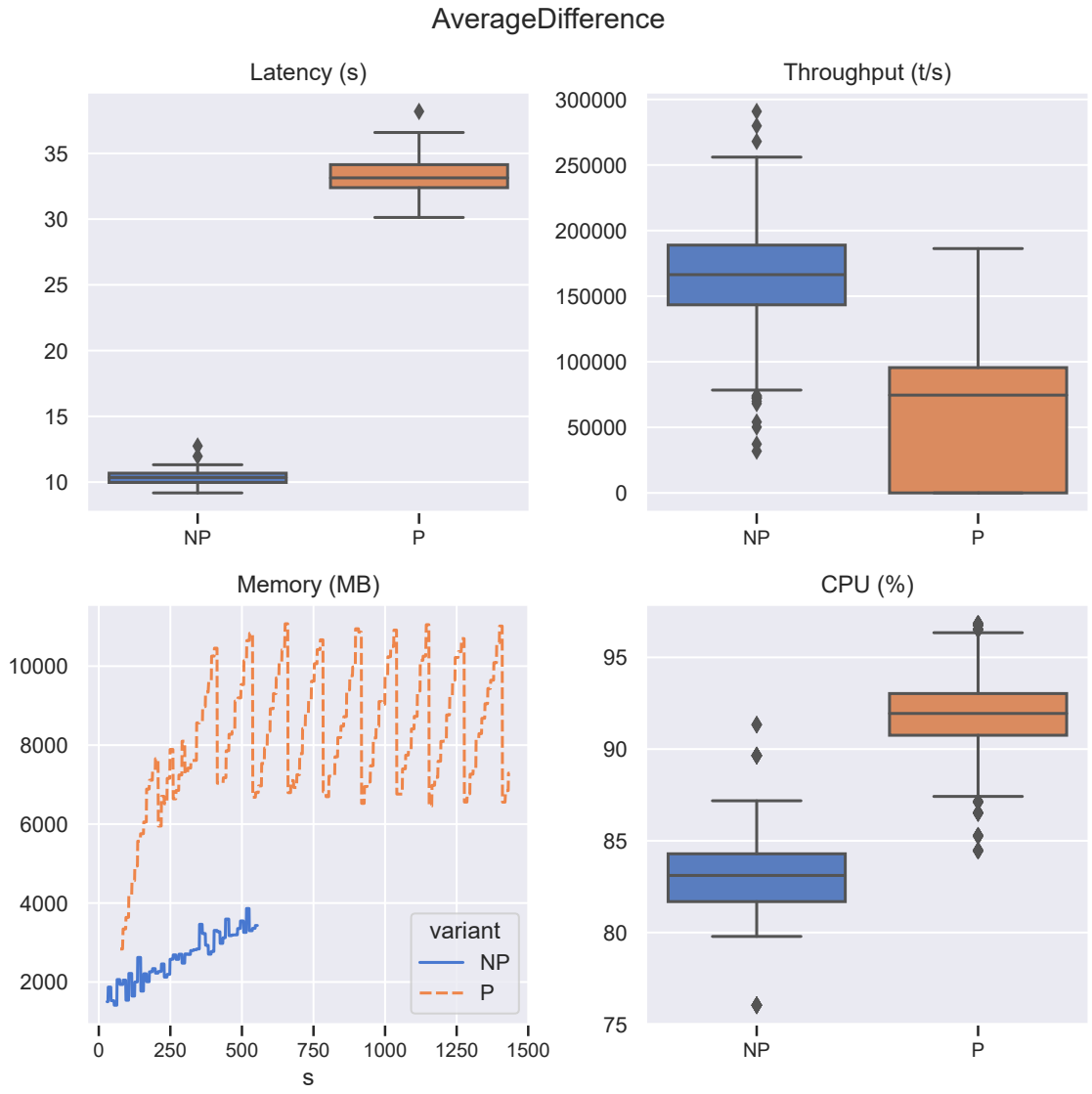



**Figure 5.6:** Performance of the `AverageDifference` query, with and without provenance for 25% of all smart meters in the AMI.

Moving on to a more complex query with multiple input data streams, we have the `AverageDifference` query (Figure 5.6). Here, by looking at the memory graph we see how the the provenance version consumes much more memory than the non-provenance version. Thus, the garbage collection is run several times during the run time of the provenance versions. Throughput frequently drops to zero for the provenance enriched query, due to the JVM stopping all operator threads before reclaiming memory that is occupied by dead objects. These dead objects are supposedly provenance no longer referenced by GeneaLog. The latency is also affected by this, since each pause in the execution brings additional delays for all data in the pipeline. Similar to other queries the CPU usage is close to the maximum.

