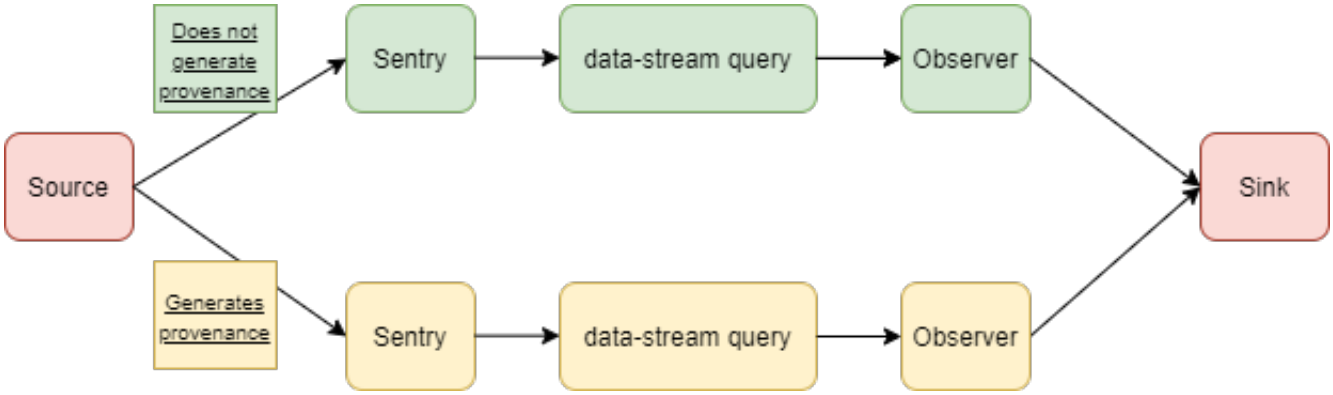
CHALMERS
UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG



Interactive Fine-Grained Provenance for Streaming-based Analysis Applications

Master's Thesis in Computer Science and Engineering

Mikael Gordani Shahri & Andréas Erlandsson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Master's Thesis 2020

Interactive Fine-Grained Provenance for Streaming-based Data Analysis Applications

Mikael Gordani Shahri & Andréas Erlandsson

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Interactive Fine-Grained Provenance for Streaming-based Analysis Applications

Supervisor: Vincenzo Massimiliano Gulisano, Department of Computer Science
Co-supervisor: Dimitris Palyvos-Giannis, Department of Computer Science
Examiner: Marina Papatriantafilou, Department of Computer Science

Master's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A modified version of the poster for the movie, Twins, which is also the name of the extension developed in this thesis.

Typeset in LaTeX
Gothenburg, Sweden 2020

Interactive Fine-Grained Provenance for Streaming-based Analysis Applications

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Streaming-based applications that process unbounded continuous streams of data, such as user activity on the web or sensor data, can be designed to detect critical events. With such an event, an application can benefit from maintaining the associated source data for further analysis. This can be achieved by fine-grained data provenance, which links each event back to the source data contributing to it.

In this thesis, the focus is on the current state-of-the-art data provenance technique called GeneaLog, which collects fine-grained data for cyber-physical systems and maintains it with low overhead. Generating provenance could be a heavy operation in certain applications, where the overhead produced will not always be negligible. Adjusting GeneaLog to become operational with the occurrence of a critical event, as opposed to always being operational, can be beneficial as it can reduce the unnecessary provenance generation.

The goal is to extend GeneaLog to generate provenance information *interactively* and evaluate during what conditions such an extension becomes beneficial. With this, GeneaLog and consequently data provenance techniques could be further introduced to a wider range of devices and applications, as it might reduce processing and memory overhead.

In this thesis, an extension for GeneaLog is proposed called Twins. To be able to activate and deactivate GeneaLog, Twins introduces a system which consists of two queries and a pair of special operators. The first query is equipped with standard operators and the second query with operators that generates provenance information. Initially, the first query processes tuples until a critical event is produced, which initiates a transition to the other query. With an absence of critical events after a transition, a transition is made back to the first query. This is performed by the special operators called the Ward operators, which are responsible to trigger and perform a transition between the queries.

A prototype of GeneaLog was used and extended in this thesis, which was built for the Stream Processing Engine Apache Flink. During the evaluation, the observed throughput of Twins resembled that of GeneaLog when provenance was active and that of a baseline query with no provenance generation when provenance was inactive. The preliminary results indicate that Twins can be beneficial in scenarios where generating provenance is not a negligible operation in terms of overhead.

Keywords: data provenance, streaming, apache flink, data analytics.

Acknowledgements

We would first like to thank our supervisors, Vincenzo Gulisano and Dimitris Palyvos-Giannis for their guidance, support and help throughout this thesis. We dare to say that it would not have been possible nor as educational without their help. We would also like to thank our friends and family, for their support throughout this thesis, motivating us to continue fighting the good fight, even though morale was low during certain periods. Lastly, we would like to thank our examiner, Marina Papatriantafilou for providing useful feedback, generally being helpful and showing interest for this thesis.

Andréas Erlandsson, Gothenburg, December 2020
Mikael Gordani Shahri, Gothenburg, December 2020

Contents

List of Figures

List of Tables

Listings

1

Introduction

The adoption of streaming analysis applications has increased over the recent years with the growth of Internet of Things (IoT) devices, as they have introduced new ways of collecting and processing data [1].

Streaming analytics works with real-time data, where it involves different mechanisms, as compared to working with historical data. Namely, it processes constantly updating data and does not necessarily maintain it [2]. Streaming-based analysis applications which are designed to detect unusual or critical outputs might benefit from saving the associated source-data, for further analysis.

When analyzing an unusual or critical output, answers to the following questions are of interest: *Where* is the source(s) that contributed to the output? *What* has been done to the data that produced it and by *whom*? Answers to these questions provide *trust in the data* by being able to verify the origin and *understandability* for a certain output. Additionally, they may also allow *reproducibility*, in order to recreate the behavior to further analyze it.

Finding relations between data in order to draw conclusions is advantageous from an analytical point of view. By being able to track the lineage of data, it makes it possible to discover the undergone *transformation* done to the data.

Data provenance or simply *provenance* is a concept that refers to the documentation of where a piece of data comes from and the processes/methodology by which it was produced [3].

Large scientific experiments can generate enormous amounts if data. For example the *ATLAS experiment* at CERN, where the amount of data produced was accumulated to be around 100 petabytes (PB) of data (both raw and processed)[4]. They concluded that provenance was needed to preserve both the data and its processing history, since it became difficult to analyze the result. Machine learning applications have used provenance data to build models for anomaly detection to prevent stealthy impersonation malware attacks in operating systems [5]. In relational databases, large portions of data is not entered manually by a user. Understanding such data can be hard without extensive knowledge about the data's origin and processing history, without additional information [6].

1.1 Problem definition

The current state-of-the-art provenance technique for data streaming, GeneaLog, is designed to always generate provenance information throughout the entirety of a data stream [7]. A streaming-based analysis application consists of several operators which creates, forwards and transforms data-points and potentially creates an output. To be able to use GeneaLog, the operators of GeneaLog are used, which are extended with extra functionality, as opposed to operators found in common SPEs. These operators annotates the data-points with metadata, as they progress through the system. This is later used to distinguish contributing or non-contributing data-points for a produced output.

However, if no output is produced, the data-points will still be annotated, resulting in extra overhead. Regardless if the overhead is considered to be minimal, without the presence of an output, there is no provenance to be generated and thus unwanted computational power and memory have been used.

1.1.1 Motivation

It could be beneficial for data provenance techniques in data streaming to generate provenance information *interactively*, namely *activate* the generation of provenance information with the occurrence of an output and later *deactivate* with an absence of output, as this could lower the computational power- and memory requirements. However, an interactive functionality comes with a cost. When the provenance generation becomes activated with the occurrence of an output, it will reduce the total amount of generated provenance information.

1.1.2 Research questions

In this thesis, the goal is to extend the GeneaLog framework to be able to generate provenance information interactively and evaluate if it results in a reduction in overhead, since the procedure of annotating the data-points is only performed when needed.

This thesis will answer three research questions:

Q1: Can GeneaLog be extended to become activated and deactivated automatically for arbitrary portions of a data stream?

Q2: When provenance information is not generated continuously, the total amount of provenance will be decreased, as opposed to a continuous generation. How much provenance is lost?

Q3: By introducing an additional functionality to GeneaLog, does the performance become affected? Does the overhead produced from the procedure of activating/deactivating GeneaLog atone for the potential reduction of overhead, achieved from the procedure?

1.2 Challenges

By extending GeneaLog, a new functionality will be introduced, which will allow the framework to *interactively* become enabled and disabled, with the occurrence of an output. This will imply the following requirements, to realize such an extension.

- *Design:* The design of a model, where the functionality of activating and deactivating provenance can be applied, using GeneaLog.
- *Preserve correctness:* At a certain point in time, the generation of provenance information will either start or stop. To preserve correctness, the ongoing computations that are taking place in the pipeline will either have to finish or be recomputed. This means that the approach is required to guarantee *no loss* nor *duplication* of output.
- *Special operator:* To activate and deactivate provenance generation, the operators in a query will be required to change their behavior. This means that there has to be a *special operator* which decides whether provenance generation should be active or inactive, based on the information carried by the data-points.
- *Operator communication:* The special operator will eventually decide whether provenance generation should become active or not, which requires that the remaining operators change their behavior. This means that when GeneaLog should become active, the special operators requires to propagate this instruction to the remaining operators, i.e., communication both upstream and downstream between the operators will be required.
- *Avoid circumlocution:* To achieve an interactive behavior, GeneaLog will be extended. As previously mentioned, new operators and a protocol for preserving the correctness will be required. This means that by satisfying the requirements, the extension will contribute with overhead. The additional overhead that is created from the extension should not atone for the potentially reduced overhead that is possibly achieved by extending GeneaLog in this matter. Furthermore, while provenance is *active*, the performance should be equal to GeneaLog and while *inactive*, it should be equal the performance when GeneaLog is not used.

1.3 Scope

The scope of this thesis is to extend a prototype of GeneaLog, implemented for the Stream Processing Engine (SPE) Apache Flink. The prototype is extended to be able to interactively activate the generation of provenance information with the occurrence of an output and later deactivate with the absence of output.

1.3.1 Limitations

In this thesis, the following limitations are set in order to limit the scope:

- *Triggers for activation and deactivation:* With different analysis applications, triggers for activation and deactivation can vary and thus will be user-defined. In this thesis, the triggers will be defined as general as possible, as the thesis focuses more on the functionality of activating/deactivating provenance generation as opposed to optimizing the triggers for when to activate and deactivate.

- *Determinism:* The GeneaLog framework is designed to work for deterministic streaming applications, since determinism is crucial to identify the source data contributing to each output event [7]. Therefore, the tuples are assumed to arrive in timestamp order.

- *Distributed execution:* The GeneaLog framework is designed to work for single node deployments as well as distributed deployments. In this thesis, the focus lies on singe node deployments.

1.4 Contribution

The overall contribution of this thesis is an extension to the current state-of-the-art data provenance framework, GeneaLog. The extension is called Twins and allows the framework to interactively activate and deactivate the generation of provenance information, for streaming-based analysis applications deployed within a single node.

Twins involves transitioning between two queries, where one is equipped with non-instrumented operators while the other is equipped with GeneaLogs instrumented operators. To guarantee that no loss-, incorrect- nor duplicated output is produced during a transition, two special operators are introduced called the Ward Operators. They consists of a pair, namely a Sentry and an Observer, which are responsible for the activation and the deactivation of GeneaLog, as they coordinate the transition between the two queries.

Furthermore, Twins was evaluated for several different scenarios to evaluate its behavior during different conditions. Twins showed to be beneficial for scenarios which involved a high processing rate of tuples, where it matched the performance of GeneaLog when active and inactive.

1.5 Outline

The report is organized as follows. Chapter 2 provides the theoretical background to the subject of this thesis, followed by methodology in Chapter 3. Chapter 4 covers the implementation of the methodology presented in the previous chapter followed by an evaluation of the implementation in Chapter 5. Chapter 6 provides a discussion regarding the evaluation and the implementation followed by Chapter 7,

which discusses related work for this thesis. Lastly, Chapter 8 concludes the thesis.

2

Preliminaries

This chapter contains the required background regarding the subject of the thesis. It covers the terminology used throughout the report, the basic building blocks of stream processing, Apache Flink, data provenance and finally, a detailed description of the GeneaLog framework.

2.1 Stream processing

Stream processing is a programming paradigm where the processing of data is done while the data is in motion, i.e., computing on data as it is produced or being received. Unlike batch processing, which stores the data and then performs computations, the data does not necessarily need to be stored in streaming applications [2].

The basic building blocks of stream processing are *streams* and *transformations*. Each stream processing pipeline starts with one or more *source(s)* and ends in one or more *sink(s)*, i.e., one or more entry point(s) and one or more end point(s).

As data continuously flows within the pipeline, transformations can be made on the data, such as computing statistical operations to, e.g., extract valuable information.

2.1.1 Data streams

A data stream, denoted S, can be characterized as an unbounded continuous flow of data. The elements of the data stream are referred to as *tuples* or *events*, denoted t. Thus, an unbounded stream can be denoted as:

$$S = t_1, t_2, t_3, \ldots \tag{2.1}$$

The tuples can be defined as records within different categories of information, where each tuple has the same set of attributes, $a_1, a_2, ..., a_n$ and a timestamp ts. Thus, a tuple can be denoted as:

$$t =< ts, a_1, a_2, \ldots, a_n > \tag{2.2}$$

As tuples undergo transformations, they will eventually reach a sink, where the tuples are referred to as *sink tuples*. These tuples are considered to be the output of a pipeline, and are referred to as *critical events*.

2.1.2 Stream operators

The entities responsible for transforming, creating and forwarding tuples are called *operators*.

The standard operators commonly found in SPEs such as [8] and [9], can be categorized in two groups, *stateless* operators and *stateful* operators. The stateless operators processes input tuples one by one while the stateful operators processes several input tuples as a group.

The most common standard stateless operators that are native in common SPEs are the following:

- `Map`, produces one or more output tuple(s) for each input tuple, by selecting one or several of its attributes, to optionally use in functions.
- `Filter`, drops tuples from the stream depending on a boolean condition.
- `Multiplex`, copies the tuples of one stream to multiple streams.
- `Union`, takes multiple input streams and merges them into one.

With stateful operators producing an output tuple that depends on multiple input tuples, a *window* is required to define a scope for its computation. The most common *stateful* operators are the following:

- `Aggregate`, takes several input tuples and produces one output tuple. The input tuples are grouped by a key, such as a common attribute. The group of tuples are aggregated with functions such as *min, max* or *sum* within a given window.
- `Join`, combines the tuples from two streams, based on their attributes and satisfying a predicate which is within a given window.

2.1.3 Queries

Pipelines are user-defined queries that contain a chain of operators with the purpose of producing a specific output. In data streaming, queries are defined as *continuous*, i.e., they are permanently installed and not executed once, as in relational databases [2].

An example query

An example use case for a streaming-based analysis application would be a pipeline which monitors the temperatures of racks in a data center. Data racks often have a

high power consumption, which can result in them being overheated [10]. Thus, it is important to monitor each rack, to ensure that they maintain a stable temperature.

A query for this example is illustrated in the Figure 2.1, where each rack sends their temperature along with their unique id to the stream. To calculate the average temperature for each individual rack, the input tuples are grouped by their *id*, to be able to separate different rack's tuples.

When they reach the `Aggregate` operator, the average temperature of the last 60 seconds for each rack is calculated every 20 seconds. A `Filter` then checks if a rack's average temperature is above a certain threshold, as it would indicate that the rack is about to overheat, which would require cooling to reduce the temperature. With the occurrence of a sink tuple, it indicates that a rack requires cooling.

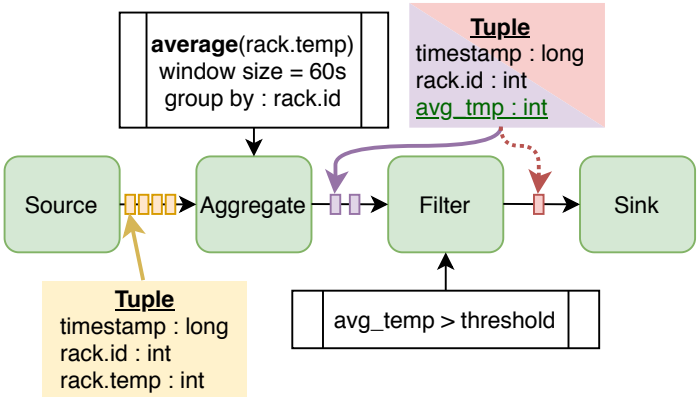


Figure 2.1: Illustration of an example query which detects overheated data racks.

2.1.4 Windows

Stateful operators require a scope to produce a tuple. The scope is referred to as *windows* which can either be *time-based* or *data-driven*.

A window's time frame (if it has one) progresses at all times, though a window is only created as soon as the first tuple that should belong to it arrives, and is *complete* when either the time has passed or when the maximum number of tuples is reached. The exact start and end point of, e.g., 30 minute windows is set in the following manner: `00:00:00.000 - 00:29:59.999`, `00:30:00.000 - 00:59:59.999`, etc [11]. The progression of windows can also vary, as they can either be *sliding windows* or *tumbling windows*. The length of a window is called *window size*, denotes as WS.

Sliding Windows

The sliding windows starts a new window every few tuples, thus overlapping the windows and in effect "sliding" with the data stream. The time between when two windows start, is called *window advance*, denoted WA. As windows overlap, tuples will be part of multiple windows. On completion of a window, all the tuples that

are not referenced in other windows will be dropped [12]. The progression of sliding windows is illustrated in Figure 2.2.

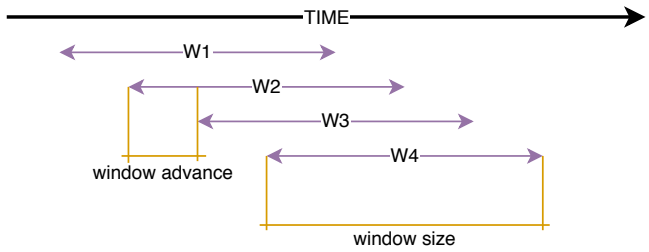


Figure 2.2: Illustration of sliding windows.

Tumbling Windows

Tumbling windows drops all tuples when filled or time is exceeded and immediately starts refilling a new window. The windows are not allowed to overlap, as illustrated in Figure 2.3.

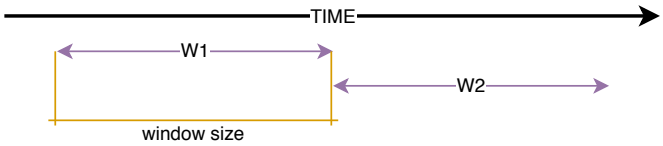


Figure 2.3: Illustration of tumbling windows.

2.1.5 Watermarks

As queries can be executed in parallel and tuples might arrive out-of-order, there is no way of knowing if each operator of the pipeline has processed all tuples or that tuples have arrived at all.

A mechanism that solves this are *watermarks*, which are unique and identifiable timestamps, W, which measure the *event time* progress of the stream, based on the timestamp of the tuples. Watermarks define a lower bound on the event time that have been processed by the current operator, this bound then acts as an allowed lateness for each tuple. Thus, if any tuple with timestamp lower than W should arrive, it will be discarded [12].

This is illustrated in Figure 2.4, which shows how tuples and its timestamp can arrive in a stream, both in-order and out-of-order. In the figure, a watermark is created with timestamp 10, $W(10)$, and placed on the stream, declaring that the stream has progressed to timestamp 10. This is then incremented to 20, $W(20)$, and attached to the stream in the same manner. Meaning that if a tuple with lower timestamp should arrive, it will be too late and consequently be dropped. An example of this is illustrated with the crossed-out tuple in the out-of-order stream in the figure, which has timestamp 19 while the watermark is $W(20)$.

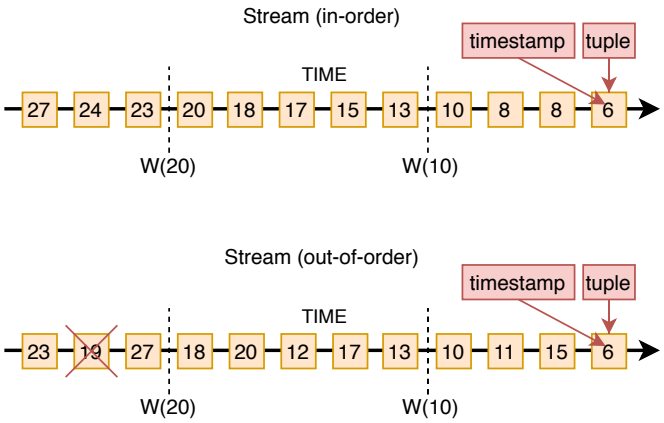


Figure 2.4: Illustration of watermarks in both in-order and out-of-order streams. In both of the streams, the tuple with timestamp 6 is the first tuple processed.

Watermarks are often generated at the source operators, and sent downstream. The other operators then utilize these to guarantee progression and completeness. If a watermark, W, appears (from upstream) at an operator, it sets its internal event time to W and then sends it along the stream, it then drops all late tuples with lower timestamp.

2.2 Apache Flink

Apache Flink is an open-source distributed processing engine and framework for stateful computations of bounded and unbounded data (i.e., batch and streaming), developed by the Apache Software Foundation [12].

When distributed, i.e., working within multiple servers (a *cluster*), it integrates with common cluster resource managers (i.e. software for managing clusters of servers), but can also be setup to run as a stand-alone cluster. It is also designed to scale and can utilize a virtually unlimited amount of CPUs, main memory, disk and network. The memory usage is optimized for local main memory access and if needed has a access-efficient on-disk data structure for extra memory.

Flink also provides fault tolerance and failure recovery through several features: consistent and efficient check-pointing of states, guarantees that data is only written out once, load balancing in case of node crashes and high-availability setup that eliminates all single-point-of-failures. It also has features to update, migrate, suspend, resume, monitor and control applications during execution.

Some use cases for Flink are: *Event-driven* applications, which are stateful queries that react on incoming events, *Data Analytics* applications, where information is extracted and presented continuously from a stream and *Data Pipeline* applications, which is a common way of converting and moving data between storage systems.

Flink also uses watermarks to synchronize operators throughout the pipeline, send-

ing sporadic watermarks that traverse trough operators unhindered and reaches all parts of the pipeline.

2.2.1 Levels of interaction

Flink provides three levels of APIs for interacting with Flink and building applications [12]:

- *SQL and Table API* - Highest level of abstraction. The Table API is a declarative domain-specific language (DSL), centered around tables and can seamlessly be converted to DataStreams/DataSets (i.e., to lower level abstractions. See below.). The SQL level allows users to define SQL queries for both stream and batch processing, which closely interact with the Table API.

- *DataStream API (and DataSet API for batch)* - This is considered the core API and in practice is where most developers interact with Flink. It provides the coding primitives used in all common stream processing operations, such as: windowing, `Aggregate`, `Filter`, `Join`, etc, and is available for both Java and Scala via functions such as: `map()`, `reduce()`, `filter()`, `aggregate()`, etc. These functions can either take lambda expressions, inner anonymous classes or a class as arguments where the logic for the operator is located.

- *Stateful Stream Processing* - It is the lowest level and most expressive function interface. It is a building block, embedded into the DataStream API and accessed via *Process Functions*, e.g., `KeyedProcessFunction<...>{}`, making it possible to integrate this level of abstraction for certain operations when coding in a higher level. It provides the most freedom to fully process events and gives full control of time and state.

2.2.2 Tasks

In Flink, a *task* is referred to as the basic unit of execution, which is responsible for a query's operators. Each task exists within a single process and consequently has its own process memory allocated. In order to execute queries more efficiently, a query can be split into several tasks, which will be several separate processes located at either the same node or distributed across different nodes [12].

When a query is divided in to two or more tasks (and/or nodes), the *Send* & *Receive* operators are used in order to forward tuples between the tasks (and/or nodes). As illustrated in Figure 2.5.

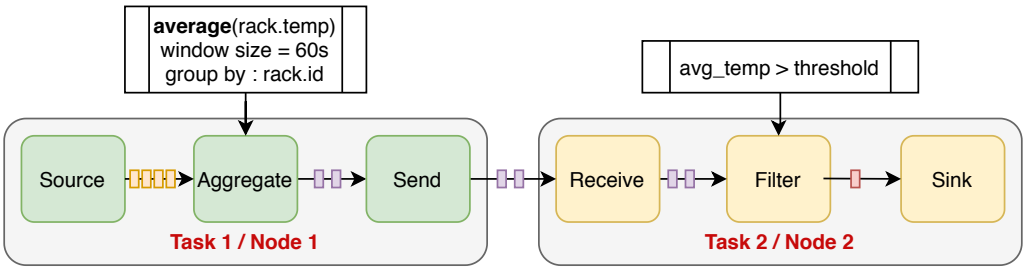


Figure 2.5: Illustration of a query divided into two tasks (and/or nodes).

A task is also split into several *parallel instances* for execution and each parallel instance processes a subset of the task's input data [13]. Defining the amount of parallel instances can either be done for individual operators or the entire task.

2.2.3 Programming model

In Listing 2.1, the example query from Section 2.1 is implemented, using the DataStream API.

Listing 2.1: Flink example

```
1  import org.apache.flink.*;
2
3  public class RackTemperature{
4      public static void main(String[] args) throws Exception {
5          int threshold = 75;
6          StreamExecutionEnvironment env =
7              StreamExecutionEnvironment.
8                  getExecutionEnvironment();
9          DataStream<RackTuple> dataStream = env
10                 .keyBy(t -> t.rackID)
11                 .window(SlidingEventTimeWindows
12                     .of(Time.seconds(60), Time.seconds(20)))
13                 .aggregate(new AverageAggregate())
14                 .filter(t -> t.temp >= threshold)
15                 .addSink(new Sink());
16         env.execute("Rack Temperature");
17     }
18 }
```

Flink is first initialized and its input is set up, at lines 1 through 8. Then the stream is declared, called `dataStream`, at line 9, after which transformations can be applied.

