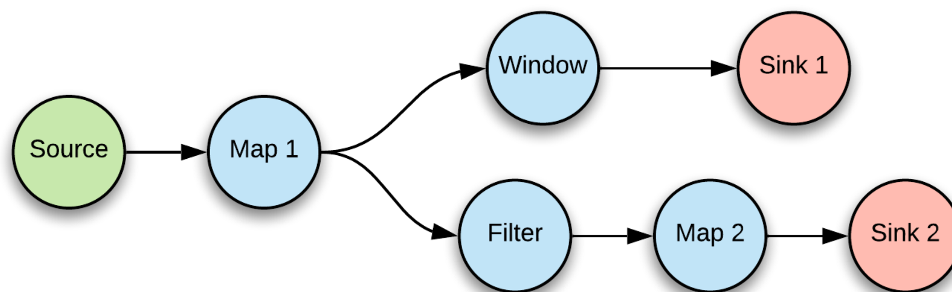
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



A GUI for application design and performance reporting of data streaming applications

Master's thesis in Computer science and engineering - Computer Systems and Networks

Rikard Teodorsson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

A GUI for application design and performance reporting of data streaming applications

Rikard Teodorsson

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

A GUI for application design and performance reporting of data streaming applications

Rikard Teodorsson

© Rikard Teodorsson, 2021.

Supervisor: Vincenzo Massimiliano Gulisano, Department of Computer Science and Engineering
Examiner: Romaric Duvignau, Department of Computer Science and Engineering

Master's Thesis 2021
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 Directed Acyclic Graph (DAG) of a streaming application

Typeset in LaTeX
Gothenburg, Sweden 2021

A GUI for application design and performance reporting of data streaming applications
Rikard Teodorsson
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

As a result of more connected devices and users and applications moving to the cloud, every year the amount of data needed to be processed by different computer systems increases. The troubles with trying to store all this information, and a growing demand for real-time processing not possible with traditional batch processing, has led to an increase in the popularity of stream processing as a way of handling large amounts of data in a real-time fashion with low latency and high throughput.
The number of Stream Processing Engines (SPEs) available for processing this streaming data is increasing and each has its own programming style and conventions, resulting in individually developed SPEs that differ in many aspects. In general, there is a lack of a common set of tools supporting developers to design, visualise and monitor streaming applications. For instance, a streaming application executed by an SPE can be modelled as a Directed Acyclic Graph (DAG), but not all SPEs have a tool to visualise this graph.
This thesis concerns the design and implementation of a framework in the form of a Graphical User Interface (GUI) as a first step towards a unified view of streaming applications, aiding developers with visual representations, statistics, and code generation for different SPEs by abstracting away the implementation details.
Using Java and JavaFX, a framework is developed and tested with two SPEs (Apache Flink and Liebre) and then evaluated in terms of performance, functionality, and generality.
Presented is a functional framework allowing for a user to visualise DAGs of existing streaming applications, viewing live and offline statistics of executions, and the possibility to design a streaming application graphically and generating the corresponding Java code. The implementation is general and can be adapted to work with other popular SPEs.
The resulting framework is a first step towards a unified view of streaming applications from different SPEs and adds a tool for developers to use, enabling for increased productivity and better understanding while working with both new and already existing streaming applications.

Keywords: GUI, SPE, stream processing, graphical user interface, framework, graph design, code generation, streaming application, query statistics

Acknowledgements

I would like to thank Vincenzo Massimiliano Gulisano for his great work as a supervisor and for guiding me and providing assistance throughout this thesis.

I also want to thank Dimitrios Palyvos-Giannas for his support with Liebre and statistics, and Romaric Duvignau, my examiner, that provided valuable feedback and ideas to improve the project.

Finally, a big thank you to my family for their support.

Rikard Teodorsson, Gothenburg, August 2021

Contents

Contents ix

List of Figures xiii

List of Tables xv

List of Listings xviii

1 Introduction 1
- 1.1 Short problem description . . . 1
- 1.2 Goal . . . 2
- 1.3 Method . . . 3
- 1.4 Outline . . . 4

2 Background 5
- 2.1 Stream Processing . . . 5
  - 2.1.1 Streams, tuples and directed graphs . . . 5
  - 2.1.2 Stream operators . . . 7
  - 2.1.3 Windows in stateful operations . . . 9
  - 2.1.4 Statistics in Stream Processing . . . 9
  - 2.1.5 Stream Processing Engines . . . 10
- 2.2 Statistics with Graphite . . . 11
  - 2.2.1 Query structure . . . 12
  - 2.2.2 The API . . . 12
- 2.3 Parsing Java code . . . 13
  - 2.3.1 JavaParser . . . 13
  - 2.3.2 Matching Java code with Regular Expressions . . . 14

3 Problem description 17
- 3.1 Shortcomings of current Stream Processing Engines . . . 17
- 3.2 Evaluating the framework . . . 19

4 Related Work 21

5 Design 25
- 5.1 The GUI elements . . . 25
  - 5.1.1 Layout and graph design . . . 26

5.1.2 Exporting and importing graphs 27
5.1.3 Text suggestions for user inputs 27
5.1.4 Graph and type analysis 27
5.2 From a graph to generated code 29
5.3 Visualising streaming applications 29
5.4 Graph and operator statistics 31
5.4.1 Statistics from Liebre 31
5.4.2 Statistics from Graphite 33

6 Implementation **37**
6.1 The GUI elements 37
6.1.1 Layout and graph design 38
6.1.2 Exporting and importing graphs 41
6.1.3 Text suggestions for user inputs 42
6.1.4 Graph and type analysis 43
6.2 From a graph to generated code 43
6.2.1 Creating a DAG 43
6.2.2 From a DAG to code 44
6.3 Visualising streaming applications 45
6.4 Graph and operator statistics 49
6.4.1 Statistics from Liebre 49
6.4.2 Statistics from Graphite 50

7 Evaluation **51**
7.1 Visualisation of streaming applications 51
7.2 Application design and code generation 56
7.3 Statistics reporting 56
7.3.1 Reliability 57
7.3.2 Processing latency 57
7.4 Performance 58
7.5 Generality 61
7.5.1 Application design and code generation 61
7.5.2 Statistics 62
7.5.3 Visualisation 62
7.6 Discussion 62
7.6.1 The GUI 62
7.6.2 Query visualisation 63
7.6.3 Application design and code generation 63
7.6.4 Performance reporting 64
7.6.5 Generality 65
7.6.6 Ethical aspects, open source and license 65

8 Conclusion **67**

Bibliography **69**

A Code listings **I**
A.1 Weather example - Flink I
A.2 Weather example - Liebre III
A.3 Weather example generated - Flink VI

B SPE JSON file **IX**

List of Figures

2.1 An example of a streaming application modelled as a directed acyclic graph, with one source, four operators and two sinks. 6
2.2 The schema for the streams in Figure 2.1. 6
2.3 A line chart from Graphite's web interface. 12
2.4 The Graphite tree structure for Liebre metrics. 13
2.5 An example of a JavaParser Abstract Syntax Tree. 14

3.1 The weather example in Apache Flink visualised using the Flink Plan Visualizer [12]. 17
3.2 The Graphite tree structure for Apache Flink metrics. 19

5.1 The launcher window of the GUI. 25
5.2 The main window of the GUI. A query corresponding to the example in Figure 2.1 has been designed and the operators connected. 26
5.3 Text suggestions shown to the user. 27
5.4 A warning is displayed to the user if errors are detected in the designed graph. 28
5.5 The visualised Flink query of the code in Appendix A.2, from the weather example in Section 2.1.1. 29
5.6 Throughput statistics for three operators in a Liebre query. 32
5.7 Latency statistics for three operators in a Liebre query. 33
5.8 Graphite statistics, collected from Liebre, showing the mean processing latency for an operator in the last minute. 34
5.9 Using wildcards in the Graphite query. 34
5.10 Query auto completion for Graphite statistics. 35

6.1 The `GraphObject` classes. 39
6.2 Two nodes in the graph have been selected, the first labeled `[FROM]` and the second `[TO]`. 40

7.1 The FraudDetection visualisation test. 52
7.2 The YSB visualisation test. 52
7.3 The weather visualisation test. 52
7.4 The LinearRoad visualisation test. 53
7.5 The StreamingJob visualisation test. 53
7.6 The Weather query, defined in a new thread, visualised for Flink. . . 54

List of Tables

2.1 The basic operators in RegEx patterns. 15

7.1 Result summary of the visualisation tests. 56
7.2 Supported operators for visualisation and for application design. . . . 56
7.3 Counting the number of received statistic values compared to how many have been written to a CSV file. 57
7.4 Measuring the Liebre CSV statistics latency for different streaming applications. 58
7.5 Measuring the statistics latency for different Graphite queries. 59
7.6 Measuring the statistics latency for test six without the graph. 59
7.7 CPU and RAM usage for the different tests. 61

List of Listings

1 Example of a Map operator in Apache Flink that receives a String and transforms this into a custom Java class. The schema changes after this operation. . . . 8
2 Example of a Filter operator in Apache Flink that filters out weather readings that have a temperature less than or equal to 21°C or are not from Gothenburg. . . . 8
3 Example of an Aggregate operator in Apache Flink that partitions the stream by city and keeps the maximum temperature reading seen for each city per day. . . . 9
4 Example of a Reduce operator in Apache Flink that combines the current tuple value with the last reduced value. . . . 9
5 Example of a sliding window in Apache Flink that starts a new 30 second window every ten seconds aggregating the maximum temperature for each city. . . . 10
6 Setting the parallelism of an operator. This operator will run as five separate parallel instances. . . . 11
7 Comparison between Liebre and Apache Flink on how to connect operators. . . . 11
8 The path for the Graphite Render API with parameters to fetch the last 24 hours of statistics of an operator. . . . 12
9 Example of a method visitor that collects all method names in a CompilationUnit. . . . 14
10 The RegEx to match connected operators in Liebre from Listing 7. . . . 15
11 A Map operation followed by a Filter operation in Apache Flink. . . . 18
12 A Map operation followed by a Filter operation in Liebre. The operators are connected using the `connect` method. . . . 18
13 A small query in Liebre. . . . 30
14 The same query in Flink. . . . 31
15 Structure of an SPE JSON file. . . . 37
16 The JSON definition of the Filter operator in Apache Flink. . . . 38
17 Using MXBeans to fetch the framework's CPU and memory usage. . . . 40
18 The JSON structure of an exported graph. . . . 41
19 The RegEx pattern used to match public Java classes, interfaces and enums with. . . . 43
20 Generated Flink code for an operator that has not been designed correctly by the user. . . . 45

21 Generated Flink code for an operator that has been designed correctly by the user. 45
22 Nodes in the JavaParser AST. Each line goes one level higher in the tree. 47
23 Nodes in the JavaParser AST. Each line goes one level higher in the tree. 47
24 The counter starts at zero, increases for each opening parenthesis and decreases for each closing parenthesis. 47
25 The (shortened) `"links"` field in the Liebre JSON file. 48
26 An example of a Flink query with five operators. 49
27 The tail listener adapter. 50
28 Creating the query in a new thread. 54
29 How the query was split up in two methods. 55
30 The query created in the constructor of a nested class. 55
31 The query created in a nested class method. 55

1 Introduction

As a result of more connected devices and users and applications moving to the cloud, every year the amount of data needed to be processed by different computer systems increases. In 2020 over 161 exabytes of data were created and consumed each day, and this number is expected to have risen to 463 exabytes per day in 2025 [1][2]. The troubles with trying to store all this information and a growing demand for real-time processing has led to an increase in the popularity of stream processing as a way of handling large amounts of data.

In stream processing data arrives continuously from one or several sources and is processed on the fly in the memory of the machine. This is different compared to traditional batch processing where the data needs to be stored before being processed. With stream processing, data can be processed in a real-time fashion with low latency (time taken to produce a result based on one or multiple pieces of input data) and high throughput (amount of data processed per time unit) [3]. This is useful in many applications, and use-cases can range from fraud detection in online credit card transactions to processing weather data from large numbers of remote sensors. The task of processing this streaming data is taken care of by a Stream Processing Engine (SPE).

A Stream Processing Engine receives continuous streams of data from one or several sources and processes these using operators. Operators perform operations on the data. The SPE executes a streaming application (or query), which can be defined as a Directed Acyclic Graph (DAG), where the streams are edges and the operators are nodes. An example of an operator is a Filter operator. This operator can filter out data that does not match a specified criterion to only send ”valid” data to the next node in the graph. SPEs and operators are further explained in Chapter 2.

Stream processing has been researched for a long time and can be traced back to the 1960’s [4]. During this time, many different SPEs have been created and some examples used in literature are Apache Storm [5], Apache Spark [6], Apache Flink [7][8], Liebre [9][10] and Amazon Kinesis [11].

1.1 Short problem description

Even though the previously stated SPEs are all used for stream processing they are quite different in their implementations. For example, Apache Flink and Liebre have a common set of supported stream processing operators but the way they are implemented in code is different and the use of one operator in Flink may require

two operators in Liebre. This can be an issue if a streaming application needs to be migrated from one SPE to another, for instance to support more operators or to possibly get better performance. Introduced following is three sub-optimal aspects of stream processing that require more work.

Since a streaming application can be modelled as a DAG, a tool that can abstract this graph visually to the user would make understanding different streaming applications easier. For example, looking at a visualised query can make understanding the flow of data between operators easier compared to having to understand the code of the query and all the implementation details. Some SPEs have such a tool built-in (Flink, Apache Storm) while others do not (Apache Spark, Liebre) [12][13][14].

When creating a complex streaming application it is most often required to manually write all the code for it. Development platforms (IDEs) help but it can still be a big task to organise the code and get an overview of large applications. A tool allowing its user to design a DAG visually and then have the ability to get the corresponding code, even if just a skeleton, generated automatically would simplify the design process.

While designing and implementing a streaming application is important, knowing if it is performing well when executed is critical. For example, a detected bottleneck or idle operator can be resolved by modifying the streaming graph to improve the performance and reduce the memory usage of the machine running the application. Most SPEs have some sort of performance reporting, although not always graphical or live.

1.2 Goal

This thesis concerns the design and implementation of a framework in the form of a Graphical User Interface (GUI) as a first step towards a unified view of streaming applications, aiding developers with visual representations, statistics, and code generation for different SPEs by abstracting away the implementation details. As comparison, designing and implementing a streaming application currently requires you to open an IDE, implement the application by writing code interleaved with running test executions to see if it is working as intended. Once done, you either watch the statistics live or write additional scripts that analyse the log files to gather stats and plot them. With this project, the goal is to make it simpler; open the GUI, create the streaming application visually and get (most of) the code generated for you, possibly fix details in an IDE, and then watch live statistics and graphs of log files seamlessly.

There are three main functionality goals for the framework. These are:

1. Query visualisation - the framework should be able to visualise a streaming application as a DAG. To do this, first a user will have to choose a directory or project containing a streaming application, and then the framework will need to identify all operators and streams and determine how these are connected.

2. Application design and code generation - a user of the framework should be able to select which SPE to use, add the operators (nodes) supported by the

current SPE and specify how they work, i.e., what type of input they receive, how they process this data and what type of data they produce and output. The operators should then be connectable using streams (edges) to produce a DAG. If the user adds and wants to connect an operator that expects a different input type than the previous operator outputs the framework should be able to detect this invalid operation and notify the user. For example, if the added operator expects a string value as input when the previous operator outputs an integer value it is not a valid stream. Then, the framework should be able to generate the corresponding code given the designed DAG.

3. Performance reporting - After parsing and visualising a streaming application the framework needs to show performance reports about it. This can be from already executed log files or gathered directly from a running streaming application. Furthermore, it should display the reports visually in the form of time series graphs. Reports can be shown for the entire application, but it should also be supported to show statistics of a single operator. For example, a useful metric is the operator's latency and the incoming versus outgoing processing throughput. The framework should be able to display statistics of its own performance, such as CPU usage and memory requirements.

Furthermore, the framework needs to be general enough so that it can be adapted to work with several different SPEs. In other words, the SPEs should have a common interface, abstracting the SPE-specific implementation details away from the framework. In this thesis, two SPEs are added and tested with the framework to evaluate its generality. The cost of running the application and measuring statistics is another important factor. For example, the CPU and memory usage need to be kept low in order to not impact the performance of a running streaming application.

1.3 Method

The thesis begins with doing background research about available tools for designing and creating graphical user interfaces and about stream processing in general. Early on, the two SPEs to use are chosen to be Apache Flink and Liebre, and these are later added to the framework. Since Liebre is written in Java and Apache Flink supports both Java and Scala [15] Java is chosen as the programming language. Because of this, the popular GUI platform JavaFX is used together with the drag-and-drop user interface builder Scene Builder, both written in Java, to design and create the GUI in the form of a standalone desktop application.

With the GUI created, directed graphs are visualised in the interface using the JavaFXSmartGraph library. For query visualisation, the open-source library JavaParser is used to parse and extract relevant information about operators and streams in a given streaming application. Statistics are added and then displayed using the ExtJFX library. Finally, the framework is evaluated in terms of functionality, performance, and generality.

1.4 Outline

The rest of this thesis is structured as follows:

- **Chapter 2** introduces the background needed to understand the rest of this thesis.
- **Chapter 3** describes the problem to be solved in more detail.
- **Chapter 4** discusses and compares this project with related work.
- **Chapter 5** explains the design requirements and functionality of the framework.
- **Chapter 6** describes the implementation details of the framework.
- **Chapter 7** presents the outcome of the thesis project and compares this with the initial goals in terms of functionality. It also includes an evaluation of the framework in terms of performance, such as memory and CPU usage, and generality.
- **Chapter 8** concludes the thesis and includes a summary of the work done and a discussion about the results and choices made.

2
Background

This chapter introduces the theory and background needed to understand the rest of the thesis.

2.1 Stream Processing

With the advance of Internet of Things there are more connected devices than ever before [16]. Applications and systems processing high-volume data streams, such as network firewalls, smart cities and stock market trading, have a demand for real-time processing as the data is time sensitive and only valuable for a short period of time [17][18][19]. This rules out traditional batch processing as early data might be too old when a batch is processed. As a result, the focus has shifted to data stream processing, which is the focus of this section.

Stream processing is the act of using streams of data from one or several sources, processing and performing operations serially, in parallel, or both, on the data, and then producing real-time results [20]. To provide continuous analysis of huge amounts of data and get results in a real-time fashion, the data in stream processing is not persisted but rather processed on the fly as it arrives [21]. In batch processing data cannot be processed incrementally and if two similar queries are run on the same data set then the overlapping set of data will be processed twice. Like a network firewall, stream processing can also be used to monitor and detect events, for example anomalies, in real-time [22].

2.1.1 Streams, tuples and directed graphs

The basic element in stream processing is a stream - an unbounded sequence of data flowing from a *source* to a *sink*. A source can be anything producing data, such as a remote sensor reporting temperature readings, or the contents of a text file being read from a computer. On its way to the sink the data can be modified, dropped or its information extracted by an *operator*. An operator can be seen as a function or computation on each data item in the stream. Finally, the sink receives the (possibly) modified data and can, for example, save this data to a log file, send it somewhere else for further processing or give it straight to the user.

A single data item in a stream is called a *tuple*, and this is a combination of a timestamp and some data. The data can, for example, be a temperature and a humidity reading from a sensor. All tuples in a stream have the same schema but since an operator can modify a stream the schema can differ between two operators.

Tuples in a stream are processed by operators in a first-in first-out (FIFO) fashion [23].

A source can have one or more output streams but no input streams. An operator can have one or more input streams, either from a source or another operator, and one or more output streams. Finally, a sink can have one or more input streams and no output streams. The connected operators and their streams together form a Directed Acyclic Graph (DAG), see the example in Figure 2.1. All streaming applications can be modelled and visually presented as a DAG.

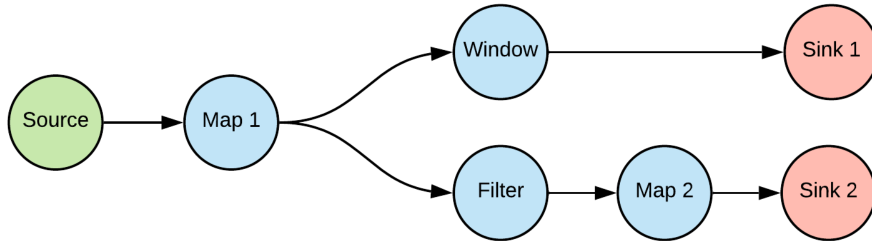


Figure 2.1: An example of a streaming application modelled as a directed acyclic graph, with one source, four operators and two sinks.