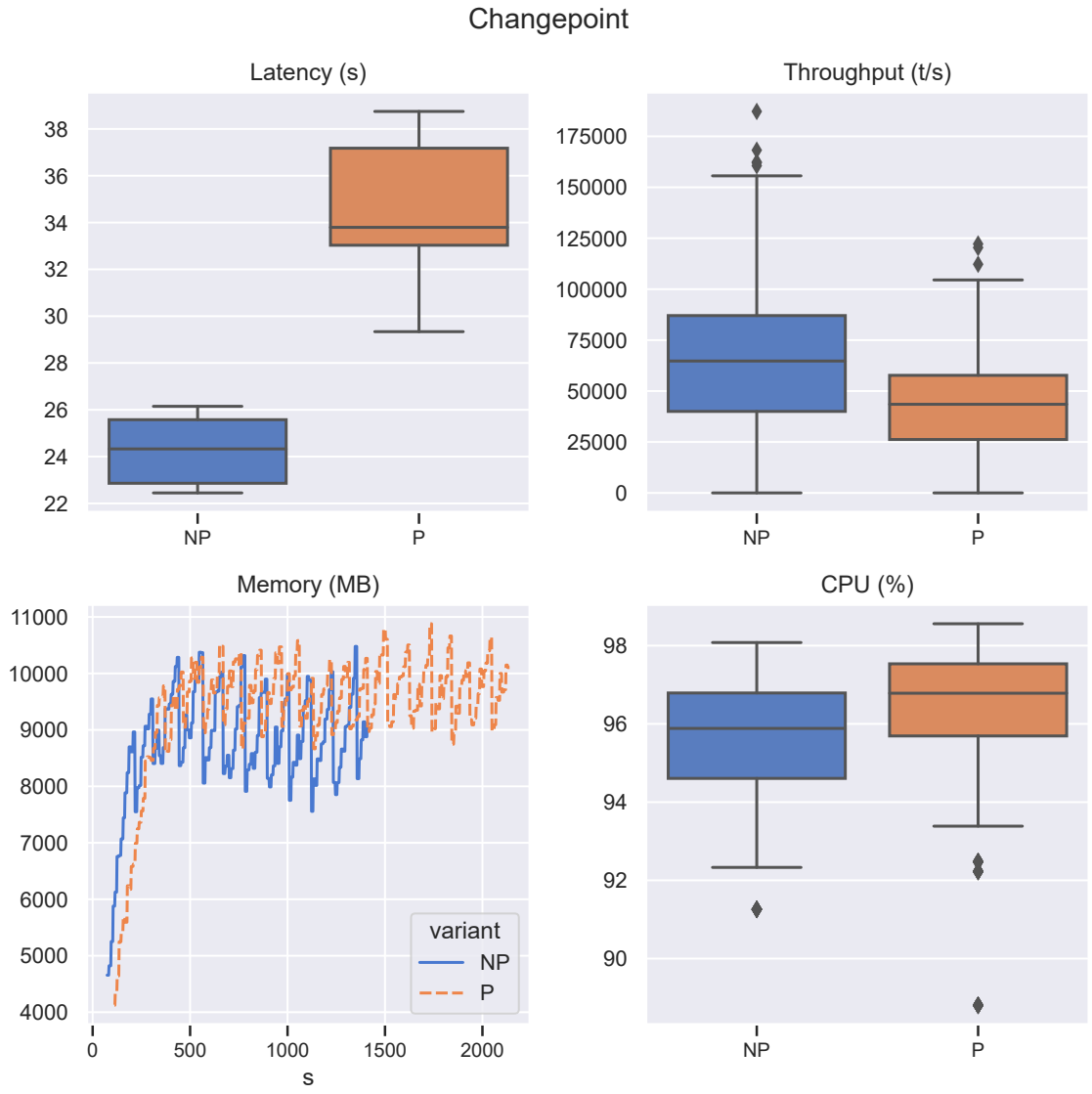



**Figure 5.7:** Performance of `Changepoint` query (Section 4.2.6), with and without provenance for *25%* of all smart meters in the AMI.

Despite processing the same data streams, the `Changepoint` query (Figure 5.7) shows less provenance overhead compared to `AverageDifference`. The memory graphs show a similar pattern in both variants because the Aggregate applying the changepoint algorithm to data around a meter change needs to keep the energy usage. This means that less data can be discarded by the garbage collector even in the case where no provenance has been applied.

### 5.2.1 Evaluation summary

To summarize, provenance capture using the transparent and extended version of GeneaLog [20] has been evaluated for six different queries. These use the different input streams found historic data from a real-world AMI. The average change in per-

formance for the statistics is presented in Table 5.2. For a more visual comparison, see the bar plot in Figure 5.8.

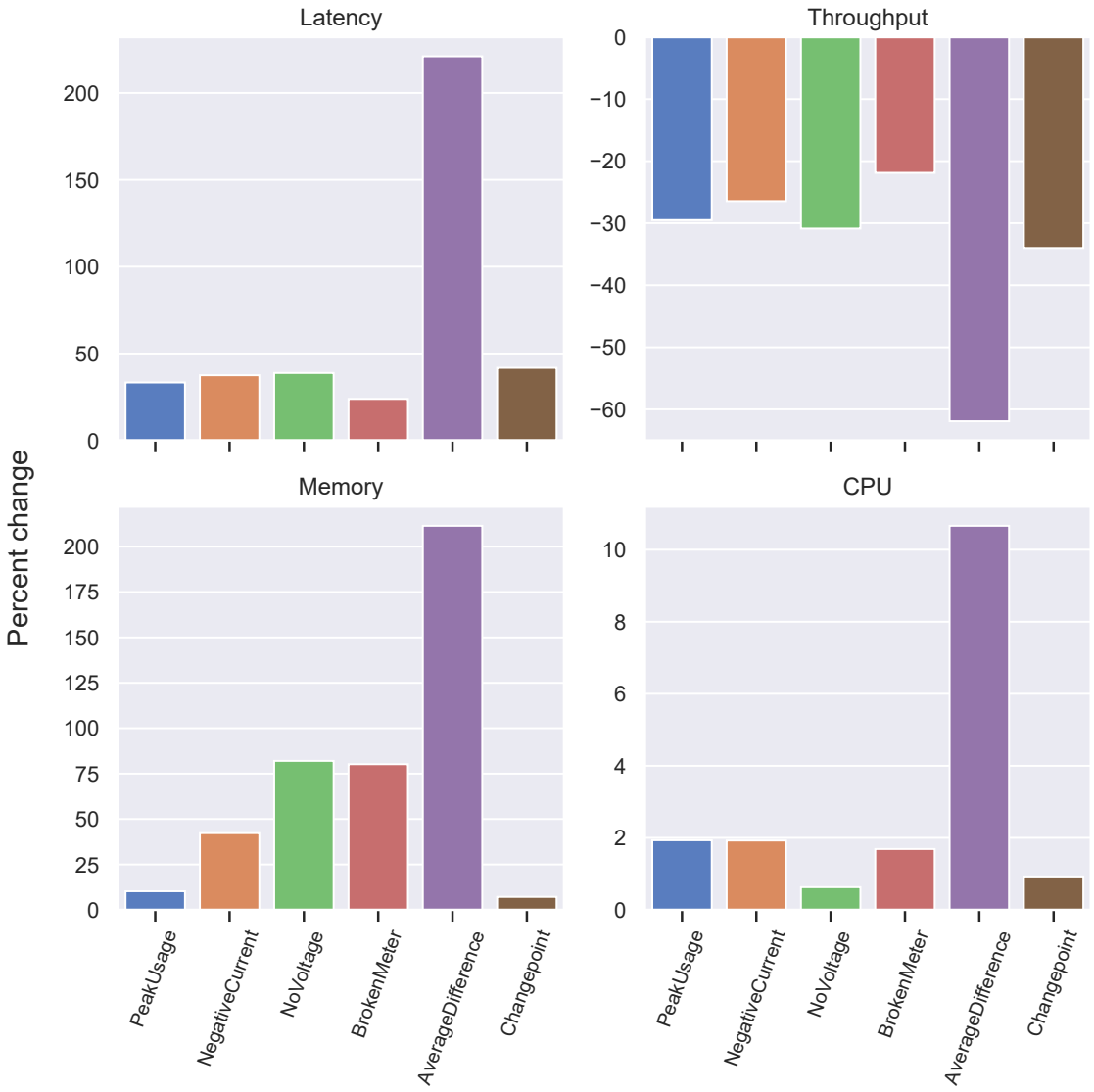



**Figure 5.8:** Bar plot representation of Table 5.2, the percent change in mean for each statistic comparing provenance with no provenance for each query.

**Table 5.2:** Change in performance for the mean of each statistic, comparing no provenance with the provenance enriched query.

| | Relative change in mean (%) | | | |
|---|---|---|---|---|
| **Query** | **Latency** | **Throughput** | **Memory** | **CPU** |
| NegativeCurrent | 37.5 | −26.4 | 42.2 | 1.9 |
| NoVoltage | 38.8 | −30.9 | 82 | 0.6 |
| BrokenMeter | 23.9 | −21.9 | 80.2 | 1.7 |
| PeakUsage | 33.3 | −29.5 | 26.3 | 1.9 |
| AverageDifference | 221.1 | −61.9 | 211.5 | 10.7 |
| Changepoint | 41.8 | −34.0 | 7.2 | 0.9 |

## 5.3 Accuracy of changepoint detection

With generated data, it was experimented with how much certainty a changepoint could be found on the day a meter change actually occurred, considering different window sizes. That is, using different amounts of data to establish a baseline for the consumption before and after a meter has been changed.