To calculate the average temperature for each rack, the `Aggregate` operator requires the stream to be *keyed*, as is done at line 10, where `keyBy()` groups the tuples by an identifier, here by `rackID`.

Then the `Aggregate` operator requires a window to work with, which is declared at line 11 and 12, where a sliding window of length 60 seconds and a new window is

started every 20 seconds, is set. The `timeWindow()` method also requires a *window function*, i.e. a stateful operator, to follow it. Thus the `aggregate()` method is declared at line 13, which takes in a class, `new AverageAggregate()`, which has all the aggregation logic and will calculate the average temperature. Because the stream is a keyed-stream the aggregate will take the average for each rack separately but within the same window.

At line 14, the `Filter` operator is applied with the method `filter()`, which can simply check if any of the average temperatures are above the given threshold.

The stream will now only contain critical events and so a sink operator can be added, which is done at line 15 with the method `addSink()` which, again, takes in a class `new Sink()` which does have the sink logic, where it can print, log, activate cooling, etc. Line 16 is the initiation of the environment and so the query will begin to execute.

Figure 2.6 is an illustration of how the code for the example query correlates to the graph used to illustrate the query.

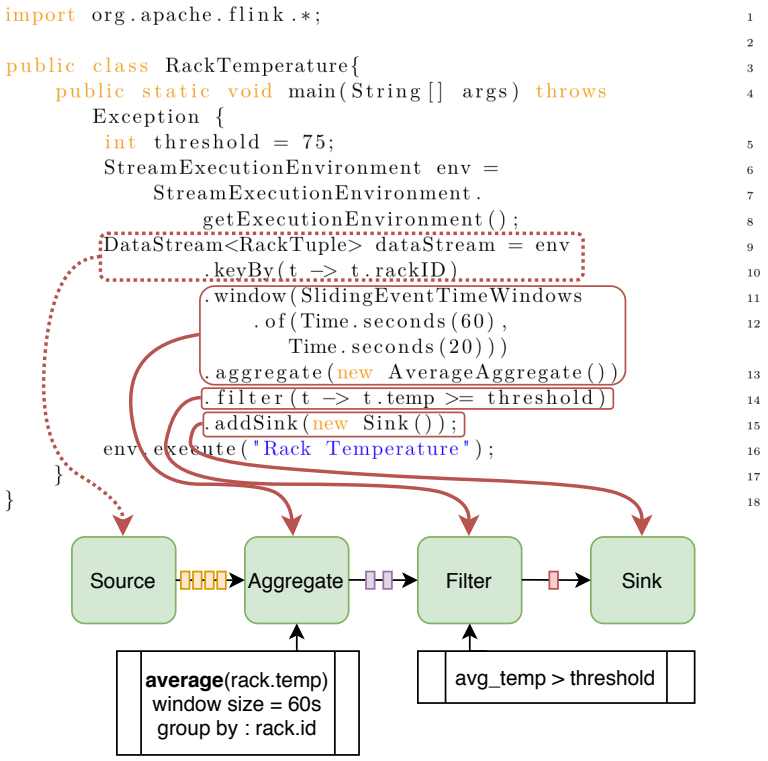


Figure 2.6: Illustration of the correlation between the graph and the code.

2.3 Data provenance

Data provenance or just *provenance*, refers to a record which contains information such as the origin of a piece of data, together with an explanation of the production

process. Data provenance techniques consists of generating *metadata* on individual pieces of data, which is referred to as *provenance information*. The *provenance data* is then created by collecting the individual pieces of contributing data using the provenance information.

In Figure 2.7, provenance for the example query from Section 2.1.3 is illustrated. As tuples are transformed throughout a query, provenance can be used to derive the relation of a sink tuple to its contributing source tuples, using the provenance information. The dotted arrow (the red arrow) shows the relation between the sink tuple and the source tuples, while the non-dotted arrow (the blue arrow) shows the relation to the intermediate tuples.

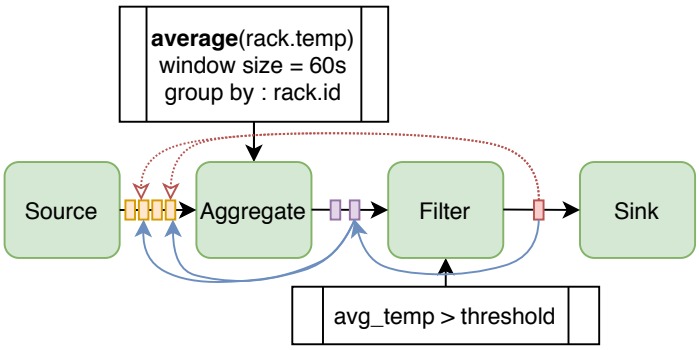


Figure 2.7: Illustration of provenance on the example query.

2.4 GeneaLog

GeneaLog is the current state-of-the-art data provenance framework for data streaming [7]. The aim of GeneaLog is to lower the requirements of provenance for resource-constrained devices, by addressing issues that were present with prior techniques, such as variable-size annotations and temporarily storing all of the input tuples [14]. With data streaming being introduced to a more wider range of devices, the requirement for provenance needs to be decreased as well [7].

GeneaLog relies on small, fixed-size annotations and makes use of memory pointers in order to distinguish contributing tuples, i.e., no need to temporarily save every input tuples and distinguish them later on.

2.4.1 Provenance information

Provenance information generated by GeneaLog consists of four attributes.

- $Type$ (T)
- $Upstream_1$ (U_1)
- $Upstream_2$ (U_2)
- $Next$ (N)

The meta-attribute *Type* describes the origin of the tuple, i.e., the operator that created it. The remaining three attributes are memory pointers, which are used to link back a tuple to a contributing tuple, which is explained in Section 2.4.2.2.

2.4.2 Instrumented operators

GeneaLog is based on a technique called *operator instrumentation*, which extends the operators with functionality to generate provenance information. With this functionality, each operator is able to annotate its output tuple with provenance information, based on the provenance annotations of its input.

The relations between an input tuple and an output tuple is defined as follows [7]:

Definition 1. An input tuple t_{IN} to an operator, OP, contributes to an output tuple, t_{OUT} of OP, depending on OP, if:

i. OP is a Filter, Union or Receive and $t_{OUT} = t_{IN}$

ii. OP is a Map, Send or a Multiplex 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}.

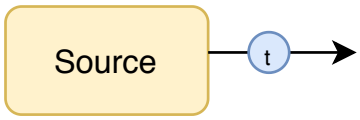
iv. OP is a Join and t_{IN} is a tuple from a pair of tuples within time distance WS for which the Join predicate holds.

Each of the operators mentioned in Section 2.1.2 are extended to generate provenance in the following way.

`Source`, creates tuples which does not depend on any other tuple. Since the source is the entry point for the tuples, the memory pointers are not set, as listed in Figure 2.8a.

- T = *SOURCE*
- U_1 = N/A
- U_2 = N/A
- N = N/A

(a) Meta-data attributes.



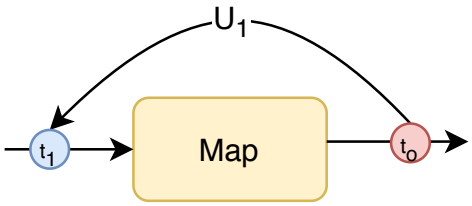
(b) Illustration of the Source operator.

Figure 2.8: Source operator.

`Map`, takes an output tuple, t_o, and points it to the input tuple t_1, which contributes to it with U_1, as listed in Figure 2.9a and illustrated in Figure 2.9b.

- T = *MAP*
- $U_1 = t_1$
- U_2 = N/A
- N = N/A

(a) Meta-data attributes.



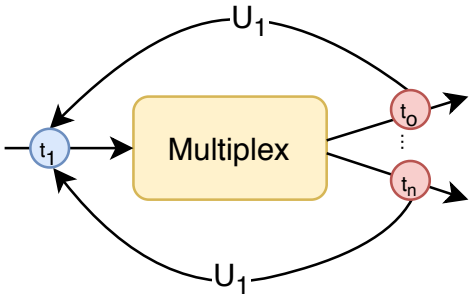
(b) Illustration of the Map operator.

Figure 2.9: Instrumented Map operator.

`Multiplex`, creates a copy of each input tuple to one or several output streams. Similarly to `Map`, each output tuple, t_o points to the tuple that contributes to it using U_1, as listed in Figure 2.10a and illustrated in Figure 2.10b.

- T = *MULTIPLEX*
- $U_1 = t_1$
- U_2 = N/A
- N = N/A

(a) Meta-data attributes.



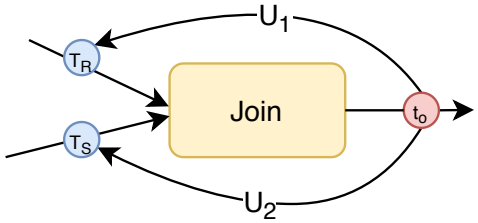
(b) Illustration of the Multiplex operator.

Figure 2.10: Instrumented Multiplex operator.

`Join`, produces an output tuple, t_o in which has exactly *two contributing tuples*, T_R and T_S. Assuming that $T_R.ts > T_S.ts$, as listed in Figure 2.11a and illustrated in Figure 2.11b.

- T = JOIN
- $U_1 = T_R$
- $U_2 = T_S$
- N = N/A

(a) Meta-data attributes.



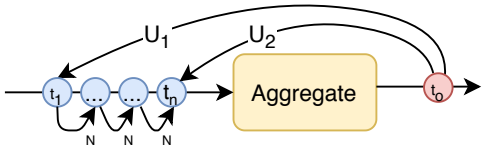
(b) Illustration of the Join operator.

Figure 2.11: Instrumented Join operator.

`Aggregate`, takes several input tuples, $t_1, \ldots, t_n$ and produces an output tuple, t_o, where t_1 refers to the earliest tuple. For the tuples in between, it sets $U_1 = t_n$ and $U_2 = t_1$, and if $n > 1$. It also sets $t_i.N = t_{i+1}$ for $i = 1, \ldots, n-1$, as listed in Figure 2.12a and illustrated in Figure 2.12b.

- T = AGGREGATE
- $U_1 = t_1$
- $U_2 = t_n$
- N = t_{i+1}

(a) Meta-data attributes.



(b) Illustration of the Aggregate operator.

Figure 2.12: Instrumented Aggregate operator.

`Filter & Union`, does not create new tuples but rather forwards them, thus no instrumentation is defined for them.

`Send & Receive`, are used to forward tuples between different processes or nodes. The Send operator will set T to $REMOTE$ as long as it is not a source tuple. The reason behind this is to locally distinguish tuples produced at other tasks.

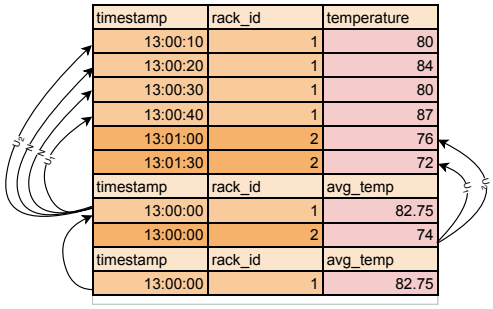
2.4.2.1 Contribution Graph

By linking tuples together with memory pointers, a path from a sink tuple to one or several source tuples can be made. This implies the existence of a *contribution graph*, where the edges represent the contributing source tuples of a sink tuple. By traversing the graph according to the provenance method listed in Appendix A.1, the *originating tuple(s)* can be retrieved.

An originating tuple is defined as the following [7]:

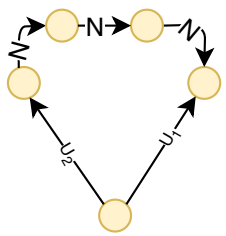
Definition 2. A tuple, t', is referred to as being an *originating tuple* of t, if t' is returned by the provenance method in Appendix A.1 as contributing to t.

This is illustrated in the table in Figure 2.13a and the graph in Figure 2.13b, which shows the example query from Section 2.1.3.



timestamp	rack_id	temperature
13:00:10	1	80
13:00:20	1	84
13:00:30	1	80
13:00:40	1	87
13:01:00	2	76
13:01:30	2	72
timestamp	rack_id	avg_temp
13:00:00	1	82.75
13:00:00	2	74
timestamp	rack_id	avg_temp
13:00:00	1	82.75

(a) Relation between tuples.



(b) Contribution graph.

Figure 2.13: Illustration of the relation between the contribution graph and the table of tuples.

2.4.2.2 Distinguishing contributing tuples

The attributes, U_1, U_2, N represent the edges in the contribution graph of a sink tuple, as well as being actual memory references. GeneaLog takes advantage of the automatic memory management found in languages such as Java or C++ for distinguishing contributing tuples [15].

GeneaLog prevents the contributing tuples from being reclaimed as long as they are (potentially) part of a sink tuple. This means that the contributing source tuples will have a reference count greater than zero, while the non contributing tuples will have a reference count of zero (sooner or later), which will be safely reclaimed by the garbage collector [16].

2.4.3 Intra-task provenance

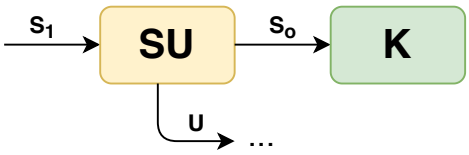
With the operators being responsible for generating provenance information, GeneaLog has a custom operator called the *single unfolder operator*, which is used to retrieve the provenance data.

Single Unfolder operator: SU

The *single unfolder operator*, denoted as SU, is an operator in GeneaLog which provides provenance in a query. It takes a single stream as input, S_1, and produces two output streams, S_0 & U, as illustrated in Figure 2.14b.

- S_1, input data stream
- S_0, an exact copy of S_1
- U, an unfolded stream of S_1

(a) Parameters for the SU-operator.



(b) Illustration of the SU operator.

Figure 2.14: Illustration of the SU operator and its parameters.

By placing a SU operator prior to each sink (with S_0 feeding the sink), provenance is provided through the *unfolded stream*.

Definition 3. An *unfolded stream* is defined as a stream where each tuple $t \in S$ is replaced by its *originating tuples* combined with t's attributes.

The unfolded stream can either be forwarded to another stream or consumed, e.g., storing it to disk.

Implementation using standard operators

SU can be implemented using the standard operators, `Map` & `Multiplex` as illustrated in Figure 2.15. The `Multiplex` is used for duplicating S_1, into S_o and S_M and the `Map` is then used to transform S_M into U, by applying the provenance method

(see Appendix A.1). This will produce, for each sink tuple, t_{SINK}, a tuple carrying t_{SINK}'s attributes and the originating source tuples.

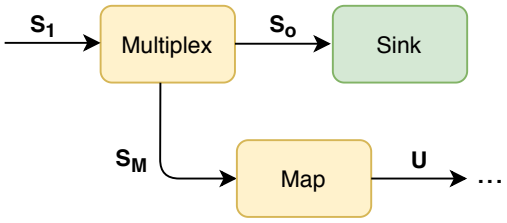


Figure 2.15: Implementation of the SU operator.

2.4.4 Inter-task provenance

For multi-task deployment, relying on memory pointers is no longer sufficient. Since tasks can be placed at different machines, shared memory is no longer present, thus providing provenance through the *SU* operator needs to be adjusted accordingly.

GeneaLog provides two different methods for inter-task deployments, *explicit inter-task provenance* and *implicit inter-task provenance*, where the provenance method is modified accordingly.

Additionally, the meta-data attributes described in Section 2.4.1 are extended by one additional meta-attribute, ID, which is an unique identifier for each tuple.

2.4.4.1 Explicit inter-task provenance

For explicit inter-task provenance, additional instrumented operators are added to be responsible for sharing each respective task's source tuples. Similar to intra-task provenance, these operators are combined in order to create the *multi-stream unfolding operator*, which is used to achieve provenance in multi-task deployments.

Multi-stream unfolding operator: MU

The *multi-stream unfolding operator*, denoted MU, is an operator in GeneaLog which provides provenance for multi-task deployments. It takes one *derived* stream and an arbitrary number of *upstream unfolded delivering streams* and produces one output stream as illustrated in Figure 2.16.

- *Unfolded delivering streams*, which initially are streams that either feeds a sink or is created by a Send operator as input. Then, processed by an SU-operator.
- *A derived stream*, contains tuples that are forwarded to the output stream, if $T = SOURCE$. Alternatively, it is replaced by the sequence: $t_1, \ldots, t_i, \ldots, t_n$ found in any of the upstreams for which $t_i.ID = t.ID_0$

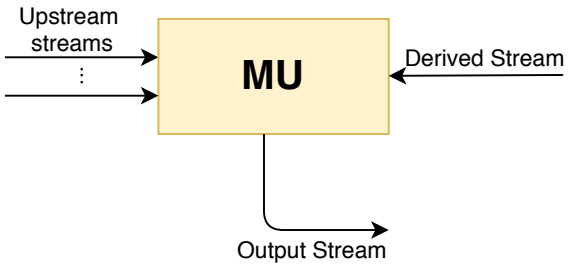


Figure 2.16: Illustration of the MU-operator and its parameters.

Implementation using standard operators

When implementing the MU operator, the `Join` operator is used, which, in the simplest scenario, merges one upstream stream S_D and one single derived stream S_T that does not contain any tuples with $T = SOURCE$. The predicate of which is used to match a tuple $t_D \in S_D$ and $t_T \in S_T$ if $t_D.ID == t_T.ID_0$. This is illustrated in Figure 2.17.

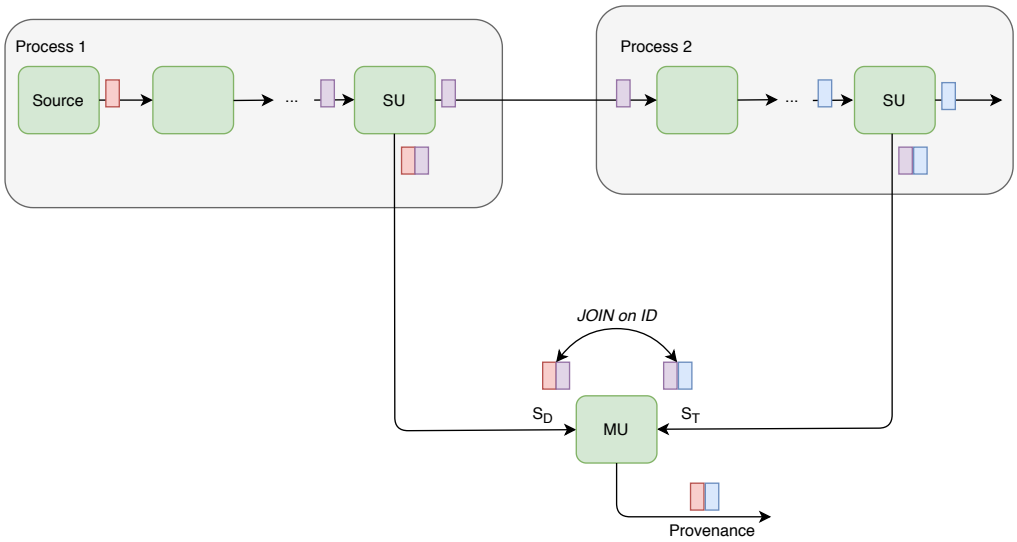


Figure 2.17: Illustration of providing provenance with a query split at two processes.

More than two processes

Depending on the number of *upstream streams* and the value of T on the tuples in the *derived stream*, the MU operator have to be extended with additional operators.

For two or more *upstream streams*, an additional `Union` operator is used. The `Union` operator deterministically merges the tuples into one stream before passing it to the `Join` operator. Additionally, if the derived stream contains tuples where $T = SOURCE$, a `Multiplex` operator, two `Filter` operators and an additional `Union` is used. The `Multiplex` operator feeds the tuples from the derived stream into both

of the Filter operators, which allows ($filter_2$) the tuples where $T = SOURCE$ to the `Union` operator and the remaining to the `Join` operator. This is illustrated in Section 2.18, where the optional operators are surrounded in gray.

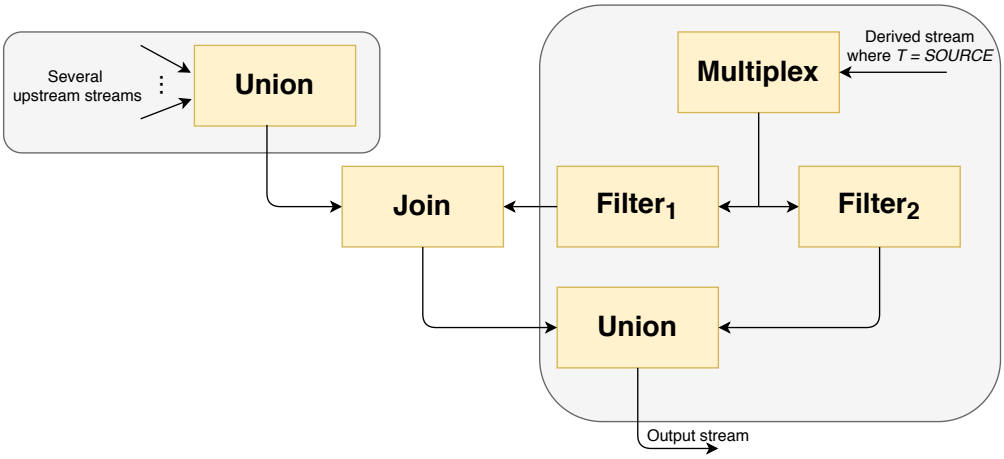


Figure 2.18: Extended MU operator for more than two processes/nodes.

By extending the MU operator with these additional operators, with more than two processes, a tree of MU-operators is created, where they are feeding each other as illustrated in Figure 2.19. Though, depending on the query and the deployment, the amount of operators can become very large.

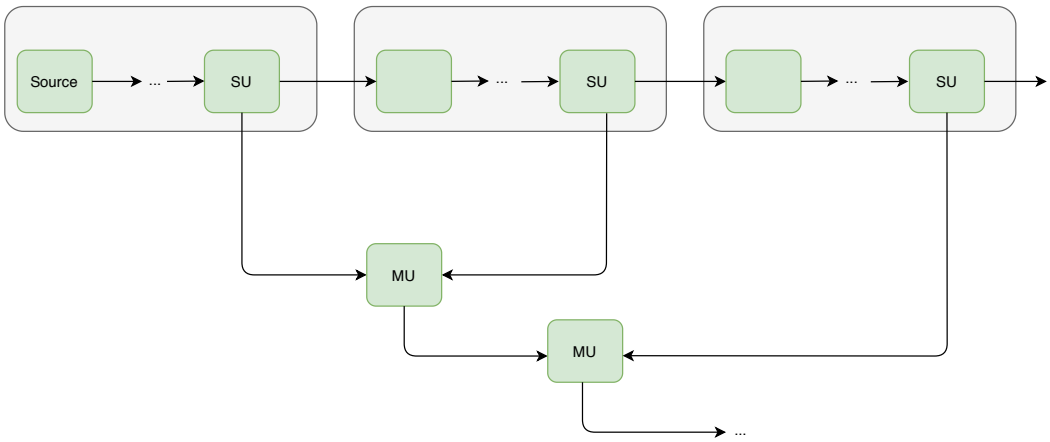


Figure 2.19: A tree of MU operators.

2.4.4.2 Implicit inter-task provenance

For implicit inter-task provenance, the goal is to solely rely on the query's operators to perform the additional operations to share source tuples among tasks instead of adding additional operators. These operators are the `Send` & the `Receive` operator.

By integrating the traversal of the contribution graph (previously used in the *SU* operator, see Section 2.4.3) each tuple with $T = REMOTE$ is serialized locally

in each `Send` operators task together with its contributing source tuples. At the corresponding `Receive` operator located at a different task, it will be received and then deserialized.

Upon deserialization, the remote tuple is reduced down to its source tuples, using the attributes U_1 & U_2 to point to the earliest and latest source tuple.

When more than one source tuple is found in the contribution graph, the source tuples' N meta-attribute is used, similarly to GeneaLog's `Aggregate` operator (See 2.4.2). Thus, the provenance method is altered in such a way that tuples with $T = REMOTE$ and $T = AGGREGATE$ follow similar semantics, see Appendix A.2. This is illustrated in the Figure 2.20.

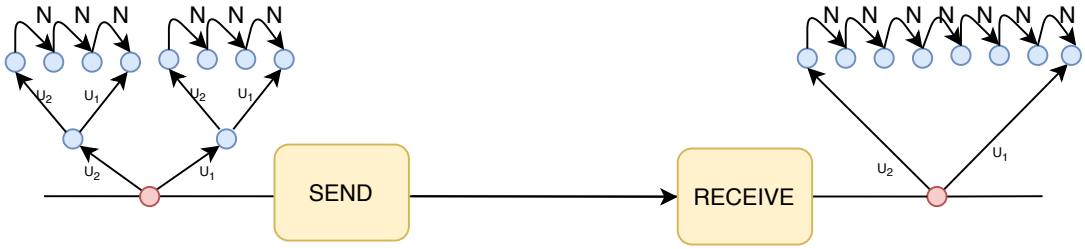


Figure 2.20: Illustration of the changed behavior of `Send` & `Receive` operator.

3

Method

This chapter covers the methodology on how to activate and deactivate GeneaLog and the solutions to the anticipated requirements presented in Section 1.2. This chapter will be more theoretical while the following chapter, Chapter 4, will contain more technical details regarding the implementation and the usage of Apache Flink's API.