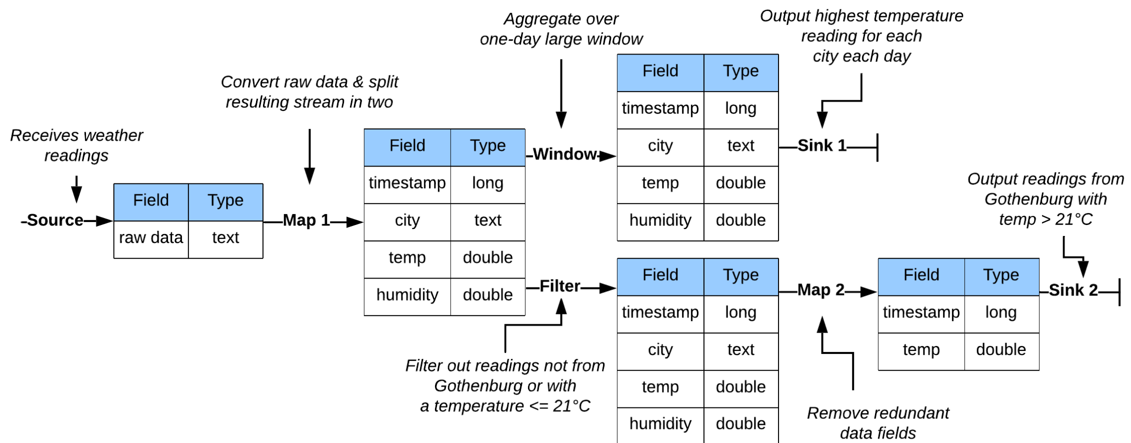


Figure 2.2: The schema for the streams in Figure 2.1.

In Figure 2.1 a source operator sends data to a Map operator (all these operators will be explained in the following section) and the stream is then split in two. In this figure it is assumed that the SPE takes care of splitting the stream. In some SPEs (Liebre) a specific operator is needed for this task. Two sinks are the last to process data in this DAG. The schema for this query is shown in Figure 2.2. This example concerns weather readings from different cities in Sweden where each reading has a timestamp indicating when it was taken, a city name, a temperature reading (in °C) and a humidity reading (0-100%). The outcome is the highest temperature reading for each city each day (Sink 1) and temperature readings above 21°C in Gothenburg (Sink 2). In this query, the source operator receives all readings as plain text strings and forwards these. The Map operator then transforms this string

into a readable data object with the different fields. The data stream is then split in two to facilitate the outcome described previously. In the first stream (top) a Window operator, as the name implies, arranges the received tuples into one day large windows, each city having its own window, and selects the reading with the highest temperature from each window. The result is one tuple for each city containing the highest recorded temperature for a specific day. This is then sent to the sink which collects these values. In the second stream (bottom) a Filter operator is applied that filters out all readings that are not from Gothenburg or have a too low temperature reading. A Map operator is then applied to remove redundant fields so that the sink only has the timestamp and the temperature reading (city is known and humidity is not used). Removing redundant fields can improve the performance of the streaming application as less data is transmitted and less memory is needed of the system processing the data. This can be important in low-performance distributed environments.

This weather example will be used throughout the thesis and its corresponding Apache Flink and Liebre Java code can be found in Appendix A.1 and Appendix A.2 respectively.

2.1.2 Stream operators

There are two groups of operators in stream processing: *stateless* and *stateful*. Stateless operators have no state, meaning that they don't keep any information about previously processed tuples, and only the current tuple is important. Stateful operators, as the name implies, maintain state. The state may be used in the computation on new incoming tuples, affecting the produced output tuple, and for each new incoming tuple the state evolves.

Stateless operators

A stateless operator processes tuples one at a time as they arrive without maintaining state. As only one tuple is processed, the operator has no information about previously seen tuples and the computations are done independently for each tuple. Because of this the operators can easily be executed in parallel to improve the performance of a streaming application. Some common stateless operators are described below.

- **Map** - the Map operator is one of the simplest in stream processing. It takes a single tuple, performs computations on it, and produces a new tuple. The produced tuple does not necessarily have the same schema as the input tuple. An example of this is the first Map operator in Figure 2.1 which receives a plain text string containing four comma-separated values; a timestamp, a city name, a temperature reading and one humidity reading, and turns this into a different schema with four fields instead of one. Listing 1 is an example of this operator in Apache Flink. Here, the custom Java class `WeatherData` represents a weather reading containing the previously described four values.

```
DataStream<WeatherData> mapStream = sourceStream.map((MapFunction<String,
    WeatherData>) value -> {
    String[] s = value.split(",");
    return new WeatherData(Long.parseLong(s[0]), s[1],
        Double.parseDouble(s[2]), Double.parseDouble(s[3]));
});
```

Listing 1: Example of a Map operator in Apache Flink that receives a String and transforms this into a custom Java class. The schema changes after this operation.

- **FlatMap** - this operator is similar to a Map operator except that it can produce zero or more output tuples, compared to exactly one in Map. An example is a sentence being split up into multiple words.
- **Filter** - the Filter operator filters out tuples in a stream based on a conditional statement. If the statement is fulfilled the tuple is forwarded without being modified, otherwise it is discarded. The operator in Listing 2 is used to only forward tuples with a temperature reading above 21°C from Gothenburg.

```
mapStream.filter((FilterFunction<WeatherData>) weatherData -> {
    return weatherData.getCity().equals("Gothenburg") &&
        weatherData.getTemp() > 21;
});
```

Listing 2: Example of a Filter operator in Apache Flink that filters out weather readings that have a temperature less than or equal to 21°C or are not from Gothenburg.

Stateful operators

Unlike stateless operators, stateful operators are allowed to keep state information. The state evolves as new tuples are processed and the current state impacts the computation on a tuple. With stateless operators two identical tuples will produce identical results while with stateful operators this might not be the case. State is maintained over a set period of time or for a fixed number of tuples. This is known as a *Window*, and is further explained in Section 2.1.3. The following list presents some frequent stateful operators.

- **Aggregate (sum, max, min, average, count)** - an aggregate operator groups together several tuples to produce a single output tuple [24]. For instance, this can be to keep track on the highest value seen, to get the average of all values or to count the number of tuples received by the operator. Listing 3 shows the Window operator from Figure 2.1 where a stream is partitioned by the city name of the measurement and then the maximum temperature reading at each city is aggregated each day.
- **Reduce** - an operator that works on keyed streams and *reduces* tuples. Specifically, it turns two tuples into one new tuple. In Listing 4 the operator takes

```
mapStream.keyBy((KeySelector<WeatherData, String>) WeatherData::getCity)
    .window(TumblingProcessingTimeWindows.of(Time.days(1)))
    .max("temp");
```

Listing 3: Example of an Aggregate operator in Apache Flink that partitions the stream by city and keeps the maximum temperature reading seen for each city per day.

as input two values, one new and the other is the previously reduced value, combines these and outputs a new value. It is a stateful operator since it keeps track on the rolling reduced value.

```
keyedStream.reduce((ReduceFunction<Integer>) (value1, value2) -> {
    return value1 + value2;
});
```

Listing 4: Example of a Reduce operator in Apache Flink that combines the current tuple value with the last reduced value.

2.1.3 Windows in stateful operations

When an operator can keep track of its state it is desirable to perform computations over periods of time, for example computing the average temperature in the last three days as in Listing 3. This is possible with the use of windows and stateful operators. All windows have a fixed *window size* specified as a unit of time, like 30 seconds, (time based) or a tuple count, for example 100 tuples (tuple based). Once a window is closed, either after a period or after receiving the specified number of tuples, it is possible to perform calculations on it. Only then will the result be forwarded to the next operator in the DAG. Two common window types are *tumbling windows* and *sliding windows*.

Tumbling windows are fixed-size non-overlapping windows. Defining a window with a size of five minutes will start a new window every five minutes. Tuples arriving during an active window belong to that window. An example in Apache Flink can be seen in Listing 3.

Sliding windows are fixed-size overlapping windows with a *window advance*. The window advance is specified as a period of time and specifies how often a new window is created. A window with size 30 seconds and advance ten seconds will start a new window every ten seconds lasting for 30 seconds, see Listing 5 for an example. Arriving tuples are assigned to all active overlapping windows.

2.1.4 Statistics in Stream Processing

Real-time processing in streaming applications is essential for many applications [18] and having statistics of the running query is beneficial. For example, downtime in a system for real-time accident monitoring and detection using data stream processing [25] is not desired and monitoring the system status, such as network bandwidth and

```java
sourceStream.keyBy("city")
    .window(SlidingProcessingTimeWindows.of(Time.seconds(30),
    ↪  Time.seconds(10)))
    .max("temp");
```

Listing 5: Example of a sliding window in Apache Flink that starts a new 30 second window every ten seconds aggregating the maximum temperature for each city.

CPU usage, can mitigate this risk. Two of the most common statistics are *latency* and *throughput*.

Latency (or processing latency) is defined as the time it takes for an operator to produce a result (a new tuple) from the moment it received the last contributing tuple [26]. This performance metric can be both for a single operator or for the whole query. In the last case it is the time from that the source receives the last contributing tuple to the query producing a result with this tuple. Latency is usually measured and presented as the average latency over a window but in some cases this can be deceiving [27]. For instance, while the average latency can be low, say 1 millisecond, there might be cases where it takes exponentially longer to produce a result. If these cases are rare the average value is barely affected. But for an external application, such as a trading or intrusion detection system, relying on receiving real-time data from the output of a streaming application, this might have severe consequences [27].

Real-time applications relying on a low latency constraint can have a low average latency while the *tail latency* is significantly higher. Tail latency can be measured as the 99 or 99.9 percentiles of long latency values, in other words the percentiles with the highest latencies. For example, a web server serving incoming user requests may handle 99% of all requests with low latency while the last 1% of requests may take longer to complete. Reporting latency in this form is important as even a few hundred milliseconds of extra latency can impact a web site's page views and revenue [28].

Throughput (or processing throughput) is the number of tuples that a query consumes per unit of time [26]. For example, this can be measured as tuples per second (`t/s`) or tuples per minute (`t/m`). Like latency this can be measured as the average throughput over a window.

Two other metrics are **CPU utilisation**, the percentage of the total available processing power that the query utilises per unit of time, and **memory consumption**, the amount of Random Access Memory (RAM) the query uses per unit of time.

2.1.5 Stream Processing Engines

Once a streaming application has been designed it needs to be implemented and there are many SPEs to choose from [29]. The two that were chosen for this project are Apache Flink and Liebre.

Apache Flink started out as a fork of the Stratosphere research project and became a top-level Apache project in 2015 [30][31] and has since become a popu-

lar high-throughput, low-latency SPE used in many business systems and research projects [32][33][34]. It is a leading open-source project having both batch and stream processing capabilities and is written in Java and Scala [35]. It supports distributed environments and scales to any size while maintaining exactly-once guarantees [36]. In Flink it is possible to set the parallelism of a single operator or for the whole environment, meaning that it can run as several separate instances either on a single or multiple machines. Each instance processes a subset of the incoming stream tuples [37]. In Listing 6 an aggregate operator has been set to run as five parallel instances.

```
1  sourceStream.keyBy("city")
2      .window(SlidingProcessingTimeWindows.of(Time.seconds(30),
       ↪  Time.seconds(10)))
3      .max("temp")
4      .setParallelism(5);
```

Listing 6: Setting the parallelism of an operator. This operator will run as five separate parallel instances.

Unlike Flink, **Liebre** is a minimal SPE able to run on single-board devices with a much smaller footprint [38]. It was developed in 2017 as a research tool and has support for a basic set of operators, both stateless and stateful, and streams, while more advanced functionality, such as parallelism and elastic scaling for operators, is lacking.

As for most SPEs the implementation details differ. For example, connecting two operators is in Liebre done with a `connect()` method call while in Flink it is done with chaining. In Listing 7 the first line is how to connect three operators in Liebre and the second line is how to do it in Flink.

```
1  query.connect(source1, map1).connect(map1, filter1);
2  source1.map(...).filter(...);
```

Listing 7: Comparison between Liebre and Apache Flink on how to connect operators.

2.2 Statistics with Graphite

Graphite is an open-source monitoring and time-series data graphing tool that operates as a web server [39]. Statistics sent from a query can be viewed as graphs (line charts) in its web interface, see Figure 2.3. These charts are rendered as images and lack the functionality to inspect individual data points.

The following two sections introduce its API and data structure; two components used by the framework created in this thesis.

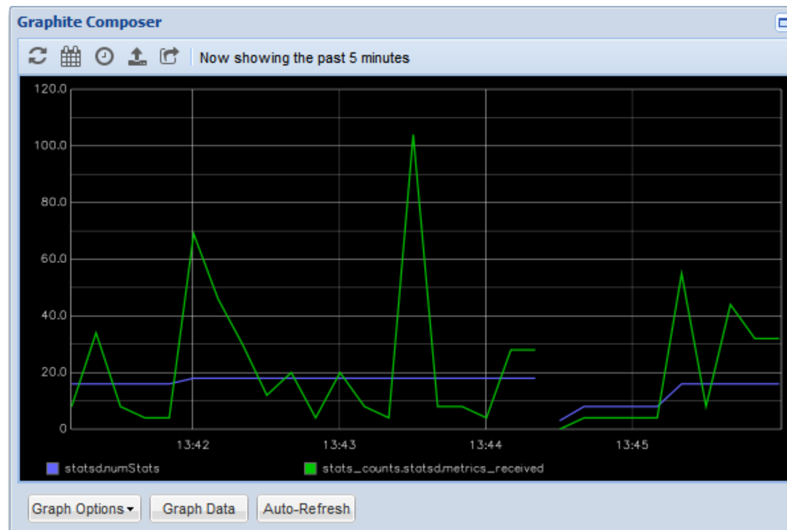


Figure 2.3: A line chart from Graphite's web interface.

2.2.1 Query structure

Graphite stores and provides statistics in a tree structure, see Figure 2.4.

The SPE decides where and how the structure looks like. Liebre statistics are stored in the second level `liebre` folder, containing sub-folders for each query. The query-level folders contain a folder for each operator and each stream. These folders in turn contain the statistics for each operator.

Querying for a statistic requires the complete path to it. A query in Graphite is a dot-separated string. In the example figure, the path to the *count* statistic is `liebre.testing.map1.EXEC.count`. Wildcards are also supported. `liebre.testing.*.EXEC.count` queries the *count* statistic for all operators.

2.2.2 The API

The web server offers an API, allowing for statistics to be retrieved. Two endpoints are of interest. The *Render URL API*, located at `/render`, provides raw data and statistics. The *Metrics API*, located at `/metrics`, provides information about the tree structure. For example, fetching the last 24 hours of the *map1* operator's *count* statistic the Render URL endpoint is used together with parameters specifying the query details as can be seen in Listing 8.

```
/render?format=json&target=liebre.testing.map1.EXEC.count&from=-24h
```

Listing 8: The path for the Graphite Render API with parameters to fetch the last 24 hours of statistics of an operator.

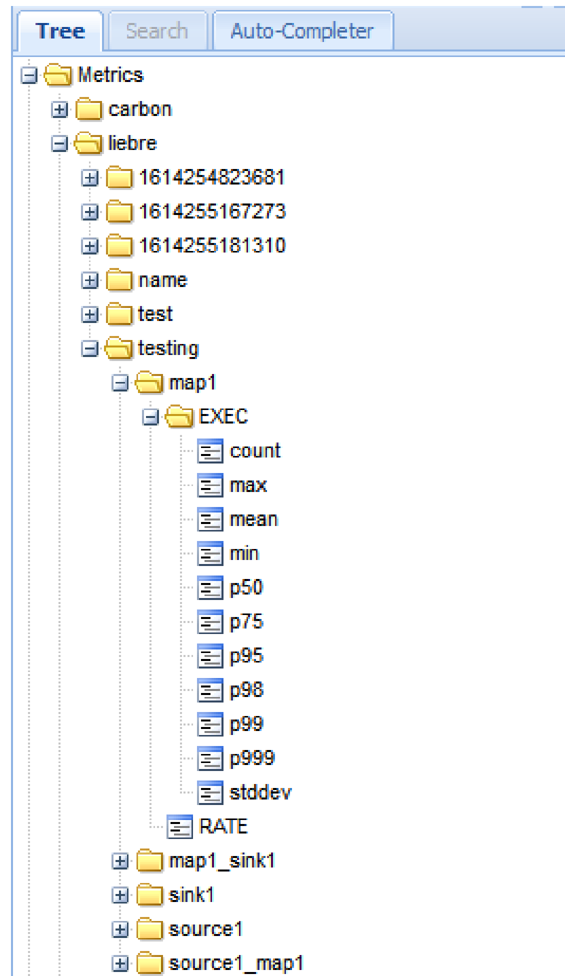


Figure 2.4: The Graphite tree structure for Liebre metrics.

2.3 Parsing Java code

To visualise DAGs of a streaming application the framework needs to parse Java code to understand what the application is doing. To aid with this task two tools were used; *JavaParser* and *Regular Expressions* (RegEx).

2.3.1 JavaParser

JavaParser is an open source Java library that is used to *parse* (turn plain Java code into a Java object representation), *analyse* (find all local variables in a method), *transform* (rename fields and variables, and change class and method access modifiers) and *generate* Java code [40]. In this thesis only the parse and analyse features are used. JavaParser transforms the contents of a Java file into an Abstract Syntax Tree (AST) [41], see Figure 2.5 for an example. Here, the `CompilationUnit` is the root of the parsed file. The root contains all child nodes, such as package declarations, import statements, top-level classes, and so on. Each of these child nodes can have further child nodes with their own data. *MethodDecl* contains everything in that method, for example the method parameters, local variables and

method calls. The tree can be acquired in Java with the line `CompilationUnit cu = StaticJavaParser.parse(new File(FILE_PATH));`.

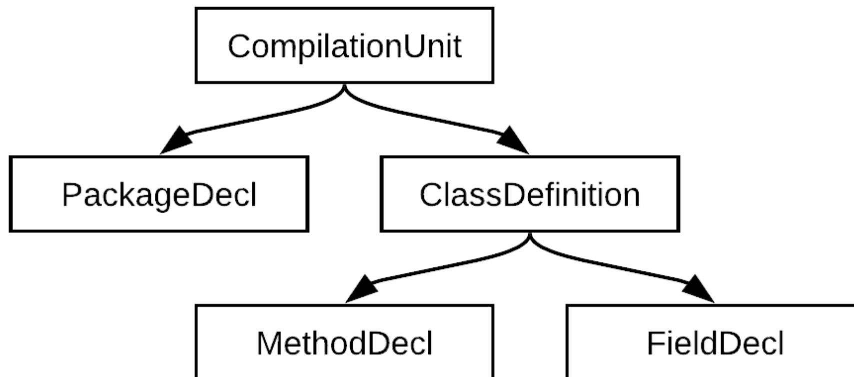


Figure 2.5: An example of a JavaParser Abstract Syntax Tree.

In JavaParser, a *visitor* is a method that traverses the CompilationUnit and searches for specific things. Consider Listing 9. Here a class `MethodVisitor` is created that extends `VoidVisitorAdapter`; a class that returns no value but has side effects - in this case adds strings to a list. Extending this class allows for overriding its over 90 different methods, such as visiting declared fields, variables, methods, method calls, parameters, comments, interfaces, and so on [41]. In this example the visitor visits and finds all declared methods and adds the method names to the `methods` list.

```
1        ...
2        List<String> methods = new LinkedList<>();
3        new MethodVisitor().visit(cu, methods)
4        methods.forEach(m -> System.out.println("Found method: " + m));
5    }
6
7    private static class MethodVisitor extends VoidVisitorAdapter<List<String>> {
8        @Override
9        public void visit(MethodDeclaration md, List<String> methods) {
10           super.visit(md, collector);
11           methods.add(md.getNameAsString());
12       }
13   }
```

Listing 9: Example of a method visitor that collects all method names in a CompilationUnit.

2.3.2 Matching Java code with Regular Expressions

Regular Expressions (RegEx) are used for pattern matching in strings. In Java, RegEx is available in the standard Java SE platform. Table 2.1 shows some basic RegEx operators used in this thesis.

Table 2.1: The basic operators in RegEx patterns.