The complete pipeline was tested with the generated data, where 1000 samples were generated with an induced changepoint and 1000 samples without a changepoint. A meter change event was generated for each sample at the date of the induced changepoints. The results of these runs can be seen in Table 5.3.

**Table 5.3:** The accuracy of changepoint-detection using different window sizes for the aggregate.

| **WS (days)** | **True positive (%)** | **False positive (%)** |
|---|---|---|
| 6 | 79 | 30 |
| 14 | 83 | 10 |
| 28 | 86 | 4 |

We can see that a larger window, and therefore more data, improved the hit-rate, as can be seen in Table 5.3. Naturally, the chance of randomly finding a changepoint within 24 hours of when a meter was changed increases as a smaller window size is considered. The chance of finding a changepoint on the day that a meter was changed by random can be calculated according to Equation 5.1, where P is the probability of randomly placing a changepoint within 24 hours from the change of a meter, and WS is the Window Size in days. It should be observed that the false positive rate in Table 5.3 approximately follows this rule.

$$P = \frac{2}{WS} \tag{5.1}$$

Furthermore, one can also observe that when the number of days regarded increases, not only does the false positive rate decrease but the true positive rate increases. This shows that a longer observation interval is largely beneficial to accurately place

changepoints to find faults. In practice, however, a longer observation interval results in a larger memory footprint as well as a longer latency. Therefore, together with GE, it was decided that 14 days of observation is a reasonable interval to get a high enough hit-rate, while limiting both false positives and latency.

# 6

# Discussion

This chapter contains a discussion around how provenance may reduce storage requirements in the AMI and the performance impact of provenance based on the results of Chapter 5. A comparison with a previous system, and what to consider when opting to use provenance in tandem with streaming. Lastly, limitations with the work are presented together with future work.

## 6.1 Storage requirements

One of the goals of using stream processing with provenance for analysis of smart meter data is to reduce the storage requirements by only storing data that is considered interesting, and discarding the rest. This can for example be the data that indicates faulty smart meters of the kind our queries aim to find. A caveat is that all consumption data, which comes from the Energy usage stream, must be stored for billing reasons, but it is nonetheless valuable for analysis as well.

The Voltage & current stream, on the other hand, is only used for analysis. Measurements can thus be discarded where no faults are found, only storing the data which contribute to detected faults. The three three queries using the Voltage & current stream, namely `NegativeCurrent`, `NoVoltage`, and `BrokenMeter`, produce output (including provenance) which occupies disk space with the size of 209 KB, 908 KB, and 242 KB respectively. Since the size of the input stream is $1,65$ GB, the ratio of what needs to be stored long term equates to roughly $0,08\%$ of the original data. Over time, as more queries are defined to analyze more types of faults, this ratio will increase but is likely to remain below a few percent.

Once the AMI fully transitions to the modern smart meters capable of measuring more properties of the network [25], more data will be generated. Many of these properties will, like the Voltage & current stream, only be used for analysis. Therefore, the consumption data will make up a smaller part of the meter data and a larger part can potentially be discarded. An estimate based on tests of modern meters at Göteborg Energi is that energy consumption will constitute only about 20% of the data. Designing queries with provenance capture as presented in this thesis will therefore enable the discarding of a larger part than currently. Assuming that analysis can discard 95% of measurements as uninteresting, giving some leeway for

what we observed with the Voltage & current stream, a utility can reduce the total long-term storage by 76%, resulting in reduced expenses. However, note that this reduction is largely dependent on how frequent alerts in the analysis are, as more frequent alerts will also give larger amounts of contributing data, and require more storage space.

## 6.2 Performance

Provenance capture naturally results in a decrease in performance. As can be seen in the evaluation summary (Section 5.2.1), the performance degradation of the queries enriched with provenance is not constant. The overhead depends heavily on the operators used, therefore a programmer who wishes to use provenance in a streaming setting must take extra precautions. Through analysis of results, the goal is to make this transparent in order to facilitate a better usage of provenance.

Unlike some cyberphysical systems, there is to a certain degree no need to minimize the latency and throughput of analysis in the AMI. This is due to the fact that tens of seconds of delay before a technician has the information that a meter is faulty has little impact. Although they are related, the statistic of main concern is memory consumption, since the accuracy of queries is dependent on the amount of data available, in many cases being dictated by the irregular nature of energy usage data.

### 6.2.1 Memory consumption

The memory consumption for most queries is significant, and in some cases higher than the evaluation hardware can handle. Part of the reason is the sheer amount of smart meters in the grid. Naturally, to find faults for specific meters, the Aggregate and Join operators have to group tuples together by the meter identifier, and as a result, there are hundreds of thousands of active windows simultaneously.

To mitigate the memory problem, it is useful to limit the analysis to segments of the streams. In this thesis, it is done by utilizing the timestamps of *Meter change* events. These events enable the `Changepoint` query to only keep tuples of one week per customer in memory, instead of the full two weeks of interest. Only once a meter change is ingested will the query start the changepoint detection. Despite this, maintaining one week of data for all 280 000+ meters in the AMI is still too much for the evaluation hardware used. Therefore this query was evaluated only on 25% of the data. Note that this problem also affects the `AverageDifference` query, where the provenance enriched version shows the same memory consumption behavior.

The memory plots for queries with provenance capture all show a sawtooth pattern. This pattern is caused by Java's garbage collection mechanism, which purges the memory from unused data, as can be seen in Figure 6.1. The only data left in memory is then data that is still being used. The vital aspects to observe to assess

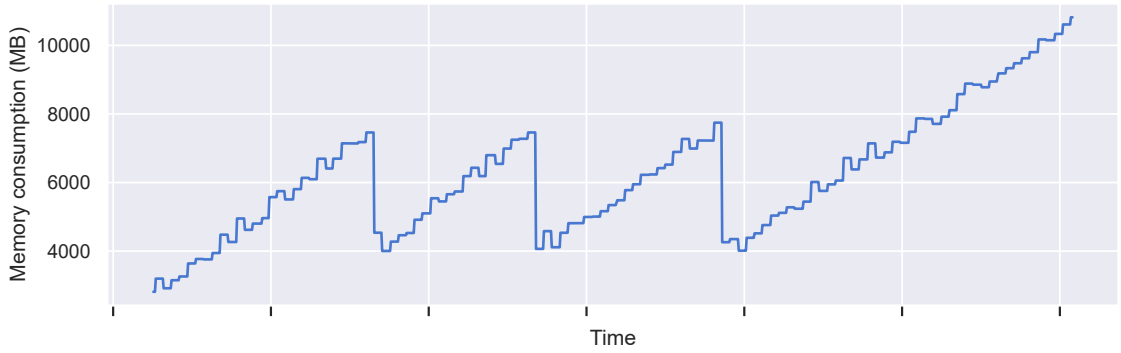



**Figure 6.1:** The distinct sawtooth pattern in memory consumption from a subset (in time) of the provenance variant of the `NegativeCurrent` query as a result of garbage collection.

the actual memory consumption of a query is therefore the rate at which occupied memory grows, and how much data that is left after a purge. A faster accumulation entails a higher memory pressure, and more garbage collection, which inherently wastes resources. Furthermore, the amount of memory kept intact after garbage collection shows how much memory is actually used in that instance of time.