In Section 3.1, the architectural model for activating and deactivating provenance is described. In Section 3.2, the special operators with the required functionality for activating and deactivating provenance are described. In Section 3.3, a technique for preventing loss and duplicated output is described. In Section 3.4, the procedure of activating and deactivating provenance is described, using the special operators and the technique for preserving correctness. In Section 3.5, an extension of the methodology is described to potentially minimize the amount of lost provenance, as mentioned in Section 1.1.2.

3.1 Twin query approach

GeneaLog creates provenance information via instrumented operators, which use up to four meta-data attributes depending on the operator (see Section 2.4.2). In the example application presented in Section 2.1.3, a critical event is defined as the average temperature of a rack exceeding a certain threshold. With the occurrence of an output, provenance becomes of interest and should be activated. This means that until provenance becomes desirable, the tuples should be processed by *non-instrumented operators*, and then by *instrumented operators*.

By having two identical queries, one using non-instrumented operators and another using instrumented operators, activating provenance can be achieved by manipulating the direction of the source tuples from the non-instrumented query to the instrumented query. Figure 3.1 illustrates the twin query approach by using the example application presented in Section 2.1.3.

Initially, the source tuples will be sent to the non-instrumented query and when a critical event is produced, the source tuples will be directed towards the instrumented query instead. This will require a protocol to ensure correctness of the

stateful computations, this will be further explored in Section 3.3.

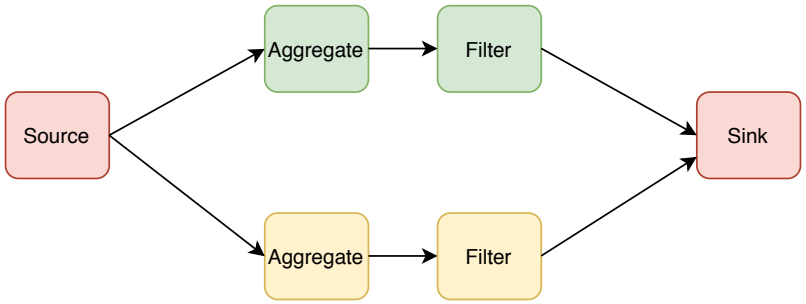


Figure 3.1: Design of the twin query approach. The upper query (colored in green) uses non-instrumented operators while the lower query (colored in yellow) uses instrumented operators.

Transition between the two queries

At first, tuples will be directed towards only one query e.g, the non-instrumented query, and if an output is produced, the tuples will be directed towards the instrumented query. Additionally, at some point, the tuples will be re-directed back towards to the non-instrumented query.

When only one query is ingesting tuples, it is considered to be active while the other query is considered to be inactive. During a transition, the deactivated query will start to ingest tuples and will be referred to as a *activating query* while the former query will eventually stop ingesting tuples, and is referred to as a *deactivating query*.

Definition 4. A query, q_A, is considered to be *activating* if it has previously been inactive i.e., it starts to ingest tuples.

Definition 5. A query, q_D, is considered to be *deactivating* if it has previously been active i.e., it stops ingesting tuples.

To achieve such a transition, a mechanism will be required which decides whether the next tuple should be sent to one query or the other (or both). This mechanism can be incorporated into a pair of special operators, which will mainly be responsible for the transitions. These special operators will be referred to as the *Ward operators*.

3.2 Ward operators

In this section, the Ward operators are presented. They handle the direction of the tuples and the transition between the queries.

As shown in Figure 3.1, both of the queries will share the same source, meaning the source tuples will be duplicated and sent to both of the queries. To ensure that only one query ingests tuples at a time, an operator will be placed in each query, allowing or denying tuples. By allowing tuples in one query while denying in the other, the

direction of the source tuples can be controlled. This operator is referred to as the *Sentry* operator, the first Ward operator. A Sentry operator has two states, active (allow tuples) or inactive (deny tuples). A transition between the queries is a change in their state.

As mentioned in Section 1.1.1, the generation of provenance information should become active with the occurrence of an output and then later become deactivated. For a transition to occur, it would require an additional operator that *observes* tuples throughout both of the queries and informs the respective Sentry operators to change their state. This operator is referred to as the *Observer* operator, the second Ward operator.

During a transition between the queries, there will be a *reconfiguration time*, which will have an impact on the amount of generated provenance information. By influencing the reconfiguration time, it could lower the amount of lost provenance information but increase the overhead.

Definition 6. The reconfiguration time, T_c, represent the time interval starting at the point in time from when a transition has been requested and ending at the point in time when a transition has been completed.

T_c could be influenced by the placement of the Ward operators. The placement for Sentries will be discussed in Section 3.2.1 and for Observers in Section 3.2.2. By increasing the amount of Ward operators in several positions in a query, a query can be *partially* activated preemptively. This could also influence the reconfiguration time. This will be further discussed in Section 3.5.

3.2.1 Sentry operators

The Sentry operator's task is to either allow tuples or deny them depending on its state, see Figure 3.2 and Listing 3.1. Both of the queries will be equipped with either one or several Sentry operators, and their placement can vary. A sentry can be defined as the following.

Definition 7. A Sentry, S, takes a tuple as its input, t, and depending on its state, S_{state}, it will either allow or deny the tuple. The value of S_{state} can either by `True` or `False`, which allows or denies the tuples respectively.

```
boolean state

function allowtuple()
function denytuple()
function changestate()
```

Listing 3.1: Sketch of the basic functionality of a Sentry operator

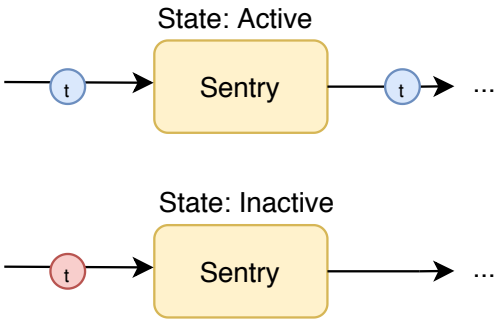


Figure 3.2: Illustration of a Sentry operator.

Placement of the Sentry ward

Notice that the placement of a Sentry could be in several positions throughout a query. By placing a Sentry close to a source, the remaining operators of the query fall under its control, i.e., they will not process any tuples unless the Sentry is activated. Furthermore, a Sentry can even be placed prior to the Sink, simply allowing or denying the output from reaching the sink.

Placing a Sentry after the source would decrease the processing overhead and result in a longer reconfiguration time, due to the type and amount of operators. If the transition is from the non-instrumented to the instrumented query, it would also result in less amount of provenance. By placing a Sentry closer to a sink, fewer operators will be dependent on its state, thus more provenance information would be generated but result in an increase of processing overhead.

For the example query presented in Section 2.1.3, the placement of a Sentry operator could be succeeding the source, as illustrated in Figure 3.3.

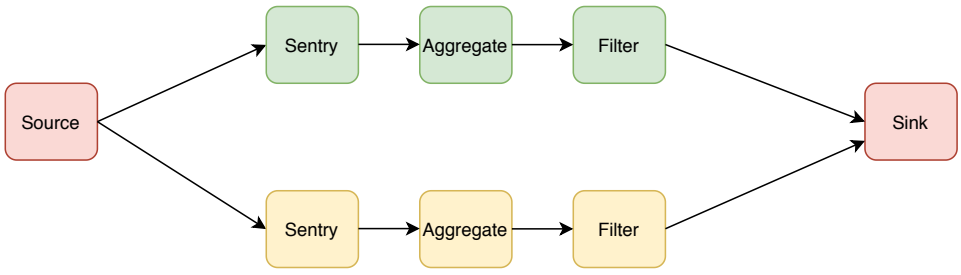


Figure 3.3: Placement of two Sentry operators, succeeding the source in both queries.

3.2.2 Observer operators

The Observer operator's task is to trigger the transition between the non-instrumented and the instrumented query. By observing incoming tuples, the Observer operator

should inform the Sentry operators to change their state accordingly, see Figure 3.4 and Listing 3.2. An Observer is defined as the following.

Definition 8. An Observer, O, takes a tuple as its input, t, and forwards the tuple to the subsequent operator(s). Depending on the tuple, an Observer will change its state, O_{state}, and trigger a transition to the other query.

```
boolean state

function changeState()
function informSentry()
```

Listing 3.2: Sketch of the basic functionality of an Observer operator.

Figure 3.4: Illustration of an Observer operator.

Placement of the Observer ward

Depending on the query, the placement of the Observer operator can vary. For the example query found in Section 2.1.3, placing the Observer operator prior to the sink would mean that the existence of the tuples themselves would be enough for the Observer to trigger a transition, as it is the end of the analysis pipeline. This placement is illustrated in Figure 3.5.

Other potential positions would be prior to the *Filter operator* or the *Aggregate operator*. When it is not placed prior to a sink, the Observer naturally can not detect critical events (sink tuples).

Assumption: The further tuples progress (or are produced) in the pipeline, the more likely it is for a critical event to happen. By placing an Observer early in the pipeline, it might imply switching query prior to the production of a sink tuple. This will be further explored in Section 3.5.

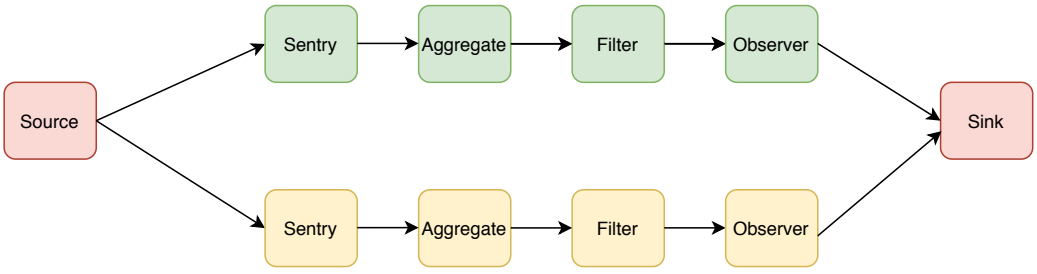


Figure 3.5: Two Observer operators placed prior to the sink in both queries.

User-defined logic for activation and and deactivation

As mentioned in Section 1.3.1, the functionality of observing tuples throughout the pipeline have to be user-defined, since with different applications, conditions for

activating provenance varies. Conditions for deactivation follows the same pattern, since with the absence of tuples, provenance should be deactivated. This is further explored in Section 3.4.2.

3.3 Correctness of output

Due to stateful operators and their associated windows, if a transition would take place without a protocol, it would result in incorrect calculations, i.e., duplicated and/or missing output. This means that the combined output from both the queries has to be the same as if every tuple was processed by just one query.

For the example query presented in Section 2.1.3, a new window is created every 20 seconds and completed after 60 seconds, which means that if a transition without any form of coordination would occur, the windows would be incomplete and could produce an incorrect output tuple (i.e., the windows would not receive the same amount of tuples that it normally would if the query would continue to ingest tuples).

Note: The windows of both queries will progress in parallel, i.e., start and end at the same time, even if there are no tuples in the pipeline (see Section 2.1.4).

In Figure 3.6, an illustration is provided to demonstrate the consequences of a transition without coordination. The illustration shows the `Aggregate` operators of the respective queries, where three windows are created. Each of the tuples are carrying the temperature of a data rack, which are aggregated to produce a tuple carrying the average temperature over a period of time.

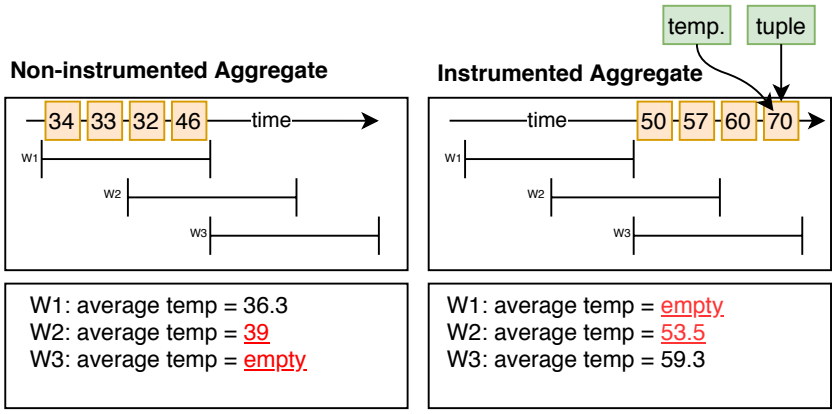


Figure 3.6: Illustration of possible miscalculations.

If a transition would occur when one or several windows are incomplete, it would result in one of the queries having the complete input for the first window, $W1$, while the other having the complete input for the third window, $W3$. The second window, $W2$, will not be correct in any query. To prevent these miscalculations, q_D is required to ingest enough tuples for the ongoing windows to be completed while q_A is required to drop the windows which has not received enough tuples to

complete its ongoing windows. Furthermore, the first complete window in q_A should correspond to the first incomplete window of q_D.

3.3.1 Overlapping ingestion of tuples

Whenever a Sentry in q_A changes its state, it will immediately start to allow tuples. This means that every tuple prior to the first tuple processed by a Sentry has already been- or is being processed by a stateful operator. Thus, the tuples resides in one (or more) ongoing window(s) in q_D. These ongoing windows need to be allowed to finish before fully transitioning to q_A, to ensure that no potential output is lost or duplicated.

Since windows can overlap (see Section 2.1.4), when a window is considered to be complete, there will still be windows present that are incomplete. During a transition, q_D will stop ingesting tuples as soon as a window is complete. However, there will still be *residual* tuples in the pipeline that contribute to one or several incomplete windows. Since the query is considered to be deactivated, whenever these windows are completed, the output that is potentially produced should be dropped. Simultaneously, there will be *early* tuples ingested in q_A which will contribute to one or several windows. These *early* windows needs to be dropped as well and only when the transition is complete will the output created by the windows no longer be dropped, i.e., output from windows in q_A are only allowed when the *last* complete window in q_D is finished. This means that during a transition, both of the queries will ingest tuples in parallel and potentially produce duplicated or incorrect output.

This is illustrated in Figure 3.7, which shows a transition from the non-instrumented query to the instrumented query and their respective windows. The box which covers $W1, W2$ and $W3$ in the non-instrumented query (the green box) represents the windows handled by the non-instrumented query, the box which covers $W4, W5$ and all future windows in the instrumented query (the blue box) represents the windows handled by the instrumented query and the red boxes are the windows that are dropped.

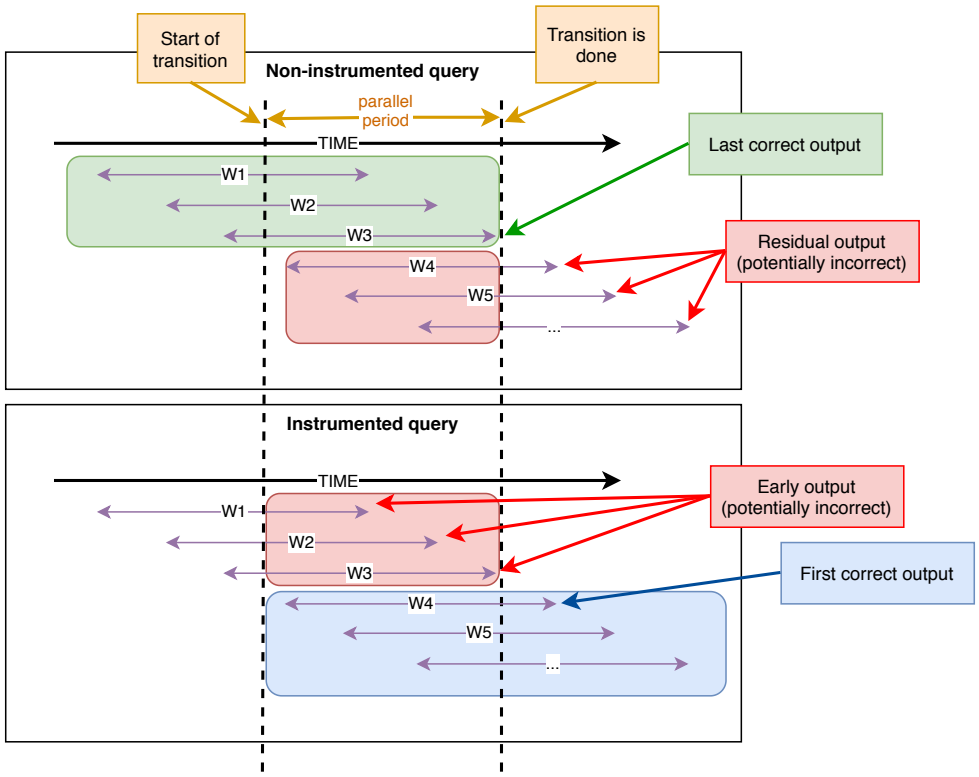


Figure 3.7: Overlapping windows from the perspective of both queries, with markings for when a transition starts and ends, as well as the last correct output and all residual output for the non-instrumented query and the first correct output and all early output for the instrumented query.

A transition which fulfills these requirements of preserving the correctness during a transition is referred to as a *safe transition*.

Definition 9. *A safe transition* is a transition from one query to another with no lost-, incorrect-, or duplicated output.

To achieve a safe transition, the usage of *timestamps* can be utilized to indicate if both queries have processed the same amount of data. As mentioned in Section 1.3.1, determinism is assumed i.e., the streams are timestamp-sorted.

3.3.2 Achieving a safe transition

As previously mentioned in Section 2.1.4, windows will be completed before even time strokes, e.g., 00:29:59.999. Additionally, the timestamp of each tuple created by a stateful operator will be the value of the point in time when the window it has contributed to has finished. This means if the point in time when a window is completed is 00:29:59.999, the tuple created by that window will have the timestamp 00:29:59.999.

During a transition, in q_D every tuple that is created at (or after) the start of the

first correct window of q_A will be considered to be *residual*, i.e., too late, and has to be dropped. Similarly in q_A, every tuple that is created prior (to the same window), is considered to be *early* and has to be dropped. The responsibility of dropping these tuples lies in the Observers, for both q_D and q_A, which will compare the timestamp of the (potential) tuples it ingest against ts_{first} as expressed in Equation 3.1, which represents the point in time when the first window in q_A has finished.

Determining residual and early tuples utilizes that the timestamp of the first tuple ingested in q_A, ts, the largest window size, WS_{max} and largest window advance, WA_{max}, is known. As mentioned in Section 3.3.1, early and residual tuples will potentially be present due to overlapping windows. The Observers determine the point in time where the first correct window is completed by following Equation 3.1.

$$ts_{first} = (ts + (WA_{max} - (ts \bmod WA_{max}))) + WS_{max} \tag{3.1}$$

With this, residual and early tuples are defined as following.

Definition 10. A *tuple*, t, is considered to be a *residual tuple*, t_r, if it has arrived at an Observer, O, after the point in time when the first correct window has finished in q_A, $ts >= ts_{first}$, as expressed in Equation 3.1 and illustrated in Figure 3.7.

Definition 11. A *tuple*, t, is considered to be an *early tuple*, t_e, if it has arrived at an Observer, O, before the point in time when the first correct window has finished in q_A, $ts < ts_{first}$, as expressed in Equation 3.1 and illustrated in Figure 3.7.

Furthermore, the Sentry in q_D must keep ingesting tuples until the last correct window is complete, which will be up until ts_{last}, as expressed in Equation 3.2. The right side of the Equation finds the end of the nearest new window from ts.

$$ts_{last} <= (ts + (WA_{max} - (ts \bmod WA_{max}))) + (WS_{max} - WA_{max}) \tag{3.2}$$

This is illustrated in Figure 3.8, where the top query (colored in green) is the deactivating query and the bottom query is the activating (colored in yellow). Note that when transitioning back from the bottom query to the top query the logic is reversed.

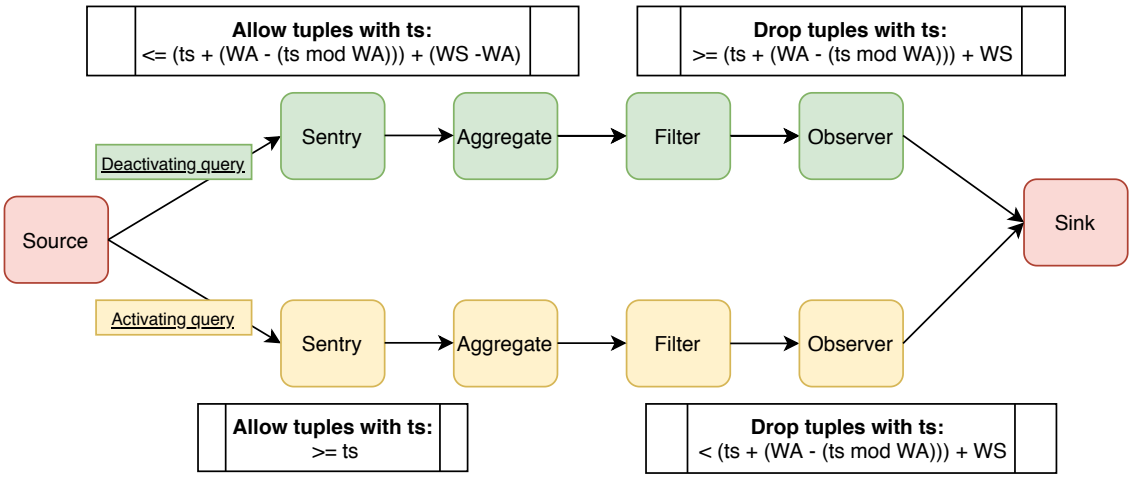


Figure 3.8: Allowed and dropped tuples based on timestamps in order to make a safe transition between the queries.

This ensures that the ongoing windows are completed and neither lost nor duplicated output is present during a transition. Furthermore, if several stateful operators are present in a query, a safe transition is achieved by utilizing the *largest window size* and its corresponding *window advance* in Equation 3.1 and 3.2.

Observation. Given a query, Q, equipped with two or more stateful operators, W_1, W_2, ..., W_n, the overlapping period required to achieve a safe transition can be defined using the largest window size, WS_{max} and window advance, WA_{max} of the respective stateful operators.

Example. During a safe transition, one last correct window needs to be finished in q_D before q_A can be considered active and reliably start to process tuples, without the possibility of producing either duplicated nor incorrect output. Assume two stateful operators, W_1 and W_2. Let the window size of W_1 be 120 and the window size of W_2 be 30.

If two or more stateful operators are placed subsequent of each other in a query, the tuple created by the former operator will be the input of the latter and so on. In this scenario, the point in time where one window of W_1 has been completed, three windows of W_2 has been completed. Additionally, the point in time where the second window of W_1 is created, the fourth window of W_2 will be created as well. This means that a tuple that is produced by the first operator, W_1 will be part of the window that is produced at the same time in the second operator, W_2 and so on.

If WS_{max}, and consequently the *last correct window*, would be based on W_2, it would imply that W_1 would never be allowed to be complete in neither q_D nor q_A. Thus the produced tuple will be incorrect and be (incorrectly) considered residual in q_D and not early in q_A. This is due to its production being after the period of what is defined as the last correct window. Notice that this will be the case regardless of

the order of the stateful operators, see Figure 3.9.

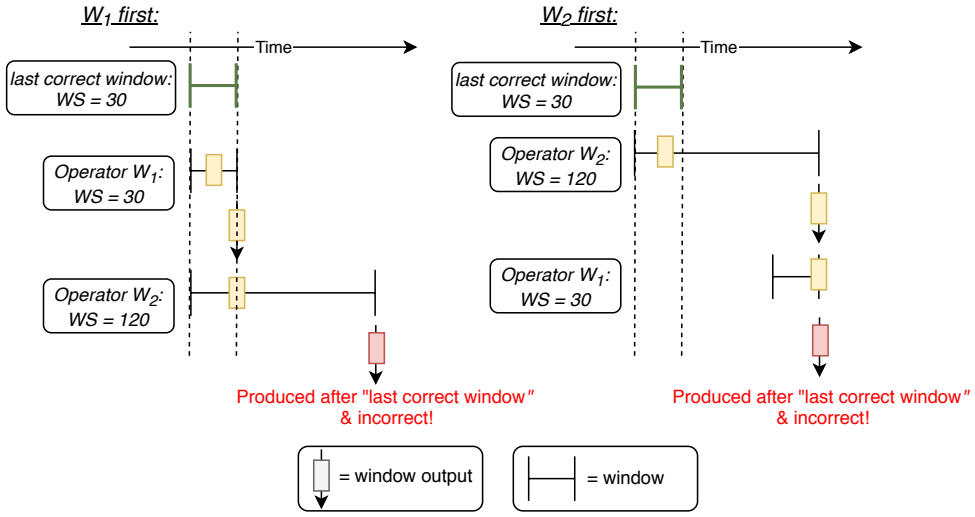


Figure 3.9: The leftmost illustration depicts W_1 as the first operator, its produced tuples will be the input of W_2. The rightmost illustration depicts W_2 as the first operator, its produced tuples will be the input of W_1. Both scenarios will produce incorrect tuples that are after WS_{max} and thus not part of the last correct window.

By basing WS_{max} on the *largest* window size, it allows q_D to create one last correct output before q_A can reliably take over. This is illustrated in Figure 3.10.