Operator	Description
a* a+ a?	0 or more, 1 or more, 0 or 1
ab|cd	match either ab or cd
(abc)	Capture group
.	matches any character except newline
\w \d \s	matches a word, digit or whitespace/newline
\(	escapes a RegEx character, matches "("

Consider the Java code in Listing 7, specifically the first line. By using the RegEx expression in Listing 10, it is possible to find which operators are connected.

```
1   .connect\((\w+),\s*(\w+)\)
```

Listing 10: The RegEx to match connected operators in Liebre from Listing 7.

The first part, `.connect\(`, matches the plain text ".connect(". The character `(` is a special character in RegEx and needs to be escaped. Next, `(\w)+` matches at least one word and the word is enclosed in a capture group. Using a capture group allows the retrieval of a specific part of the matched string, in this example the variable name of the connected operator. Then, a comma is matched followed by zero or more whitespaces and then another capture group to extract the second connected operator. Finally, a closing parenthesis is matched.

3

Problem description

With the relevant background introduced, this section provides a more in-depth explanation on the problem which is to be solved. Section 3.1 discusses the limitations of current SPEs, what they lack and what is required, and Section 3.2 gives an outline on how the resulting framework will be evaluated and tested.

3.1 Shortcomings of current Stream Processing Engines

Lack of graph visualisation tools and (mostly) manual coding

A large streaming application can be difficult to understand at first glance. Graph visualisation tools can be helpful but not all SPEs have such a feature. Liebre does not, but for Apache Flink the Flink Plan Visualizer exists [12]. This tool, available online, visualises a JSON object representing a Flink query (produced with the line `env.getExecutionPlan()` in Java). In Figure 3.1 the weather example has been visualised using this tool.

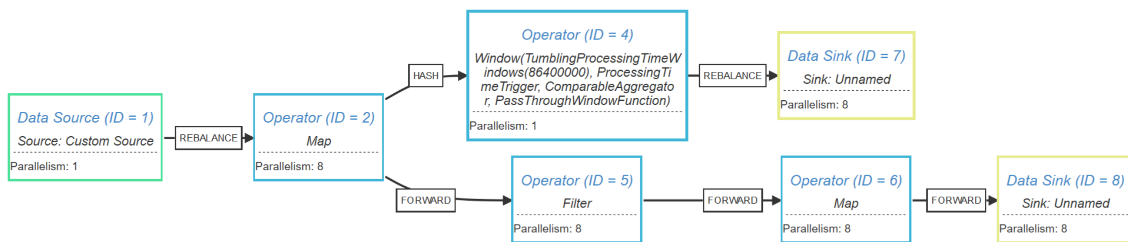


Figure 3.1: The weather example in Apache Flink visualised using the Flink Plan Visualizer [12].

Appendix A.1 contains the corresponding Flink streaming application in Java code. Compared to this code, in the figure it is easier to see the application DAG and its structure. Appendix A.2 contains the Liebre code for the same query.

One drawback with the Flink Plan Visualizer is that it only works in one direction, that is it is not possible to modify the visualised graph and get the Java code for the new graph generated. Both Liebre and Flink require manual coding as they do not have tools to automate this task.

Creating a streaming application generally requires writing the Java code for it. Flink additionally supports the use of SQL to create streaming queries [42].

This extends the user base from just programmers to anyone with general SQL knowledge. However, SQL queries are limited to simpler queries and it is often not possible to create the complex queries achievable by pure programming [43]. Therefore, this project focuses on the programmatic API and the Flink SQL API will not be supported.

Implementation differences

Even though most SPEs support a common set of operators how they are implemented in code differs. As an example, consider the Java code in Listing 11 where a Map operator is followed by a Filter operator in Apache Flink. Compare this with the same streaming application implemented in Liebre, seen in Listing 12. As can be seen, Apache Flink uses method *chaining* to connect the two operators. Liebre on the other hand does not use this and new operators must be defined separately and connected using a specific `connect(srcOp, dstOp)` method. Another thing to note is that in Flink the main idea revolves around streams while in Liebre it is operators. Separating the connection of operators from their declaration can make the streaming application easier to understand but it has the drawback of requiring slightly more code to write. Nevertheless, moving a streaming application from one SPE to another will require some work as each is unique and there exists no standard design or model for how an SPE can be implemented or designed.

```
1  DataStream<Double> stream = intStream
2      .map((MapFunction<Integer, Double>) value -> value * Math.PI)
3      .filter((FilterFunction<Double>) value -> value > 2);
```

Listing 11: A Map operation followed by a Filter operation in Apache Flink.

```
1  Operator<Integer, Double> mOp = query.addMapOperator("map1", value -> value *
   ↪  Math.PI);
2  Operator<Double, Double> fOp = query.addFilterOperator("filter1", value ->
   ↪  value > 2.0);
3  query.connect(mOp, fOp);
```

Listing 12: A Map operation followed by a Filter operation in Liebre. The operators are connected using the `connect` method.

Differences in performance reporting

As with the implementation differences, the reporting of statistics and metrics differ between SPEs. For example, Liebre can report statistics to CSV files, plain text files and to Graphite. Apache Flink supports Graphite in addition to several other formats [44]. The problem lies in that there is no established common set of statistics and one SPE may report statistics under a different name.

For example, comparing the Graphite tree structure for a Liebre application, as shown in the previous chapter in Figure 2.4, with a Flink application's, seen in

Figure 2.4, shows that Flink has a totally different structure. As such, using the same Graphite query for both SPEs is not possible. Here, Flink collects statistics under a *taskmanager* folder. Below is the job ID, job name and then operators belonging to the job. Only a sink operator is shown here and the folders named 0 and 1 are statistics for each parallel operator.

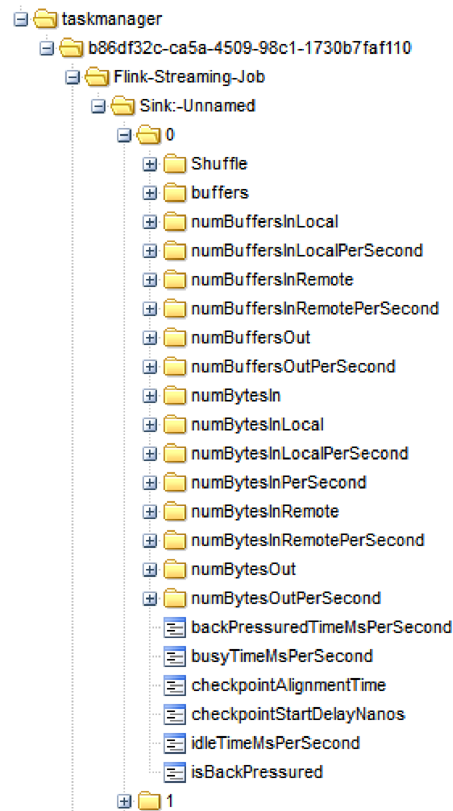


Figure 3.2: The Graphite tree structure for Apache Flink metrics.

3.2 Evaluating the framework

To assess the validity of the results and to be certain that the framework behaves as intended it is evaluated, both to measure how well it solves the problems listed above but also in terms of performance and the cost of using it. The four evaluation points are:

1. **Visualisation** - Since code can be written in many ways and formats it is necessary to test the framework to make sure that it reliably finds and visualises queries correctly. For example, does it find all operators and their stream relations in a given streaming application and visualises them correctly? The evaluation is carried out on several different applications, ranging from small and simple to larger and more complex ones.

2. **Application design and code generation** - For application design it is evaluated if the framework supports the functionality of stream processing,

such as using sources, sinks, operators and streams, and if it is possible to design a streaming application by adding and connecting different operators to create a directed acyclic graph.

As it is not certain that generated code can be executable without user modification, the framework is evaluated in terms of how *correct* the generated code is. For example, if the user writes a closing parenthesis in the input field of an operator the syntax of the generated code is incorrect.

3. **Statistics** - Statistics are evaluated on how reliable they are (for example, if the framework receives, processes and displays the information correctly and not misses tuples leading to incorrect statistics) and the delay, that is, how long time it takes from that the streaming application produces data to the GUI displaying the statistics for it.

4. **Performance** - The performance is measured and evaluated to know the *cost* of the application, in terms of how much CPU and memory usage is used when fetching and displaying statistics for operators in a running streaming application.

The framework is also evaluated regarding its generality, that is how much work is needed to support a new SPE.

The evaluations can be found in Section 7.

4
Related Work

This chapter discusses and compares previous work and research done in the three main areas of this thesis.

Apache Flink has a visualisation tool for streaming applications [12]. It displays a directed graph given an execution plan, generated by calling `env.getExecutionPlan()` (`env` is the stream execution environment, that is where the code is executed) in the Java code of a Flink streaming application. The code for this is open source [35] and resembles the work done for the query visualisation goal in this thesis. Although the Flink visualisation tool works well for Flink it is not easy to adapt it to work with other SPEs as it is integrated into the Flink streaming library and cannot run without it. This thesis aims to provide a common query visualisation interface that can be used by many different SPEs, and if Flink is using its own implementation it is not ideal. The Flink Plan Visualizer also simplifies the visualised query as can be seen in Figure 3.1 where the query from Figure 2.1 has been visualised using this tool. What it names *Operator* is the Window which consists of three steps, namely `keyBy()`, `window()` and `max()`. These three steps have been merged into one operator even though all of them are listed as separate operators in the Flink documentation [45]. This thesis aims to visualise all steps as it can make it easier for the user to understand the query. For example, if the `max()` step is not visible the user has no idea what actions are performed on the window.

SPADE is a declarative SPE, programming language and compiler, first presented in [46] in 2008. The SPE relies on System S, a data stream processing middle-ware from IBM Research [46]. Acting as a front-end to this system, SPADE provides rapid application development by having its own language and compiler. The programming language is used to create data flow graphs, using a set of standard streaming operators, that the compiler transforms into System S applications. The compiler acts as a code generation tool outputting optimised code, both in terms of performance and scalability but also security and scheduling, so that the programmer does not have to worry about implementation or performance details when creating streaming applications [47]. Rapid development is made possible with the language's stream-centric programming model, where a block diagram containing lists of data flows (streams) is automatically turned into the application skeleton. However, only a skeleton is created, and the rest of the application still needs to be created by coding in the SPADE language and there seems to be no tool that lets you create or view the graph visually. While SPADE does additional optimisations for the generated code, this project focuses on generating code "as designed" and no effort is made to improve it beyond that. Instead, the user has full ability to modify the code before it is generated.

In [48], the authors present a visual debugging system for SPADE streaming applications allowing a user to visualise queries, inspect data streams and, more importantly, visually trace how a tuple flows from a source to a sink. This system is mainly targeted for debugging purposes such as finding out how a tuple is routed or what processing is done to it at each operator in the query. Tracing tuples and inspecting live streaming applications is achieved by the debugging system being used as a plugin in the Eclipse IDE [48], unlike this project where the framework is a standalone application and does not have direct access to the streaming application that an IDE plugin offers. Comparing the query visualisation, the framework in this project offers the user to inspect all the details about an operator in a query while the debugging application just displays the operators and streams and focuses on giving the user information about the data in the streams, such as the tuple contents.

In [49] the author presents StreamCloud, an SPE based on the Borealis SPE. It includes a Visual Integrated Development Envrionment (IDE) for designing streaming applications visually. With this IDE a user can add operators, define their use, connect them to form a directed graph and then get the execution script generated automatically. A query in the visual editor is based on XML files. Comparatively, this thesis uses JSON files as they are easier to read, requires less characters to state the same thing, is faster and does not use as much memory [50]. StreamCloud's visual editor is limited to specifically work with StreamCloud while the framework in this thesis aims to be general and work for multiple SPEs.

While the previously discussed tools mostly target a single SPE some try to to be more general. Apache Beam is a tool that abstracts away the implementation details of different SPEs and provides a single interface to them all [51]. Beam is a tool that allows a user to design and code a ”pipeline” (the structure of a streaming application), for example add and define operators and an execution plan (code) is then generated and executed by one of the supported SPEs, including Apache Flink and Apache Spark. As such, an application created using Beam can easily be moved between the different SPEs while the user only has to learn a single interface. Sadly, research shows that Beam comes with a cost. In [52] the authors found that using Beam to create and run a streaming application heavily impacts the run-time performance when executing the query. In average they noticed a slow-down of at least three times compared to creating and executing the same query directly in the SPE. Similar to Beam, this project's framework is also an abstraction layer to different SPEs with the difference that it does not run the code but only generates it to let the SPE execute it. The generated code is directly targeting an SPE and does not come with the negative performance impact of Beam.

The authors of [53] present Reactive Vega, a system built on the declarative language Vega, that constructs a visual data-flow graph (similar to a graph of operators and streams in data streaming) from a declarative specification and displays graphs of the incoming data. A user of the system can click on the different nodes in the graph to display data flowing through that point as a time-series line chart, scatter plot or bar charts. While not targeting SPEs specifically, this system has similar functionality to the framework in the sense that it lets the user see metrics and statistics of streaming data and correlating this to the visualised graph. A feature not in the framework is the ability to dynamically modify the data-flow graph based

on the observed streaming tuples, such as adding and removing nodes and edges. This is possible as the system investigates the contents of each tuple while the framework does not receive tuples but only statistics data from the SPEs and as such does not have any information about what data is sent in the tuples.

5

Design

This chapter presents the design and functionality of the framework. The chapter starts with the design, layout, and vital user functionality of the GUI in Section 5.1. Section 5.2 presents how code is generated from a DAG of operators. Section 5.3 concerns how a streaming application in Java is parsed and visualised. Lastly, Section 5.4 presents the design of the graph and operator statistics.

The implementation of this design can be found in Chapter 6.

5.1 The GUI elements

The GUI design focuses on simplicity to make it uncomplicated for the user and consists of three main windows. The first is a launcher window where a user can select which SPE to use. The second is the main window where graphs can be designed and visualised and most of the functionality lies. The third and last is the metrics window, letting a user see statistics for visualised queries. These first two windows are shown in Figure 5.1 and Figure 5.2 while the metrics window is presented later in Section 5.4.

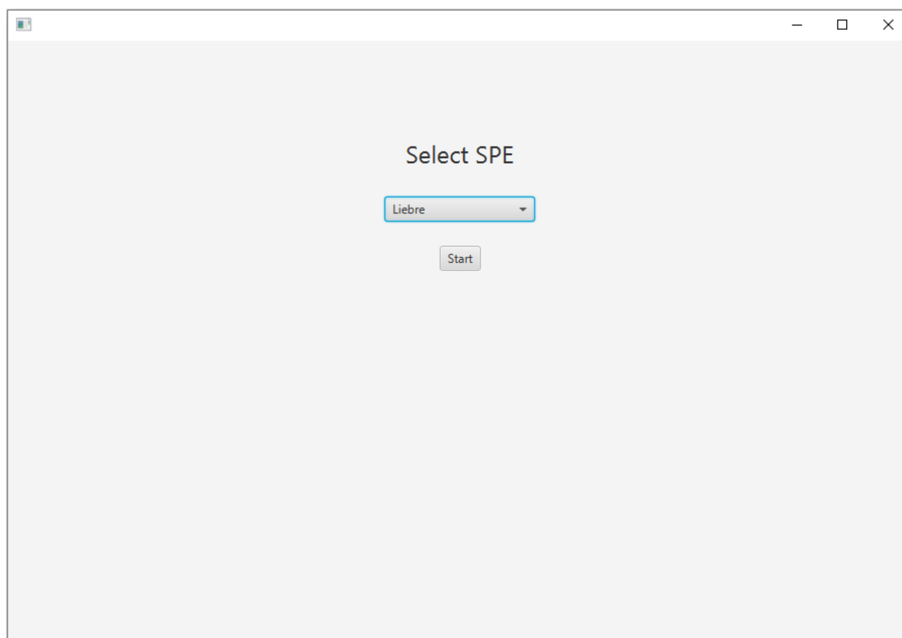


Figure 5.1: The launcher window of the GUI.

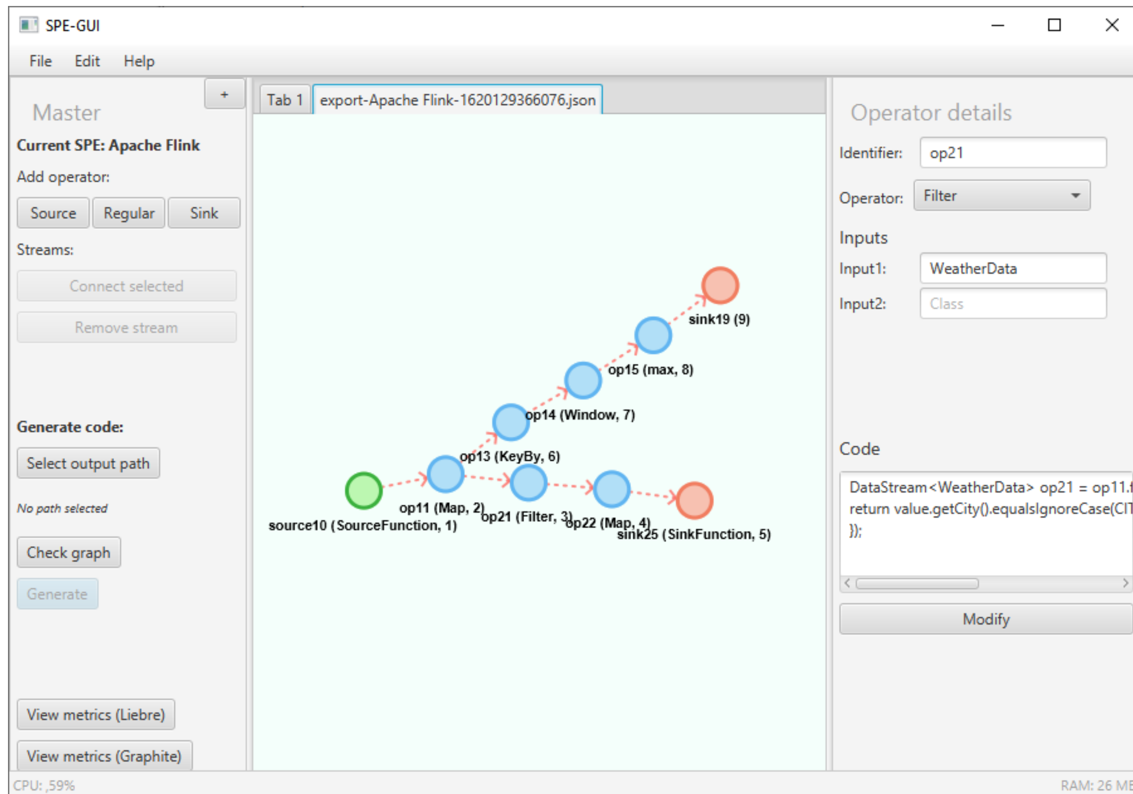


Figure 5.2: The main window of the GUI. A query corresponding to the example in Figure 2.1 has been designed and the operators connected.

5.1.1 Layout and graph design

The main window, which is displayed after an SPE has been selected in the launcher window, consists of three views; one *Master* view (left), one *graph* view (center) and one *Operator details* view (right). There is also a top menu with additional functionality.

The graph view contains the designed and visualised DAGs. Orange arrows dictate streams and their directions, and the nodes are coloured differently based on being a source, sink or regular operator. Beneath each node is a string displaying the operator's identifier, what operator it is and a unique ID.

Details about an operator is displayed in the right view. A user can here modify the operator's identifier, set the operator from a list of available operators, specify what kind of input and output data it receives and produces, and modify its Java code. In Figure 5.2 the Filter operator *op21* receives a stream of `WeatherData` tuples.

The functionalities to add nodes to and connect operators in the graph are provided by the master view. For instance, clicking on the *Sink* button adds a new node, coloured red, to the graph view. Connecting two operators is performed by double clicking them in the graph, first the one producing the stream and then the one receiving it, and clicking on the *Connect selected* button. Further down are buttons for code generation, where the user first must select an output path, viewing metrics and analysing the designed graph for user errors. The latter is further explained in

Section 5.1.4. These functionalities will be explained later in this chapter. At the top right of the Master view the user can click the plus button to create a new tab in the graph view. Thus making it possible to design and visualise several queries simultaneously. The buttons affect the currently selected tab only.

Additionally, displayed at the bottom of the main window are the CPU and RAM usages of the framework (the Java process). These values are updated once every second and allows the user to get a hint on the cost of running the framework.