We can see that for all queries, the provenance enriched version consumes significantly more memory. This can be seen in both higher accumulation rate, and more memory kept intact after garbage collection. Also, note that the time series for the provenance enriched queries are longer than for those without provenance. This is caused by the longer runtime required by the provenance enriched queries, resulting in more measurement points.

### 6.2.2 Latency

A latency of several seconds and in some cases tens of seconds is considered very high for streaming applications. This can partly be explained by the high CPU usage for all queries, but for the queries `AverageDifference` and `Changepoint`, the utilization of Flink's Session windows is the main culprit. This will be made clear in the discussion that follows.

The latency is measured as the time from when the last contributing tuple has been ingested, to that of the result being produced, as done in [8] and [20]. This works well for queries with only stateless operators. However, for queries that utilize windowed stateful operators, Flink is not allowed to start actual calculations until the window has closed. That is, Flink will have to wait until any tuple with a timestamp that exceeds the closing time of the window has been received.

This means that the latency will be forcibly inflated with the time left until the window closes, that is, the separation between the last contributing tuple and the closing of the window. We call this separation the *Window Separation*. In sliding or tumbling windows this separation is difficult to estimate since the tuples' placement

within the window with respect to the closing of the window is arbitrary. However, in session windows, we know that the session gap must expire before the calculation can be triggered. Thus, large windows, and large session gaps in session windows, will increase latency disproportionally, decreasing the perceived latency added by provenance. The difference between the reported latency and the computational latency can be seen in Figure 6.2, and the calculation for the reported latency can be seen in Equation 6.1.

$$\textit{Reported latency} = \textit{Window Separation} + \textit{Computational latency} \tag{6.1}$$

While this method is practical and easy to implement, it doesn't do justice to Flink's abilities. This is since the latency clock is started, and Flink is then forced to wait for a while until it is allowed to start calculations. Our main interests are in the computational latency, and not in how long we wait for the window to close.

Ideally, latency should be started first when the calculation is allowed to start when a watermark exceeding the window limit has been received, thus only the computational latency would contribute to the reported latency. However, Flink has no mechanisms that allow this, and a custom implementation to provide this functionality is outside the scope of the thesis.

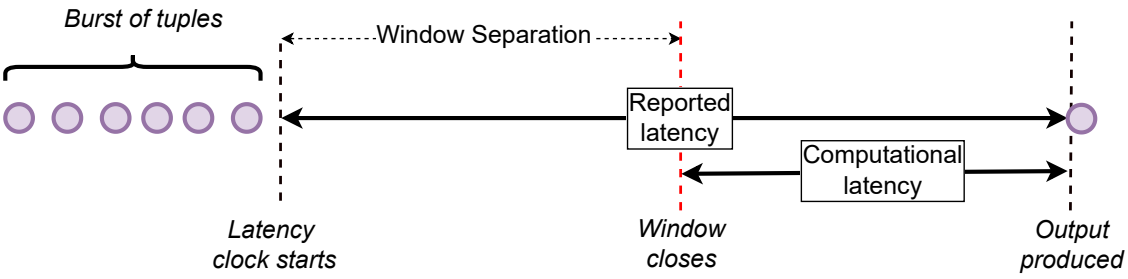



**Figure 6.2:** The computational restraints of measuring latency in windowed operations. Although the latency clock starts as soon as the last contributing tuple has been ingested, windowed operations introduce latency by delaying the triggering of calculations. The reported latency, and the actual computational latency, can then differ greatly.

### 6.2.3 Throughput and CPU usage

For the queries using the Voltage & current streams, the relative decrease in mean for throughput is in the range of 20–27% which is acceptable given that only two tuples per meter are ingested by the system per day. We note that the `BrokenMeter` query has less throughput overhead compared to its similar counterparts `NegativeCurrent` and `NoVoltage` because the very first operator, a Filter, discards tuples such that there is less provenance that needs to be maintained. Nevertheless, this only amounts to about 2–5% difference.

Feeding the system as fast as possible causes the absolute values of the metrics to be CPU bound, since the usage always hovers around 95-100% regardless of query variant. But this applies to all queries such that the relative difference between other metrics are still significant.

## 6.3 Improvements compared to Göteborg Energi's system

Provenance in the current system is provided in a simple, but somewhat unpractical fashion. A script that observes which events have been generated by the Flink job the previous 24 hours is run daily. The script then fetches the data from the temporary time-series database which contributed to the alarms, with a 7-day period padding before and after the alarm, and saves that data long-term. This naive method assumes that all contributing data falls within 7 days from the created event, and does not distinguish between contributing and non-contributing tuples.

While the current strategy works in practice, it lacks precision and consumes unnecessary resources. The lack of distinguishing between contributing and non-contributing tuples results in it being more difficult to diagnose faults in the AMI in general, as well as more challenging to debug queries in particular. Furthermore, the system requires maintaining its time-series database to temporarily store all incoming data for several days. The effort needed to maintain this database will likely rise when considering the increased data-loads related to the increased reporting capabilities of new meters, which will be installed to comply with new regulations [13]. According to an expert at GE, the amount of data handled can then increase by a factor of 100.

Stream processing with provenance offers significant benefits compared to the currently used system. While the current system only runs an analysis every 24 hours, the continuous calculations offered in stream processing present the possibility of almost instant results. Additionally, the added precision which follows from in defining only the contributing tuples entails easier fault detection in the AMI by technicians, as well as easier debugging of queries. More importantly, however, the need for a temporary database, as is used currently, can be eliminated, since all contributing data is kept as a result of the relevant queries. On the other hand, the temporary database currently allows technicians to fetch data as they see fit, enabling a more thorough investigation of faults. This is unfeasible using only provenance since it only saves tuples that specifically contributed to an alert.

This thesis further also focuses on the specific use case aimed at detecting faultily installed smart meters. The explored changepoint detection strategy offers significant improvements compared to the current system. The currently applied strategy relies on technicians manually reviewing metrics if time allows it. This typically entails that a technician manually fetches data with a script, to then review the data by hand. This is a somewhat viable strategy when the number of newly installed meters is low, currently tens of meters per day. However, it becomes unsustainable

once all the 280.000 meters in the AMI have to be changed to comply with new regulations [13].

With our query, analysis for faultily installed meters can instead run continuously, requiring only the occasional check to determine if an alarm sent by the query justifies a visit by a technician. In the analyzed two months of real-world data, just over 1000 meter changes were recorded. Out of these, 99% of the meter changes could be filtered out as unsuspicious, only sending an alarm on 10 of the changes. Compared to the current system, both latency, throughput, as well as time spent by technicians to find faults, are greatly improved.

## 6.4 Provenance: ease of use vs. performance

While the performance of a program is critical for a query's intended use, the complexity of writing it also matters a great deal. Programs that are harder to write take longer to produce, increasing costs and introducing delays. However, the tools which provide provenance in this thesis impose limits on how a program can be written, requiring them to only utilize standard operators [20, 8]. User-defined methods which give a programmer greater freedom in how a program is expressed are therefore forbidden. While most programs can be written with standard operators exclusively, it can pose a great challenge when constructing queries that could more easily be defined with a custom user-defined function.