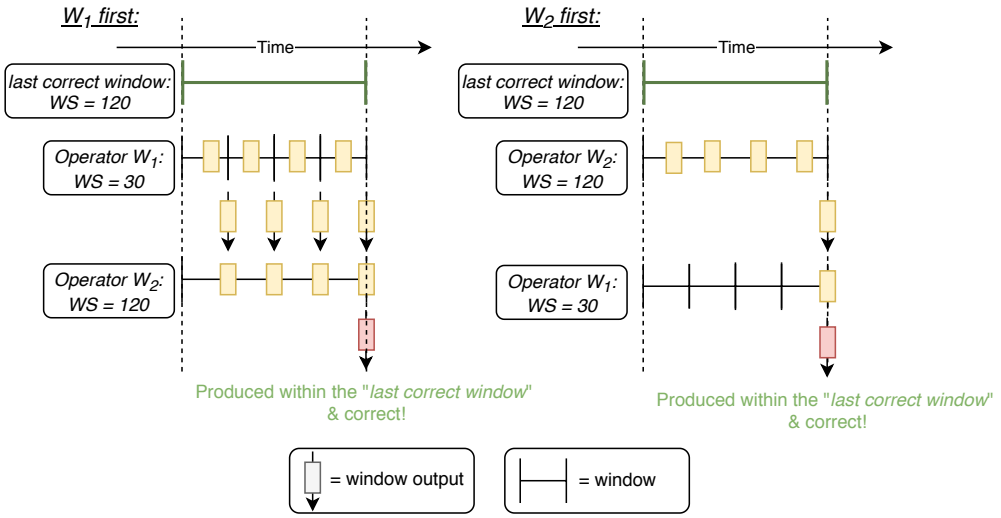


Figure 3.10: The leftmost illustration depicts W_1 as the first operator, its produced tuples will be the input of W_2. The rightmost illustration depicts W_2 as the first operator, its produced tuples will be the input of W_1. Both scenarios will produce correct tuples that are within WS_{max}, and thus part of the last correct window.

The last correct window will also be dependent on a window advance as it will

set how often windows are finished and how many windows run in parallel, thus it changes how windows are aligned, no matter the size of them.

Note: When all operators window advance (and size) are multiples of each bigger advance (and window), then the bigger advance will always end at the same time as one with smaller advance. The smaller will always finish one or more windows before the bigger finishes one.

Thus to ensure that windows still ends at the same time, in the same manner as Figure 3.10 illustrates, the same logic for choosing the largest window size holds for window advance as well; setting the WA_{max} to the largest window advance ensures that tuples from the shorter windows advance are part of a bigger and the bigger is allowed to finish.

Note: In this thesis, the use cases used in the evaluation does not involve window sizes nor window advances which are not multiples of each bigger size and advance respectively. Therefore, Theorem **??** does not cover such a scenario either.

3.4 Activating and deactivating provenance

By using the Twin query approach together with the Ward operators (Sentry & Observer), activating/deactivating provenance can be achieved by transitioning between the non-instrumented and the instrumented query. To preserve the correctness of the potential output produced during a transition, the safe transition technique presented in Section 3.3.2 is used.

In this section, the required steps of activating and deactivating provenance are presented. The placement of the Sentry operators will be succeeding the source, while the Observer operators will be prior to the Sink. This is referred to as the *default scheme*.

3.4.1 Activating provenance

Regardless of the placement of the Ward operators, activating provenance, i.e., transitioning to the instrumented query will consist of three steps. Initially, the Sentry located in the non-instrumented query will allow tuples while the Sentry located in instrumented query will deny tuples.

First, whenever an Observer is triggered (based on its user-defined condition) in the non-instrumented query, it will notify a Sentry placed in the instrumented query to start allowing tuples. At this point, a transition between the queries will begin. Secondly, the Sentry operators will communicate at which point in time the latter query should stop ingesting tuples, to achieve a safe transition without loss nor duplication of potential output. Lastly, the Observer operators placed in both queries will determine which tuples are considered to be residual/early and deny them.

3.4.2 Deactivating provenance

By activating provenance with the occurrence of an output, deactivation should occur with the absence of output. A user-defined condition can be used to decide when provenance is no longer required, e.g., after a given time. Furthermore, an Observer in the instrumented query can delay the transition back to the non-instrumented query, if output continues to be present. This will be dependent on how the condition is defined, and will naturally be different depending on the type of applications.

Deactivation will occur when the used-defined condition in an Observer placed in the instrumented query is met. It will notify a Sentry in the non-instrumented query to start to allowing tuples. Similarly to the activation procedure, the Sentry operators will communicate to achieve a safe transition and the Observers will determine which tuples are defined as early/residual in both of the queries.

Adjusting the example query presented in Section 2.1.3, to be able to both activate and deactivate provenance is illustrated in Figure 3.11.

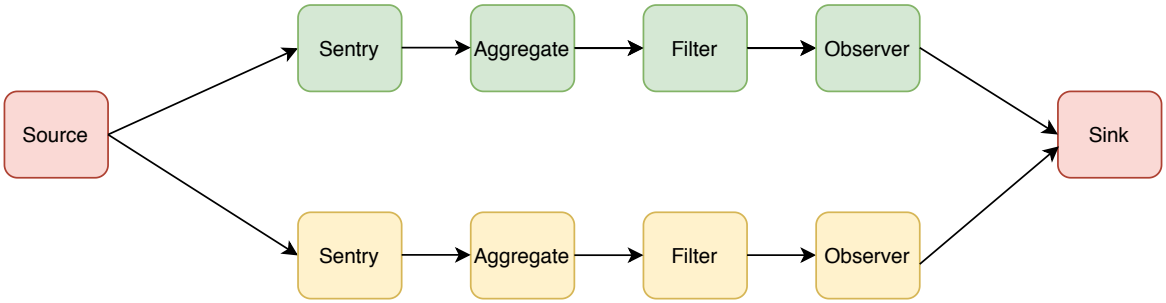


Figure 3.11: Query which can both activate and deactivate provenance by having Sentry- and Observer operators placed in both the non-instrumented- and the instrumented query.

3.5 Partial activation and deactivation

As tuples progress further through a pipeline, i.e., appearing closer to a sink, it becomes more likely for a tuple to actually reach a sink. This means that it might be more likely for a critical event to become produced.

This can be utilized to make the activation and deactivation of the instrumented query more dynamic and in a *partial manner*. Parts of the instrumented query can be activated preemptively, i.e., before a sink tuple is produced, if it seems likely that a critical event is about to happen, and reversely, parts can be deactivated as the situation changes and a critical event seems less likely.

Note that either the non-instrumented query or the instrumented query has to be active at all time, thus whenever the entire instrumented query is not active, the entire non-instrumented query has to be active.

This might mean that some instrumented operators might run in vain and slightly increase the hardware demand, but it could lower the reconfiguration time, and consequently increase the amount of provenance enough to motivate it.

3.5.1 Partial activation

Partial activation can be achieved by adding more than one Observer to the non-instrumented query. The instrumented query will then be split into *parts*, where a Sentry is responsible for each part. Each Observer is then connected to a Sentry in the instrumented query, which is responsible for allowing or denying tuples to one or several operators. Thus, each Observer activates one part of the query, until the entire query is active and only then deactivate the entire non-instrumented.

The placement of the Sentries does not necessarily have to be at the corresponding Observer placement in the opposite query.

For the example query in Section 2.1.3, an Observer can be added after the Aggregate operator, *NI_Obs1* in Figure 3.12, which observes tuples in that *part* of the query depending on a user-defined logic. *NI_Obs1* will then activate the first instrumented Sentry, *I_Sentry1*, which will allow tuples to reach the instrumented Aggregate operator. Then *NI_Obs2*, will activate the remaining *part* with *I_Sentry2*. When the entire instrumented query is activated and ready to take over, the non-instrumented will be deactivated, as illustrated in Figure 3.12.

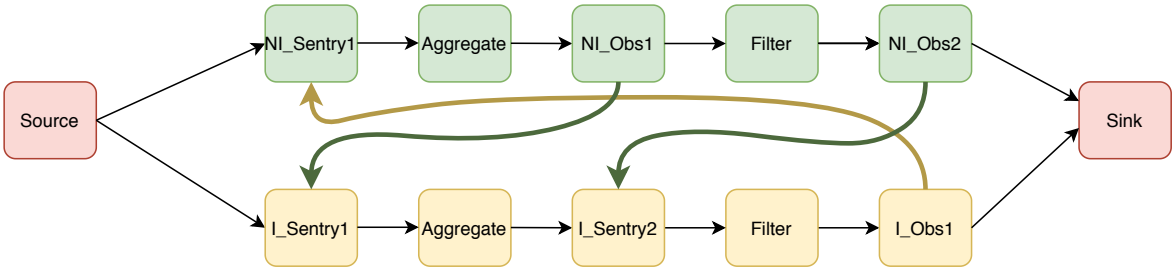


Figure 3.12: Illustration of multiple Observers being distributed in a query. The arrows points towards which Sentry each Observer control.

With this, the activation latency can be significantly lowered and be done in a partial manner, where if the Observer is placed nearer the sink in a query, it indicates that the chance of a tuple reaching a sink, i.e., a critical event, is more likely. Thus the placement of the Observers becomes a function of how likely a critical event is and how long the reconfiguration time becomes.

Reverting activated parts

When the instrumented query is not fully active and the tuples do not traverse further in to the instrumented query fast enough, or some other user-defined condition says that a critical event is unlikely, then one or more Sentries that had been activated temporarily can be deactivated again.

3.5.2 Partial deactivation

The core idea of partially activating the instrumented query is to minimize lost provenance by lowering the reconfiguration time. Partially activating the non-instrumented query when deactivating the instrumented query will not affect the reconfiguration time with regards to lost provenance. Minimizing lost provenance can only be achieved by having parts of the instrumented query already active whenever it should be activated.

Thus, there is nothing to gain in partial deactivation, i.e., partially activating the non-instrumented query preemptively when the instrumented query is about to deactivate. A lower reconfiguration time from instrumented to non-instrumented will not affect the reconfiguration time from non-instrumented to instrumented, instead it would only add unnecessary overhead, i.e., more processing and memory usage that does not lead to gained provenance.

4

Algorithmic implementation

This chapter covers the implementation of the methodology presented in the previous chapter. The implementation of the Ward operators will be described as well as how a safe transition is achieved. In this thesis, a prototype of GeneaLog is used [7], in order for it to be extended for the methodology presented in the previous chapter.

In Section 4.1, communication among the Ward operators is explained with the usage of shared variables. In Section 4.2, the implementation of the Ward operators is described with the usage of Apache Flinks API. In Section 4.3, the implementation on a safe transition between the queries is described followed by illustrations presenting the transition of states when activating and deactivating provenance. In Section 4.4, implementation optimizations are discussed.

4.1 Communication between the Ward operators

In Flink, a task is split into several parallel instances, by adjusting the parallelism-parameter for either an entire query or separate operators (see Section 2.2.2). Since each parallel instance is a copy of the same object (such as operators or global variables), the initial objects will be deserialized and then serialized, resulting in additional instances of it, depending on the degree of the parallelism [17].

References assigned to the initial objects will not work as intended, since the actual objects that are being used are the ones that have been serialized, while the references belongs to the initial objects (which have been deserialized).

Furthermore, as of Apache Flink 1.10.0 [12], there is no upstream or downstream communication functionality between the operators. In this implementation, the communication between the Ward operators is done via a centralized entity using shared variables. Note that this restrains the extent of achievable distributed execution.

To allow the Ward operators to communicate with each other via shared variables, a class is used to hold the variables, which follows the singleton design pattern to ensure that only one instance of the class is created [18]. Furthermore, to ensure that the shared variables are only allocated memory once and that there will not be several instances of it, they are declared *static*.

4.1.1 Shared variables

The shared variables used in this implementation consists of maps and boolean variables, which are used for the Ward operators to share their state and timestamp of specific tuples.

Observer states

Initiating a transition between the two queries will depend on the state of the Observers. The state of an Observer is represented as an `ObserverState` object, containing two boolean variables, `initTransition` and `initTransitionAcknowledge`. It is used for the Ward operators to differentiate whether a transition should take place or not. The functionality of initiating and acknowledging a transition will be further explain in Section 4.2.2.

The state of the Observers is stored into a shared map, called `observerStates`, where the key will be the ID of the Observer and the value will be their `ObserverState`.

Sentry states

When a Sentry notices a change in state in its Observers state, a transition to the other query has been requested. The Sentries in each respective query will have to coordinate to achieve a safe transition. They require to know the *timestamp* of the latest processed tuple and the state of each Sentry, as describe in Section 3.3.1. With this, two additional maps are required, `sentryStates` and `timestamps`, which contains the state of each Sentry and the timestamp of its first processed tuple upon activation, `startTS`.

Direction variables

Activation and deactivation shares the same semantics, but their respective procedure needs to be distinguishable, as a transition can either be from the instrumented to the non-instrumented query or vice versa.

This is due to the uncertainty of sink tuples with the placement of an Observer being prior to the sink in the instrumented query. Thus, an Observer can not solely be responsible for deactivating provenance, this is further explained in Section 4.3.2. This means that the Ward operators have to be able to distinguish between an activation of provenance and a deactivation, when executing a transition.

To distinguish whether a transition is activating or deactivating, two global variables, `isProvActivating` and `isProvDeactivating` are used. Both variables are represented as `TransitionState` objects containing two boolean variables, `early` and `residual`, representing if early and/or residual tuples are potentially present in any of the two queries. Thus, if both `early` and `residual` are `false`, the state of a direction variable is `false` and that a transition in that "direction" is *not* ongoing.

When a transition is initiated, the state of both `early` and `residual` is set to `true`,

and as long as at least one of them remain `true`, the whole state will remain `true` as well. This meaning that as long as there might be residual and/or early tuples left in q_D the transition is still considered ongoing.

User-defined condition for deactivation

In this implementation, the user-defined condition for deactivating the instrumented query consist of transitioning back to the non-instrumented query, if there is no output produced within a user-defined amount of event time, called `timeTrigger`. If an output is present in the Observer, the deactivation will occur when a Sentry has processed a tuple with a timestamp greater than the timestamp of the *latest sink tuple* added with `timeTrigger`. The timestamp of the latest sink tuple is represented as a global variable called `latestSinkTuple`, which initially is set to 0 and will be updated if the Observer in the instrumented query processes a tuple.

Concurrency control

With multiple threads that will try to access the maps simultaneously, synchronization is required to make it *thread-safe*. This is achieved by using an alternative implementation of the map data structure, namely `ConcurrentHashMap` [19].

Additionally, to prevent inconsistency, threads must be guaranteed to read the latest updated value. When multiple threads are reading from/writing to the same variable, they may copy it from main memory into a CPU cache, for performance reasons. As threads can belong to different CPUs, these values can in turn also reside at different CPU cache. Thus, there is no guarantee that the value that reside in the different CPU caches, is the same value as the one stored in main memory. Declaring the variables as `volatile`, will ensure that the threads will read and write the value of the variable from/to main memory ,instead of the CPU caches [20].

4.2 Ward operators

The Ward operators are both based on the Filter operator in Apache Flink. In Flink's API, a custom operator-function can be declared *Rich*, to be able to use additional methods for initialization and tear-down. The `open()`-method is called prior the actual working-method (e.g. filter, map), which makes it suitable for initialization for the Ward operators. For each respective Ward operator, the `open()`-method is used to insert its initial state into the maps. As both of them require the `open()`-method, they both extend the abstract class *RichFilterFunction* [21].

Note: In Flink, `Filter` operators are stateless operators, however in this implementation they will be considered active or inactive (meaning allowing or dropping tuples), which will be represented as a change in state. Thus, the Ward operators can be considered stateful in this implementation.

As tuples will eventually be transformed throughout a query, they will be represented as different *Java objects*. Since the location of the Ward operators are not set, an

operator will need to handle different *tuple types*. In Flink, generics are used to reference the tuple types, thus the Ward operators are declared as *generic*.

4.2.1 Sentry operators

A Sentry operator is created by defining a new instance of the `Sentry class`. It is then placed in a query, acting as a custom `Filter` operator.

When creating a new instance of a Sentry, it is assigned an ID and an initial state. The ID will be used as the key in the shared maps, which the Ward operators will use to retrieve the state and timestamp of a specific Sentry, to achieve a safe transition.

If a Sentry is located prior to one or several stateful operators, they will be dependent on the Sentry's state, since the Sentry controls the progression of tuples for the remaining operators in a query. Thus, to achieve a safe transition, the Sentry operators (in each query) are responsible for these stateful operators and will have to coordinate at which point in time they should start and stop ingesting tuples respectively. This means that a Sentry requires the ID of another Sentry, defined as a *neighboring* Sentry, as well as its Observer and its neighboring Observer.

Furthermore, the Sentry operator requires to know the largest window size and its window advance of the stateful operator(s) under its control.

4.2.2 Observer operators

An Observer is created by defining a new instance of the `Observer` class. Similarly to a Sentry, it is placed in a query as a custom `Filter` operator.

The Ward operators communicate with each other by using their unique ID, which is required for an Observer as well. Furthermore, an Observer requires an initial state, which is represented as an `ObserverState` object.

ObserverState

An Observer's state is represented as an object which consists of two boolean variables, `initTransition` and `initTransitionAcknowledge`. Both of these two variables are used when transitioning between the queries by the Sentry operators. The `initTransition` variable is initially set to `false` and set to `true` when a transition is requested, and set back to `false` when the request has been acknowledged. Similarly, `initTransitionAcknowledge` is initially set to `false` and set to `true` when a transition has been acknowledged by a Ward operator in the other query, and set back to `false` when the transition is underway.

A Sentry knows if a transition have been requested with a change of state in `initTransition`, and only then will change the value of `initTransitionAcknowledge` to `true`, to inform the Sentry in the opposite query that it has acknowledged and started the initiation of a transition.

Dropping residual and early output

During a safe transition, the Observers are responsible for dropping the (potential) residual or early outputs produced in their respective query (See Section 3.3). An Observer needs to know the ID of the closest upstream Sentry in the same query, to know whenever if it has changed its state. Additionally, it needs to know the ID of a neighboring Sentry, to find the timestamp of the first processed tuple. Lastly, to fully be able to drop residual or early tuples, an Observer requires to know the largest window size and the largest window advance.

Observable tuples

An Observer operator should be able to *observe* each of its incoming tuples and depending on the information it is able to derive from the tuple, change its state. With several different tuple types present in a query, an Observer needs to be able to *observe* them separately, since different tuple types will require different *observe* logic (since they are produced at different stages in a pipeline).

To achieve this, an interface is used containing one method called `observe()`. By extending each of the tuple classes with this interface, each of them can have user-defined logic for the Observer to use, to decide whether to change its state.

The `observe()`-method is then used in an Observers `Filter` function.

4.3 Safe transition between the queries

Initially, the application will run the non-instrumented query and then transition to the instrumented query, when an Observer placed in the former query changes its state. Depending on the condition defined in an Observer placed in the instrumented query, a transition back to the non-instrumented query could happen.

When transitioning from one query to the other, a deactivated Sentry will start ingesting tuples as soon as it sees a change in the opposite query's Observer state. Simultaneously, it will communicate with its neighboring Sentry by informing it of the timestamp of the first tuple it ingested. With this, the Sentry in the former query will define its scope, as presented in Section 3.3.2, to ensure that its stateful operators produce correct results. Based on the scope, the Observers in both queries will determine what tuples are residual or early respectively.

With transitions being changes in the direction variables and the Sentry operators state, a safe transition can be illustrated as a state machine, see Figure 4.1. As shown in the Figure, the instrumented Sentry is denoted as `I_S` and the non-instrumented Sentry as `NI_S`. `State 1` to `State 3` represent the states for activating provenance, while `State 5` to `State 7` represent the states for deactivating provenance. `State 0` represents the state when the provenance is deactivated while `State 4` represents the state when provenance is active.

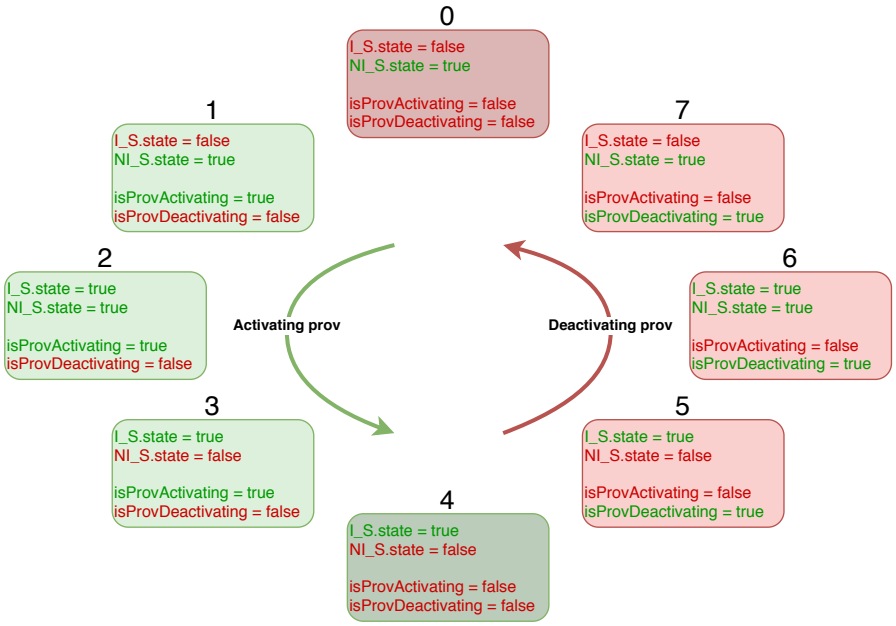


Figure 4.1: Representation of the possible states in the transition state machine.

In the following sections, achieving a safe transition with the progression of states will be explained when *activating* and *deactivating* for the default *scheme* presented in Section 3.4. To ease the explanation, the Ward operators located in the instrumented query will be denoted as S_I for the Sentry and O_I for the Observer, for the non-instrumented query, S_{NI} and O_{NI}.

4.3.1 Activating provenance

As soon as O_{NI} receives an input (sink tuple), a transition to the instrumented query will happen. Before transitioning to the first state of activating provenance, `State 1`, O_{NI} checks the following conditions to ensure that its in the correct state, `State 0`, see Listing 4.1.

```
instrumentedSentry.state = false
nonInstrumentedSentry.state = true
isProvActivating = false
isProvDeactivating = false
```

Listing 4.1: Conditions for `State 0`.

When the following conditions are met, the initiation of a transition can start.

4.3.1.1 Initiating a transition: Non-instrumented query

Before O_{NI} initiates a transition (while still in `State 0`), it has to verify that there is no other transition that has been requested, or that itself already has requested

a transition. If not, O_{NI} will initiate a transition by changing the state of both `initTransition` and `isProvActivating`, see Listing 4.2 and Figure 4.2.

```
o_ni = NonInstrumentedObserver.observerState                    1
                                                                2
if (in State 0)                                                 3
    if( !o_ni.initTransition                                    4
    && !o_ni.initTransitionAcknowledge)                         5
        o_ni.initTransition = true                              6
        isProvActivating = true                                 7
```

Listing 4.2: Pseudocode for initiating a transition request to the instrumented query. This is only executed by O_{NI} in `State 0`.

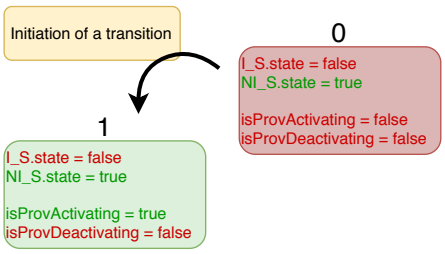


Figure 4.2: Initiation of a transition. `State 0` → `State 1`.

With this, the transition changes it state from `State 0` to `State 1`.

4.3.1.2 Initiating a transition: Instrumented query

When in `State 1`, S_I will notice that O_{NI} has requested a transition, with the change in state of `initTransition`. Furthermore, it has to verify that it has not already acknowledged the transition, `initTransitionAcknowledge`. If not, it will acknowledge the transition request, by changing the state of `initTransitionAcknowledge` and `initTransition`.

When the transition has been acknowledged, S_I will change its own state and start to ingest tuples. The timestamp of the first tuple it processes will be written to the shared map, `timestamp`, which is accessible to the Ward operators in the opposite query, see Listing 4.3 and Figure 4.3.

```
o_ni = nonInstrumentedObserver.observerState
s_i= instrumentedSentry.state
ts = timestamps
startTS = firstTupleProcessed.timestamp

if(in State 1)
    if(o_ni.initTransition
    && !o_ni.initTransitionAcknowledge)
        o_ni.initTransition = false
        o_ni.initTransitionAcknowledge = true
        s_i = true
        ts.put(startTS)
```

Listing 4.3: Pseudocode for acknowledging the transition request. This is only executed by S_I in `State 1`.

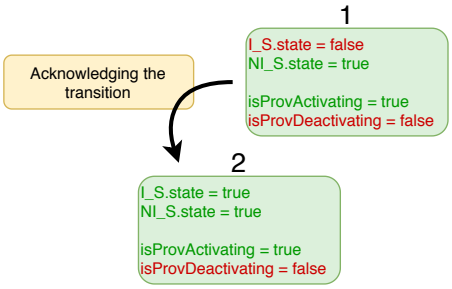


Figure 4.3: Acknowledging the initiation of a transition. `State 1` → `State 2`.

Now, with both queries processing tuples, the transition changes it state from `State 1` to `State 2`, and the procedure of achieving a *safe transition* can begin.

4.3.1.3 Safe transition: Deactivating the non-instrumented query

In `State 2`, S_{NI} will notice that S_I has become active, with the change of state in `initTransitionAcknowledge`. S_{NI} resets the state of `initTransitionAcknowledge`, thus completing the initiation of the safe transition. Simultaneously, it deactivates itself with the occurrence of a timestamp exceeding the defined threshold (see Equation 3.2), see Listing 4.4 and Figure 4.4.

Note: In both `State 0` & `State 3`, the value of both `initTransition` & `initTransitionAcknowledge` is `false`. This is the condition for initiating a new transition (to instrumented query), however the Observer will not be able to start a new transition, as it also requires the transition to be in `State 0`.

```
o_ni = nonInstrumentedObserver.observerState
s_ni = nonInstrumentedSentry.state
startTS = instrumentedSentry.startTS
tuple = current tuple

if (in State 2)
    if(o_ni.initTransitionAcknowledge
    && !o_ni.initTransition
    && canDeactivate(tuple))
        o_ni.initTransitionAcknowledge = false
        s_ni = false

boolean canDeactivate(tuple t)
    return (t.timestamp
            >= startTS
            + (windowAdvance
            − (startTS mod windowAdvance))
            + (windowSize − windowAdvance))
```

Listing 4.4: Pseudocode to mark that a transition has been acknowledged and initiated. This is only executed by S_{NI} in `State 2`.