5.1.2 Exporting and importing graphs

To prevent work from being lost, a vital part when designing queries is the ability to save and restore progress. From the top menu of the main window, a user can both export and import graphs. Exporting creates a new file containing the DAG from the current tab. This file can then be backed up or moved to other machines. An exported file can also be imported again to restore the graph and all operators' details.

5.1.3 Text suggestions for user inputs

As a user with only basic knowledge of the supported SPEs should be able to use the framework, the GUI gives the user hints when modifying operators. For example, when the user changes an operator's input and output types a list of suggested entries is displayed as a drop-down list, letting the user choose and know what is available. This is shown in Figure 5.3. The entries are both common Java types and types specific to the selected SPE.

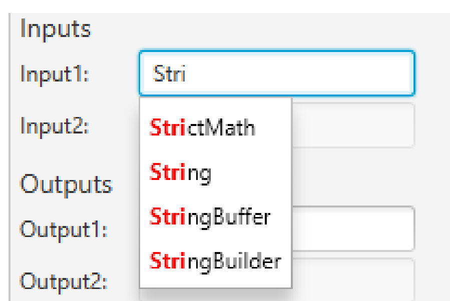


Figure 5.3: Text suggestions shown to the user.

5.1.4 Graph and type analysis

As user mistakes is a common cause of errors in computer programs, the framework provides both automatic and user-initiated error checks when designing graphs. The automatic prevents the user from creating cyclic graphs and adding streams originating from a sink or ending at a source operator. Kahn's algorithm for topological sorting [54] is used to detect cycles. Its pseudo code can be seen in Algorithm 1. A graph is acyclic if and only if it can be sorted in a topological order.

A user can run additional checks with the *Check graph* button. When clicked, an analysis is performed on the designed DAG and any found errors are reported in a

popup window, see Figure 5.4 for an example. Errors include non-unique operator identifiers and operators receiving an unexpected type of data from an incoming stream.

Algorithm 1 Kahn's algorithm for topological sorting

Require: $L \Leftarrow$ Empty list that will contain the sorted elements
Require: $S \Leftarrow$ Set of all nodes with no incoming edge
Require: $G \Leftarrow$ The graph

1: **while** S is not empty **do**
2: $\quad n \Leftarrow$ remove node from S
3: $\quad L \Leftarrow n \cup L$
4: $\quad$ **for each** node m with an edge e from n to m **do**
5: $\quad\quad$ remove edge e from G
6: $\quad\quad$ **if** m has no other incoming edges **then**
7: $\quad\quad\quad S \Leftarrow m \cup S$
8: $\quad\quad$ **end if**
9: $\quad$ **end for**
10: **end while**
11: **if** G has edges **then**
12: $\quad$ **return** Error
13: **else**
14: $\quad$ **return** L $\qquad \triangleright$ a topologically sorted order
15: **end if**

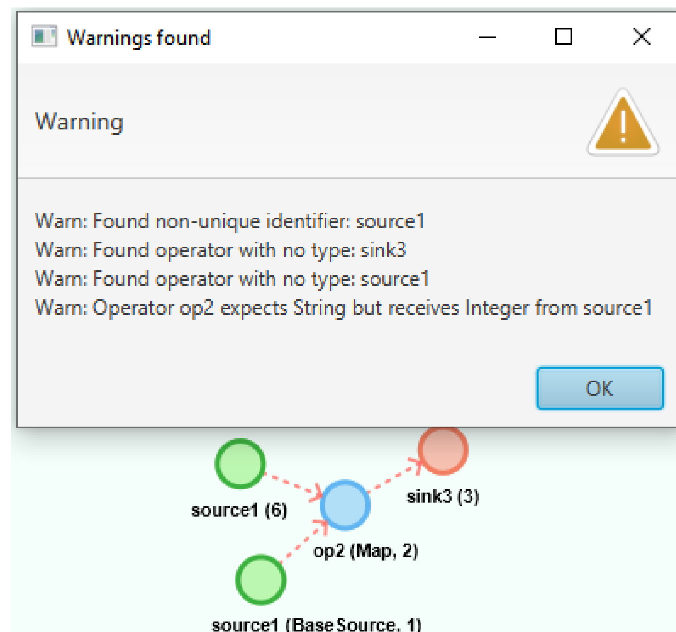


Figure 5.4: A warning is displayed to the user if errors are detected in the designed graph.

5.2 From a graph to generated code

The process of generating code for a designed DAG begins when the user clicks on the *Generate* button in the GUI.

First, the user is prompted to select a directory in which a new `.java` file is then created. Second, depending on which SPE is used, either a Flink or a Liebre query is written to the file. Third, the DAG is traversed, and code for each operator is added to the query. Lastly, for Liebre, operators need to be linked with a `connect()` method call (see line three in Listing 12) and the framework adds this to the query. For Flink an added operator is connected to its predecessor by a chained method call (see Listing 11) and does not need extra code. The Java file then contains the query corresponding to the designed DAG.

5.3 Visualising streaming applications

Figure 5.5 shows a visualised Liebre query, namely the weather example from Section 2.1.1, after the operators and streams have been found in the Java file. When an operator in the graph is clicked, the *Operator details* view displays its information. The information includes what file and class it was found in, the type of operator and the operator's Java code.

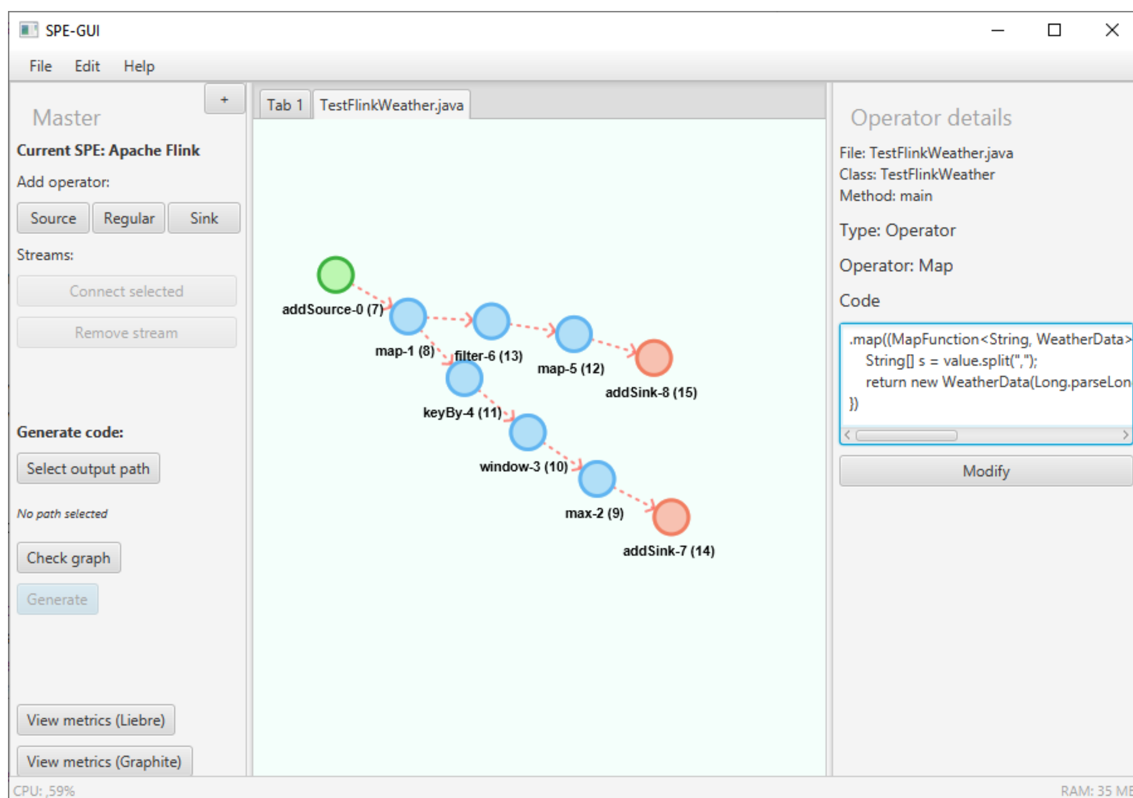


Figure 5.5: The visualised Flink query of the code in Appendix A.2, from the weather example in Section 2.1.1.

Visualising a query begins with the user clicking on the menu item *Visualise from*

file. This opens a dialog, letting the user choose a `.java` file. The framework then begins the process of parsing said file to extract the DAG. As the SPEs differ in their implementations, analysing a Java file to find the streaming query requires a great deal of "SPE-specific" code. But both Liebre and Flink share a common pattern, consisting of three steps:

1. **Finding the query variables** - In this step the framework searches for the line of code and the Java variable that defines the start of a query. For Liebre it is in the form `Query query = new Query();`, where `query` is the variable. In Flink it can look like `StreamExecutionEnvironment query = StreamExecutionEnvironment.getExecutionEnvironment();`. By finding this variable the starting point of the query is known. This information is needed to find the operators affiliated with the query.

2. **Finding the operators and their definitions** - In this step the query operators and their definitions (input and output data types and the function) are found. The framework has knowledge of how the operators are named and created in Java. For example, in Flink a Map operator is created with a `query.map()` method call (see Listing 11) while in Liebre it is instead created with a `query.addMapOperator()` method call (see Listing 12).

   Following explains how the framework finds this information, first for Liebre and then for Flink.

   Liebre - Consider the small Liebre query in Listing 13, consisting of two connected operators. From the previous step the query variable is known. All operators in Flink are created with a method call using the `query` variable. For instance, when searching through the file the framework will find the method call `query.addBaseSource(...);` and then knows that a BaseSource operator is created. In the same way, it is found that the operator is saved in the `src` variable and perform a function on decimal values.

   Flink - Listing 14 contains the corresponding Flink query. In this code the `map` method is chained after the `addSource` method. In addition to searching for method calls using the query variable, the framework searches for chained method calls.

```
1  Query query = new Query();
2  Source<Double> src = query.addBaseSource("mySource", () -> {
3      return Math.random() * 100;
4  });
5  Operator<Double, Double> mOp = query.addMapOperator("myMap", integer ->
   ↪  integer * Math.PI);
6  query.connect(src, mOp);
```

Listing 13: A small query in Liebre.

3. **Finding how the operators are connected** - Once the operators have been identified the last step targets how they are connected.

```
StreamExecutionEnvironment query =
    StreamExecutionEnvironment.getExecutionEnvironment();
SingleOutputStreamOperator<Double> stream = query.addSource(new
    SourceFunction<Double>() {
    @Override
    public void run(SourceContext<Double> ctx) throws Exception {
        ctx.collect(Math.random() * 100);
    }
    @Override
    public void cancel() {}
}).map((MapFunction<Double, Double>) value -> value * Math.PI);
```

Listing 14: The same query in Flink.

With Liebre, a `query.connect(op1, op2)` method call connects the two operators `op1` and `op2`, with a stream from the first to the second. Only operators found in such a method call are relevant to the query and the visualisation, even though other operators may exist in the code.

With Flink, the framework instead has to search among all method calls. As an example, in Listing 14 it can find that the Source operator sends a stream of data to the Map operator. Operators created with the query variable are root nodes in the DAG. In this code, the Source operator is a root. The Map operator is here saved in the `stream` variable. Any subsequent operators created with this variable are children of the Map in the DAG. For instance, the Filter operator in `stream.filter(...)` receives streams of data from it.

The result of these steps is a DAG of the found operators in the query. This DAG is then used to create all nodes and edges (or operators and streams) in the graph view, allowing the user to inspect it and the operators' details.

5.4 Graph and operator statistics

A user can view statistics about a designed or visualised query to evaluate its performance. The main GUI window has two buttons for this task: one for viewing Liebre statistics and the other for Graphite statistics.

5.4.1 Statistics from Liebre

With the correct configuration, a Liebre query reports statistics to CSV files. The metrics consist of the throughput and latency for each operator and the throughput for each stream. Liebre names the files as follows (**X** and **Y** are operator identifiers):

- Stream input throughput in **X__Y.IN.csv** - `[t,count]` - each line contains a timestamp and an integer, counting the rate at which tuples are added to the stream by operator **X**.

- Stream output throughput in **X_Y.OUT.csv** - `[t,count]` - each line contains a timestamp and an integer, counting the rate at which tuples are removed from the stream by operator **Y**.

- Operator throughput in **X.RATE.csv** - `[t,value]` - each line contains a timestamp and the operator's throughput.

- Operator latency in **X.EXEC.csv** - `[t,count,max,mean,min,stddev,p50, p75,p95,p98,p99,p999]` - each line contains a timestamp, three integers (`count`, `max` and `min`), and several decimal values. `count` counts the number of tuples while `max`, `mean` and `min` are the maximum, mean and minimum times in nanoseconds to process a tuple. `stddev` is the standard deviation and the `p` values are percentiles of it. For example, `p99` is the 99th percentile with the highest processing time of all tuples.

With each stream producing two files and each operator an additional two, the total number of files read by the framework is $2S + 2O$, where S is the number of streams and O the number of operators. For the Weather example in Section 2.1.1 there would be one source, five operators, two sinks and seven streams for a total of 30 files.

As the filenames depend on the operator identifiers, the query needs to be visualised before the framework can read the correct files. Additionally, the directory where the files are created is user-defined in the Liebre query. As such, before viewing statistics, the user manually selects the correct directory. Once done, the framework opens a new statistics window, see Figure 5.6.

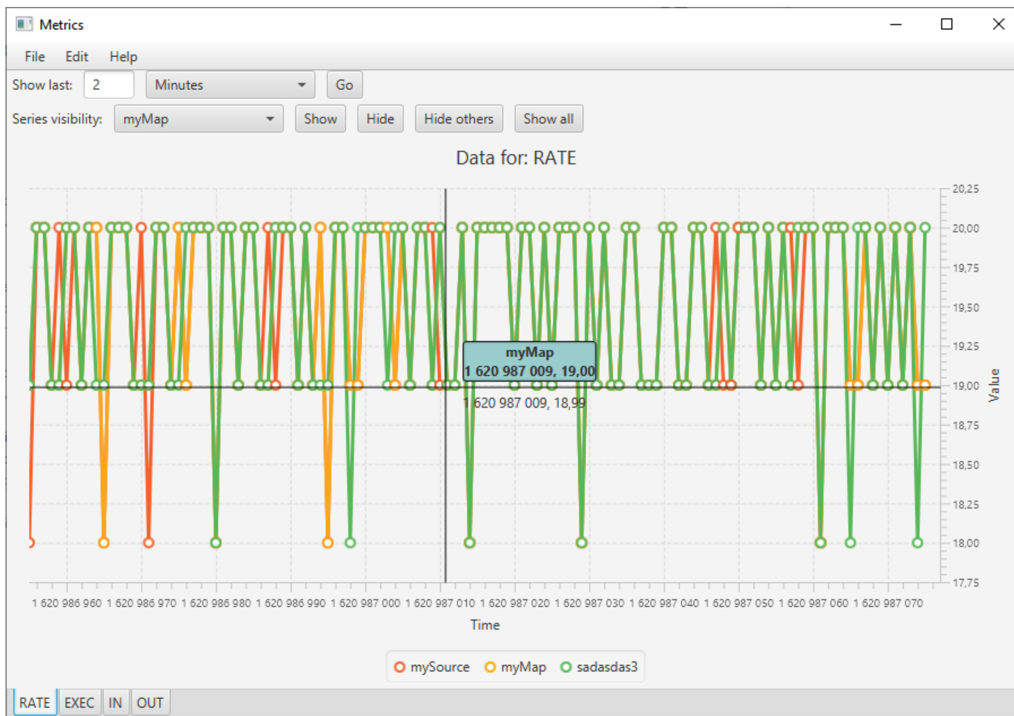


Figure 5.6: Throughput statistics for three operators in a Liebre query.

This window displays statistics, updating with new values every second, for three operators that have differently coloured series. The X-axis shows the timestamps and the Y-axis the values, here oscillating between a throughput of 18 to 20 tuples per second. The user can specify the time range shown and hide and show specific series in the graph. One series is the statistics for an operator or stream. At the bottom of this window there are multiple tabs, one for each type of file.

As the EXEC files contain a great amount of statistics the GUI has additional tabs, one for each operator, for these metrics, see Figure 5.7. By hovering the mouse over a point in the graph it displays its data.

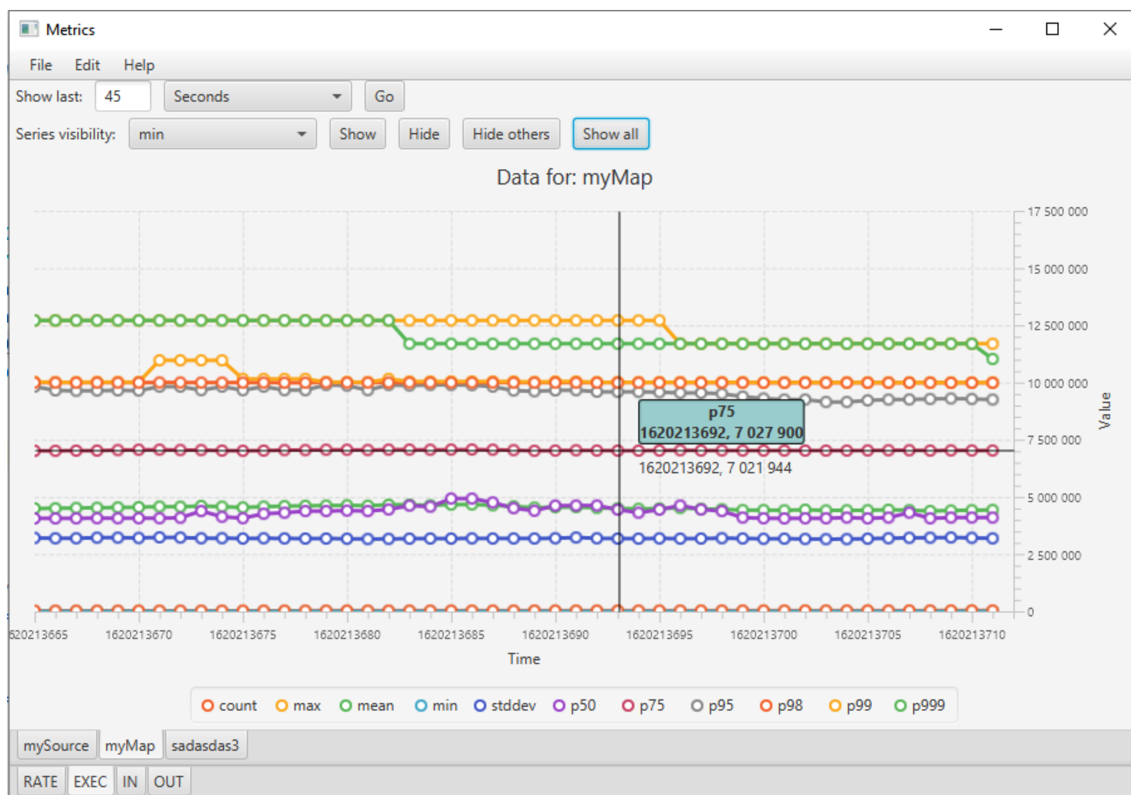


Figure 5.7: Latency statistics for three operators in a Liebre query.

5.4.2 Statistics from Graphite

The second button opens another window, see Figure 5.8. At the top, the user can write a Graphite API query and specify a time range. The *Go* button makes the framework query Graphite and create a new tab, in which the statistics are displayed. Differently from the Liebre statistics, the user manually fetches new data. If a wildcard is used in the query string the GUI displays a separate series for each retrieved statistic, see Figure 5.9. To ease the life of the user the GUI displays text suggestions when writing a query, see Figure 5.10. These suggestions are based on the list of available Graphite statistics, fetched using the Metrics API.