A particularly intrusive limitation with the wrappers introduced by [20] is that the Flink-native `process` function can't be wrapped. Its native implementation gives great freedom to the developer, by not giving any predefined relation between input and output. It is therefore not possible to apply the same semantics to this operator as to the other standard operators. The developers of [20] have thereby opted to not support provenance for the process operator. Our solution when wrapping existing queries was to attempt to convert these processes to either maps or aggregates. These operators can sometimes provide similar functionalities as processes, albeit with a few limitations. However, we recommend building queries with provenance in mind from the ground up, since queries can then purposefully be built using only standard operators, in order to avoid including incompatible operators already in the planning stage.

Furthermore, performance will always degrade when introducing provenance. This is inevitable since provenance in its essence is extra meta-data, which inherently takes up extra space and resources. Therefore running queries with provenance might not always be ideal, if the provenance is not needed. In particular, performance degrades significantly where there are many different input tuples for each output tuple, meaning more provenance data is generated and stored, as well as where the windows are larger and longer, meaning that provenance data is stored for longer. Thus, the focus on improving performance should lie on limiting the number of contributing tuples, and limiting the size of the windows in queries.

Often, the above-mentioned requirements may be incompatible with the query's intended purpose, and thus its usability. Provenance might be valuable to find events with lots of contributing tuples collected over a large window of time; an analysis that would have been painstaking for a human to perform but often trivial for a streaming query. However, the query's state may grow prohibitively large due to the number of contributing tuples as well as the window size's demand for long-time storage.

## 6.5 Limitations and future work

This thesis has several important limitations to take into account. Most of these limitations can be addressed through further study. One limitation is that the scalability of the system is not fully considered. As stated in Section 5.1.3, evaluation was performed on hardware that more or less represents what GE uses currently, however it does not fully leverage the SPEs possibility of distributing the deployment in the grid. It would therefore be interesting to for example use the Collector units to process tuples closer to the meters and only send the analysis results (with provenance) to the central servers.

As a response to the new regulations coming into effect in 2025 [13], utility companies in Sweden are replacing their existing smart meters with ones that are more fully featured. This is not only in terms of how often values are sensed but also in the number of attributes measured. This opens up the possibility of developing new queries using streams with different properties. Provenance capture should then be evaluated with this new analysis for a better understanding of its effects, although the results are likely to be in line with ours.

Another limitation is that the survey conducted only tests a single implementation of provenance, namely the extended GeneaLog [20] running on Apache Flink. This method was chosen due to its superior performance compared to its alternatives, as well as the close support from its creators throughout this project. However, provenance capture can be implemented in other SPEs and then evaluated on the same analysis as was done in this thesis.

Lastly, this thesis explored how provenance behaves in queries applied to the AMI. There are many more viable application areas that generate large data loads but with different performance requirements, such as data center logging, vehicle sensors etc., that could greatly benefit from applied provenance.

# 7

# Related Work

In this chapter, we discuss past research related to the different aspects of this thesis. Since the aim was to apply stream processing in the AMI, Section 7.1 discusses other works where this approach has been the main study. Section 7.2 highlights methods related to applying provenance in streaming environments, while Section 7.3 highlights work done in the field of changepoint detection.

## 7.1 Stream processing in the AMI

Lohrmann and Kao [27] proposed a number of requirements for processing smart meter data streams. These include scalability, availability, support for low-latency processing, and data management. To address these requirements, the authors use a cloud computing framework called Nephele [28] that enables the execution of parallel stream processing. Kao was also involved in the development of Nephele, which predates Flink but has a similar architecture with Job Managers and Task Managers (see Appendix A.2). For evaluation, the authors implemented a real-time pricing application that uses simulated data from one million smart meters. Deployed on commodity server hardware, a cluster of 19 virtual machines is managed by the Eucalyptus [29] cloud computing platform. With this setup a new price update is provided approximately every 10 seconds. This thesis focuses neither on stream processing on a cloud computing platform nor on real-time pricing based on smart meter data. However, the implementation presented here facilitates deployment to a cloud platform such that Lohrmann and Kao's conclusions about scalability are likely applicable to this thesis as well.

Stream processing in the AMI with Apache Storm is explored by Joseph and Jasmin [30]. Their work proposes an application which is designed to detect power outages in generated power meter data. They achieved a throughput of 6250 tuples per second and a latency of 3 seconds, using a data set of 200.000 sample values.

In another paper, Joseph and Jasmin [31] explores the possibility of analyzing data from an AMI in the realm of Big Data using Hadoop. They conclude that in order to create the largest value from the data, real time processing must be leveraged. Hadoop, which is mainly oriented around batch processing, is therefore seen as unfit to carry out the task. In another study, [32] instead opts to using Apache Spark

to analyze AMI data. They argue that Spark's streaming properties, as well as its flexibility and scalability, makes it a valuable SPE to handle the challenges in an AMI.

Furthermore, Carvalho et. al [33] proposes distributed stream processing in the AMI with Apache Storm. Although Flink was available to them as an alternative of Storm, the authors state that since Flink's windowing features were not yet fully developed, Storm was a better alternative at that point in time. Using generated data, Carvalho et. al showed that their system produced high throughput (in the tens of thousands tuples/s) and was highly scalable.

An interesting application is put forward in [34], where adaptive messaging-rates from smart meters are explored. Using IBM Infosphere, another SPE, they implement a query which aims to balance demand and supply of energy in the AMI. When the AMI has enough power, the messaging rates can be lowered in order to decrease bandwidth and computational resources. But when energy consumption is near capacity, and accurate readings of the consumption are critical in order to prevent blackouts, messaging-rates are increased to offer higher granularity in consumption data. Using this technique, the authors show that the consumed bandwidth can be decreased by 50%, while still meeting accuracy requirements when load is high.

## 7.2 Data provenance

In this thesis, we enriched streaming queries with provenance capture for queries aimed at finding faulty smart meters in an energy distribution network. An early work that combines provenance with streaming applications is [17], where Glavic et al. introduce the idea of *Operator instrumentation*, which means that the standard operators of SPEs are extended. Their implementation of this method, named Ariadne, achieves provenance collection by annotating tuples with variable size metadata, which has the drawback that the annotations may grow arbitrarily large. Although this thesis does not focus on improving provenance capture it shows that there are other potential methods, and we note that Ariadne was only evaluated on generated tuples unlike this thesis.

Most closely related to this thesis is the extended version of GeneaLog [8] presented in Ananke [20], both authored by Palyvos-Giannas et al. Like Ariadne, this tool (extended GeneaLog) leverages operator instrumentation and has indeed already been evaluated on queries from a smart grid, specifically to hourly energy consumption data from smart meters. In this thesis we build on this body of work by capturing provenance in new AMI data streams with different rates such as voltage and current data. Further, we implement a larger amount of queries this domain, with varied semantics for a broader evaluation of the provenance capture method.