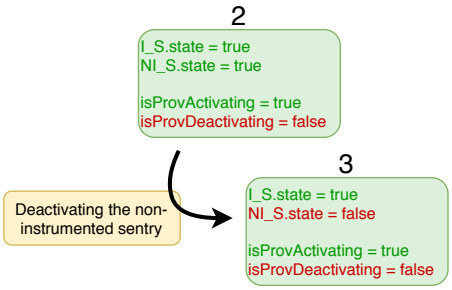


Figure 4.4: Deactivating S_{NI}. `State 2` → `State 3`.

With this, S_{NI} has stopped ingesting tuples and S_I has started ingesting tuples. This means that the transition will progress to `State 3` and the potentially produced *early* tuples in the instrumented query- and the *residual* tuples in the non-instrumented query needs to be dropped.

Note: When S_I starts to ingest tuples in `State 2`, the potentially early output can be produced before or during the production of the last correct window in q_D. Similarly, tuples will be considered residual after the last correct window is completed, when reaching `State 3`. This means that even though the early tuples are dropped in q_A, there might still be residual tuples present in q_D. Thus, dropping early tuples can occur both in `State 2` and `State 3`, while dropping residual tuples only occurs in `State 3`.

4.3.1.4 Safe transition: Dropping early tuples in the instrumented query

The early output produced in q_A has to be dropped, since the initial output might be incorrect (see Section 3.3).

O_I determines if a tuple is considered to be early by following Equation 3.1 (presented in Section 3.3.2). If the timestamp of the incoming tuples is lower than the timestamp of the *first complete window* in the q_A, the tuples are considered to be early and are dropped, see Listing 4.5 and Figure 4.5.

```
startTS = instrumentedSentry.startTS                          1
tuple = current tuple                                         2
                                                              3
if (in State 2 || State 3)                                    4
    if isEarlyTuple(tuple.timestamp)                          5
        drop tuple                                            6
                                                              7
boolean isEarlyTuple(tuple t)                                 8
    return t.timestamp < startTS                              9
    + (windowAdvance                                          10
    − ( startTS mod windowAdvance))                           11
    + windowSize                                              12
```

Listing 4.5: Pseudocode for determining and dropping early tuples. This is executed by O_I in both `State 2` and `State 3`.

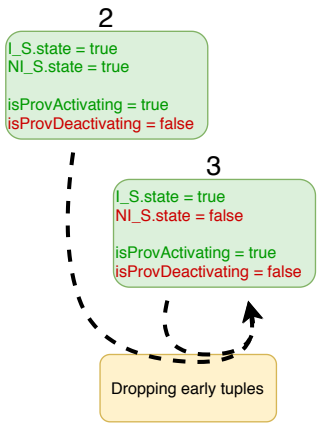


Figure 4.5: Dropping early tuples. Starts in `State 2` and is ongoing during `State 3`.

Note: By placing the Observer prior to the sink, the Observers actions are dependent on the (potential) sink tuples. Since there is no guarantee that sink tuples are produced, the transition might not progress further than `State 3`. This means that a transition back to the non-instrumented query would not be possible. Thus, to ensure the progression to `State 4`, the Observers utilize watermarks (see Section 2.1.5). This will be further explained in Section 4.3.1.6.

Before progressing to `State 4`, all residual tuples has to be dropped.

4.3.1.5 Safe transition: Dropping residual tuples

O_{NI} determines if a tuple is considered to be residual by following Equation 3.1 (presented in Section 3.3.2). If the timestamp of the incoming tuples is larger or equal to the timestamp of the *first complete window* in q_A, the tuples are considered to be residual and are dropped, see Listing 4.6 and Figure 4.6.

```
startTS = instrumentedSentry.startTS                          1
tuple = current tuple                                         2
                                                              3
if (in State 3)                                               4
    if (isResidualTuple(tuple.timestamp))                     5
        drop tuple                                            6
                                                              7
boolean isResidualTuple(Tuple t)                              8
    return t.timestamp >= startTS                             9
    + (windowAdvance                                          10
    − (startTS mod windowAdvance))                            11
    + windowSize                                              12
```

Listing 4.6: Pseudocode for dropping residual tuples. This is executed by O_{NI} in `State 3`.

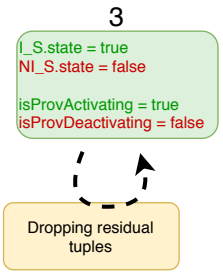


Figure 4.6: Dropping residual tuples. Occurs in `State 3`.

4.3.1.6 Ensuring progression to `State 4`

Due to the potential lack of tuples in the Observers, to ensure progression of state, watermarks can be utilized as they will progress through a query, regardless of if a query is *active* or *inactive*. The goal of watermarks is to synchronize the operators, which means that watermarks are not processed as normal tuples. This means that even if no sink tuples are produced, watermarks will still reach the Observers.

The functionality of watermarks can be used by the Observers to ensure that the transition progresses to `State 4`, even if no sink tuples are produced. If the timestamp of a watermark indicates that all early and residual tuples are done, then the Observers in each query changes the state of `early` and `residual` in `isProvActivating` respectively, and the transition progresses to `State 4`.

Early tuples finished

O_I will stop consider the tuples to be early with the occurrence of a watermark. As mentioned in Equation 3.1, every tuple processed before the point in time where the *first correct window* has finished in q_A, tuples are considered early, and are dropped. When a watermark arrives with a value representing the point in time where the first correct window in q_A have been completed, the tuples are no longer considered to be early. With this, the transition is complete from q_A's point of view and the state of `early` in `isProvActivating` is changed, see Listing 4.7 and Figure 4.7.

```
startTS = instrumentedSentry.startTS                    1
var timeTrigger = user-defined                          2
    deactivation condition in event time
wm = current watermark                                  3
                                                        4
if (in State 3)                                         5
    if (isEarlyTuplesFinished(wm))                      6
        isProvActivating.early = false                  7
                                                        8
boolean                                                 9
    isEarlyTuplesFinished(Watermark
    w)
    return w.timestamp >= startTS                       10
    + (windowAdvance                                    11
    - ( startTS mod windowAdvance))                     12
    + windowSize                                        13
```

Listing 4.7: Pseudocode to ensure that the transition can move into `State 4`. This is only executed by O_I in `State 3`.

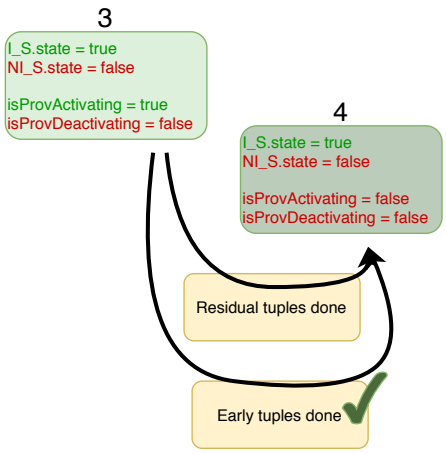


Figure 4.7: With an absence of early tuples, the transition progressed to its final state. `State 3` → `State 4`.

However, the transition can not move to the final state of activating provenance, `State 4`, until the state of `residual` in `isProvActivating` is changed.

Residual tuples finished

Similarly, O_{NI} determines that no more residual tuples will arrive with the occurrence of a watermark. As mentioned in Equation 3.1, every (potential) tuple which arrives at O_{NI} after the point in time where the *first correct window* in q_A is complete, are considered residual and needs to be dropped. This means that there will be either one or several windows which will contribute to potential residual tuples.

When a watermark arrives with the value representing the point in time where every ongoing windows that contributes to residual tuples are complete. With this, the transition is complete from the q_D's point of view and the state of `residual` in `isProvActivating` is changed, see Listing 4.8 and Figure 4.8

```
startTS = instrumentedSentry.startTS                                    1
var timeTrigger = user−defined deactivation                             2
      condition in event time
watermark = current watermark                                           3

if (in State 3)                                                         5
      if (isResidualTuplesFinished(watermark))                          6
            isProvActivating.early = false                              7

boolean                                                                 9
      isResidualTuplesFinished(Watermark w)
      return w.timestamp >= startTS                                     10
      − (startTS mod windowAdvance))                                    11
      + 2 ∗ windowSize                                                  12
```

Listing 4.8: Pseudocode to ensure that the transition can progress to `State 4`. This is only executed by O_{NI} in `State 3`.

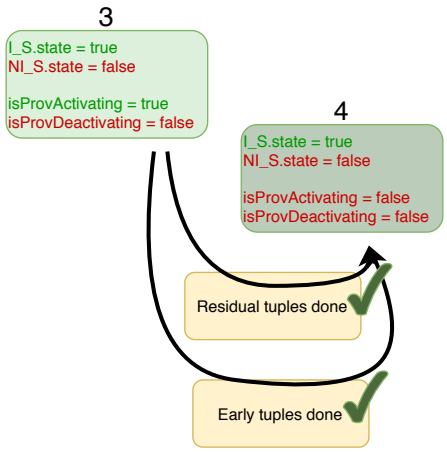


Figure 4.8: Transitioning to `State 4`, with the absence of residual tuples.

Note: In Listing 4.8, the method *isResidualTuplesFinished()* finds the start of the last correct window in q_D and adds two whole windows sizes, in order to find the end of last window with potential tuples in it, i.e., one whole window after the last correct window in the q_D.

With a state change in both `early` and `residual`, the transition is finished and provenance has become activated.

4.3.1.7 Summary

In conclusion, activating provenance is the shift from `State 0` to `State 4`, as illustrated in Figure 4.9, which includes all necessary steps to perform a safe transition from the non-instrumented query to the instrumented query.

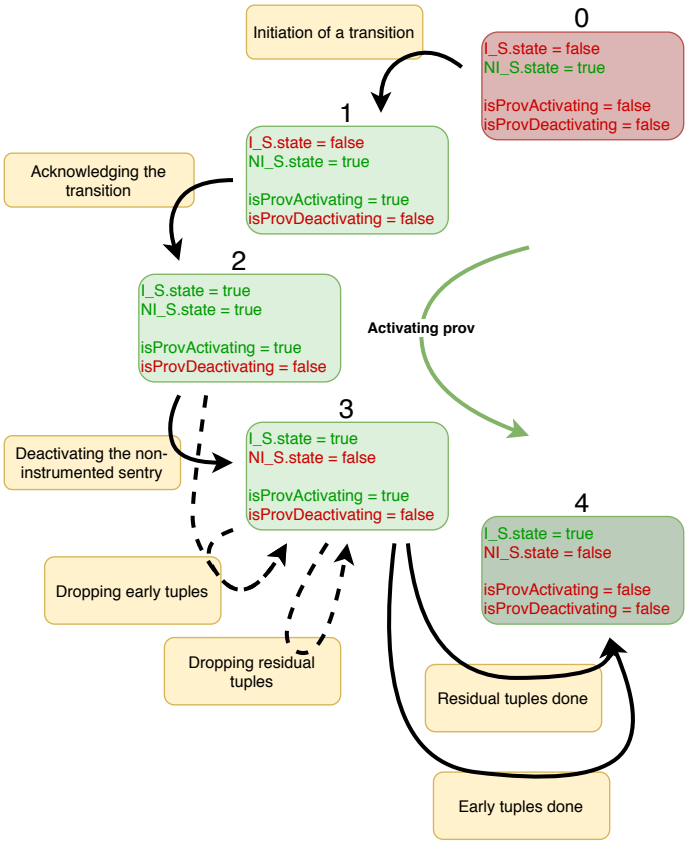


Figure 4.9: The transition in states to activate provenance and achieving a safe transition.

4.3.2 Deactivating provenance

With the occurrence of an output, the query should naturally keep ingesting tuples and delay the transition. With the absence of output, a transition back to the non-instrumented query should happen.

To deactivate, the transition has to be in `State 4`, see Listing 4.9.

```
instrumentedSentry.state = true
nonInstrumentedSentry.state = false
isProvActivating = false
isProvDeactivating = false
```

Listing 4.9: Conditions for `State 4`.

Note: The procedure of activating and deactivating provenance is symmetrical except for the triggers in the respective Observers, O_{NI} and O_I. O_{NI} will trigger a transition with the occurrence of a tuple, while O_I will trigger a transition with the absence of tuples. With this, O_I is required to have different behavior which

is not dependent on tuples. As mentioned in Section 4.3.1.6, Observers can utilize watermarks to perform actions with the absence of tuples. Furthermore, aside from Section 4.3.2.1, the procedure explain in the remaining sections are identical to those presented from Section 4.3.1.2 to Section 4.3.1.6.

4.3.2.1 Initiate a transition: Instrumented query

A transition back to the non-instrumented query should occur if no tuple has been processed by O_I for a certain period of time, which will be known utilizing watermarks.

The period of time before it deactivates is set by the timestamp of the latest processed tuple in O_I, `latestSinkTuple` and the user-defined condition `triggerTime`. If no tuples arrive at O_I, `latestSinkTuple` will never be set, then the condition will instead be based on the first tuple processed by S_I, `startTS`, together with the user-defined condition `triggerTime`.

If the timestamp of a watermark indicates that no sink tuple has been processed by O_I during the user-defined period of time, a transition to the non-instrumented query is initiated, see Listing 4.10 and Figure 4.10.

```
o_i = instrumentedObserver.observerState                    1
startTS = instrumentedSentry.startTS                        2
st = instrumentedObserver.latestSinkTuple                   3
var timeTrigger = user-defined deactivation                 4
    condition in event time
watermark = current watermark                               5
tuple = current tuple                                       6
                                                            7
if (in State 4)                                             8
    st = tuple                                              9
                                                            10
    if(!o_i.initTransition                                  11
    && !o_i.initTransitionAcknowledge                       12
    && deactivateCondition(watermark))                      13
        o_i.initTransition = true                           14
        isProvDeactivating = true                           15
                                                            16
boolean deactivateCondition(watermark w)                    17
    return (w.timestamp > (st.timestamp +                   18
        timeTrigger))
        && (w.timestamp > (startTS +                        19
            windowSize + timeTrigger))
```

Listing 4.10: Pseudocode for initiating the deactivation of the instrumented query. This is only executed by O_I in `State 4`.

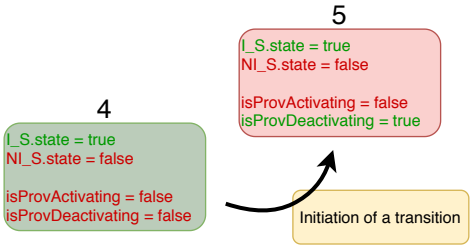


Figure 4.10: Initiation of a transition. `State 4` → `State 5`.

Note: The following sections include the procedure for performing a safe transition from the instrumented query to the non-instrumented query. This is identical to the procedure explained in Section 4.3.1.2 to Section 4.3.1.6.

4.3.2.2 Initiate a transition: Non-instrumented query

When in `State 5`, S_{NI} will notice that a request for a transition has been made by the instrumented query.

To transition into `State 6`, S_{NI} will change its own state and acknowledge the request by changing the state of `initTransitionAcknowledge`. Additionally, it will save the timestamp of the first tuple it processes, see Listing 4.11 and Figure 4.11.

```
o_i = instrumentedObserver.observerState                    1
s_ni = nonInstrumentedSentry.state                          2
ts = timestamps                                             3
startTS = firstTupleProcessed.timestamp                     4
                                                            5
if (in State 5)                                             6
    if( o_i.initTransition                                  7
    && !o_i.initTransitionAcknowledge)                      8
        o_i.initTransition = false                          9
        o_i.initTransitionAcknowledge =                     10
            true
        s_ni = true                                         11
        ts.put(startTS)                                     12
```

Listing 4.11: Pseudocode to acknowledge the transition request and starting to ingest tuples. This is only executed by S_{NI} in `State 5`.

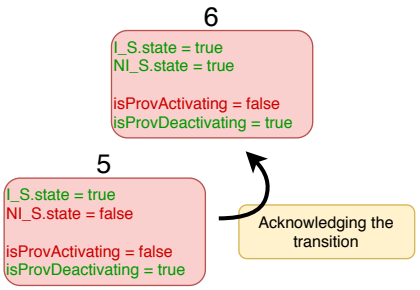


Figure 4.11: Acknowledging the initiation of a transition. `State 5` → `State 6`.

4.3.2.3 Safe transition: Deactivating the instrumented query

In `State 6`, S_I will notice that S_{NI} has become active. S_I resets the state of `initTransitionAcknowledge`, thus completing the initiation of the safe transition. With the occurrence of a timestamp of a tuple exceeding the defined threshold (See Equation 3.1), S_I changes its state, see Listing 4.12 and Figure 4.12.

```
o_i = instrumentedObserver.observerState                          1
s_i = instrumentedSentry.state                                    2
startTS = instrumentedSentry.startTS                              3
tuple = incoming tuple                                            4
if (in State 6)                                                   5
    if(o_i.state.initTransitionAcknowledge                        6
    && !o_i.initTransition                                        7
    && canDeactivate(Tuple))                                      8
        s_i = false                                               9
        initTransitionAcknowledge = false                         10
                                                                  11
boolean canDeactivate(Tuple t)                                    12
    return (t.timestamp                                           13
            >= startTS                                            14
            + (windowAdvance                                      15
            − (startTS mod windowAdvance))                        16
            + (windowSize − windowAdvance))                       17
```

Listing 4.12: Pseudocode to mark that a transition has been acknowledged and initiated. This is only executed by S_{NI} in `State 6`

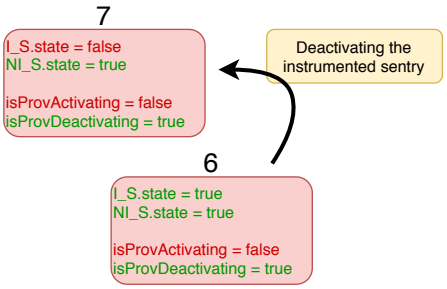


Figure 4.12: Deactivating the S_{NI}. `State 6` → `State 7`.

Similarly to the activation procedure, dropping the (potential) early tuples will occur in both `State 6 & 7` and for the (potential) residual tuples in `State 7`, as described in Section 4.3.1.4 and Section 4.3.1.5. Similarly to Section 4.3.1.6, the state machine will progress to `State 0` with the absence of early and residual tuples.

4.3.2.4 Safe transition: Dropping early tuples in the non-instrumented query

When the transition has reached `State 6`, both of the queries are active and are processing tuples. As previously mentioned, the (potential) early tuples produced in the non-instrumented query needs to be dropped.

O_{NI} determines if a tuple is considered to be early by following Equation 3.1, presented in Section 3.3.2. If the timestamp of the incoming tuple is lower than the timestamp of the *first complete window* in the non-instrumented query, the tuples are considered to be early and are dropped, see Listing 4.13 and Figure 4.13.

```
startTS = firstTupleProcessed.timestamp                    1
tuple = current tuple                                      2
                                                           3
if (in State 6 || State 7)                                 4
    if isEarlyTuple(tuple.timestamp)                       5
        drop tuple                                         6
                                                           7
boolean isEarlyTuple(tuple t)                              8
    return t.timestamp <                                   9
    + (windowAdvance                                       10
    − (startTS mod windowAdvance))                         11
    + windowSize                                           12
```

Listing 4.13: Pseudocode to find and drop early tuples. This is executed by O_{NI} in both `State 6` and `State 7`.

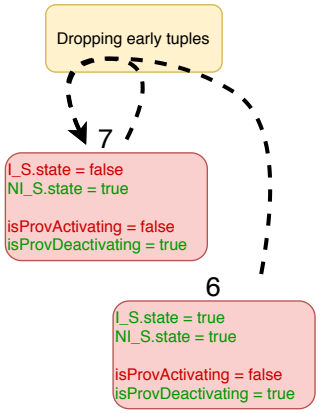


Figure 4.13: Dropping early tuples. Starts in `State 6` and is ongoing during `State 7`.

4.3.2.5 Safe transition: Dropping residual tuples in the instrumented query

O_I determines if a tuple is considered to be residual by following Equation 3.1, presented in Section 3.3.2. If the timestamp of the incoming tuple is larger or equal to the timestamp of the *first complete window* in the non-instrumented query, the tuples are considered to be residual and are dropped, see Listing 4.14 and Figure 4.14.

```
startTS = nonInstrumentedSentry.startTS                    1
tuple = current tuple                                      2
                                                           3
if (in State 7)                                            4
    if (isResidualTuple(tuple.timestamp))                  5
        drop tuple                                         6
                                                           7
boolean isResidualTuple(tuple)                             8
    return t.timestamp >= startTS +                        9
            (windowAdvance − (startTS                      10
                mod
            windowAdvance)) + windowSize                   11
```

Listing 4.14: Pseudocode to find and drop residual tuples. This is executed by O_I in `State 7`.

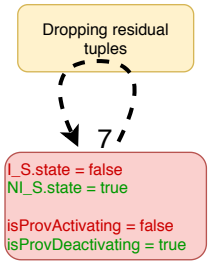


Figure 4.14: Dropping residual tuples. Runs in `State 7`.

4.3.2.6 Ensuring progression to `State 0`

As explained in Section 4.3.1.6, to ensure the progression of state, the Observers utilize watermarks to determine when the residual and early tuples have been dropped.

If the timestamp of a watermark indicates that all early and residual tuples are done, then the Observers in each query changes the state of `early` and `residual` in `isProvDeactivating` respectively, and the transition can progress to `State 0`.

Early tuples finished

O_{NI} will stop treating the tuples as early when a watermark arrives with a value representing the point in time where the first window in q_A have been completed, see Listing 4.15 and Figure 4.15.

```
startTS = instrumentedSentry.startTS                    1
var timeTrigger = user-defined deactivation             2
    condition in event time
watermark = current watermark                           3
                                                        4
if (in State 7)                                         5
    if (isEarlyTuplesFinished(watermark))               6
        isProvActivating.early = false                  7
                                                        8
boolean isEarlyTuplesFinished(Watermark                 9
    w)
    return w.timestamp >= startTS                       10
    + (windowAdvance                                    11
    - ( startTS mod windowAdvance))                     12
    + windowSize                                        13
```

Listing 4.15: Pseudocode to ensure that the transition can progress to `State 0`. This is only executed by O_{NI} in `State 7`.

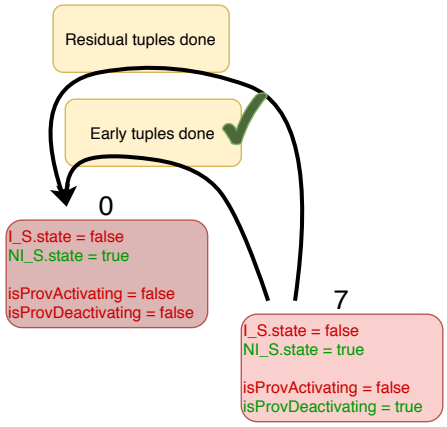


Figure 4.15: Dropping early tuples. Runs in `State 7`.

However, the transition can not move to the final state of activating provenance, `State 0`, until the state of `residual` in `isProvDeactivating` is changed.

Residual tuples finished

Similarly, O_I determines that no more residual tuples will be present when a watermark arrives with the value representing the point in time where every ongoing window that will potentially contribute to the residual tuples are complete, see Listing 4.15 and Figure 4.15.

```
startTS = instrumentedSentry.startTS                                    1
var timeTrigger = user−defined deactivation                             2
      condition in event time
watermark = current watermark                                           3
                                                                        4
if (in State 7)                                                         5
      if (isNotEarlyTuple(watermark))                                   6
            isProvActivating.early = false                              7
                                                                        8
boolean                                                                 9
      isResidualTuplesFinished(Watermark w)
      return w.timestamp >= startTS                                     10
      − (startTS mod windowAdvance))                                    11
      + 2 ∗ windowSize                                                  12
```

Listing 4.16: Pseudocode to ensure progression to `State 0`. This is only executed by O_I in `State 7`.

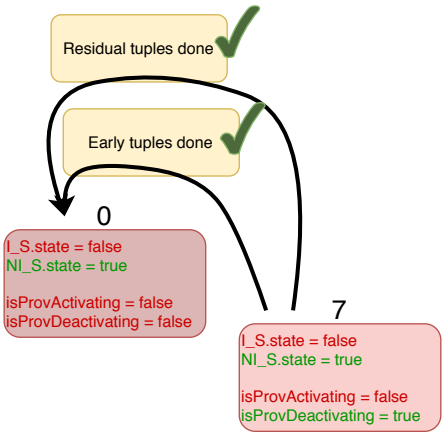


Figure 4.16: Completed dropping the residual tuples. Runs in `State 7`.

Summary

Deactivating provenance is achieved by shifting from `State 4` to `State 0`, which includes all necessary steps to perform a safe transition from the instrumented query to the non-instrumented. Thus, combining all states, as in Figure 4.1 in Section 4.3, and all state changes for both activating and deactivating provenance is illustrated in Figure 4.17.