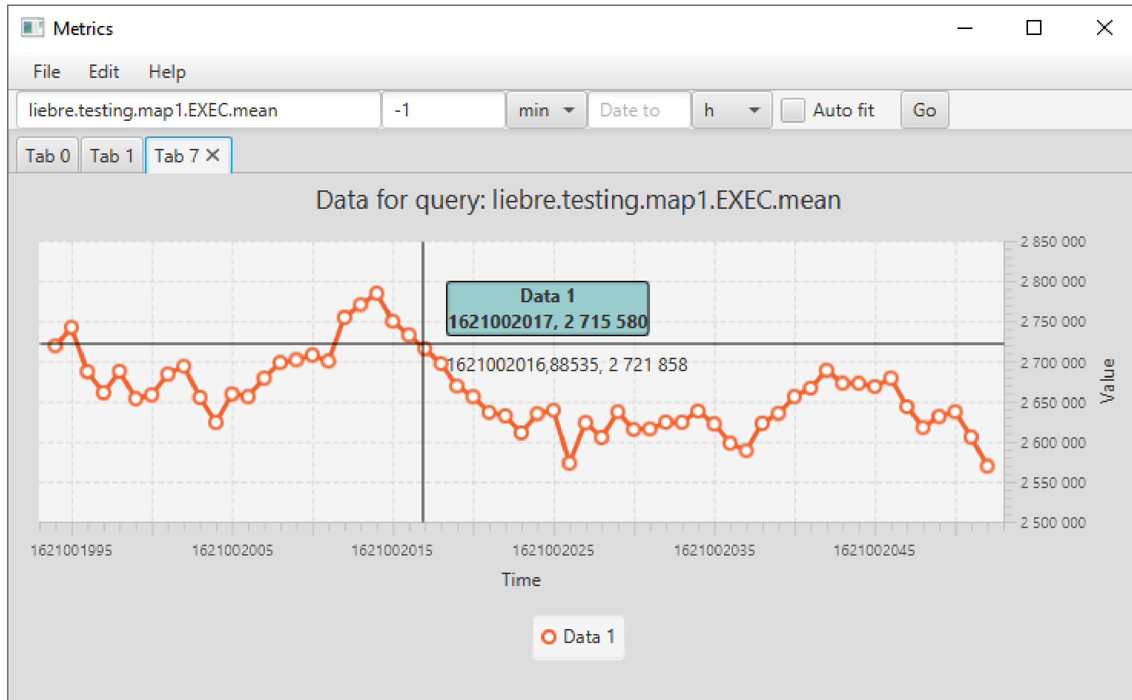


Figure 5.8: Graphite statistics, collected from Liebre, showing the mean processing latency for an operator in the last minute.

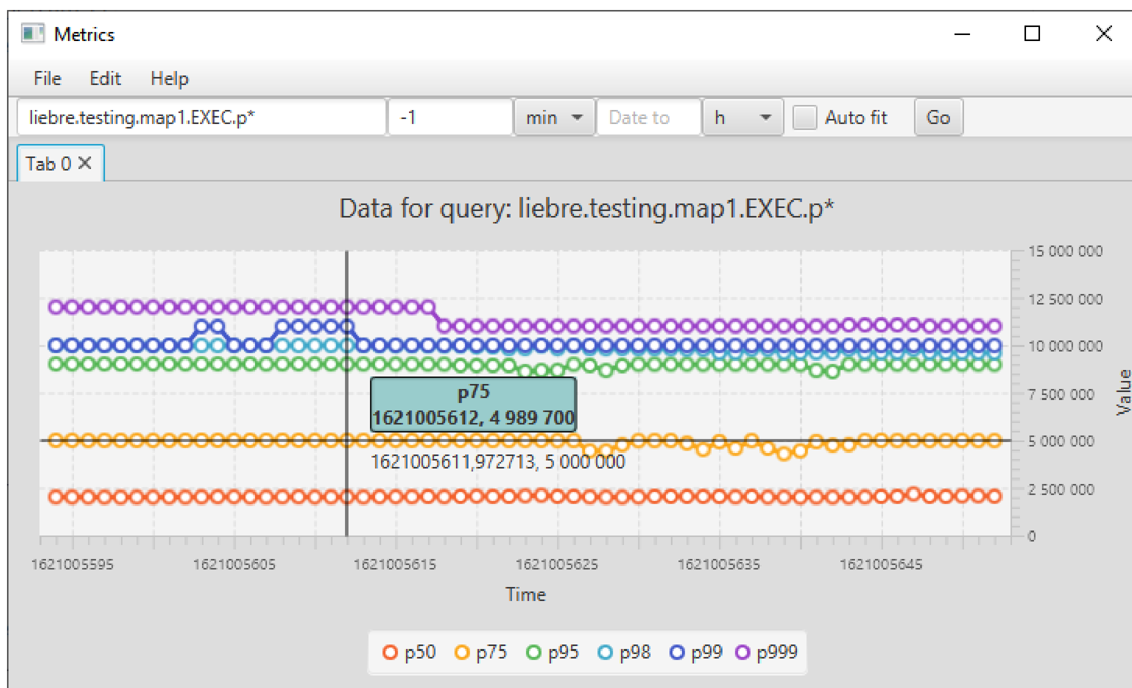


Figure 5.9: Using wildcards in the Graphite query.

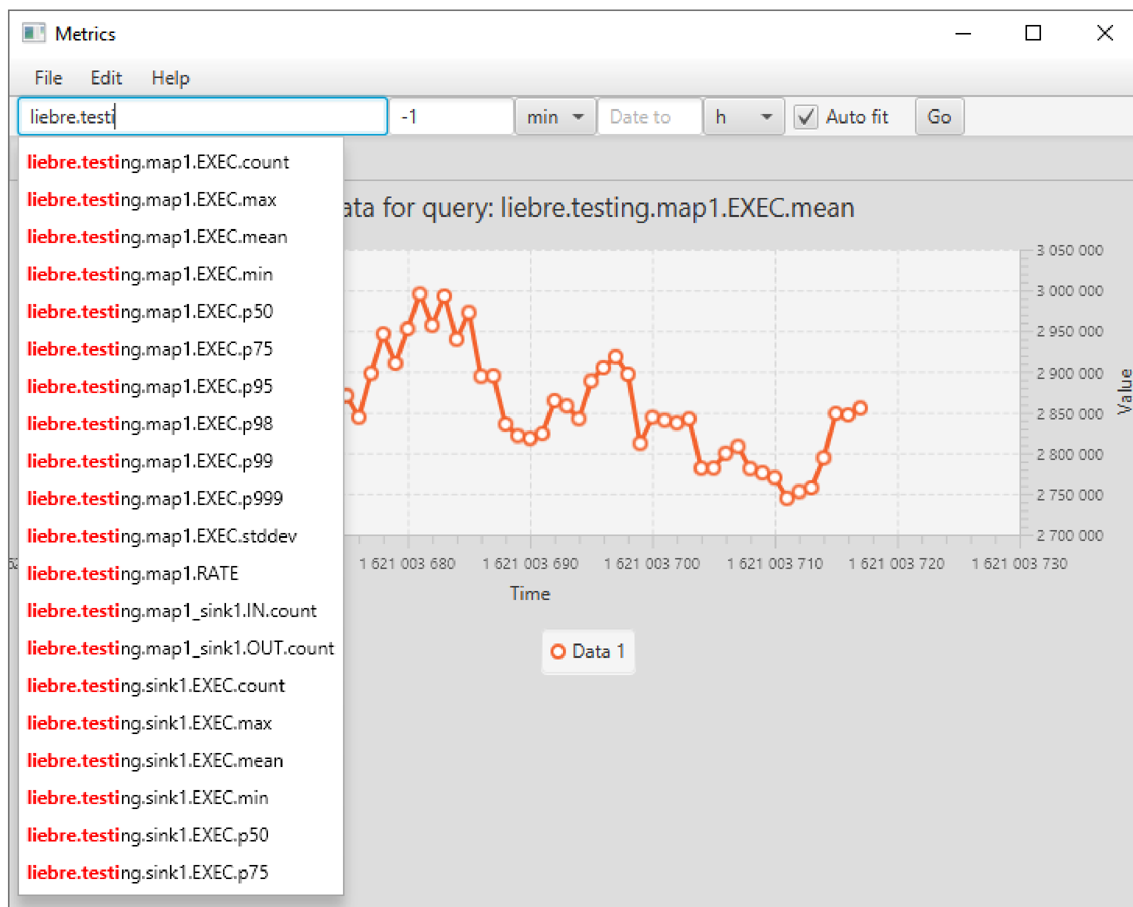


Figure 5.10: Query auto completion for Graphite statistics.

6

Implementation

This chapter concerns the implementation details of the design presented in Chapter 5. It begins with describing the basic implementation of the GUI followed by a more in-depth explanation on how the different goals were achieved. The project code can be found on GitHub [55].

6.1 The GUI elements

Developing a GUI from scratch without any third-party library or tool would be a time-consuming task. This was one reason why JavaFX [56] was chosen as the platform to build the GUI with. JavaFX is a widely used Java platform with many open-source libraries that can be used to speed up the design process [57]. Another reason is Scene Builder, a drag-and-drop user interface designer for JavaFX, which makes designing the GUI more visual compared to simply coding [58].

When the user selects an SPE and clicks the *Start* button the framework loads data associated with the SPE from a `.json` file. One JSON file for each SPE was created and they contain all information needed for the framework to operate. To keep track on the loaded JSON data an abstract Java class `ParsedSPE` was created. This class contains all the information from the JSON file and several methods that are used when generating code. Also created was a Java class for each SPE, namely `ParsedLiebreSPE` and `ParsedFlinkSPE`, extending this class. These subclasses contain no additional data but override the methods used for code generation. This will be further explained in Section 6.2.2. An instance of one of the extending classes is kept during the window's lifetime.

The JSON file for Liebre can be seen in Appendix B. The structure of these files is presented in Listing 15.

```
1  {
2    "name": "Liebre",
3    "operators": {},
4    "links": {},
5    "imports": {},
6    "definition": {}
7  }
```

Listing 15: Structure of an SPE JSON file.

First, `"name"` is simply the name of the SPE. `"operators"` is a JSON object con-

taining three lists of operators, one for sources, one for regular operators and one for sinks. This so that the GUI can distinguish between the three different types. "links" is used when visualising queries and will be explained in Section 6.3. In short it links Java method names to operators. For example, the Liebre method addMapOperator links to a regular operator Map. "imports" contains a list of all imports required by each operator. For example, the Map operator needs the import import component.operator.in1.map.MapFunction;. And lastly, "definition" defines the code for each operator, used for the code generation functionality of the framework. A definition includes information about the number of inputs and outputs and how the code looks like.

Listing 16 shows the definition of the Filter operator in Apache Flink. Here, "before" and "after" define the Java code that creates a Filter operator. "middle" is the part of the definition that the user can code, the other parts are not modifiable. In "placeholders" the inputs, outputs and operator identifier are defined. For instance, when the user changes the input of the Filter operator the placeholder values are used to replace the correct part of the code with the user input, see the *Code* text area in the bottom right of Figure 5.2. In this figure the placeholders @IN1, @ID and @PID have been replaced with WeatherData, op21 and op11, respectively. The "prev_identifier" refers to the operator this operator was ”chained” from, and is not present in the Liebre JSON file. In Figure 5.2, operator op21 is chained from operator op11.

```
1  "Filter": {
2      "before": "DataStream<@IN1> @ID = @PID.filter((FilterFunction<@IN1>)
   ↪  value -> {",
3      "middle": "return false;",
4      "after": "});",
5      "placeholders": {
6          "input": ["@IN1"],
7          "output": [],
8          "identifier": "@ID",
9          "prev_identifier": "@PID"
10     }
11 }
```

Listing 16: The JSON definition of the Filter operator in Apache Flink.

All this information is used throughout the GUI. In Figure 5.2, under the *Operator details*, the user can select an operator's type from a dropdown list. This list is populated with data from the JSON file, specifically from the "operators" object.

6.1.1 Layout and graph design

The launcher window (Figure 5.1) is what the user sees first when launching the framework. Here the user can select from the available SPEs (Flink and Liebre) from a drop-down menu. Once the *Start* button is pressed the framework remembers which SPE was selected and switches scene to the main window (Figure 5.2). The three views of the main window will now be further presented.

- **Graph** - contains the designed and visualised DAGs. A modified version of the JavaFXSmartGraph library is used for this task [59]. This library is responsible for displaying the nodes and edges and it automatically arranges their positions and distances from each other. With the availability to style the graph using CSS files, the nodes were coloured according to the type of operator. Further, generic Java types are used for the nodes and edges, meaning that any custom class can be passed as a node or edge. The label beneath each node contains the `toString()` value from the custom class. The framework uses a custom class hierarchy of `GraphObject`s, which structure can be seen in Figure 6.1.

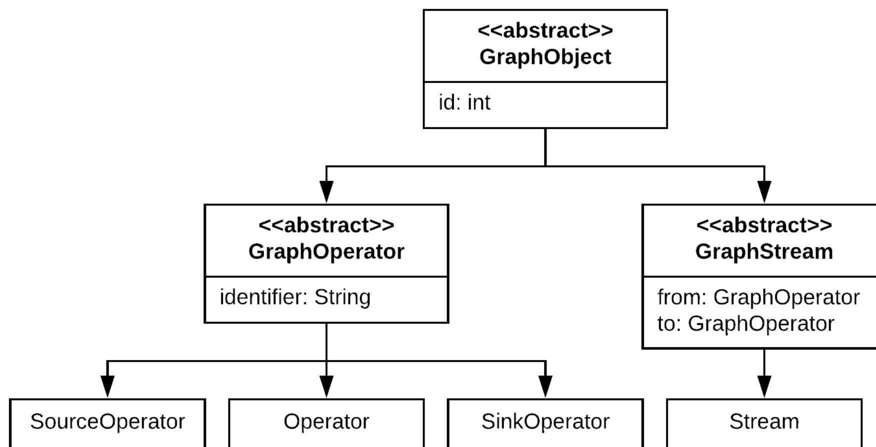


Figure 6.1: The `GraphObject` classes.

Specifically, it uses subclasses of `GraphOperator` for the nodes and subclasses of `GraphStream` for the edges. Each `GraphObject` has a unique ID which is displayed in the parenthesis below each node in the graph and is used to distinguish and compare the nodes and edges in the code. Going down in the hierarchy, each `GraphOperator` has a string identifier. This identifier is modifiable in the *Operator details* in the GUI and is used for both code generation and visualisation but is also shown in the label below the corresponding node in the graph view. A source operator is then an instance of the `SourceOperator` class, a stream an instance of `Stream`, and similarly for the other types.

- **Master** - contains functionality to modify the graph, generate code and view statistics. At the top, the user can add operators to the graph view. For example, clicking on the *Sink* button adds a new node, as an instance of `SinkOperator`. Below these buttons, the user can connect operators and remove streams. This is possible by double clicking the operators in the graph, first the operator which the stream should originate from and then the operator which is to receive its data, and clicking on *Connect selected*. This calls a graph library method `insertEdge(from, to, stream)` which creates an edge of type `Stream` between the two `GraphOperator`s. The GUI displays which nodes have been selected by changing their colour and adding the text `[FROM]` or `[TO]`, depending on which node was double clicked first, to their labels, see

Figure 6.2. A modification to the JavaFXSmartGraph library was performed so that it, in addition to single clicking, supports double-clicking nodes.

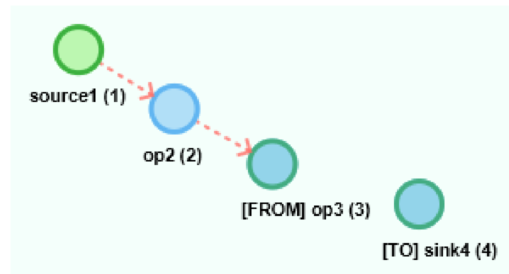


Figure 6.2: Two nodes in the graph have been selected, the first labeled `[FROM]` and the second `[TO]`.

These remaining functionalities will be explained later in this chapter.

- **Operator details** - contains details about the a clicked operator, populated with data from the SPE JSON file and any modifications done by the user, stored in the `GraphOperator` instances.

Additional functionality in the top menu includes exporting and importing graphs and visualising Java queries. Also possible is to change the current SPE without having to restart the framework.

When designing a graph, the framework prevents invalid graphs from being created. Firstly, it is not possible to add an edge ending at a source or starting at a sink as it prevents a source operator from being selected as `[TO]` and a sink from being selected as `[FROM]`. Then, when the *Connect selected* button is pressed, it checks if adding a new edge creates a cycle in the graph using Kahn's algorithm, as presented in the previous chapter.

Metrics for the framework's CPU and memory usage is displayed in the bottom left and bottom right corner, respectively, in the GUI. These values update once every second and are retrieved from the MXBeans `OperatingSystemMXBean` and `MemoryMXBean` classes, as seen in Listing 17. The memory usage is returned in bytes, so it is converted into MB by dividing it with 1048576 ($1024 * 1024$).

```
1  OperatingSystemMXBean osBean =
   ↪  ManagementFactory.getPlatformMXBean(OperatingSystemMXBean.class);
2  MemoryMXBean memoryBean =
   ↪  ManagementFactory.getPlatformMXBean(MemoryMXBean.class);
3
4  double usedCPUPercentage = osBean.getProcessCpuLoad() * 100;
5  long usedMemoryMB = memoryBean.getHeapMemoryUsage().getUsed() / 1048576;
```

Listing 17: Using MXBeans to fetch the framework's CPU and memory usage.

6.1.2 Exporting and importing graphs

The process of exporting a graph begins when the user clicks on the menu item *Export to file* from the *File* menu. First, the designed graph is converted into a DAG (explained in Section 6.2.1) which in turn is made into one large JSON object representation. Its structure is presented in Listing 18. Here, `"spe"` is the name of the SPE and `"nodes"` is an array of all root operators (not only sources as, for instance, a sink operator may not yet have been connected with another operator) as JSON objects. Each such object contains the operator's identifier (`"name"`) and type (`"type"` (which is 0 for sources, 1 for regular operators and 2 for sinks) and another JSON object `"ops"`. This last object contains any user changes made to the operator's inputs, outputs and code. Finally, each root operator has an array, `"suc"`, containing its successors, meaning the operators it has output streams to.

```
1   {
2     "spe": "Liebre",
3     "nodes": [{
4           "suc": [...],
5           "data": {
6               "name": "source1",
7               "type": 0,
8               "ops": {
9                   "BaseSource": {
10                      "def": {
11                          "in": [ ],
12                          "mid": "return (Math.random() * 60) + \"\";",
13                          "out": ["String"]
14                      },
15                  "name": "BaseSource",
16                  "prev_name": ""
17                  "type": 0
18                  }
19              }
20          }
21      }]
22  }
```

Listing 18: The JSON structure of an exported graph.

Generating this JSON object was implemented by each `GraphOperator` having a `toJsonObject()` method. This method creates a JSON representation of the operator (the `"data"` object) and calls this method recursively for all its successors and collects them in the `"suc"` object. Once the JSON object has been generated it is written to a user-selected directory as a `.json` file.

Importing a `.json` file is done in a similar way. When a file has been selected by the user, the framework confirms that the `"spe"` field equals the currently in-use SPE. If not, it is not imported. Otherwise, the framework parses the data, extracts the operators, and adds them to the graph view.

6.1.3 Text suggestions for user inputs

Displaying text suggestions for user inputs was implemented using the Java class `AutoCompleteTextField` [60] extending `TextField` - the basic text input field in JavaFX. The available suggestions are are all public Java classes from the implemented SPEs and some common Java classes. Pseudo code for finding these classes can be seen in Algoritm 2.

Algorithm 2 Finding public java classes

```
Require: F ⇐ A directory to search for classes in
Require: P ⇐ A RegEx pattern to match java classes with
 1: procedure FindJavaClassesIn(F)
 2:     R ⇐ An empty concurrent list that will contain the found class names
 3:     T ⇐ An empty concurrent list that will contain spawned threads
 4:     FindClassFilesIn(R, T, F)
 5:     for each thread t in T do
 6:         Wait for t to die.
 7:     end for
 8:     return R
 9: end procedure
10:
11: procedure FindClassFilesIn(R, T, F)
12:     for each file f in F do
13:         if f is a directory then
14:             FindClassFilesIn(R, T, f)
15:         else
16:             n ⇐ name of f
17:             if n is a .java file then
18:                 t ⇐ spawn new thread
19:                 call FindClassesInFile(R, f) from t
20:                 T ⇐ T ∪ t
21:             end if
22:         end if
23:     end for
24: end procedure
25:
26: procedure FindClassesInFile(R, f)
27:     for each line l in f do
28:         if p matches l then
29:             n ⇐ extract class name
30:             R ⇐ R ∪ n
31:         end if
32:     end for
33: end procedure
```

Shortly described, the method `findJavaClassesIn(File directory)` was called manually with `directory` set to the root code folder of each SPE. The files in the

directory were then traversed. If a new directory was found it recursively calls the method with the new directory as argument. If a `.java` file is found a new thread is spawned that traverses the file line-by-line and tries to match each line with the RegEx pattern in Listing 19. This pattern matches lines that declare a public class, interface or enum, that can be static or final. Here, capture group number eight, `(\w+)`, is used to extract the name of the class. When all threads have completed the result is returned as a list of strings. Finally, the list is converted into a comma-separated string and saved in a `.txt` file, `flink-classes.txt` for Apache Flink and `liebre-classes.txt` for Liebre. The entries in these files are then read upon starting the main window and fed to the `AutoCompleteTextField` class.

```
(\s+)?(public\s+((static|final)(\s+))*(class|interface|enum)(\s+)(\w+)(<.*>)?(.*))
```

Listing 19: The RegEx pattern used to match public Java classes, interfaces and enums with.