## 7.3 Changepoint detection

Another changepoint detection algorithm, the *Bayesian Changepoint Detection* is a well known algorithm for finding changepoints. The Bayesian method relies on statistically calculating the probability of a value belonging to the same distribution as previous values. In [35], this method is modified and tailored towards being run in a streaming environment, by modifying it to run *online*. That is to say that it reacts to changes directly, without having the entire time series ready to analyze, significantly reducing the latency of detecting a changepoint. However, online changepoint detection algorithms suffer from decreased accuracy compared to *offline* algorithms. For our and GE's purposes, minimizing latency is not a vital aspect in detecting a changepoint, but rather the accuracy. Furthermore, the most practical way to implement this functionality in Flink is to place all functionality into an aggregate. The aggregate function will receive all data collected in the window at once, essentially making it run offline, yielding results when the entire time series has been collected.

As an alternative to changepoint detection for detecting reversal of wires, Jokar et al. [3] propose an advanced method of surveying the AMI in search for customers deliberately tampering with their own smart meter. The method works by first observing a given neighborhood from a master node, to estimate technical energy losses (losses in wiring and transformers) as well as observing the usual consumption behaviors of the areas inhabitants. When this training phase is completed, the method transitions into a surveillance mode. If the supplied power into the local network suddenly differs from the billed power, as well as the algorithm detecting abnormal consumption behavior, an alert is created. While this method bears promising results and great capabilities, we deem the complexity to be out of scope of this thesis to attempt to implement a similar application in Flink. Furthermore, the ethical and legal circumstances concerning profiling the consumption patterns of single customers are problematic at best, and criminal at worst.

# 8

# Conclusion

The AMI provides excellent opportunities with regard to continuous measurement and fault detection. However, its great capacity also entails a vast amount of data that needs to be processed. Recent Swedish regulations demand more frequent updates from all meters in the AMI [13], thus also increasing the demand for processing the data. In these settings, stream processing is a suitable tool. In order not to lose the information about the cause of analysis results, one must maintain it during the process through provenance capture.

In this thesis, a number of queries have been enriched with state-of-the-art data provenance capture using an extended GeneaLog [8, 20]. The queries utilize different streams available in the AMI to find common faults affecting the smart meters. The accuracy of one query was studied in particular to detect faultily installed meters in the AMI by applying changepoint detection. The results show a relatively high sensitivity, finding 83% of induced changepoints in generated data, making it a useful method combined with provenance.

Using two months of recorded data from a real-world AMI, the evaluation of the system show varied performance on a per-query basis. Latency, throughput, memory footprint, and CPU usage were measured for each query with and without provenance. The overhead resulting from provenance is mostly manageable, manifesting an increased latency between 24–42% and a decreased throughput in the range of 22–34%. However, one of the tested queries shows significant degradation, where the overhead reaches above 200% in latency and memory consumption. It was found that a major contributing factor to the added overhead of provenance for all queries is increased memory consumption which in turn affects both latency and throughput negatively.

Enhancing queries with provenance is viable, albeit special care should be taken when constructing these queries in order to avoid instances where significant overhead occurs. It is therefore critical to determine if the provenance is needed at all, in order to avoid enriching queries that do not have any use for it and will thus only waste resources. Furthermore, one of the larger constraints when enriching queries using [20] is the limitation of using standard operators. This puts an extra burden on the programmer and might decrease its efficiency where more suitable methods are made off-limits.

Finally, we show that the storage needs can be dramatically reduced by utilizing stream processing with provenance capture. For data unrelated to billing customers, we show that storage needs can be decreased by up to 99% by only storing the analysis result and its provenance. However, this reduction is dependent on the number of queries in use and how frequent the alerts are, as well as the ratio of data that is used for billing. As the adoption of more modern smart meters progress, this ratio will reduce and we, therefore, expect greater gains in terms of reducing storage needs, by using the system studied in this thesis.

# Bibliography


[1] Joris van Rooij. "Data stream processing meets the Advanced Metering Infrastructure: possibilities, challenges and applications". Licentiate Thesis. Gothenburg, Sweden: Chalmers University of Technology and Gothenburg University, 2020. 90 pp. URL: https://research.chalmers.se/publication/519945 (visited on 11/15/2021).

[2] Caio César Oba Ramos et al. "A New Approach for Nontechnical Losses Detection Based on Optimum-Path Forest". In: *IEEE Transactions on Power Systems* 26.1 (Feb. 2011), pp. 181–189. ISSN: 0885-8950, 1558-0679. DOI: 10.1109/TPWRS.2010.2051823. URL: http://ieeexplore.ieee.org/document/5530391/ (visited on 05/03/2022).

[3] Paria Jokar, Nasim Arianpoo, and Victor C. M. Leung. "Electricity Theft Detection in AMI Using Customers' Consumption Patterns". In: *IEEE Transactions on Smart Grid* 7.1 (Jan. 2016). Conference Name: IEEE Transactions on Smart Grid, pp. 216–226. ISSN: 1949-3061. DOI: 10.1109/TSG.2015.2425222.

[4] Viktor Botev et al. "Detecting non-technical energy losses through structural periodic patterns in AMI data". In: *2016 IEEE International Conference on Big Data (Big Data)*. 2016 IEEE International Conference on Big Data (Big Data). Washington DC,USA: IEEE, Dec. 2016, pp. 3121–3130. ISBN: 978-1-4673-9005-7. DOI: 10.1109/BigData.2016.7840967. URL: http://ieeexplore.ieee.org/document/7840967/ (visited on 12/02/2021).

[5] Ulrich Greveler et al. "Multimedia Content Identification Through Smart Meter Power Usage Profiles". In: (), p. 8.

[6] Joris van Rooij, Vincenzo Gulisano, and Marina Papatriantafilou. "LoCoVolt: Distributed Detection of Broken Meters in Smart Grids through Stream Processing". In: *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*. DEBS '18: The 12th ACM International Conference on Distributed and Event-based Systems. Hamilton New Zealand: ACM, June 25, 2018, pp. 171–182. ISBN: 978-1-4503-5782-1. DOI: 10.1145/3210284.3210298. URL: https://dl.acm.org/doi/10.1145/3210284.3210298 (visited on 04/14/2022).

[7] Paris Carbone et al. "Apache Flink™: Stream and Batch Processing in a Single Engine". In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* (2015), p. 12.

[8] Dimitris Palyvos-Giannas, Vincenzo Gulisano, and Marina Papatriantafilou. "GeneaLog: Fine-grained data streaming provenance in cyber-physical systems". In: *Parallel Computing* 89 (Nov. 2019), p. 102552. ISSN: 01678191. DOI: 10.1016/j.parco.2019.102552. URL: https://linkinghub.elsevier.com/retrieve/pii/S0167819119301437 (visited on 10/25/2021).

[9] Hyo-Sang Lim, Yang-Sae Moon, and Elisa Bertino. "Research issues in data provenance for streaming environments". In: *Proceedings of the 2nd SIGSPATIAL ACM GIS 2009 International Workshop on Security and Privacy in GIS and LBS - SPRINGL '09*. the 2nd SIGSPATIAL ACM GIS 2009 International Workshop. Seattle, Washington: ACM Press, 2009, p. 58. ISBN: 978-1-60558-853-7. DOI: 10.1145/1667502.1667516. URL: http://portal.acm.org/citation.cfm?doid=1667502.1667516 (visited on 04/29/2022).