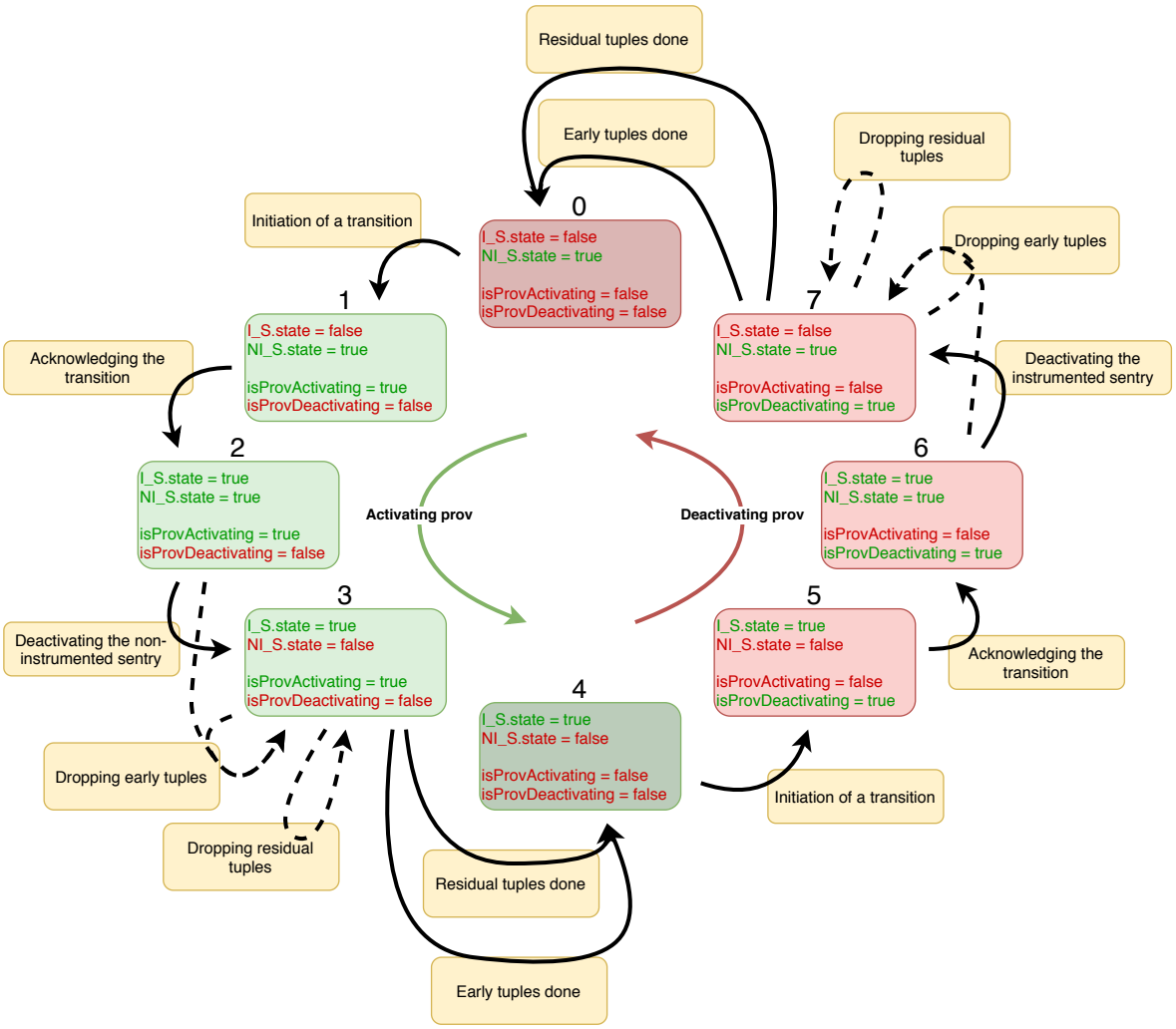


Figure 4.17: The full transition of states of activating provenance and deactivating provenance.

4.4 Implementation optimization

The communication between the Ward operators is based on reading/writing to/from variables placed in a centralized class (Singleton). As mentioned in Section 4.3, the location of both Sentry operators is succeeding the source, which means that both of the operators will make a decision on whether to allow or deny a tuple, for every source tuple. Notice that for every tuple, the Sentries will poll the state of the Observer and the other Sentry, to check if it should initiate a transition. This could introduce a bottleneck in the implementation, which introduces extra overhead and affects the performance of the approach.

4.4.1 Passive polling

Polling the state of the Observer for every incoming tuple will contribute to extra overhead, since a function call will be made to fetch the state from the Singleton-

class, for every tuple. By lowering the level of *aggressiveness* of the polling, less function calls will be made and thus generate less amount of overhead. This was done by introducing a variable, `frequency`, which controls how often the Sentries should check the state of the Observer.

A Sentry operator is required to know both the largest `windowSize` and the largest `windowAdvance` of the stateful operator(s), to perform a safe transition. This means that the Sentry is able to determine the periodicity of when an output tuple will potentially be produced by a stateful operator. If a stateful operator is placed prior to an Observer, the input of the Observer will be the output of the stateful operator. This means that the value of `frequency` can be configured in such a way that the Sentry will poll whenever it knows there is a possibility that the Observer has changed state, which is every time a window finishes. This is how the frequency is set for this implementation.

5

Evaluation

In this chapter, the evaluation of the Twin query implementation described in Chapter 4 is presented.

The goal of this thesis (see Section 1.1.2) is to extend GeneaLog to be activated/deactivated and evaluate if such extension could contribute to lower overhead.

As mentioned in Section 1.2, the procedure of activating/deactivating will produce overhead and should not atone for the possible overhead that is lowered. The purpose of the evaluation is to determine under what conditions an interactive behavior is beneficial, by measuring the approach against GeneaLog with a set of metrics and under different variants of three different use cases. Lastly, as opposed to having GeneaLog generating provenance information continuously, the amount of provenance information will be decreased.

5.1 Experiments

To evaluate the Twin query approach, different types of experiments are conducted. Each experiment is based on a use case that have been used during the evaluation of GeneaLog [7].

As previously mentioned, the goal of the evaluation is to explore the different scenarios where an interactive behavior is beneficial. With this, the use cases presented in Section 5.2 are modified to introduce different scenarios which will be important to determine under what conditions it becomes beneficial. The results from these experiments are compared to experiments without any modifications. The different scenarios include the following modifications.

- *Different trigger conditions:* Depending on how the trigger for deactivation is defined, it will have an impact on the amount of provenance generated as well as the number of transitions. By increasing the duration of the condition (`timeTrigger`), it could result in an increase of provenance and a less amount of transition as opposed to an decrease of provenance and a higher number of transitions. This can in turn can have an impact on the performance, since achieving a safe transition will contribute to overhead.

- *Increasing the workload of the query:* Every use case presented in Section 5.2 is lightweight, with regards to the complexity the analysis. Due to this, the source becomes a bottleneck, as the production of source tuples, serialization and deserialization is what consumes most of the CPU. Similarities in the use cases are that succeeding the source, a `Filter` is used which will only allow tuples to progress from the source if they meet a certain condition. By removing the the `Filter` in every variant, more tuples will progress the pipeline which shifts the bottleneck from the source to the `Aggregate` instead. With this, more tuples will be annotated and introduces a scenario where provenance becomes a heavy operation.

- *Different hardware:* As GeneaLog is originally designed for resource constrained devices, evaluating the twin query approach on a more powerful machine such as a laptop would produce inaccurate results. By using a more powerful machine, tuples will be processed faster, as more resources are available for Flink and its threads. With this, the reconfiguration time will be affected which in turn affects the amount of provenance which is lost during a transition. The experiments are conducted on two devices, namely an Odroid and a laptop.

5.1.1 Hardware setup

The experiments are performed on two different machines, namely: An embedded device, Odroid-XU4 with a Samsung Exynos5422 Cortex-A7 Octa core CPU 2Ghz with 2 GB of memory, running Ubuntu 16.04.4. The second machine is a laptop, ThinkPad X240 with a `Intel i3-4030U (4) @ 1.900 GHz` CPU and 8 GB of memory, running Manjaro Arch Linux. See Table 5.1 for the configuration of the cluster. See Table 5.1 for the configuration of the cluster.

Each experiment is executed on a locally deployed Flink cluster, with the configuration presented in Table 5.1.

Apache Flink version	1.10 Stable
Heap size	1024 Mb
Memory process size	2000 Mb
Task slots	1
Parallelism	1

Table 5.1: Flink cluster configuration.

5.1.2 Evaluation metrics

To evaluate the Twin query approach, the following measurements are of interest.

1. *Throughput*: The average number of tuples per second that both queries can process.

2. *Latency*: The average time between the production of each sink tuple and the reception of the latest source tuple contributing to it.
3. *Reconfiguration time:* The average time between the initiation of a transition and its completion, between the two queries.
4. *Memory*: The average and maximum size of memory used by the process running both queries.
5. *Provenance*: The amount of sink tuples that are produced, together with their respective contributing source tuples.
6. *Lost provenance*: The amount of sink tuples that are lost by transitioning between the two queries.

5.1.3 Experimental setup

Each experiment will consist of at least three different variants of a use-case, namely:

1. *No provenance, NP:* Only using standard operators.
2. *GeneaLog, GL:* Only using instrumented operators.
3. *Twins, TW:* Two queries where one is equipped with standard operators while the other with instrumented operators.

As mentioned in Section 4.3.2.1, the user-defined condition for deactivating provenance is if no sink tuple has been present within the time of the completion of four windows, a transition back to the non-instrumented query happens. In order to determine during which conditions an interactive behavior is feasible, different trigger conditions are evaluated, to observe the differences between on how much time is spent in the instrumented query as opposed to the non-instrumented query.

The following different trigger conditions is evaluated.

1. *Two windows:* With an decrease in the amount of time Twins remains in the instrumented query, it would result in an higher amount of transitions and a decreased amount of generated provenance information.
2. *Four windows:* Baseline condition.
3. *Eight windows:* With an increase would result in a lower amount of transition and an larger amount of generated provenance information.

The experiments performed on the Odroid are at least five minutes long and the results are averaged on at least five runs, while the experiments on the laptop are at least one minute long and the results are averaged on at least five runs.

5.2 Use cases

In this section, the different use-cases, their structure and the results from the experiments are presented.

The first two use cases, *broken-down cars* and *car accidents* are based on theLinear road benchmark, a standard for studying SPE performance. It simulates vehicu-

lar traffic on linear expressways, composed of predefined segments [22]. Every 30 seconds, position reports are forwarded by the cars traveling in the highway, and carries a timestamp, unique id of the car, its speed and position.

$$< timestamp, car_{id}, speed, position > . \tag{5.1}$$

The third use-case, *long-term blackout detection* is based on a real-world Smart grid infrastructure. Each source tuple are simulated measurements forwarded by smart meters, every hour that carries a timestamp, unique id of the smart meter and its consumption.

$$< timestamp, meter_{id}, consumption > \tag{5.2}$$

5.2.1 Broken-down cars query

The first query aims to detect *broken-down vehicles.* A car is considered to be stopped if at least four consecutive position reports from the same car which report zero speed and the same position.

The query consists of a `Filter`, which only forward tuples which contain `speed == 0`. In order to determine that a car has stopped, an `Aggregate` is placed subsequently, which groups each computation by the car_{id}. The Aggregate maintains a window size of 120 seconds and a window advance of 30 seconds. Lastly, a `Filter` is placed which forwards tuples that contain a count of four in the same unique position.

To adjust the query for the methodology presented in Chapter 3, a Sentry is placed succeeding the source and an Observer is placed prior to the sink, similarly to the example query presented in Section 2.1.

This is illustrated in Figure 5.1, where the lower operators, colored in yellow, represent the instrumented operators while the upper operators, colored in green, represented non-instrumented operators.

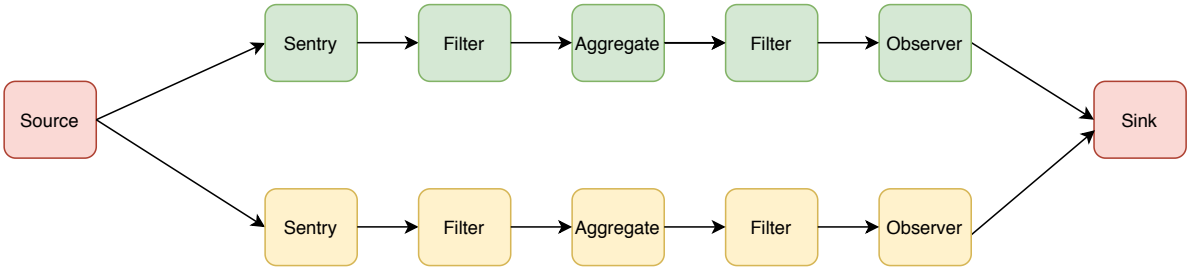
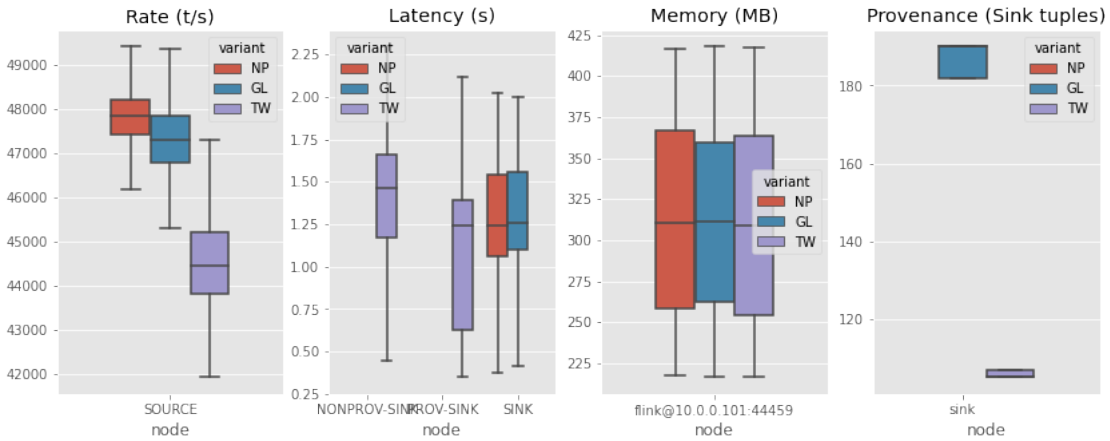


Figure 5.1: Query for detecting broken down cars.

Experiment 1 - No modifications



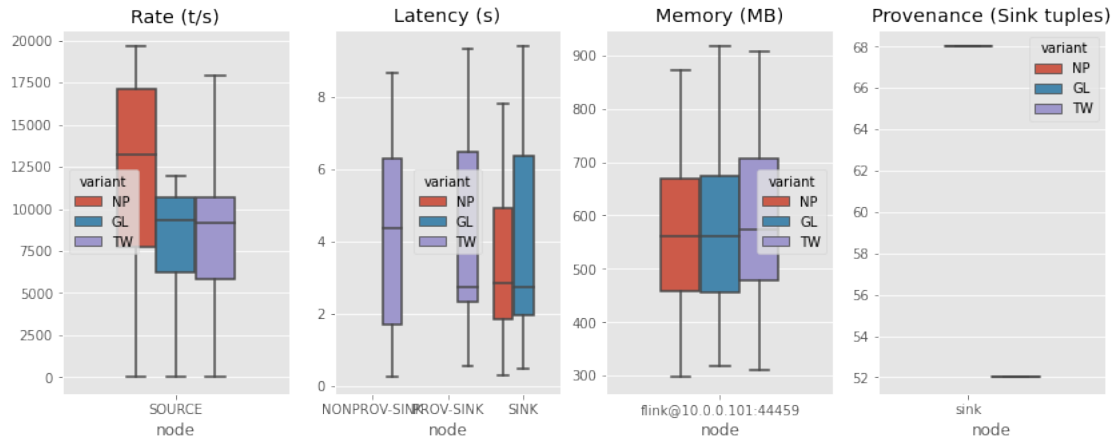
(a) Box plot representation.

(b) Graph representation.

Figure 5.2: Results from evaluating the first use case with no modifications to the query.

In Figure 5.2, the results from the first experiment for the first use case is presented. Notice that in Figure 5.2b, TW show oscillating behavior in terms of throughput, which is expected, since in theory it should transition between NP and GL. However, it has a lower throughput compared to NP and GL. The overall provenance generated by TW is approximately 60%, compared with GL.

Experiment 2 - Increasing the workload



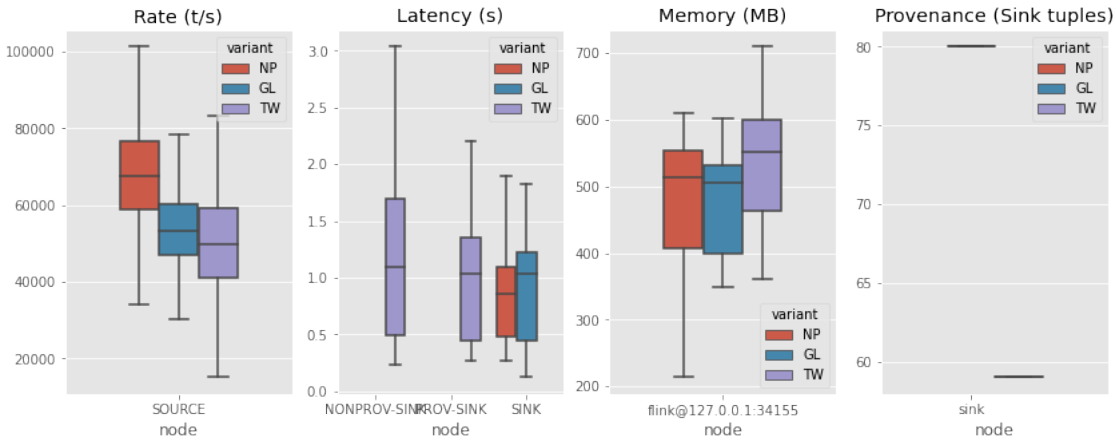
(a) Box plot representation.

(b) Graph representation.

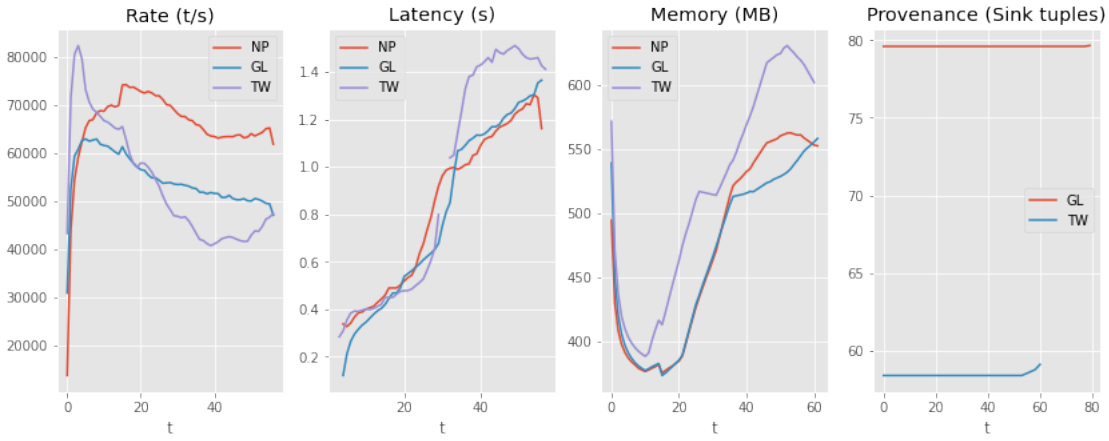
Figure 5.3: Results from evaluating the first use case where the first `Filter` have been removed, to increase the workload.

In Figure 5.3, the results from the second experiment is presented, where a higher workload has been introduced to the query. As opposed to the previous experiment, TW shows to oscillate between NP and GL, rather than below them, which is a more desirable behavior. Notice that this change of behavior emerged by introducing a heavier workload to the query. The overall provenance generated by TW is approximately 75%, compared with GL. From Figure 5.3b, the amount of transitions can be observed, which shows that the total amount of transitions performed is two, i.e., provenance is activated, deactivated and then activated again.

Experiment 3 - Different hardware



(a) Box plot representation.

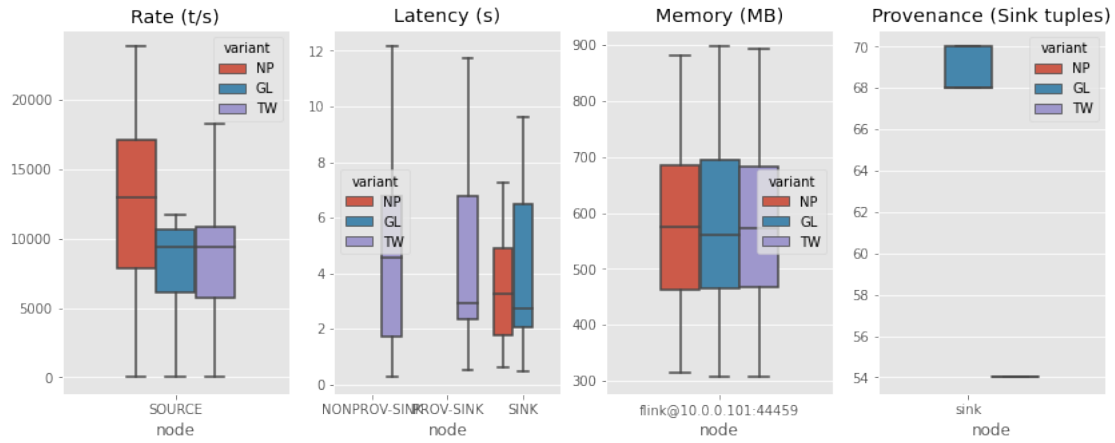


(b) Graph representation.

Figure 5.4: Results from re-running experiment 2 but on the a laptop instead of an embedded device.

In Figure 5.4, the results from third experiment is presented. Here, the setup of the second experiment is used again, but executed on a laptop instead of the Odroid. Unalike the previous behavior, the oscillating behavior of TW cannot be observed and the overall throughput is continuously decreased until the end of the experiment, where it converges with GL. The total amount of provenance generated is approximately 74%, compared with GL.

Experiment 4 - Different trigger conditions



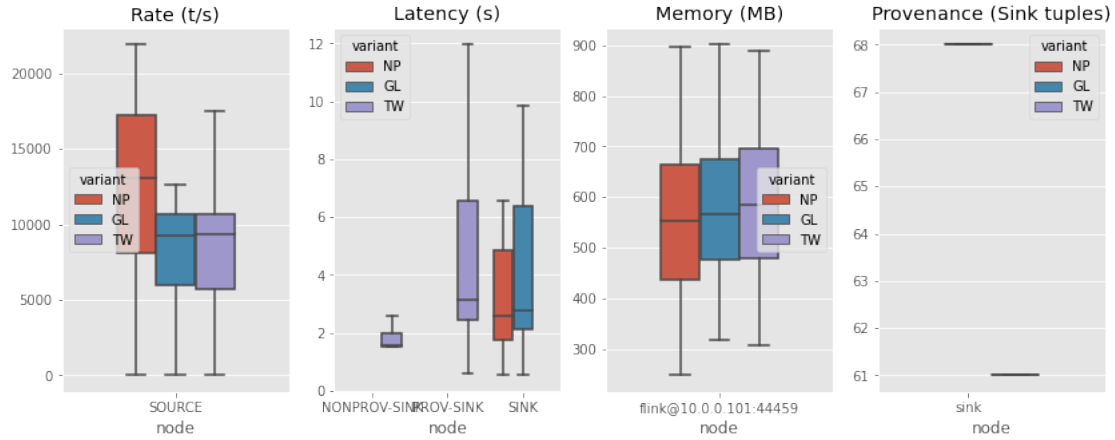
(a) Box plot representation.

(b) Graph representation.

Figure 5.5: Results from evaluating the first use case where the trigger condition for deactivating is decreased from four windows to two windows.

In Figure 5.5, the results from the fourth experiment is presented. In this experiment, the workload has been increased as well a decrease in the amount of time before provenance becomes deactivated. Similarly to the previous experiment, TW shows oscillating behavior between NP and GL, and the same amount of transitions occur. By lowering the trigger condition, an increase in the number of transitions back to the non-instrumented query can be expected, but comparing the results with Figure 5.3, the same amount of transitions occurred. The amount of provenance generated by TW is approximately 78% compared with GL, which is an 3% increase as opposed to previous experiment.

Note: Notice that the last transition to the non-instrumented query occurs earlier as opposed to the previous experiment, but results in the same amount of provenance.



(a) Box plot representation.

(b) Graph representation.

Figure 5.6: Results from evaluating the first use case where the trigger condition for deactivating is increased from four windows to eight windows.

By increasing the duration, TW would generate provenance information for a longer period of time, as opposed to the previous experiment. This can be observed in Figure 5.6b, as a single transition to the instrumented query has taken place. This resulted in an increased amount provenance, which is approximately 90% compared with GL, which is an 12% increase compared with the previous experiment.

5.2.2 Car accidents query

The second query is an extension of the previous query, which aims to detect car accidents. An accident is defined if at least two broken-down cars are found in the same position at the same time. It has the same structure of operators, with an additional `Aggregate` and `Filter`. The `Aggregate` groups each computation by the tuples position attribute, with a window size and advance of 30 seconds. The output produced carries the number of stopped vehicles observed for each position in the same time window. In order to detect at least two broken-down cars, the `Filter` only forwards tuples carrying a count value of at least 2.

Similarly to the previous query, the Ward operators are placed alike. This is illustrated in Figure 5.7.