6.1.4 Graph and type analysis

Clicking on the *Check graph* button in the *Master* view makes the framework analyse the designed graph for errors. First, it creates a DAG of the design and three "error lists". The first list contains non-unique operator identifiers, the second contains nodes that have not been assigned an operator type, and the third contains operators that expect a certain type of data as input from a stream but receives another type from the operator at the opposite end of the stream. For example, a Map operator expecting String values but receives Integer values is an error. Then, the DAG is traversed and the lists are filled with any found errors. Once traversed, a popup window displays the result, see the example in Figure 5.4.

6.2 From a graph to generated code

This section starts with describing how a designed graph is turned into a DAG and ends with explaining how the DAG is used to generate the corresponding Java query.

6.2.1 Creating a DAG

The graph library stores the nodes and edges as separate lists and does not support creating a DAG automatically. To do this, a Java class `DirectedGraph` was created. Its main method is `static DirectedGraph fromGraphView(Graph<GraphOperator, GraphStream> graph)`, which takes as input a `Graph` object (the designed graph, a class from the JavaFXSmartGraph library) and returns an instance of the class. The instance has a list `List<Node<GraphOperator>>`, containing the root operators in the graph. The `Node` class contains two objects: a `GraphOperator` and a list `List<Node<GraphOperator>>`, with all the node's successors. Together, the result forms a DAG.

Algorithm 3 presents how the `fromGraphView()` method creates the DAG. In the first step all root operators are found. For each such operator, it stores the operator and its successors, found using the recursive FindSuccessors method, in a list (`List<Node<GraphOperator>>`). When finished, the list (the DAG of the graph) is returned.

Algorithm 3 From a graph to a DAG

Require: $R \Leftarrow$ Empty list that will contain the DAG
Require: $G \Leftarrow$ The graph

1: **procedure** GenerateDAG(G)
2: **for each** node n in G **do**
3: **if** n has no incoming edges **then**
4: $(n, successors) \Leftarrow$ FindSuccessors(n)
5: $R \Leftarrow R \cup (n, successors)$
6: **end if**
7: **end for**
8: **return** R
9: **end procedure**
10:
11: **procedure** FindSuccessors(n)
12: $S \Leftarrow$ Empty list of $(node, successors)$
13: **for each** node m in G **do**
14: **if** n has an outgoing edge to m **then**
15: $S \Leftarrow S \cup$ FindSuccessors(m)
16: **end if**
17: **end for**
18: **return** (n, S)
19: **end procedure**

6.2.2 From a DAG to code

When the DAG has been created the framework begins the process of generating the Java code using the `ParsedSPE` instance. This class has an abstract method `String generateCodeFrom(DirectedGraph directedGraph, String fileName)` that each instance overrides in order to meet the SPE-specific implementation requirements. The returned data is a string of the Java file contents. This method begins with creates a `StringBuilder` instance and the process of filling it with Java code works as follows:

1. With the information from the SPE specific instance of `ParsedSPE`, the required query imports are added, both for the query itself and for the operators.

2. Added next is a new class and the start of a main method (`public static void main(String[] args){`).

3. Following, the start of a query is added. For Liebre it is the line `Query q = new Query();`.

4. Then, the code structure for each operator is retrieved from the JSON file. Each operator's identifier, inputs, outputs, and code are replaced with the user inputs from the GUI. The resulting code is added to the file. As an example, Listing 20 shows the code generated for a Filter operator where the user has made no changes to the operator data, while in Listing 21 a "correct" operator has been designed and generated.

```
1 DataStream<@IN1> op1 = @PID.filter((FilterFunction<@IN1>) value -> {
2     return false;
3 });
```

Listing 20: Generated Flink code for an operator that has not been designed correctly by the user.

```
1 DataStream<Integer> op1 = source1.filter((FilterFunction<Integer>) value -> {
2     return value > 20;
3 });
```

Listing 21: Generated Flink code for an operator that has been designed correctly by the user.

5. The `ParsedFlinkSPE` class connects the operators by chaining and the `ParsedLiebreSPE` class connects the operators using `q.connect(from, to);`.
6. Almost complete, the query needs to be executed. In Flink this is done by adding the line `env.execute();` and `q.activate();` in Liebre.
7. Two closing brackets are then added, one closing the main method and the other the class. As a last step, the string is formatted using a Google Java style library [61], which cleans up and indents the code and removes any duplicate import statements.

When completed, a new `.java` file is created in the user-selected directory and the string data is written to it.

6.3 Visualising streaming applications

Once a file is selected for visualisation, JavaParser parses it and produces a CompilationUnit. Then, all class definitions in the file are collected using a `VoidVisitorAdapter`, overriding the `public void visit(ClassOrInterfaceDeclaration n, Void arg)` method. From this, all class methods are obtained using the `ClassOrInterfaceDeclaration` method `getMethods()` that returns a list of `MethodDeclarations`. Two methods were created for each SPE: `methodParserFindVariables()` and `methodParserFindDefinitions()`. The first searches the code for query variables and the second operator definitions. Extracting the relevant information using these two adapters is done in three steps:

1. **Finding the query variables** - The first adapter overrides the method `public void visit(VariableDeclarator n, Void arg)`. The task is to find the start of the query and the query variable, that is on the form `Query query = new Query();` for Liebre. By using the `VariableDeclarator` method `n.getType()`, it compares any found variables with `Query`. When a match is found the variable name is saved.

   Specifically for Liebre, the same adapter also overrides the method `public void visit(MethodCallExpr n, Void arg)`, visiting all method calls in order to find connected operators on the form `query.connect(op1, op2)`. The `MethodCallExpr` object has no information about the type of the variable, meaning that it does not know that a `query.connect()` call was made from a `Query` object. For this reason, the framework resorts to checking if the method name is `connect` and the number of parameters is two. When found, the two parameters are the variable names of the operators and these are saved along with the information that they are connected. In Flink the connections are found in the next step.

2. **Finding the operators and their definitions** - In both Flink and Liebre operators are created using method calls, for example `stream.map(...)` in Flink and `query.addMapOperator(...)` in Liebre. As such, the second adapter overrides the method `public void visit(MethodCallExpr n, Void arg)`.

   The **Liebre** implementation of this method is the simplest of the two. As the operator variables have already been found the adapter simply checks if the currently visited method is called with one of the found variables. If it is, the adapter extracts the relevant operator information. The `MethodCallExpr n` only gives information about the method call and not if the result of the call is saved in any variable. For example, if it visits the code in Listing 13 the data in the method parameter `n` only contains the part `query.addBaseSource("mySource", ...);`. Finding that it is saved in the variable `src` requires some more work.

   The JavaParser AST is built up of nodes (objects extending the JavaParser `Node` class) and `MethodCallExpr` is such a node. All nodes have a method `getParentNode()` that returns the parent (the node one level higher in the tree). See Listing 22 for an example of the code at different levels in the tree, where the query is located in the top-level of a method body. The first line is the original `MethodCallExpr`, the second is a `VariableDeclarator` and the third line is of type `VariableDeclarationExpr`. In this example, two levels further up is the entire method body. As in the first step, the `VariableDeclarator` contains all the necessary information so the framework stops at this level.

   Flink requires some more work as operators can be chained and does not necessarily have to be saved in a variable. For example, a line `stream.map(...).filter(...).flatMap(...);` is a valid definition of three operators. Consider the example in Listing 23. Here, the lines one to three are of type `MethodCallExpr` and the fourth `ExpressionStmt`. As before each line goes one level higher in the AST.

   The Flink adapter calls `getParentNode()` as long as the type it finds at

```
query.addBaseSource("mySource", ...)
src = query.addBaseSource("mySource", ...)
Source<Double> src = query.addBaseSource("mySource", ...)
Source<Double> src = query.addBaseSource("mySource", ...);
{the entire method body}
```

Listing 22: Nodes in the JavaParser AST. Each line goes one level higher in the tree.

```
intStream.map(...)
intStream.map(...).filter(...)
intStream.map(...).filter(...).flatMap(...)
intStream.map(...).filter(...).flatMap(...);
{the entire method body}
```

Listing 23: Nodes in the JavaParser AST. Each line goes one level higher in the tree.

the new level is of type `MethodCallExpr`. Once it stops, the parent for the node at the current level should either be of type `VariableDeclarator` or `ExpressionStmt`. In the first case the stream is stored in a variable and in the second it is not.

As multiple operators can be chained, a method `List<Pair<String, String>> getMethods(String node)` was created to extract information about the chained operators. It takes the string representation of the node (got by calling `node.toString()`) as input and traverses it from left to right while counting the number of opening and closing parentheses. The return data is a list of pairs, where each pair contains an operator and the operator's code. The counter starts at zero and for each opening parenthesis the counter increases by one and for each closing parenthesis it decreases by one. If it goes from a positive number to zero a complete operator has been found. Listing 24 shows how the counter moves.

```
intStream.filter( (FilterFunction<Double>) value -> value > 2);
0                1 2                      1                 0
```

Listing 24: The counter starts at zero, increases for each opening parenthesis and decreases for each closing parenthesis.

By keeping track on the positions where the counter is set to one or to zero the adapter can extract the operator's code. With these positions it is also easy to find the type of operator. In this example the filter operator can be found between the position of the dot and the position where the counter increases from zero to one. When an operator has been found the adapter searches for it in the SPE JSON file to make sure that it actually is an operator. If it is found the framework adds its name and data as a new pair in the resulting list. Otherwise it is just a regular chained method call. For example, the Flink code `stream.map().assignTimestampsAndWatermarks().filter();` contains a

method call that is not an operator but is still relevant to the query. When the adapter finds such a method call it appends it to the code of the previous operator, in this case `map`, thus making it visible in the GUI.

When an operator has been found the adapter determines what kind of operator it is. For example, in Listing 22 the Liebre method name `addBaseSource` is "translated" into a BaseSource operator. This information is contained in the `"links"` field of the SPE JSON file. Listing 25 shows a shortened version of it for Liebre. Each key links to its corresponding operator type and the operator. For instance, by searching for a key named `addBaseSource` it is found that the type is a source operator, and the operator is a BaseSource.

```
"links": {
    "addBaseSource": "source:BaseSource",
    "addMapOperator": "op:Map",
    "addFilterOperator": "op:Filter",
    "addBaseSink": "sink:BaseSink"
}
```

Listing 25: The (shortened) `"links"` field in the Liebre JSON file.

3. **Finding how the operators are connected** - With all operators located the framework finds how they are connected. In Liebre this is simple. As all connections have been found in step one, the framework iterates over the list of operators and arranges them into a new DAG-structured list using the `Node` class.

   For Flink the process is a bit different. For each operator the framework knows which other operator "created" it and in which variable it is saved in. For instance, `map1` in `DataStream<String> map1 = source1.map(...);` was created using `source1` and they are thus connected. Consider the example in Listing 26. Here, five operators are created but only two variables exist. From this, the framework knows that `source1` is a root in the DAG as it was created using the `query` variable. In addition to knowing an operator's position in the chain, the framework also knows the order of the operators. Here, it is known that the source operator sends data to the Filter operator which sends to the Map operator which sends to the FlatMap operator. Finally, the output stream is stored in the `op1` variable. Since the Filter and Map operator are chained and not saved in any variable they can also not create any other streams. The framework then knows the final order of these four operators. However, the FlatMap operator can create additional streams as it is the one linked to the variable. When the framework sees that `op1` creates a sink operator the two are connected. With this information a list of `Nodes` is created.

Once the information has been extracted, the framework creates a new tab in the graph view, traverses the `Node` list, adds the operators to the graph and creates streams to connect them. The user can then view the Java query visualised as a DAG.

```
StreamExecutionEnvironment query =
    StreamExecutionEnvironment.getExecutionEnvironment();
DataStream<String> source1 = query.addSource(...);
DataStream<Integer> op1 = source1.filter(...).map(...).flatMap(...);
op1.addSink(...);
query.execute();
```

Listing 26: An example of a Flink query with five operators.

6.4 Graph and operator statistics

The open-source graph library ExtJFX is used to display statistics [62]. It contains several components, such as line charts, bar charts and heat maps, but only the line chart has been used in the GUI as it is a good fit for displaying these types of statistics. While JavaFX contains line charts, this library is an extension to add more functionality such as crosshairs and data point tooltips.

6.4.1 Statistics from Liebre

When the user selects the CSV files directory, JavaFX returns a `File` object pointing to it. From this, together with the knowledge of which operators exist in a visualised query, the correct CSV files are located. Four tabs are then created and the *EXEC* tab creates a tab for each operator. Each tab is populated with an instance of `XYChartPane<Number, Number>` from the ExtJFX library. The two objects in the diamond specify the type of data for the X- and Y-axis, respectively. As the statistics are integer or decimal values the `Number` class is used.

To read the file contents the Java `Tailer` class, from the Apache Commons IO library, is used [63]. Instead of manually reading a file, for example to every second check if any new lines have been added, this class makes it possible to "listen" to it and get notified for new lines. For this task, an instance is created as `Tailer tailer = new Tailer(file, tailerListener, delayMillis, fromEndOfFile);`. The first argument specifies the file to listen to, the second is a *tailer listener*, explained shortly, the third how often the file is checked for new content and the fourth is a Boolean value specifying if the listener should start from the end of the file. If the last argument is set to false, the tailer will start from the beginning of the file and notify for each line. The delay was set to 500 milliseconds and the graph can then update up to two times per second.

The `tailerListener` is a custom class extending `TailerListenerAdapter`. Listing 27 shows the implementation. The framework creates an instance of it for each file. The first argument to the constructor is a metric listener and the second is the name of the file. The file name is necessary as all the files use the same metric listener. The name is then used to determine which tab and graph the data should be added to. From the `TailerListenerAdapter` class, only the `handle(String line)` method is of importance. The tailer class calls this method when a new line is discovered. In this method, when a line is received the data is extracted and saved in a `FileData` class together with the file name. This is then passed to the metric

listener. Originally being executed on (and blocking) the UI thread, the framework runs each adapter in a new thread.

```
private static class MyTailListener extends TailerListenerAdapter {
    private final IOnNewMetricDataListener listener;
    private final String name;

    private MyTailListener(IOnNewMetricDataListener listener, String name) {
        this.listener = listener;
        this.name = name;
    }
    @Override
    public void handle(String line) {
        listener.onNewData(new FileData(extractData(line), name));
    }
}
```

Listing 27: The tail listener adapter.

The ExtJFX library provides methods to change the bound of an axis. When the user sets a time range in the GUI the framework parses the given information and changes the lower bound of the X-axis accordingly. The Y-axis changes height to fit the data in the selected bound automatically.

6.4.2 Statistics from Graphite

In the Graphite window, when the *Go* button is pressed the framework creates a new tab populated with an instance of `XYChartPane<Number, Number>` and queries the Graphite Render URL API with the user parameters. The response is JSON data containing a JSON array with data points for each statistic. Using a wildcard in the query will return an array for each matched query. The framework creates a series for each array and displays this in the graph. For example, the query `liebre.testing.map1.EXEC.p*` matches all percentile latency values for the `map1` operator. As a result, six series will be created. See Figure 5.9 for how this can look.

To display query suggestions in the query text field, the framework queries the Graphite Metrics API at the URL `/metrics/index.json`. It returns a JSON array containing all the queries. The data is then used together with the `AutoCompleteTextField` class. See Figure 5.10 for an example.

7

Evaluation

This section presents the outcome of the thesis work and evaluates the different functionalities, the performance and the generality of the framework. First, Section 7.1 evaluates the visualisation goal on different streaming applications. Following, in Section 7.2 the application design and code generation is evaluated in terms of the number of operators that are supported and can be used to create queries with. Then, Section 7.3 evaluates the statistics reporting for the two SPEs and Section 7.4 tests the framework's performance, such as CPU and memory usage, in different scenarios. Next, in Section 7.5 the generality of the framework is evaluated along with a discussion of the work required to support a new SPE. Finally, Section 7.6 discusses the results and outcome of the thesis.

7.1 Visualisation of streaming applications

To evaluate the visualisation goal, five different queries of varying complexity were used: three from StreamBenchmark [64] (*FraudDetection*, *YSB* and *LinearRoad*), the weather example presented in Section 2.1.1, and *StreamingJob*, found in the "testing" folder on the project's GitHub page [55].

The tests are presented below, ordered by complexity. The numbers in parenthesis is the, for Flink, number of sources, operators and sinks in the query.

1. **FraudDetection** (1-2-1) - Both Flink and Liebre visualise this query correctly, see Figure 7.1. Note that Liebre does not have a KeyBy operator.

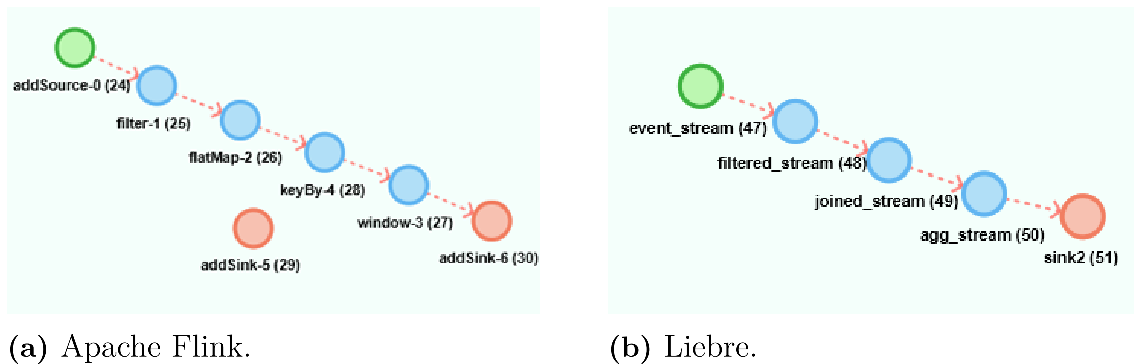
2. **YSB** (1-5-2) - This query contains both stateless and stateful operators as well as a *side output* that forwards late tuples to a sink. Figure 7.2 shows the result. For Liebre the result is correct as it does not support side outputs from windows and only has one sink. As can be seen for Apache Flink, the framework failed to connect the Window operator with one of the sinks. The cause is a missing Fold operator which has not been added to the JSON file. See Figure 7.2c for the correct DAG, However, as explained in Section 6.3, the code for the Fold operator is added to the previous operator's code and can still be seen in the GUI.

3. **Weather** (1-6-2) - This query is the weather example from Section 2.1.1 and Figure 7.3 shows the result. For this test both the Flink and Liebre queries have been visualised correctly. For Liebre an extra router operator has been added to split the stream and the top branch only has one Aggregate operator instead of Flink's three.

(a) Apache Flink.

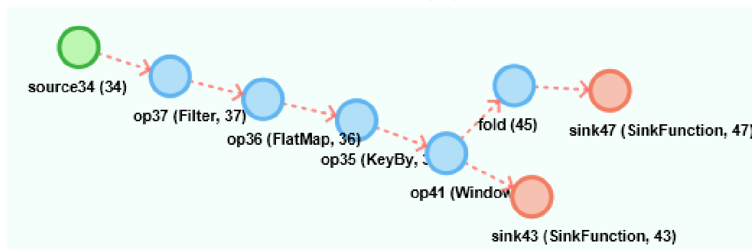
(b) Liebre.

Figure 7.1: The FraudDetection visualisation test.