[10] Qi Wang et al. "You Are What You Do: Hunting Stealthy Malware via Data Provenance Analysis". In: *Proceedings 2020 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. San Diego, CA: Internet Society, 2020. ISBN: 978-1-891562-61-7. DOI: 10.14722/ndss.2020.24167. URL: https://www.ndss-symposium.org/wp-content/uploads/2020/02/24167.pdf (visited on 12/09/2021).

[11] Patrick McDaniel and Stephen McLaughlin. "Security and Privacy Challenges in the Smart Grid". In: *IEEE Security & Privacy Magazine* 7.3 (May 2009), pp. 75–77. ISSN: 1540-7993. DOI: 10.1109/MSP.2009.76. URL: http://ieeexplore.ieee.org/document/5054916/ (visited on 02/07/2022).

[12] Ramyar Rashed Mohassel et al. "A survey on Advanced Metering Infrastructure". In: *International Journal of Electrical Power & Energy Systems* 63 (Dec. 2014), pp. 473–484. ISSN: 01420615. DOI: 10.1016/j.ijepes.2014.06.025. URL: https://linkinghub.elsevier.com/retrieve/pii/S0142061514003743 (visited on 02/14/2022).

[13] Infrastrukturdepartementet. *Förordning (1999:716) om mätning, beräkning och rapportering av överförd el*. URL: https://www.riksdagen.se/sv/dokument-lagar/dokument/svensk-forfattningssamling/forordning-1999716-om-matning-berakning-och_sfs-1999-716 (visited on 11/29/2021).

[14] Robert Czechowski and Anna Magdalena Kosek. "The most frequent energy theft techniques and hazards in present power energy consumption". en. In: *2016 Joint Workshop on Cyber- Physical Security and Resilience in Smart Grids (CPSR-SG)*. Vienna, Austria: IEEE, Apr. 2016, pp. 1–7. ISBN: 978-1-5090-1164-3. DOI: 10.1109/CPSRSG.2016.7684098. URL: http://ieeexplore.ieee.org/document/7684098/ (visited on 05/26/2022).

[15] Don Carney et al. "Monitoring Streams — A New Class of Data Management Applications". In: *VLDB '02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 2002, pp. 215–226. ISBN: 978-1-55860-869-6. DOI: 10.1016/B978-155860869-6/50027-5. URL: https://linkinghub.elsevier.com/retrieve/pii/B9781558608696500275 (visited on 02/18/2022).

[16] Vincenzo Gulisano et al. “StreamCloud: An Elastic and Scalable Data Streaming System”. In: *IEEE Transactions on Parallel and Distributed Systems* 23.12 (Dec. 2012). Conference Name: IEEE Transactions on Parallel and Distributed Systems, pp. 2351–2365. ISSN: 1558-2183. DOI: 10.1109/TPDS.2012.24.

[17] Boris Glavic et al. “Efficient Stream Provenance via Operator Instrumentation”. In: *ACM Transactions on Internet Technology* 14.1 (July 15, 2014), pp. 1–26. ISSN: 1533-5399, 1557-6051. DOI: 10.1145/2633689. URL: https://dl.acm.org/doi/10.1145/2633689 (visited on 11/07/2021).

[18] *Apache Storm*. URL: https://storm.apache.org/ (visited on 02/18/2022).

[19] Maninder Pal Singh, Mohammad A. Hoque, and Sasu Tarkoma. “A survey of systems for massive stream analytics”. In: *arXiv:1605.09021 [cs]* (June 4, 2016). arXiv: 1605.09021. URL: http://arxiv.org/abs/1605.09021 (visited on 05/05/2022).

[20] Dimitris Palyvos-Giannas et al. “Ananke: a streaming framework for live forward provenance”. In: *Proceedings of the VLDB Endowment* 14.3 (Nov. 2020), pp. 391–403. ISSN: 2150-8097. DOI: 10.14778/3430915.3430928. URL: https://dl.acm.org/doi/10.14778/3430915.3430928 (visited on 02/04/2022).

[21] Melanie Herschel, Ralf Diestelkämper, and Houssem Ben Lahmar. “A survey on provenance: What for? What form? What from?” In: *The VLDB Journal* 26.6 (Dec. 2017), pp. 881–906. ISSN: 1066-8888, 0949-877X. DOI: 10.1007/s00778-017-0486-1. URL: http://link.springer.com/10.1007/s00778-017-0486-1 (visited on 01/27/2022).

[22] Y Richard Wang and Stuart E Madnick. “A Polygen Model for Heterogeneous DatabaseSystems:” in: *Proceedings of the VLDB Conference* 16 (Jan. 1990), pp. 519–538. (Visited on 05/17/2022).

[23] Charles Truong, Laurent Oudre, and Nicolas Vayatis. “Selective review of offline change point detection methods”. In: *Signal Processing* 167 (Feb. 2020), p. 107299. ISSN: 01651684. DOI: 10.1016/j.sigpro.2019.107299. URL: https://linkinghub.elsevier.com/retrieve/pii/S0165168419303494 (visited on 03/14/2022).

[24] A. J. Scott and M. Knott. “A Cluster Analysis Method for Grouping Means in the Analysis of Variance”. In: *Biometrics* 30.3 (1974), pp. 507–512. URL: http://www.jstor.org/stable/2529204.

[25] Kamstrup. *Smarta elmätare med mobil IoT | OMNIA® e-meter*. URL: https://www.kamstrup.com/se-se/ellosningar/smarta-elmatare/meters/omnia-e-meter (visited on 02/04/2022).

[26] Joris van Rooij, Vincenzo Gulisano, and Marina Papatriantafilou. “TinTiN: T ravelling in t ime ( i f n ecessary) to deal with out-of-order data in streaming aggregation”. In: *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*. DEBS ’20: The 14th ACM International Conference on Distributed and Event-based Systems. Montreal Quebec Canada: ACM, July 13, 2020, pp. 141–152. ISBN: 978-1-4503-8028-7. DOI:

10.1145/3401025.3401769. URL: https://dl.acm.org/doi/10.1145/3401025.3401769 (visited on 05/05/2022).

[27] Bjorn 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*. 2011 2nd IEEE PES International Conference and Exhibition on "Innovative Smart Grid Technologies" (ISGT Europe). Manchester, United Kingdom: IEEE, Dec. 2011, pp. 1–8. ISBN: 978-1-4577-1421-4 978-1-4577-1422-1 978-1-4577-1420-7. DOI: 10.1109/ISGTEurope.2011.6162747. URL: http://ieeexplore.ieee.org/document/6162747/ (visited on 05/26/2022).

[28] 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 - MTAGS '09*. the 2nd Workshop. Portland, Oregon: ACM Press, 2009, pp. 1–10. ISBN: 978-1-60558-714-1. DOI: 10.1145/1646468.1646476. URL: http://portal.acm.org/citation.cfm?doid=1646468.1646476 (visited on 05/26/2022).

[29] Daniel Nurmi et al. "The Eucalyptus Open-Source Cloud-Computing System". In: *2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*. 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid. Shanghai, China: IEEE, 2009, pp. 124–131. ISBN: 978-1-4244-3935-5. DOI: 10.1109/CCGRID.2009.93. URL: http://ieeexplore.ieee.org/document/5071863/ (visited on 05/26/2022).