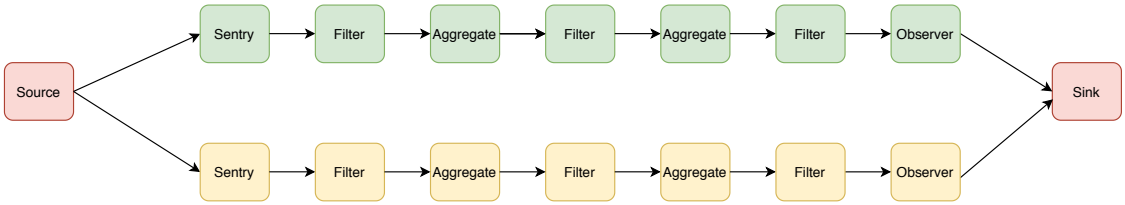
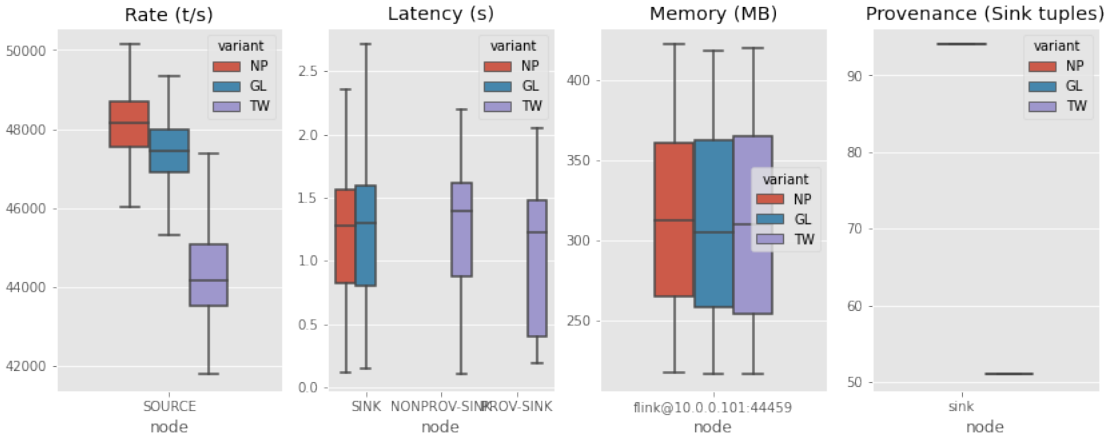


Figure 5.7: Query for detecting car accidents.

Experiment 1 - No modifications



(a) Box plot representation.

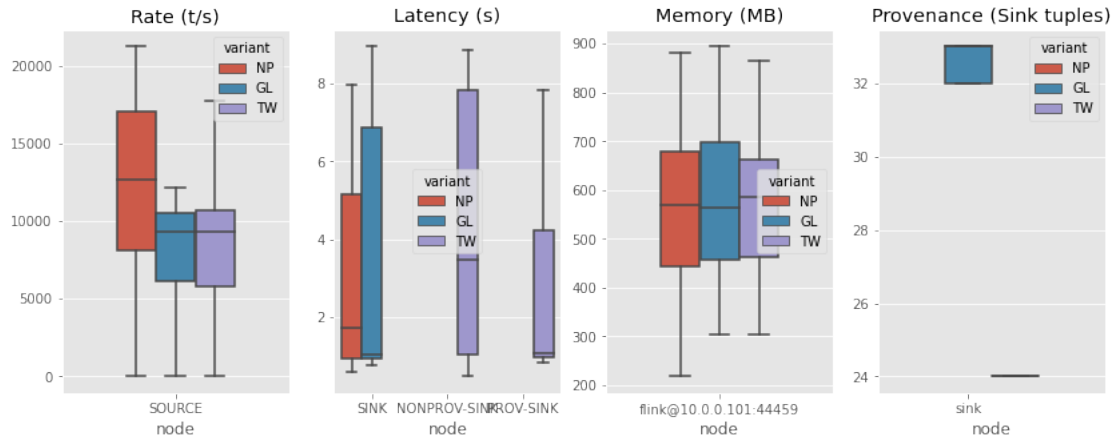
(b) Graph representation.

Figure 5.8: Results from evaluating the second use case with no modifications to the query.

In Figure 5.8, the results from the first experiment is presented. As previously observed in Figure 5.2, TW shows the same behavior by oscillating below both

GL and NP rather than between them. The amount of provenance generated is approximately 54% compared with GL.

Experiment 2 - Increasing the workload



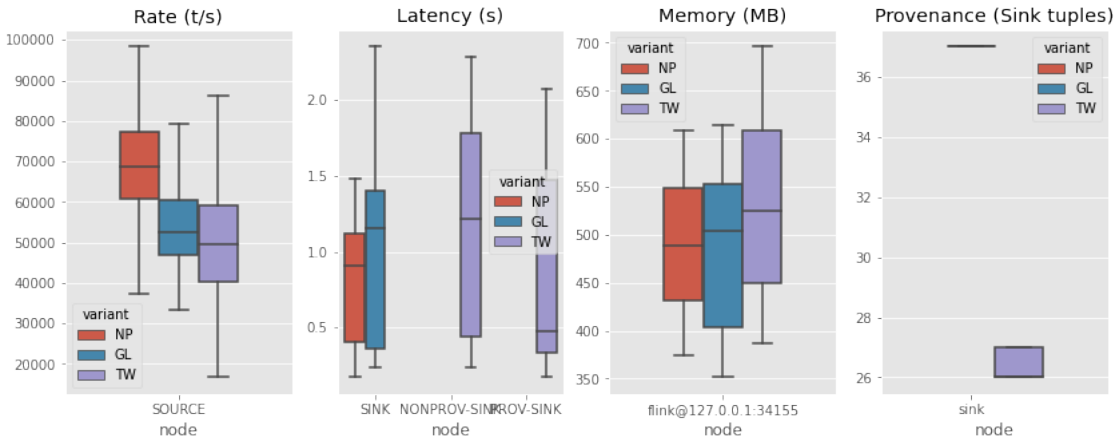
(a) Box plot representation.

(b) Graph representation.

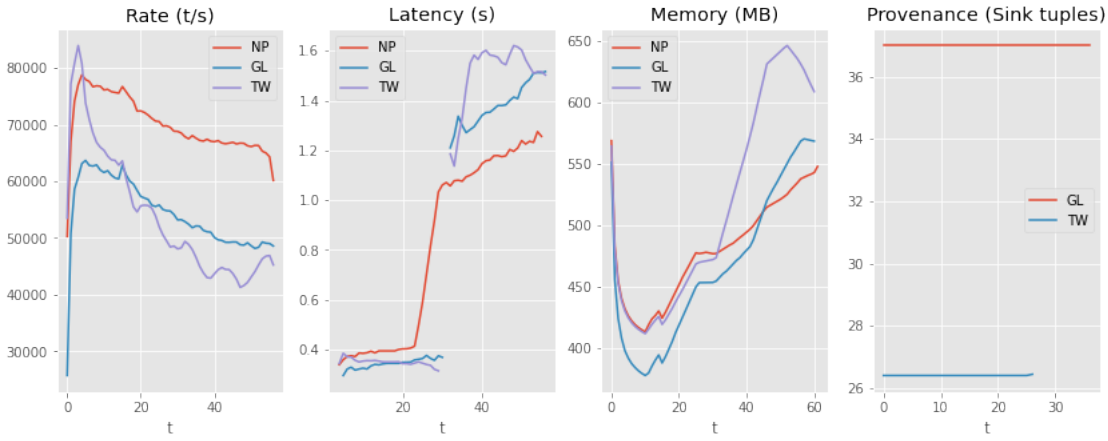
Figure 5.9: Results from evaluating the second use case where the first `Filter` is removed, to increase the workload.

In Figure 5.9, the results from the second experiment is presented. As observed in Figure 5.9b, TW shows oscillating behavior between NP and GL rather than below then, when the workload of the entire query is increased. Notice that this occurred for the first experiment as well, as observed in Figure 5.3. The amount of provenance generated by TW is approximately 73%, compared with GL.

5.2.2.1 Experiment 3 - Different hardware



(a) Box plot representation.

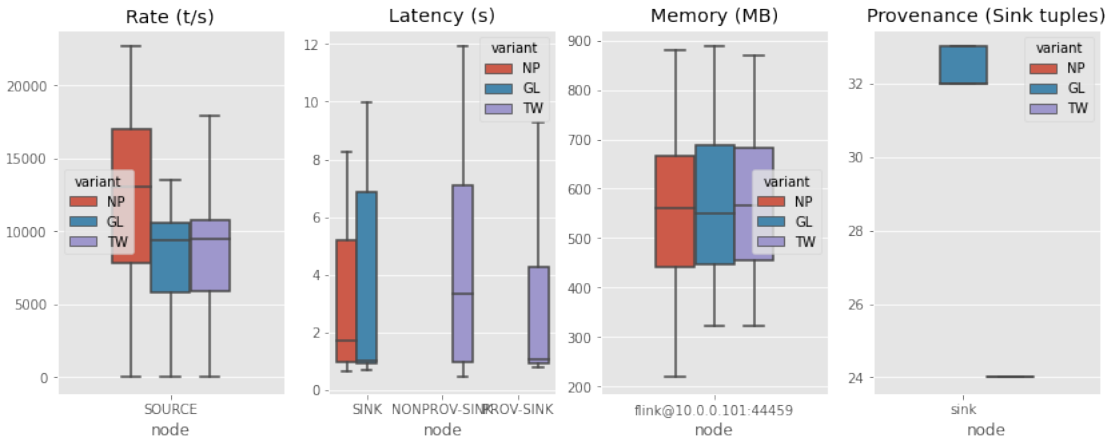


(b) Graph representation.

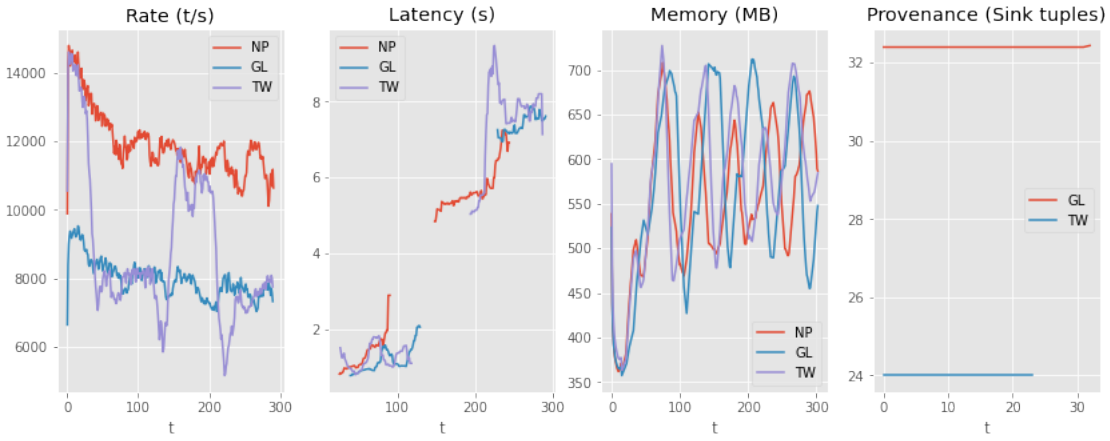
Figure 5.10: Results executing the second use case with modifications, on a laptop instead of an embedded device.

In Figure 5.10, the results from the third experiment is present. Similar to the results from the previous use case, TW does not produce oscillating behavior as the previous experiment, but rather a continuous decrease. The total amount of provenance generated is 70%, compared with GL.

Experiment 4 - Different trigger conditions



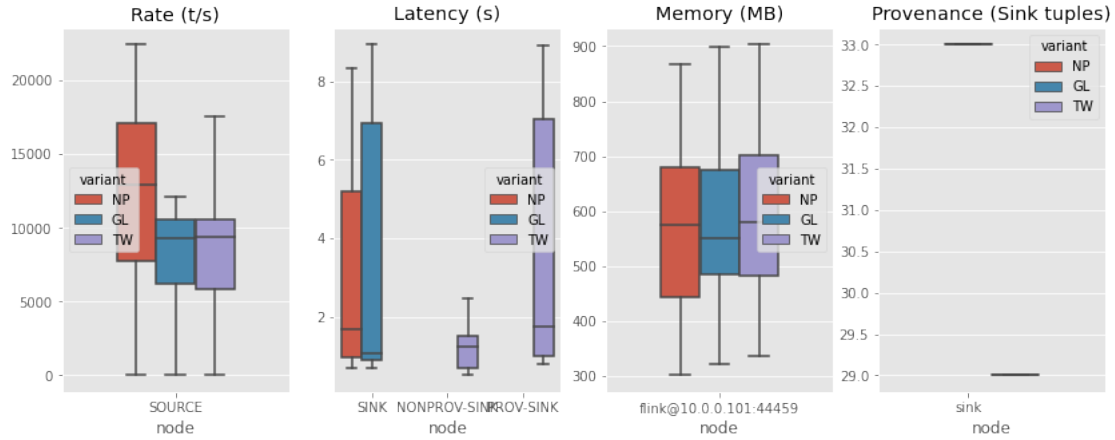
(a) Box plot representation.



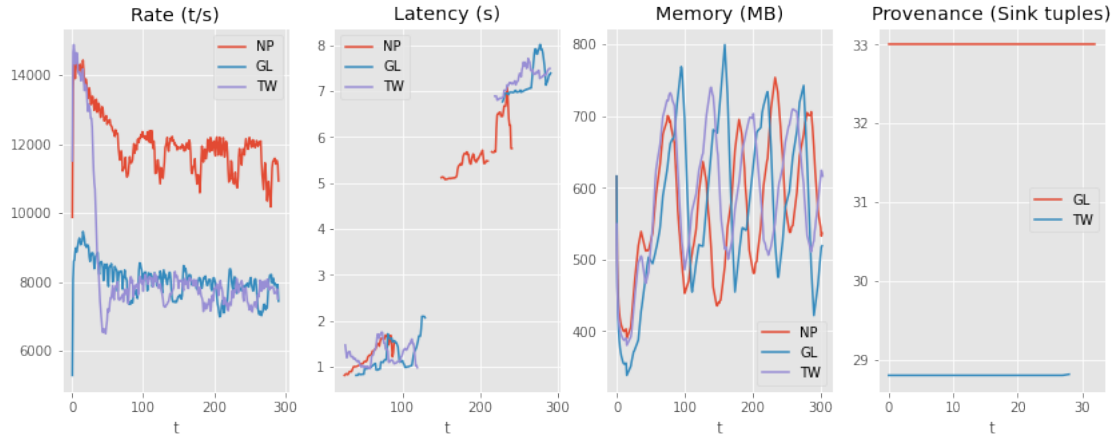
(b) Graph representation.

Figure 5.11: Results from evaluating the second use case where the trigger condition for deactivation is decreased from four windows to two windows.

In Figure 5.11, the trigger condition has been lowered from four windows to two windows. Notice in Figure 5.11b, TW achieves an earlier transition but generates approximately the same amount of provenance as in Figure 5.9, which is 73%, compared with GL.



(a) Box plot representation.



(b) Graph representation.

Figure 5.12: Results from evaluating the second use case where the trigger condition for deactivation is increased from four windows to eight windows.

As observed in Section 5.2.1, by increasing the trigger condition to eight windows, only one transition to the instrumented query will occur. With this, the total amount of provenance is increased, which is approximately 88% compared with GL, which is an 15% increase compared with the previous experiment, as seen in Figure 5.6.

5.2.3 Blackout detection query

The third query aims to detect blackouts in Smart Grid systems. Initially, the source data is grouped by each smart meter which are forwarded every hour. The tuples are then summed throughout each day by an `Aggregate`. Following, a `Filter` forwards the tuples with zero consumption to a second `Aggregate`, where the window size and advance is one day. An output is produced if there are more than seven meters which report zero consumption for a whole day.

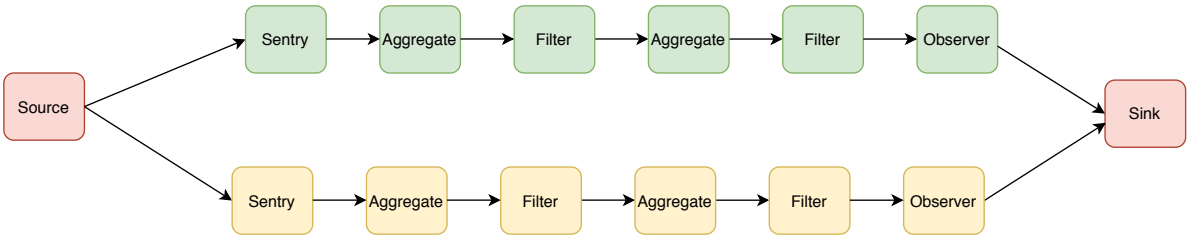
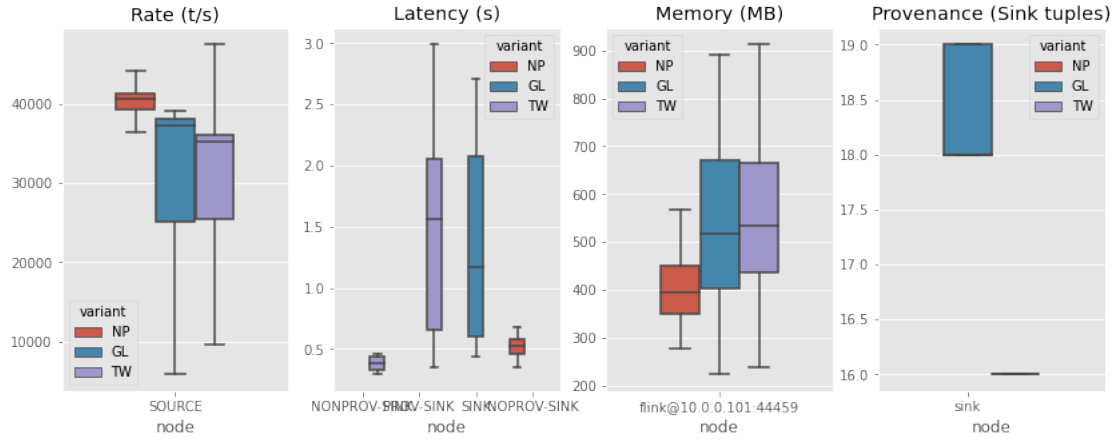
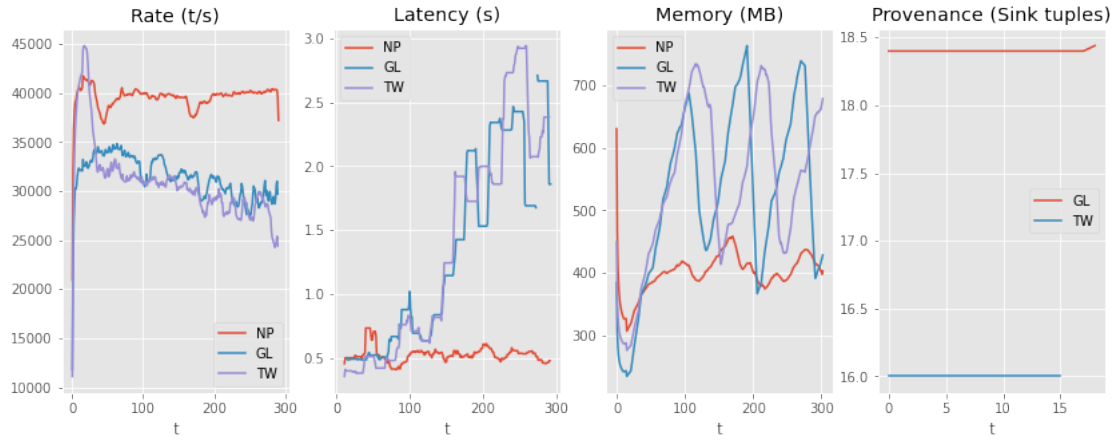


Figure 5.13: Query for detecting a long-term blackout.

Experiment 1 - No modifications



(a) Box plot representation.



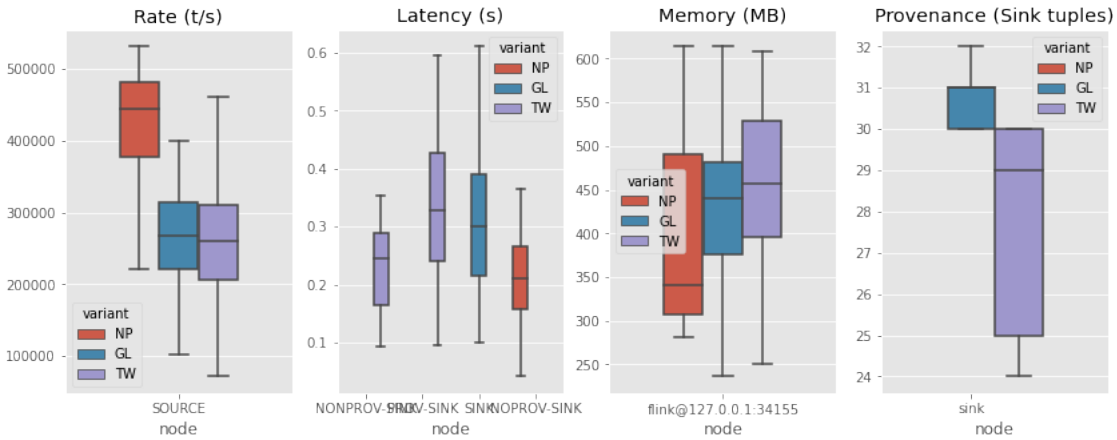
(b) Graph representation.

Figure 5.14: Results from the first experiment of the third use case where no modifications is made to the query.

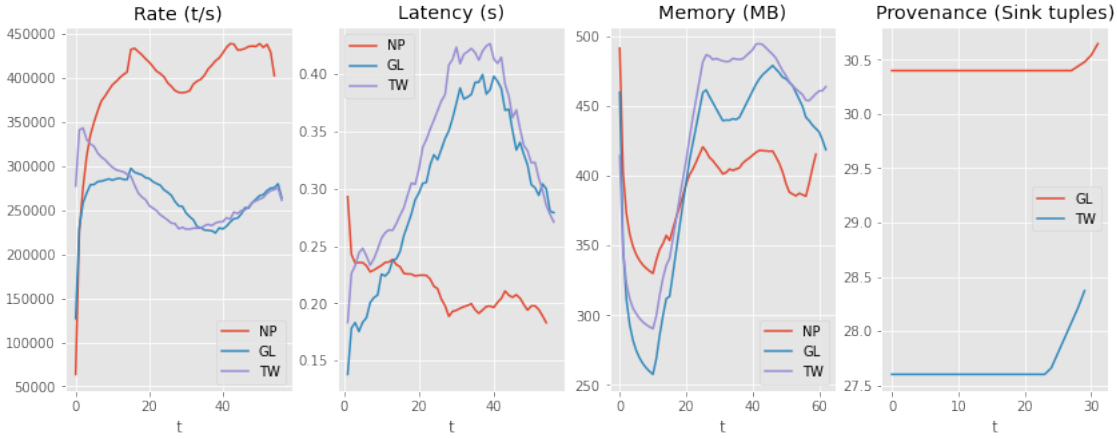
In Figure 5.14, the results from the first experiment is presented. Unalike the first experiment of the use cases based on the LinearRoadBenchmark, TW oscillates between NP and GL rather than below them, as observed in Figure 5.14a. With this, the query is already characterized as a heavy, thus only two experiments are

conducted for this use case, i.e., where different trigger conditions are used. The amount of provenance generated by TW is approximately 89% compared with GL.

Experiment 2 - Different Hardware



(a) Box plot representation.

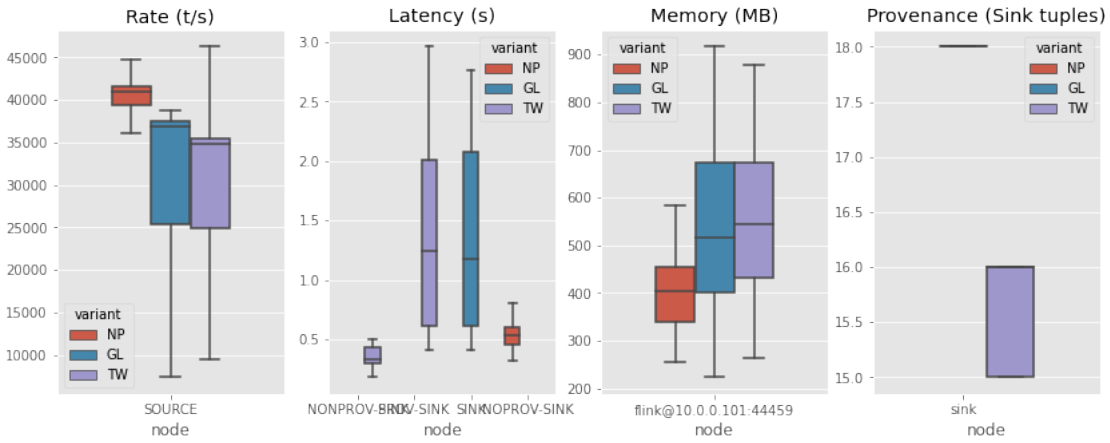


(b) Graph representation.

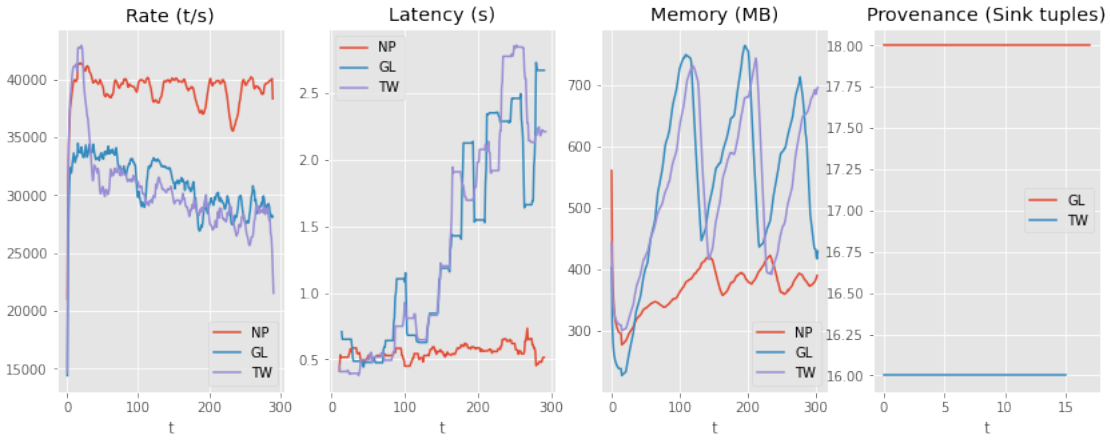
Figure 5.15: Results executing the third use case with no modifications, on a laptop instead of an embedded device.