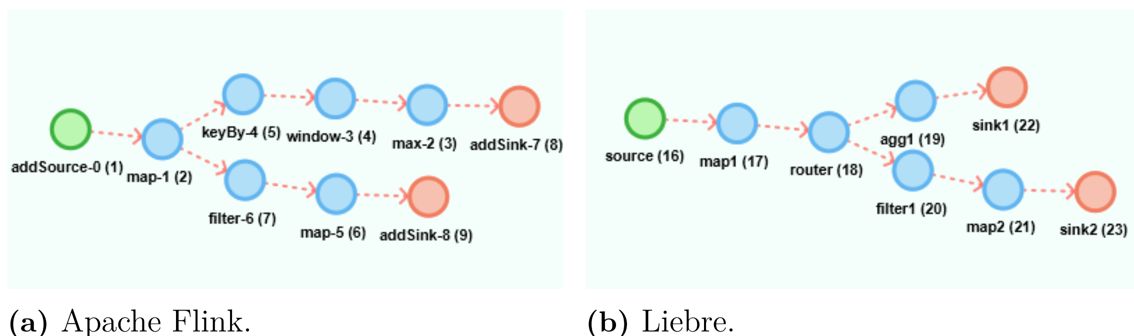
(a) Apache Flink.

(b) Liebre.



(c) The correct Apache Flink query.

Figure 7.2: The YSB visualisation test.

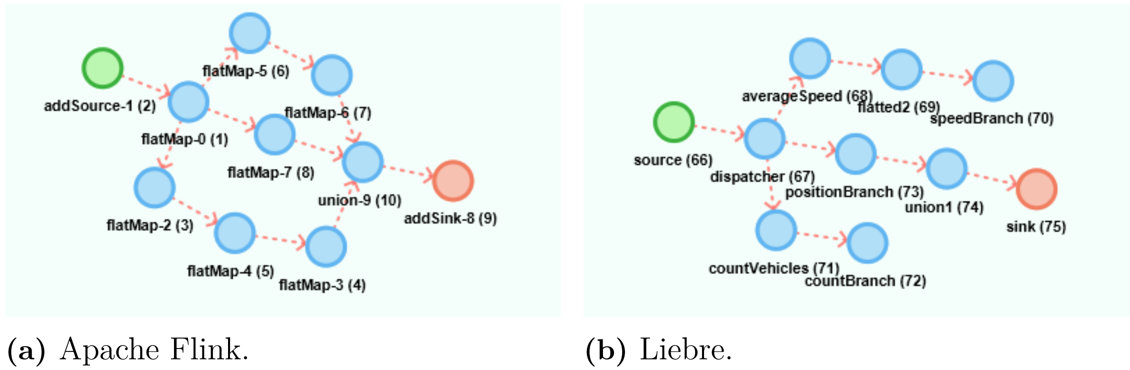


(a) Apache Flink.

(b) Liebre.

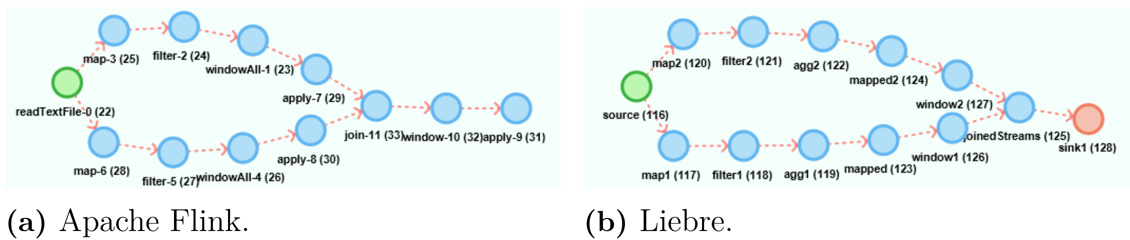
Figure 7.3: The weather visualisation test.

4. **LinearRoad** (1-8-1) - This query splits the stream into three branches after the first FlatMap operator and these are then joined together with a Union operator just before the sink, see Figure 7.4. This is visualised correctly for



(a) Apache Flink. **(b)** Liebre.

Figure 7.4: The LinearRoad visualisation test.



(a) Apache Flink. **(b)** Liebre.

Figure 7.5: The StreamingJob visualisation test.

Flink. For Liebre, the issue is that the framework has no logic to join streams that are not in a `connect` method call. When a Union operator has been created with `query.addUnionOperator("union1")` the other stream is joined with the method `union.addInput(otherStream)`. As the framework is not searching for these methods it does not have the needed information. In comparison, in Flink the Union operator simply takes the streams to be joined as the method parameters directly, e.g. `positionBranch.union(speedBranch, countBranch)` and this is found by the framework.

5. **StreamingJob** (1-11-1) - This query reads data from a text file, performs operations in two branches and joins the results in the end, see Figure 7.5. Here, Flink joins the streams earlier than Liebre but performs additional operations after the join. The framework succeeds in visualising the query correctly for Liebre but not completely for Flink. After the Apply operator the Flink query uses a `stream.writeAsText(...)` sink operator which has not been added the Flink JSON file. Liebre instead uses a `query.addTextFileSink(...)` method call which is captured.

These five tests were all fully or partially successful. In common, the whole query is contained in a single top-level method in the root class of the Java file. None of them are split up into different methods or, for example, created in nested classes. The FraudDetection and YSB queries are enclosed in an `if-else` statement but does not affect the result.

To further investigate the usability of the visualisation functionality, modifications were made to the weather query and three additional tests were performed on it:

6. **Creating the query in a thread** - This tested whether the framework correctly visualised a query defined in a new thread, see Listing 28. For Liebre, the result is equal to Figure 7.3b, that is, correct. For Flink, the query is only partially visualised and the two sink operators are missing, see Figure 7.6.

```
1  public static void main(String[] args) {
2      new Thread(new Runnable() {
3          @Override
4          public void run() {
5              Query query = new Query();
6              ...
7          }
8      }).start();}
```

Listing 28: Creating the query in a new thread.

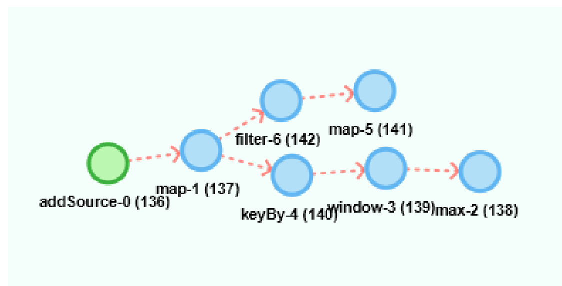


Figure 7.6: The Weather query, defined in a new thread, visualised for Flink.

7. **Part of the query in a separate method** - Splitting up a long code segment into multiple methods is a way to make code more readable. As such, this query was setup to test whether the visualisation works correctly when a part of it is defined in a separate method. More precisely, the query root (`Query query = ...` for Liebre), the source operator and the Map operator were created in the main method while the rest were created in another class method, see Listing 29 for the structure. The framework successfully visualised this Java class, both for Liebre and for Flink.

8. **Created in the constructor of a nested class** - This setup, shown in Listing 30, tested whether the framework can visualise queries defined in a nested (inner) class. Because JavaParser only visits methods and not class constructors, this test failed and no operators were visualised.

9. **Created in a nested class method** - Similar to the previous test, this query was defined in a method of the nested class instead of the constructor. Listing 31 shows the setup. It was successful for both SPEs and all operators were found and visualised correctly.

These nine tests are summarised in Table 7.1. Tests labelled *Correct* were visualised correctly with all operators and streams, *Partially correct* tests found a majority of the operators and streams and *Failed* tests found no operators at all.

```
1  public class WeatherLiebre {
2      public static void main(String[] args) {
3          Query query = new Query();
4          // Source
5          // Map
6          createQuery(query, source1, map1);
7          query.execute();
8      }
9
10     private void createQuery(...) {
11         // create the rest of the operators
12         query.connect(...); // connect them all
13     }
14 }
```

Listing 29: How the query was split up in two methods.

```
1  public class WeatherLiebre {
2      public static void main(String[] args) {
3          new WeatherClass();
4      }
5      private static class WeatherClass {
6          public WeatherClass() {
7              Query query = new Query();
8              // ...
9              query.execute();
10         }
11     }
12 }
```

Listing 30: The query created in the constructor of a nested class.

```
1  public class WeatherLiebre {
2      public static void main(String[] args) {
3          new WeatherClass();
4      }
5      private static class WeatherClass {
6          public WeatherClass() {
7              run();
8          }
9          private void run() {
10             Query query = new Query();
11             // ...
12             query.execute();
13         }
14     }
15 }
```

Listing 31: The query created in a nested class method.

Table 7.1: Result summary of the visualisation tests.

#	Liebre	Flink
1	Correct	Correct
2	Correct	Partially correct
3	Correct	Correct
4	Partially correct	Correct
5	Correct	Partially correct
6	Correct	Partially correct
7	Correct	Correct
8	Failed	Failed
9	Correct	Correct

7.2 Application design and code generation

Table 7.2 contains all supported operators for visualisation and code generation. The second column contains all operators that have been added to the `"links"` field in the SPE-specific JSON file and are recognised by the framework. The third column contains the operators that can be used when designing a streaming application, that is, they have been defined in the `"definition"` field of the JSON file.

Table 7.2: Supported operators for visualisation and for application design.

SPE	Supported operators for visualisation	Supported operators for application design
Liebre	*BaseSource, TextFileSource, BaseSink, TextFileSink, BaseOperator1In, BaseOperator2In, Map, FlatMap, Filter, Router, Aggregate, Join*	*BaseSource, TextFileSource, BaseSink, TextFileSink, BaseOperator1In, BaseOperator2In, Map, FlatMap, Filter, Router, Aggregate, Join*
Flink	*SourceFunction, TextFileSource, Map, Max, Min, FlatMap, Filter, KeyBy, Window, WindowAll, Window Join, Apply, Union, SinkFunction*	*SourceFunction, TextFileSource, Map, Max, Min, MaxBy, MinBy, Sum, FlatMap, Filter, KeyBy, Reduce, Window, SinkFunction*

As Liebre is a minimal SPE most its operators are supported. The Union operator is one that has not been implemented, but, as seen in Figure 7.4b, the framework still adds unknown Liebre operators to the graph as it was found in a `connect()` method call. Unknown operators are reported as *Operator: null* in the *Operator details* view of the GUI. For Flink only the most common operators are supported.

7.3 Statistics reporting

An evaluation of the statistics reporting goal was performed with two tests. The first evaluated whether any statistics readings were lost, that is, reported by a query

but not handled by the framework. The second measured the processing latency, that is, the time between the query reporting a statistic to it being processed by the framework and visible in the GUI.

7.3.1 Reliability

Graphite was considered reliable at storing and reporting metrics so this evaluation was performed only for the Liebre CSV statistics. To measure the reliability, the framework's code was modified to print the total number of received data points for each CSV file to the IDE console. The test was performed as follows:

1. Create and start a Liebre query, reporting its statistics to CSV files every second.
2. Visualise this query in the GUI and open the statistics window.
3. Run the test for a few minutes and then stop the query.
4. From the console logs, compare if the number of received tuples is the same as the number of lines in the CSV file.

This test was run twice and the result is shown in Table 7.3. In the first test the query and the framework started at the same time, that is, the CSV files were empty when the framework started to read them. In the second the framework was started one minute after the query to see if it read all ”old” values. As can be seen, the framework is successful in reading all statistics.

Table 7.3: Counting the number of received statistic values compared to how many have been written to a CSV file.

#	Statistics received	# of lines in CSV file
1	234	234
2	195	195

7.3.2 Processing latency

The second test measured the processing latency for both Liebre CSV statistics and for Graphite statistics.

Liebre

To measure the latency for reading statistics from CSV files, some modifications were made to the framework code. First, the `handle(String line)` method of the `Tailer` adapter was updated to print the received line, filename and the current system time to the IDE console. Then, when the data has been sent to the correct statistics tab and added to a graph, the framework prints similar data. The time

difference between two prints is the processing latency. With these changes, a query was started and the GUI was setup to listen to the created CSV files. The tests were run for one minute and the minimum, average, median and max values were calculated and the results are shown in Table 7.4, with values in milliseconds. The first test is for a query with three operators and ten files. The second and third is for the weather query with eight operators and 30 files.

Table 7.4: Measuring the Liebre CSV statistics latency for different streaming applications.

#	Min	Max	Median	Average	Readings
1	0,16	17,49	0,26	0,92	60
2	0,05	13,83	0,30	0,52	60
3	0,14	5,49	0,29	0,98	60

The results show that the processing latency is less than one millisecond in most cases. But, as described in Section 6.4.1, the `Tailer` instances were configured to check for file changes once every 500 milliseconds and in the worst case this adds another 500 milliseconds to the latency. However, this only results in the statistics being visible an instance later for the user.

Graphite

To evaluate the Graphite statistics, six tests were performed and both the network latency and the total latency were measured. Each test consisted of a Liebre query with three operators reporting statistics to Graphite every second. The total latency is the time from that the user clicks the *Go* button in the GUI to the returned data are displayed. The network latency is the time between querying Graphite to getting a result back. Three Graphite queries were used. Test one and two used `liebre.testing.map1.EXEC.mean` (a single statistic), test three and four used `liebre.testing.map1.EXEC.*` (eleven statistics) and test five and six used `liebre.testing.*.EXEC.*` ($3 * 11 = 33$ statistics). The first test in each pair queried for the last 60 seconds of statistics and the second queried for all statistics in the last 15 minutes. Table 7.5 presents the results with values in milliseconds.

As can be seen, the framework is very fast for small queries but once a query contains several thousand data points it slows down drastically. The network latency grows slower than the total latency so the main bottleneck is the framework. To investigate the cause of the slowdown, test seven, presented in Table 7.6, shows the result of performing test six without sending the received data to the graph library (ExtJFX). Comparing test seven to test six, it can be seen that the network latency is roughly the same but the total latency dropped substantially.

7.4 Performance

To evaluate the framework's performance and potential impact on the system it runs on, several tests were performed and the the CPU and memory usages were

Table 7.5: Measuring the statistics latency for different Graphite queries.

	Total latency				Network latency				
#	**Min**	**Max**	**Median**	**Avg**	**Min**	**Max**	**Median**	**Avg**	**Data points**
1	30	42	39	38	28	40	37	36	60
2	62	95	74	76	30	46	40	40	899
3	162	250	221	214	110	203	186	181	650
4	976	4768	3060	2704	180	255	210	214	9890
5	600	800	696	683	423	555	521	506	1950
6	6325	15685	13091	10837	594	630	615	609	29700

Table 7.6: Measuring the statistics latency for test six without the graph.

	Total latency				Network latency				
#	**Min**	**Max**	**Median**	**Avg**	**Min**	**Max**	**Median**	**Avg**	**data points**
7	539	726	631	613	512	697	600	584	29700

measured. The following tests ran on a Windows 10 desktop computer with an Intel i7 4790K (4 GHz) four core, eight thread processor and 16 GB 1600 MHz DDR3 RAM. To be certain that the IDE did not limit the framework's performance it was compiled into, and run as, a standalone JAR file. Between each test the JAR file was restarted to release any memory used.

Twelve tests were performed and in each the current CPU and memory usages were printed to the console every second.

1 **Idle at startup** - This test measured the idle CPU and memory usage of the framework over a minutes time. It was measured directly after selecting an SPE when the main scene is displayed.

2-3 **Visualising the weather data example** - This test measured the performance when visualising the weather example. Test #2 is for Liebre and #3 for Flink. The measuring started with an empty main scene, from that the *File* menu item was pressed and ended two seconds after the visualisation was finished. To mitigate outliers this test was performed five times for each SPE.

4-5 **Idle after visualising** - This test measured the idle CPU and memory usage of the framework over a minutes time. It was measured directly after the query from test 2-3 had been visualised.

6 **Code generation** - This test measured the performance when generating the code for the weather example. It was run only for Liebre as both SPEs have a very similar process when generating code. The test started when the *Generate* button was clicked and ended as soon as the code generation completed.

7-8 **Statistics from Liebre** - This test measured the performance of reporting live Liebre statistics of the weather query's CSV files. Test 7 was for initially empty CSV files and test 8 started reading from the beginning of files that the query had been filling for an hour (one line per second gives 3600 lines per file). With eight operators and seven streams there was a total of 30 files being listened to. The test ran for one minute once the window with graphs and tabs had loaded properly. For test 8, a total of 107 thousand lines were read.

9-10 **Statistics from Liebre (no graph)** - This test is like test 8-9 but without giving data to the ExtJFX graph library. All statistics were kept in memory but no series or data points were shown in the graph with the purpose of measuring the library's impact on the performance.

11-12 **Loading the statistics window for test 8 and 10** - This test measured the performance while the statistics window, with tabs and graphs, was loading. The test started when the window opened and ended when the first data point in a graph was visible. This test measured the time taken before any data was displayed.

Table 7.7 presents the results. Test 1 shows that some CPU power is used even when the framework is idle. For test 2-3, a low average CPU usage is observed but the max reading is higher. However, visualising a Java file takes roughly 200 milliseconds and as shown by test 4-5 the CPU usage is lower once it completes. Visualising the same file again only takes around 30 milliseconds, most likely due to the file being cached. Test 4-5 also shows a higher CPU and memory usage compare to test 1, as is expected when more data is held by the framework and the graph items are recomputed each frame. Test 6 shows a higher CPU usage than the previous tests but as the code generation is fast it only has a single reading and no average could be taken. Generating the code and writing the file took 180 milliseconds.

Test 7 shows that the framework needs around 4% CPU power and 100 MB memory to display a small amount of CSV statistics. Test 8 however is very different. While the average CPU usage was only 6% the memory usage was significantly higher. The query created 30 files with roughly 3600 lines each, totalling up to 108 thousand lines to be read and parsed. Evaluating with test 11, it took 529 readings before the file had been processed and loaded completely. One reading per second gives a total time of around nine minutes. As loading the data takes place on the UI thread the GUI was not responding during this time. The memory usage peaked at 3,5 GB.

Moving to test 9-10, where no data was sent to the graph library, the CPU usage is roughly the same as in test 7-8 but the memory usage is lower, especially for test 10. For test 12, only eleven readings (eleven seconds) were taken before for the window had loaded, a big improvement compared to nine minutes in test 11. With this information, and with the data from Table 7.6, it is concluded that the graph library is the framework's major performance bottleneck.

Table 7.7: CPU and RAM usage for the different tests.

#	Test	Avg CPU (max)	Avg RAM (max)	Readings
1	Idle at startup	0,4 (1,0) %	22,4 (25) MB	60
2	Visualising the weather data example (Liebre)	6,5 (40,8) %	32,6 (53) MB	83
3	Visualising the weather data example (Flink)	6,8 (42,3) %	31,5 (50) MB	79
4	Idle after visualising (Liebre)	4,5 (18,5) %	39,5 (60) MB	60
5	Idle after visualising (Flink)	4,6 (10,2) %	39,0 (60) MB	60
6	Code generation of the weather data example	29,5 (29,5) %	34,0 (34) MB	1
7	Statistics from Liebre (small files)	4,3 (19,1) %	94,2 (160) MB	60
8	Statistics from Liebre (big files)	6,2 (31,9) %	3022,0 (3458) MB	60
9	Statistics from Liebre (small files, no graph)	2,1 (16,9) %	63,8 (81) MB	60
10	Statistics from Liebre (big files, no graph)	6,2 (39,3) %	112,3 (160) MB	60
11	Loading the statistics window for test 8	15,3 (69,5) %	1389,6 (3317) MB	529
12	Loading the statistics window for test 10	24,3 (66,7) %	121,1 (135) MB	11

7.5 Generality

This section evaluates the generality of the framework in terms of the amount of work required to support a new SPE.

7.5.1 Application design and code generation

Designing a query and generating its code is largely supported by the JSON files. To support a new SPE, a new JSON file containing all its operator and visualisation information needs to be created. Then, a new Java class extending `ParsedSPE` needs to be added, overriding the `generateCodeFrom()` method. Remaining steps include adding the SPE to the launcher window's dropdown list and, for text auto completion, a `.txt` file needs to be added with all the SPE's public classes.

7.5.2 Statistics

If the SPE supports sending metrics to graphite no additional changes are required. Otherwise the framework needs to be extended to support reading statistics from the SPE directly, such as from CSV files in Liebre, and a button needs to be added to the main window to view its statistics.

7.5.3 Visualisation

Out of these three points, the visualisation functionality requires the most work. If the new SPE has a code style like Liebre or Flink, the steps from Section 5.3 can be applied and some code reused. Otherwise, a whole new SPE-specific implementation needs to be written. As an example, the current implementation consists of 177 Lines of Code (LOC) that are shared between both Liebre and Flink while they have 120 and 257 LOC respectively that are unique to each SPE.

7.6 Discussion

This section discusses the created framework and the results of the evaluation with a comparison to the initial goals.