[30] Shibily Joseph and E. A. Jasmin. "Stream computing framework for outage detection in smart grid". In: *2015 International Conference on Power, Instrumentation, Control and Computing (PICC)*. 2015 International Conference on Power, Instrumentation, Control and Computing (PICC). Thrissur, India: IEEE, Dec. 2015, pp. 1–5. ISBN: 978-1-4673-8072-0. DOI: 10.1109/PICC.2015.7455744. URL: http://ieeexplore.ieee.org/document/7455744/ (visited on 05/05/2022).

[31] Shibily Joseph, Jasmin E.A., and Soumya Chandran. "Stream Computing: Opportunities and Challenges in Smart Grid". In: *Procedia Technology* 21 (2015), pp. 49–53. ISSN: 22120173. DOI: 10.1016/j.protcy.2015.10.008. URL: https://linkinghub.elsevier.com/retrieve/pii/S2212017315002364 (visited on 05/25/2022).

[32] Shyam R. et al. "Apache Spark a Big Data Analytics Platform for Smart Grid". In: *Procedia Technology* 21 (2015), pp. 171–178. ISSN: 22120173. DOI: 10.1016/j.protcy.2015.10.085. URL: https://linkinghub.elsevier.com/retrieve/pii/S2212017315003138 (visited on 05/25/2022).

[33] Otávio Carvalho, Eduardo Roloff, and Philippe O.A. Navaux. "A Distributed Stream Processing based Architecture for IoT Smart Grids Monitoring". In: *Companion Proceedings of the10th International Conference on Utility and Cloud Computing*. UCC '17: 10th International Conference on Utility and Cloud Computing. Austin Texas USA: ACM, Dec. 5, 2017, pp. 9–14. ISBN:

978-1-4503-5195-9. DOI: 10.1145/3147234.3148105. URL: https://dl.acm.org/doi/10.1145/3147234.3148105 (visited on 05/06/2022).

[34] Yogesh Simmhan et al. "Adaptive rate stream processing for smart grid applications on clouds". In: *Proceedings of the 2nd international workshop on Scientific cloud computing - ScienceCloud '11*. the 2nd international workshop. San Jose, California, USA: ACM Press, 2011, p. 33. ISBN: 978-1-4503-0699-7. DOI: 10.1145/1996109.1996116. URL: http://portal.acm.org/citation.cfm?doid=1996109.1996116 (visited on 02/01/2022).

[35] Ryan Prescott Adams and David J. C. MacKay. "Bayesian Online Changepoint Detection". In: *arXiv:0710.3742 [stat]* (Oct. 19, 2007). arXiv: 0710.3742. URL: http://arxiv.org/abs/0710.3742 (visited on 02/11/2022).

[36] *Liebre SPE*. URL: https://github.com/vincenzo-gulisano/Liebre (visited on 05/17/2022).

# A

# Appendix 1

## A.1 Data provenance in stream processing

GeneaLog [8], is a tool that solves the problem of fine-grained data provenance in the SPEs Apache Flink [7] and Liebre [36]. GeneaLog uses operator instrumentation to enrich queries with provenance, which adds the needed meta-data to tuples to ensure provenance. Each tuple is enriched with four fields: Type (T), $\text{Upstream}_1$ ($U_1$), $\text{Upstream}_2$ ($U_2$), and Next (N). T states which type of operator created the tuple, while the other attributes are references to other tuples that contributed to its own existence. For stateless operators, only $U_1$ will be set, to the tuple which created it. For the Join operator, $U_1$ is set to the tuple from the first stream, while $U_2$ is set to the tuple from the second stream.

Aggregation is from a provenance perspective the most challenging since it can have an arbitrary amount of contributing tuples. Provenance here is handled by setting $U_2$ to the first contributing tuple, and $U_1$ to the last contributing tuple. The N attribute for all contributing tuples is then set to the next contributor, if there is one, effectively creating a linked list. Moreover, the Filter operator, which doesn't create it's own output tuples but only forward them doesn't change the meta-data, since the tuple stays identical.

By recursively tracing the referenced tuple(s) in the output, one unravels all contributing parts of the result.

Through this strategy, GeneaLog can provide backward provenance with constant per-tuple overhead. The limited overhead leads to acceptable costs in terms of latency, throughput, and memory.

### A.1.1 Ananke

Ananke [20] is a tool which for our purposes can be seen as an extension of GeneaLog. Unlike GeneaLog, Ananke provides *live forward provenance* to find which sink tuples originate from a source tuple.

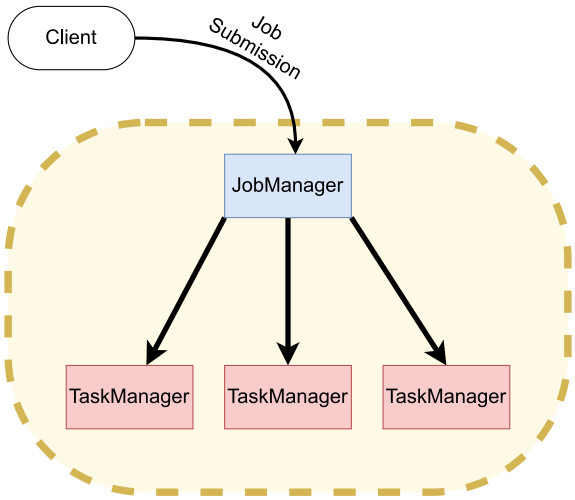



**Figure A.1:** The architecture of Apache Flink. A client submits a job to the Flink *cluster*. The *JobManager* assigns it to its one or more *TaskManagers*. Here there are three TaskManagers.

## A.2 Apache Flink

Apache Flink (or simply Flink), is a modern state-of-the-art Stream Processing Engine (SPE). It is built to handle both streaming and batch data. Flink is designed with a focus on throughput, latency, and scalability. It is an open source project, build upon the philosophy that most programs can be expressed through a pipeline. Additionally, Flink supports deployment in distributed systems, as well as handling out of order data. Used in production in several companies, it is a popular and mainstream SPE [7].

### A.2.1 Flink cluster

A Flink cluster consists of a JobManager process and one or more TaskManager processes, as can be seen in Figure A.1.

- **JobManager** works as a coordinator in the cluster, supervising the TaskManagers. The JobManager process takes care of allocating resources, managing jobs and exceptions, and communications with clients (which is a UI where you can submit jobs), etc.
- **Taskmanager**, or worker, takes care of the actual execution of the query.

All Flink applications run on a *cluster*. Currently, there are three types of clusters:

- **Job cluster** on demand when you only want to run one Flink job. Once the job has been executed, the job cluster will terminate itself. This is the preferred method if you aim to deploy a single long-running job where the added startup time doesn't matter, while at the same time having high stability requirements.

- **Application cluster** is similar to a job cluster, in that it only serves a single job, here called "application" for some reason. Here the cluster is started automatically, where you submit a jar-file with your job. The cluster lives until the job terminates, then shuts down. This approach is preferred in environments where the application is deployed on eg Kubernetes.

- **Session cluster** is a long-term cluster, which can run indefinitely. Here you submit your job to the cluster for execution, but the cluster lives on when your job has terminated and waits for new jobs to be submitted. This is suitable if you aim to run many small jobs, where a long startup time would be detrimental to overall performance.