In Figure 5.15, the results from the second experiment is presented. Similarly to the former experiments performed on the laptop, TW does not show oscillating behavior but rather a continuous decreased, which converges with GL at the end. The total amount of provenance generated is 95%, compared with GL.

Experiment 3 - Different trigger conditions



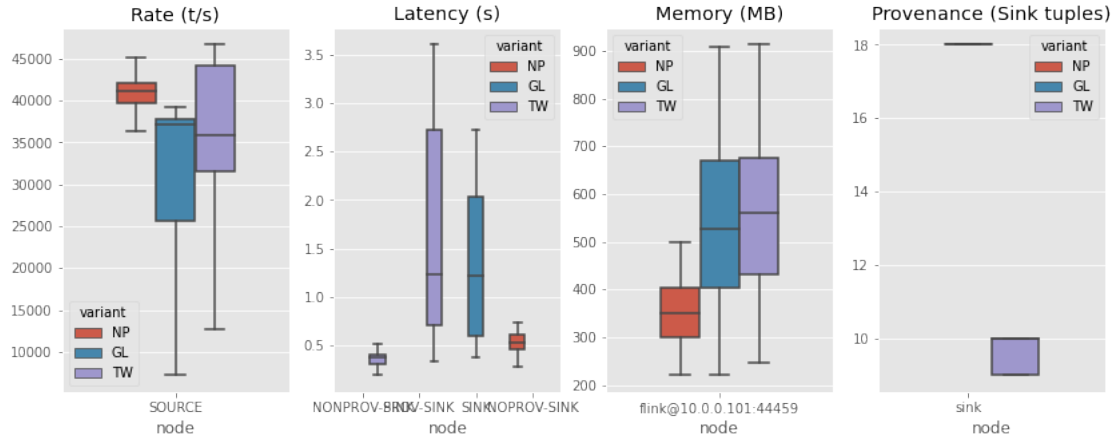
(a) Box plot representation.



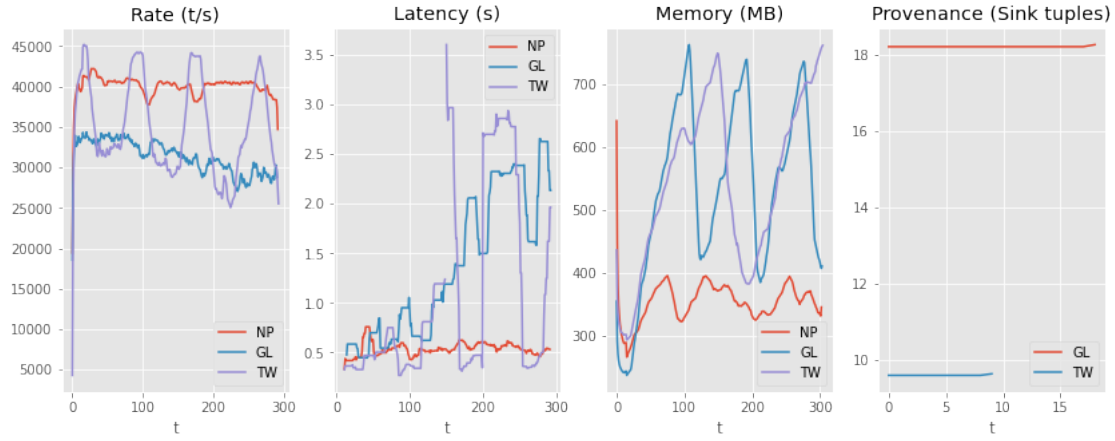
(b) Graph representation.

Figure 5.16: Results from evaluating the third use case where the trigger condition for deactivation is decreased from four windows to two windows.

In Figure 5.16, the results from the third experiment is presented. Similarly to the previous experiment, only one transition occurs and approximately the same amount of provenance is generated. With no change in behavior with a lower trigger value, performing an experiment with an increased trigger condition would not show any difference. Thus, the next experiment will the trigger condition be decreased further, to one window to observe if more than one transition occurs.



(a) Box plot representation.



(b) Graph representation.

Figure 5.17: Results from evaluating the third use case where the trigger condition for deactivation is decreased from two windows to one window.

By decreasing the trigger condition from two windows to one window, the amount of transition increased significantly from one transition to six transitions, as seen in Figure 5.17. This shows a more oscillating behavior, as observed in both Figure 5.2 and Figure 5.8. The total amount of provenance generated is approximately 55%, compared with GL.

5.2.4 Summary

To summarize, the Twin query approach has been evaluated for three different use cases with different modifications to introduce contrasting scenarios, to determine during which conditions it becomes feasible for such an approach. An outline of the results is presented in Table 5.2 for the first use case, Table 5.3 for the second use case and Table 5.4 for the last use case.

Broken-down cars	Throughput (t/s)	Rate (s)	Memory (MB)	Provenance (t)
No modifications	42500	1.30	320	60%
Increased workload	8000	3.0	560	75%
Different hardware	50000	1.1	550	74%
Decreased trigger	9000	3.0	590	78%
Increased trigger	9000	2.5	590	90%

Table 5.2: Results for the use case, broken-down cars.

Car accidents	Throughput (t/s)	Rate (s)	Memory (MB)	Provenance (t)
No modifications	44200	1.4	320	53%
Increased workload	9000	3.0	590	73 %
Different hardware	50000	1.1	540	70 %
Decreased trigger	9000	2.0	590	73 %
Increased trigger	9000	1.5	590	88 %

Table 5.3: Results for the use case, car accidents.

Blackout detection	Throughput (t/s)	Rate (s)	Memory (MB)	Provenance (t)
No modifications	35000	1.0	520	89 %
Different hardware	250000	0.3	460	95 %
Decreased trigger	35000	0.8	550	88%
Decreased trigger 2	35000	0.8	550	55 %

Table 5.4: Results for the use case, long-term blackout detection.

6

Discussion

In this chapter, a discussion regarding the evaluation and the implementation is presented. The methodology of Twins was implemented for single node deployments which relies on shared memory, which becomes infeasible for distributed deployments. In Section 6.1, different implementation constraints and unexamined directions are discussed. Furthermore, the evaluation of Twins showed for most of the scenarios that it is on par with GeneaLog in terms of memory consumption, which originally was hypothesized to decrease. This will be discussed in Section 6.2.

6.1 Implementation

A prototype of GeneaLog was implemented for the SPE Apache Flink, which has been used in this thesis. As mentioned in Section 1.2, to achieve an extension which allows GeneaLog to *interactively* generate provenance information would require extra functionality, which is not present in Flink. In this section, different implementation directions for future work are discussed.

6.1.1 Distributed execution

GeneaLog allows provenance generation for both non-distributed and distributed deployments [7]. As mentioned in Section 4.1, there is no upstream or downstream communicating functionality between operators in a query. Due to this constraint, a limitation was set in the beginning of the thesis to evaluate enabling and disabling the method for single node deployments, utilizing shared memory. The communication between operators relies on shared variables using a centralized entity (singleton). Such an approach is not feasible for distributed deployments, since the singleton resides within a separate JVM than the other distributed tasks, where references would be lost upon forwarding tuples across tasks.

An alternative for distributed deployments would be to substitute the Singleton class with a database, where the shared variables is stored. Naturally, this would result in a significant increase in the reconfiguration time, due to network latency. If an application allows a more relaxed provenance generation, it would be an alternative.

6.1.2 Reconfiguration time

Enabling and disabling the generation of provenance information naturally results in a decrease of provenance. Additionally, since a transition cannot occur immediately, the procedure of achieving a safe transition will result in lost provenance information as well, with regards to the stateful operators and their ongoing computations.

If the tuples are read from a predefined data set in a file, the reconfiguration time will be sensitive to the amount of tuples that are produced by the source, as it will affect the progression in time between the Sentries and the Observers. Additionally, the Sentries periodically poll the state of the Observer to detect if a transition has been requested, which can result in a delay for when the transition request has been noticed by a Sentry. All this accumulates into a latency between when provenance is requested to when its actually achieved, which affects the reconfiguration time.

In a real-world scenario, where tuples are produced in real time and retrieved from sensors (as opposed to a data set), the ingestion of tuples would be slower, which would in turn lower the latency and thus lower the reconfiguration time during a transition. This could result in a faster transition to the respective queries which in turn could affect the throughput, memory consumption and generated provenance in Twins, depending on the amount of transitions.

However, with a reduced reconfiguration time, the difference in performance could still be insignificant. Since the reconfiguration time will for the majority of time depend on the windows of the stateful operators, thus the reconfiguration time can not be reduced lower than the windows themselves, for a safe transition to work.

In short, the reconfiguration time will affect the performance of Twins, but it is anticipated that it would result in insignificant difference with the chosen placement of the Ward operators, thus the reconfiguration time was never evaluated. However, as discussed in Section 3.2.1, the reconfiguration time could be lowered or entirely removed by relocating the Sentries so that the stateful operators are always ready to take over execution, though this will add a lot of overhead.

6.1.2.1 Partial activation and deactivation

Partial activation as explained in Section 3.5, can (in theory) lower the amount of lost provenance and decrease the reconfiguration time significantly as there is no need to perform a safe transition. By allowing the stateful operators in both queries to process tuples, there is no need for synchronization, as the transition could be simpler. However, such a modification to the procedure would introduce significant overhead, as both of the queries are required to run in parallel during a longer period of time. There is a high possibility that the overhead generated with such a modification, would result in a decreased performance for Twins.

In conclusion, by partially transition to the other query, the amount of provenance could be increased, but result in a significant increased amount of overhead. With Twins averaging on 12 % lower provenance generation and showed acceptable

throughput, partial activation was not implemented and evaluated.

6.1.3 Aborting transitions

In the implementation presented in Section 4, the Observer operator is mainly responsible for initiating a transition. After a transition has been made to the instrumented query, a transition back to the non-instrumented will occur if there has not been a sink tuple produced within the completion of four windows, which is the baseline deactivation for the implementation (see Section 5.1.3) . Since the possibility exists that during the transition to the non-instrumented query, one final output could be produced. Ideally, this would cancel the deactivation and remain in the instrumented query, but due to restrain of an Observer not being able to either cancel or initiate another transition, the deactivation will occur regardless.

An optimization, or rather extension, to the protocol would be to allow the Observer in the instrumented query to potentially cancel a transition. If the production of a sink tuple would occur after an initiation of a transition to the non-instrumented query has happened, then the instrumented query should keep ruining and the transitions should be canceled.

6.1.4 User interaction for deactivation

As this thesis explores the possibility of activating and deactivating provenance based on tuples, another alternative direction would be to base the functionality on user interaction. Detected outputs are used for notifying human supervisors that undesired behavior has occurred, then the decision to activate provenance rests in the hands of the user. This would result in extra functionality where the supervisors themselves can control the amount of provenance, which could be desired.

6.2 Evaluation

The goal of this thesis is to evaluate whether GeneaLog can be extended to generate provenance information interactively and if this will result in a reduced amount of overhead, i.e., a lower memory consumption and an increase in throughput. As observed in Section 5.2.4, the memory consumption of Twins compared to GeneaLog shows to be on par for every use case, except when performing the experiments on a more powerful device. Similarly, this is also the case in terms of throughput.

This was observed during the original evaluation of GeneaLog as well, as the authors of GeneaLog mention that they did not notice any notable changes in the average and maximum memory consumption when GeneaLog is enabled [7]. Furthermore, they also mention that there were slightly no difference between the two different LinearRoad use cases, but an increase was observed for the Smart grid use case. This is due the amount of events produced and the size of the windows, which results in a larger amount of tuples contributing to a window and thus an increase of provenance data.

During the evaluation, a modification was made to the use cases based on Linear-Road, to introduce an increased workload by removing the first `Filter` and introducing a scenario where provenance becomes a heavier operation. This resulted in a overall decrease in throughput for each variant (NP, GL and TW). Furthermore, Twins shows a more desirable behavior, since it oscillates between NP and GL rather than below them. However, even though an amplification to the amount of tuples allowed to progress from the source was made, there were still no notable changes in the average and maximum memory consumption for either use case and Twins showed to be on par with GeneaLog in terms of throughput.

In theory, the memory consumption of Twins should be lowered compared with GeneaLog, as the generation of provenance information is not continuous. Unable to observe any notable changes in memory between the different variants, the original hypothesis presented in Section 1.1.1 cannot be validated (*Q3)*. This would require further evaluation of GeneaLog and consequently Twins on other SPEs, to prove this hypothesis.

7

Related Work

In this chapter, past research and related works for data provenance will be discussed. To the best of our knowledge, there has not been any research conducted in activating/deactivating provenance for data streams. However, there are still interesting related research in other fields of data streaming that will be important for this thesis.

In Section 7.1, research and works in data provenance techniques will be discussed. As the objective of this thesis is to evaluate the cost of activating and deactivating provenance throughout a data stream, analyzing prior techniques and their approaches is of interest.

In Section 7.2, related research in fault tolerance techniques for data streams and load balancing by switching node mid-stream will be discussed. The approach described in this thesis is based on having two identical queries, where switching from one to the other should ensure the correct state. Research within fault tolerance and load balancing for data streaming is of interest, as this thesis shares similar functionality.

7.1 Data provenance techniques

In relational databases, the work by [6] present a system based on a technique called *query rewrite*. Their approach is to generate provenance by rewriting queries. For a given query, they would generate another query extended with additional attributes used to store provenance data.

This approach was not applicable for streaming based applications, since there are a number of challenges present for data streaming that are not addressed, as mentioned in the works by [23]. They proposed a new approach called *Operator instrumentation*, where provenance generation is added as an extension to the standard operators found in stream processing engines. They realized this in Ariadne, a provenance-aware extension which implements the technique.

As [23] addresses the challenges for provenance techniques for data streams, their solution uses variable-length annotations which are represented as sets of tuple identifiers. Provenance is then later achieved by reconstructing the input tuples which

contribute to an output tuple from these identifiers. In order for this to work, all of the input tuples needs to be temporarily stored. For applications that run on resource-constrained devices, this could become troublesome, since the annotations could grow arbitrary large.

This thesis is a continuation of the works by [7], the current state-of-the-art data provenance technique for data streaming. They proposed a new technique based on operator instrumentation, which is more suitable for a wider range of devices, such as cyber-physical systems (CPSs) which use resource-constrained devices. As their annotations are fixed-size instead of variable-size, it also does not require to store all of the input tuples, but uses memory pointers to distinguish the contributing tuples only.

As discussed in Chapter 6, extending GeneaLog to be enabled and disabled throughout a data stream showed to resemble the performance of GeneaLog while active, and that of a baseline query with standard operators while inactive. By being able to shift the throughput implies a more efficient usage of resources, which is a common research question for data analysis applications, as they range from high-end servers to embedded edge devices (e.g., Odroid) [24], [25]. With this, GeneaLog and consequently data provenance techniques can be extended to introduce a more resource-flexible behavior and thus be introduced to a wider range of devices and applications.

7.2 Synchronization between stateful operators

In this section, the works by [26] and [27] is discussed. The authors of both papers encounter problems similar to Section 3.3, for synchronization of states between parallel instances of the same stateful operator.

In [26], they propose *active replication* of their operators, where entire- or parts of a query is replicated, in order to provide fault tolerance i.e., they route the source into multiple identical operators. This adds fault tolerance when one operator fails, since another can take its place. This means that each replica has to be in the exact same state as the primary operator at all times.

To ensure consistency among all the replicas and solve synchronicity problems with the stateful operators local states, events (tuples) must be processed in the same order and at the same time, in all replicas where the computations must be deterministic. To solve this, they introduce *requirements* on the usage of an ordered message delivering mechanism (e.g., atomic broadcast) to ensure all nodes agree on the order of the messages and that they receive all messages. This adds significant latency.

The synchronicity problem with stateful operators arises in [27] as well, where a new parallel-distributed SPE, *StreamCloud*, is introduced. This SPE utilizes dynamic load balancing, which offloads computations from a congested node, to another less

congested node.

This involves changing to a parallel identical query, which will require their stateful operators to synchronize, to preserve the correctness of the analysis. The author suggested utilizing timestamps, `startTS` and `endTS`, which represents the start and end of a window. These are communicated upstream to ensure a correct transition to the other query.

This is illustrated in Figure 7.1, where the point marked as *"1"* is where the transfer starts. `startTS` is then set to when the new query (or node) should start, with the requirement that it is at the start of a new window. In the Figure this is the point marked as *"3"*. `endTS` is then set to when the current window ends, effectively overlapping the two timestamps, i.e. the point marked as *"2"*. All tuples with timestamp, `ts` > `startTS` are sent to the new query, and all tuples with `ts` < `endTS` are sent to the congested node. Thus, as the queries both handle the tuples during the transition period, no windows that end in-between will be lost and the transfer is complete. This is also illustrated in the figure, where the upper box (the green box) illustrates all windows handled by the congested node, and all new windows are handled by the less congested node, illustrated by the lower (blue box).

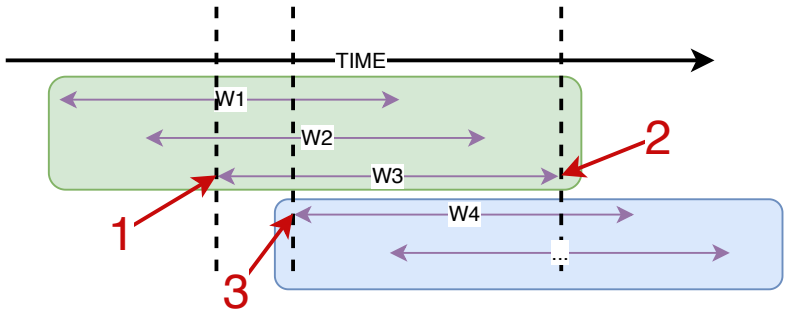


Figure 7.1: Illustration of the window synchronization problem.

This technique introduces a lower demand on hardware as well as a lower latency than [26]. As Twins involves transitions between two queries, the transition is required to be *safe*, to preserve correctness. Furthermore, as GeneaLog targets resource-constraint devices, the transition technique requires to be efficient and not resource-demanding.

In this thesis, the safe transition technique presented in Section 3.3.2 takes inspiration from the usage of timestamps and windows to ensure a safe transition. However, during a transition, the deactivating query does not terminate but rather becomes idle, as a transition back could eventually occur. This means that tuples will still reach both queries but is only processed in one, the active query, and the other query is considered to be inactive. Furthermore, the technique proposed in this thesis introduces communication between the operators to detect whether incorrect or duplicated output has been produced during a transition, as transitions could occur back and forth.

8

Conclusion

In this thesis, an extension to the current state-of-the-art data provenance technique, GeneaLog, has been developed and implemented, called Twins. Twins simulates activation and deactivation of GeneaLog by using two identical queries, one using GeneaLogs instrumented operators while the other query uses non-instrumented operators. By allowing only one query to be active and transitioning between them, the generation of provenance information can be enabled and disabled. With this, *Q1* as presented in Section 1.1.2 is answered.

Enabling and disabling GeneaLog shows a decreased amount of generated provenance, as opposed to generating provenance information continuously. Twins showed to on average produce 12% less provenance (though this is subject to the user-defined condition) than GeneaLog during its evaluation. With this, *Q2* is answered.

During the evaluation, Twins was studied using three different use cases where different contrasting scenarios were introduced, to evaluate during which conditions the functionality of Twins showed to be beneficial. Twins showed to oscillate in terms of both throughput and memory consumption, which was expected, as it will simulate enabling and disabling GeneaLog. However, such behavior was only present for the experiments which involved the use cases with an increased workload. With this, the performance of Twins matches the performance of the original query when provenance is inactive and matches GeneaLog when provenance is active. Thus, to summarize and answer *Q3*, the performance does not become significantly affected while using Twins, as long as the query is subjected to a high amount of tuples. Namely, in scenarios where the procedure of recording provenance becomes a resource-demanding operation, roughly 30% lower throughput compared to a query with non-instrumented operators.

Bibliography

[1] Streaming Analytics Market Size, Share and Global Market Forecast to 2024 | MarketsandMarkets, Feb 2020. [Online; accessed 25. Feb. 2020].

[2] The Big Data Debate: Batch Versus Stream Processing - The New Stack, Jun 2018. [Online; accessed 25. Feb. 2020].

[3] Ands. Data provenance, May 2020. [Online; accessed 6. May 2020].

[4] Kyle Cranmer, Lukas Heinrich, Roger Jones, and David M. South. Analysis Preservation in ATLAS. *J. Phys. Conf. Ser.*, 664(3):032013, 2015.

[5] Qi Wang, Wajih Ul Hassan, Ding Li, Kangkook Jee, Xiao Yu, Kexuan Zou, Junghwan Rhee, Zhengzhang Chen, Wei Cheng, C Gunter, et al. You are what you do: Hunting stealthy malware via data provenance analysis. In *Proc. of the Symposium on Network and Distributed System Security (NDSS)*, 2020.

[6] Boris Glavic and Gustavo Alonso. Perm: Processing Provenance and Data on the Same Data Model through Query Rewriting. *undefined*, 2009.

[7] Dimitris Palyvos-Giannas, Vincenzo Gulisano, and Marina Papatriantafilou. GeneaLog: Fine-grained data streaming provenance in cyber-physical systems. *Parallel Comput.*, 89:102552, Nov 2019.

[8] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The design of the borealis stream processing engine. In *Cidr*, volume 5, pages 277–289, 2005.

[9] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[10] Site Planning Guide for Sun Servers, Apr 2020. [Online; accessed 11. Apr. 2020].

[11] Windows (flink 1.10-SNAPSHOT API), Apr 2020. [Online; accessed 11. May. 2020].

[12] Apache Flink: Stateful Computations over Data Streams, Feb 2020. [Online; accessed 13. Feb. 2020].

[13] Apache Flink 1.10 Documentation: Parallel Execution, Apr 2020. [Online; accessed 16. Apr. 2020].

[14] Ariadne | Proceedings of the 7th ACM international conference on Distributed event-based systems, Mar 2020. [Online; accessed 23. Mar. 2020].

[15] Emil Vassev and Joey Paquet. Aspects of Memory Management in Java and C++, Jun 2006. [Online; accessed 18. Jun. 2020].

[16] Eva Andreasson. JVM performance optimization, Part 3: Garbage collection. *JavaWorld*, Oct 2012.

[17] Apache Flink: Juggling with Bits and Bytes, Apr 2020. [Online; accessed 15. Apr. 2020].

[18] Design patterns : elements of reusable object-oriented software : Gamma, Erich : Free Download, Borrow, and Streaming : Internet Archive, Apr 2020. [Online; accessed 27. Apr. 2020].

[19] ConcurrentHashMap (Java Platform SE 8), Mar 2020. [Online; accessed 16. Apr. 2020].

[20] Atomic Access (The Java™ Tutorials > Essential Classes > Concurrency), Apr 2020. [Online; accessed 20. Apr. 2020].

[21] RichFilterFunction (flink 1.10-SNAPSHOT API), Apr 2020. [Online; accessed 27. Apr. 2020].

[22] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: A stream data management benchmark. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, page 480–491. VLDB Endowment, 2004.

[23] Boris Glavic, Kyumars Sheykh Esmaili, Peter M. Fischer, and Nesime Tatbul. Efficient Stream Provenance via Operator Instrumentation. *ACM Trans. Internet Technol.*, 14(1):1–26, Jul 2014.

[24] Dimitrios Palyvos-Giannas, Vincenzo Massimiliano Gulisano, and Marina Papatriantafilou. Haren: A Framework for Ad-Hoc Thread Scheduling Policies for Data Streaming Applications. *Chalmers Research*, 2019.

[25] Bastian Havers, Romaric Duvignau, Hannaneh Najdataei, Vincenzo Massimiliano Gulisano, Marina Papatriantafilou, and Ashok Chaitanya Koppisetty. DRIVEN: A framework for efficient Data Retrieval and clustering in Vehicular Networks. *Chalmers Research*, 2020.

[26] Andrey Brito, Christof Fetzer, and Pascal Felber. Multithreading-enabled active replication for event stream processing operators. In *2009 28th IEEE International Symposium on Reliable Distributed Systems*, pages 22–31. IEEE, 2009.

[27] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patiño-Martínez, Claudio Soriente, and Patrick Valduriez. StreamCloud: An Elastic and Scalable Data Streaming System. *IEEE Trans. Parallel Distrib. Syst.*, 23(12):2351–2365, Dec 2012.

A

Appendix 1

A.1 Traversal of the contribution graph

Listing A.1: Contribution graph traversal.

```
Set findProvenance(root):
Set provenance
Set visited
Queue q
q.addLast(root)
    while(!q.isEmpty()) :
     Tuple t = q.removeFirst()
     switch(t.type) :
      case SOURCE or REMOTE:
        result.add(t)
      case MAP or MULTIPLEX:
        enqueueIfNotVisited(t.U1, q, visited)
      case JOIN:
        enqueueIfNotVisited(t.U1, q, visited)
        enqueueIfNotVisited(t.U2, q, visited)
        break;
      case AGGREGATE:
        enqueueIfNotVisited(t.U2, q, visited)
        Tuple temp = t.U2.N;
        while(temp != null && temp != t.U1)
         enqueueIfNotVisited(temp, q, visited)
         temp = temp.N;
         enqueueIfNotVisited(t.U1, q, visited)
    return result;

void enqueueIfNotVisited(tuple, queue, visited :
    if(!visited.contains(tuple)):
     visited.add(t)
    queue.addLast(t)
```

A.2 Implicit inter-task provenance method

Listing A.2: Combining the behavior of Remote and Aggregate for implicit inter-task provenance.

```
case SOURCE:                                                    1
 result.add(t)                                                  2
case AGGREGATE or REMOTE:                                       3
 enqueueIfNotVisited(t.U2, q, visited)                          4
 Tuple temp = t.U2.N;                                           5
 while (temp != null && temp != t.U1)                           6
  enqueueIfNotVisited(temp, q, visited)                         7
  temp = temp.N;                                                8
 enqueueIfNotVisited(t.U1, q, visited)                          9
```