7.6.1 The GUI

The framework was created as a standalone desktop application. As comparison, a web application usable with a regular web browser has the benefits of reaching more users and does not require downloading and executing an application. Further, to visualise a streaming application the user could simply copy Java code and paste it in the web application to visualise it directly. Also, the application design and code generation can be done ”online” and if the SPE JSON file needs updating it is simply a matter of updating it on the web server and all users instantly gets the updated version. For a desktop application, this requires the download of an updated version. However, in this project a desktop application has the benefit that it is possible to directly connect to a locally running SPE to fetch statistics from it, such as reading Liebre's CSV files. A web application would be limited by the performance of the web browser running it and it is difficult for it to access local files. If needed, a JavaFX application can be converted to run on a web server compatible with regular browsers, for example using [65].

Some aspects of the GUI could be improved in terms of user friendliness. For instance, when writing code for an operator the current implementation uses a simple text area which is not ideal when writing Java code. There are only a few JavaFX libraries that can provide source code styled editors [66], however they do not provide code analysis or type checking but only syntax highlighting (colouring a method name blue, for example).

Since the graph library (SmartGraphFX) displays all nodes as circles, the nodes have been coloured to differentiate between the three operator types. However, for users with colour blindness this does not help, and a better visual feature is

to have the different types as different shapes. But, since the library uses custom classes that extend the JavaFX `Circle` class this is not possible without a major modification to the library. Because of this, and that it offers no improvement in terms of functionality, other features have been prioritised.

7.6.2 Query visualisation

The evaluation shows that the query visualisation functionality performs well for queries in both Liebre and Flink. Almost all tests were completely successful in terms of finding all the operators and connecting them in the right order. There were no tests where non-operators were visualised or streams went in the wrong direction or to the wrong operator. However, not all tests were visualised correctly. Tests 6 and 8 showed that it is difficult to cover all cases as code can be written in so many ways. Tests 2, 4 and 5 were only partially correct with the reason that not all operators have been added to the JSON files and that the visualisation implementation has not been extended to cover all cases and operators.

A limitation is that only a single `.java` file can be visualised at a time. An extension can be to make it possible for the user to select a directory instead of a single file, and the framework can then search through all files in the directory for queries and visualise them all at once. It does also not check if the selected file contains a query before visualising it. Because of this, the framework can create a new empty tab without any nodes in it.

Further, the framework has not been tested for Java files containing, perhaps unusual, multiple queries. This is expected to work if the query variables have different names, as that is what links them together. The framework does not consider in which method or Java class the operators are in so if two queries are in two different methods but share some common variable names the framework will, most likely, create an invalid DAG.

Lastly, the framework does not allow a visualised DAG to be modified and converted back into a new Java query. It is possible to add new nodes and add and remove edges in the GUI but the code for the visualised operators cannot be edited. As such, it is also not possible to generate new Java code given a visualised graph. The reason is that the framework does not convert the found Java code into a ”real” node in the graph, i.e. a node with data from the JSON file, but simply creates an empty node containing information about the found class data.

7.6.3 Application design and code generation

The framework supports a small set of streaming operators, but the use of JSON files to store information makes it easy to add new, or modify already existing, operators, without editing the framework's Java code. With the `"before"`, `"middle"` and `"after"` fields it is possible to define what parts of an operator's code the user can edit and what is ”static”. This works well if an operator has a simple definition for which the user can write the code for. But for more complex ones, such as a Window operator in Liebre that requires a custom class, it becomes more difficult. For instance, it is not possible to wholly define a Window operator in the GUI as

only the operator's method can be modified and not the code surrounding it, that is, the `public static void main()` method or the surrounding class.

It is possible to design DAGs for many different applications and several checks prevent the user from designing invalid graphs. For example, the framework prevents streams from being created in the wrong direction, connecting operator types that should not be connected (like a stream from a sink to a source), prevents cycles and the *Check graph* button allows for additional audit. However, it does not check the created operators for errors such as Java syntax errors, if a specified input value for an operator is a valid Java class or if two operators can be connected.

With the JSON files containing the required imports for the different operators and their code structure, the generated code is in most cases executable directly. But this requires that the user has written code for the operators in the GUI as only a code skeleton is generated without it.

7.6.4 Performance reporting

Besides from a slow and CPU-heavy chart library the statistics reporting meets the initial goal. It is possible to reliably watch statistics for both Liebre and Flink for the smaller queries that were tested. Its reliability and performance with large scale applications and queries are unknown but given the evaluation it is likely not optimal. For Liebre a user can view both live and offline statistics. For Flink there was not enough time to implement ”direct” statistics using the Flink API and it now relies on Graphite. This has the drawback of requiring the user to understand the Graphite query structure where for the Liebre statistics the user can simply switch between the different tabs and view the metrics directly.

The project was not successful with providing a common interface to statistics. For instance, a user still needs to know the separate Liebre and Flink Graphite queries and, for example, there is no explicit ”Latency” button, that when clicked retrieves and displays the latency for the operators. An improvement in terms of user friendliness is the ability to zoom and to pan, that is, inspect specific points in the chart. Currently, if there are a lot of data points some may overlap, making it difficult to hover the mouse over the chart to inspect a particular one.

Both the latency and the reliability tests gave positive signs that the framework is fast in displaying live statistics. Due to the current solution being sub-optimal, for larger Graphite queries it is slow and has a high latency. For instance, all the statistics (data points) are kept in memory and are used in the graph library at all times, even when many points are not visible in the selected time range. A modification so that only the data points in the visible time range are processed would be a great improvement.

The evaluation shows that the CPU usage is around 2-6% when viewing live Liebre statistics. Such a low cost is acceptable but for big CSV files the long loading times and high memory usage is not. The main cause for this is the chart library and its data structure. Looking into the library it was found that the underlying list, that stores the data points, is an `ArrayList`. Such a list is very inefficient when mainly adding values because when the list is full it needs to be copied into a new list of bigger size. It is not possible to change this list to, for example, a

`LinkedList` that is faster when adding many values as the graph library relies on the `ObservableArrayList` from the JavaFX library. Another issue is that for each frame it recalculates which data points are visible in the selected time-range by iterating over all the points and comparing their values with the X and Y axis bounds. Changing to another library would most likely drastically improve the performance and lower the CPU usage, resulting in a smaller impact on other applications running on the same system and a less frustrated user.

7.6.5 Generality

The generality of the code is high for the application design and code generation functionalities, meaning that not much work is required to support a new SPE. However, the query visualisation functionality is the major obstacle as it requires a more detailed Java implementation.

The framework is not limited to work with just SPEs written in Java. The JSON files support any kind of programming language, but the code generation and query visualisation implementations need to be adapted for the new language. For instance, JavaParser is used for visualisation, but it only works with Java code.

7.6.6 Ethical aspects, open source and license

There are few ethical aspects to consider for this project. The code for the framework is open source [55] and can easily be inspected to check that it does not contain any malware or other damaging code. Finally, all the source code is licensed under the MIT license, making it possible for others to use and build upon it.

8
Conclusion

Several SPEs have been developed in the recent years for the task of processing large amounts of data with low latency and high throughput. However, they all target different use-cases and come with their own implementations and functionalities. This thesis aimed to ease the process of creating, visualising, and understanding the performance of streaming applications by designing and implementing a framework in the form of a GUI that can unify different SPEs under a single interface. The goals were to let a user visualise queries from existing Java files, design queries entirely in the framework and get the corresponding code generated automatically, and view both live and offline statistics for different queries and SPEs.

By using Java, JavaFX, and several open-source libraries a framework was created that lets a user perform these tasks. In this project the two SPEs Apache Flink and Liebre were added and supported, and the framework's functionality, performance and generality were evaluated.

With the use of JSON files to store information about the SPEs and supported operators a user can visually design queries, consisting of many different operators, specify how they work and connect them using streams to form a DAG. The framework generates correct and compilable Java code provided that the user has written code defining the method body for each operator. If no code is written in the GUI only a skeleton of the query is generated. Using JavaParser, the framework visualises queries from Java files as DAGs in the GUI. In most cases the visualised DAGs are correct and match the ones in code. However, queries containing certain operators, such as a Union or Join operator, did not always visualise correctly, as seen in the evaluation. Statistics reporting of queries using Liebre CSV files or Graphite is reliable and fast for the small-scale queries designed and used in this thesis. For larger queries, the framework was found to be slow and further development and optimisation is needed. Live and offline statistics are supported for Liebre's CSV files, and by fetching data from Graphite additional metrics can be seen for both SPEs.

This thesis has shown that it is possible to create the basis for a unified framework for different SPEs. By designing the framework as general as possible it is in most areas relatively simple to support additional SPEs. The outcome is a tool for developers to use, enabling for increased productivity and better understanding while working with both new and already existing streaming applications.

Bibliography

[1] IDC. (2020). "IDC's global datasphere forecast shows continued steady growth in the creation and consumption of data," [Online]. Available: https://www.idc.com/getdoc.jsp?containerId=prUS46286020 (visited on 03/18/2021).

[2] Raconteur. (2021). "A day in data," [Online]. Available: https://www.raconteur.net/infographics/a-day-in-data/ (visited on 03/18/2021).

[3] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic scaling for data stream processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1447–1463, 2013.

[4] R. Stephens, "A survey of stream processing," *Acta Informatica*, vol. 34, no. 7, pp. 491–541, 1997.

[5] J. S. Van Der Veen, B. Van Der Waaij, E. Lazovik, W. Wijbrandi, and R. J. Meijer, "Dynamically scaling apache storm for the analysis of streaming data," in *2015 IEEE First International Conference on Big Data Computing Service and Applications*, IEEE, 2015, pp. 154–161.

[6] H. Karau and R. Warren, *High performance Spark: best practices for scaling and optimizing Apache Spark.* " O'Reilly Media, Inc.", 2017.

[7] A. Katsifodimos and S. Schelter, "Apache flink: Stream analytics at scale," in *2016 IEEE international conference on cloud engineering workshop (IC2EW)*, IEEE, 2016, pp. 193–193.

[8] D. Battulga, D. Miorandi, and C. Tedeschi, "Fogguru: A fog computing platform based on apache flink," in *2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, IEEE, 2020, pp. 156–158.

[9] D. Palyvos-Giannas, V. Gulisano, and M. Papatriantafilou, "Genealog: Fine-grained data streaming provenance at the edge," in *Proceedings of the 19th International Middleware Conference*, 2018, pp. 227–238.

[10] D. Palyvos-Giannas, V. Gulisano, and M. Papatriantafilou, "Haren: A middleware for ad-hoc thread scheduling policies in data streaming," in *Proceedings of the 20th International Middleware Conference Demos and Posters*, 2019, pp. 19–20.

[11] J. Evermann, J.-R. Rehse, and P. Fettke, "Process discovery from event stream data in the cloud-a scalable, distributed implementation of the flexible heuristics miner on the amazon kinesis cloud infrastructure," in *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, IEEE, 2016, pp. 645–652.

[12] The Apache Software Foundation. (2021). "Flink plan visualizer," [Online]. Available: https://flink.apache.org/visualizer/ (visited on 02/09/2021).

[13] Hortonworks. (2021). "Working with Storm Topologies," [Online]. Available: https://docs.cloudera.com/HDPDocuments/HDF3/HDF-3.4.0/storm-topologies/content/tuning_an_apache_storm_topology.html (visited on 03/22/2021).

[14] R. Ramamonjison. (2021). "The graph visualization," [Online]. Available: https://www.oreilly.com/library/view/apache-spark-graph/9781784391805/ch03s02.html (visited on 03/22/2021).

[15] The Apache Software Foundation. (2021). "Project configuration," [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-stable/dev/project-configuration.html (visited on 03/23/2021).

[16] P. Newman. (2021). "The internet of things 2020," [Online]. Available: https://www.businessinsider.com/internet-of-things-report (visited on 03/25/2021).

[17] R. Tönjes, P. Barnaghi, M. Ali, A. Mileo, M. Hauswirth, F. Ganz, S. Ganea, B. Kjærgaard, D. Kuemper, S. Nechifor, *et al.*, "Real time iot stream processing and large-scale data analytics for smart city applications," in *poster session, European Conference on Networks and Communications*, sn, 2014, p. 10.

[18] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *ACM Sigmod Record*, vol. 34, no. 4, pp. 42–47, 2005.

[19] W. Wingerath, F. Gessert, S. Friedrich, and N. Ritter, "Real-time stream processing for big data," *it - Information Technology*, vol. 58, no. 4, pp. 186–194, 2016. DOI: doi:10.1515/itit-2016-0002. [Online]. Available: https://doi.org/10.1515/itit-2016-0002.

[20] I. Hazelcast. (2021). "What is stream processing?" [Online]. Available: https://hazelcast.com/glossary/stream-processing/ (visited on 03/25/2021).

[21] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[22] D. Namiot, "On big data stream processing," *International Journal of Open Information Technologies*, vol. 3, no. 8, 2015.

[23] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A catalog of stream processing optimizations," *ACM Computing Surveys (CSUR)*, vol. 46, no. 4, pp. 1–34, 2014.

[24] F. Gutierrez, K. Beedkar, A. Souza, and V. Markl, "AdCom: Adaptive Combiner for Streaming Aggregations," in *EDBT 2021 - 24th International Conference on Extending Database Technology*, Nicosia, Cyprus, Mar. 2021. [Online]. Available: https://hal.archives-ouvertes.fr/hal-03156337.

[25] R. Sahal, J. G. Breslin, and M. I. Ali, "Big data and stream processing platforms for industry 4.0 requirements mapping for a predictive maintenance use case," *Journal of manufacturing systems*, vol. 54, pp. 138–151, 2020.

[26] D. Palyvos-Giannas, B. Havers, M. Papatriantafilou, and V. Gulisano, "Ananke: A streaming framework for live forward provenance," *Proceedings of the VLDB Endowment*, vol. 14, no. 3, pp. 391–403, 2020.

[27] M. H. Javed, X. Lu, and D. K. Panda, "Cutting the tail: Designing high performance message brokers to reduce tail latencies in stream processing," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, 2018, pp. 223–233. DOI: 10.1109/CLUSTER.2018.00040.

[28] H. He. (2019). "Storage: How "tail latency" impacts customer-facing applications," [Online]. Available: https://www.computerweekly.com/opinion/Storage-How-tail-latency-impacts-customer-facing-applications (visited on 04/26/2021).

[29] M. Zhang. (2021). "Awesome Streaming," [Online]. Available: https://manuzhang.github.io/awesome-streaming/ (visited on 04/26/2021).

[30] The Apache Software Foundation. (2019). "StratosphereProposal," [Online]. Available: https://cwiki.apache.org/confluence/display/incubator/StratosphereProposal (visited on 05/02/2021).

[31] The Apache Software Foundation. (2015). "The Apache Software Foundation Announces Apache Flink as a Top-Level Project," [Online]. Available: https://blogs.apache.org/foundation/entry/the_apache_software_foundation_announces69 (visited on 05/02/2021).

[32] The Apache Software Foundation. (2021). "Powered by flink," [Online]. Available: https://cwiki.apache.org/confluence/display/FLINK/Powered+by+Flink (visited on 02/09/2021).

[33] D. García-Gil, S. Ramírez-Gallego, S. García, and F. Herrera, "A comparison on scalability for batch big data processing on Apache Spark and Apache Flink," *Big Data Analytics*, vol. 2, no. 1, pp. 1–11, 2017.

[34] R. Sahal, M. H. Khafagy, and F. A. Omara, "Big data multi-query optimisation with apache flink," *International Journal of Web Engineering and Technology*, vol. 13, no. 1, pp. 78–97, 2018.

[35] The Apache Software Foundation. (2021). "Apache flink," [Online]. Available: https://github.com/apache/flink (visited on 02/18/2021).

[36] The Apache Software Foundation. (2021). "What is Apache Flink? — Applications," [Online]. Available: https://flink.apache.org/flink-applications.html#building-blocks-for-streaming-applications (visited on 05/02/2021).

[37] The Apache Software Foundation. (2021). "Parallel Execution," [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-stable/dev/parallel.html (visited on 05/02/2021).

[38] V. Gulisano. (2021). "The Liebre Stream Processing Engine," [Online]. Available: https://vincenzogulisano.com/2017/07/09/the-liebre-stream-processing-engine/ (visited on 04/26/2021).

[39] Graphite. (2021). "Graphite," [Online]. Available: https://graphiteapp.org/ (visited on 05/11/2021).

[40] JavaParser.org. (2020). "Javaparser," [Online]. Available: https://javaparser.org/ (visited on 12/01/2020).

[41] N. Smith, D. van Bruggen, and F. Tomassetti, *JavaParser: Visited*. Leanpub, 2021.

[42] The Apache Software Foundation. (2021). "SQL," [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/dev/table/sql/overview/ (visited on 05/24/2021).

[43] M. Pathirage, J. Hyde, Y. Pan, and B. Plale, "Samzasql: Scalable fast data management with streaming sql," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 1627–1636. DOI: 10.1109/IPDPSW.2016.141.

[44] The Apache Software Foundation. (2021). "Metrics Reporter," [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/deployment/metric_reporters/ (visited on 05/11/2021).

[45] The Apache Software Foundation. (2021). "Operators," [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/stream/operators/ (visited on 04/05/2021).

[46] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo, "Spade: The system s declarative stream processing engine," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08, Vancouver, Canada: Association for Computing Machinery, 2008, pp. 1123–1134, ISBN: 9781605581026. DOI: 10.1145/1376616.1376729. [Online]. Available: https://doi.org/10.1145/1376616.1376729.

[47] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu, "Elastic scaling of data parallel operators in stream processing," in *2009 IEEE International Symposium on Parallel Distributed Processing*, 2009, pp. 1–12. DOI: 10.1109/IPDPS.2009.5161036.

[48] W. De Pauw, M. LeŢia, B. Gedik, H. Andrade, A. Frenkiel, M. Pfeifer, D. Sow, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. Pace, G. Roşu, O. Sokolsky, and N. Tillmann, "Visual debugging for stream processing applications," in *Runtime Verification*, H. Barringer, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 18–35, ISBN: 978-3-642-16612-9.

[49] V. Gulisano, "StreamCloud: an elastic parallel-distributed stream processing engine," Ph.D. dissertation, Universidad Politécnica de Madrid, 2012.

[50] S. Zunke and V. D'Souza, "Json vs xml: A comparative performance analysis of data exchange formats," *Int J Comput Sci Netw*, vol. 3, no. 4, pp. 257–261, 2014.

[51] The Apache Software Foundation. (2021). "Apache Beam Overview," [Online]. Available: https://beam.apache.org/get-started/beam-overview/ (visited on 05/10/2021).

[52] G. Hesse, C. Matthies, K. Glass, J. Huegle, and M. Uflacker, "Quantitative impact evaluation of an abstraction layer for data stream processing systems," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, IEEE, 2019, pp. 1381–1392.

[53] A. Satyanarayan, R. Russell, J. Hoffswell, and J. Heer, "Reactive vega: A streaming dataflow architecture for declarative interactive visualization," *IEEE transactions on visualization and computer graphics*, vol. 22, no. 1, pp. 659–668, 2015.

[54] A. B. Kahn, "Topological sorting of large networks," *Commun. ACM*, vol. 5, no. 11, pp. 558–562, Nov. 1962, ISSN: 0001-0782. DOI: 10.1145/368996.369025. [Online]. Available: https://doi.org/10.1145/368996.369025.

[55] R. Teodorsson. (2021). "SPE-GUI," [Online]. Available: https://github.com/hej2010/SPE-GUI (visited on 05/04/2021).

[56] Oracle Corporation. (2021). "Openjfx project," [Online]. Available: https://openjdk.java.net/projects/openjfx/ (visited on 03/22/2021).

[57] GitHub. (2021). "JavaFX libraries," [Online]. Available: https://github.com/search?q=javafx (visited on 03/22/2021).

[58] Gluon. (2021). "Scene builder," [Online]. Available: https://gluonhq.com/products/scene-builder/ (visited on 02/10/2021).

[59] B. Silva and R. Teodorsson. (2021). "JavaFXSmartGraph," [Online]. Available: https://github.com/hej2010/JavaFXSmartGraph (visited on 04/16/2021).

[60] C. Brinkman and F. Ochmann. (2021). "Javafx textfield auto-suggestions," [Online]. Available: https://stackoverflow.com/a/56173327/7232269 (visited on 03/29/2021).

[61] Google Inc. (2021). "google-java-format," [Online]. Available: https://github.com/google/google-java-format (visited on 05/06/2021).

[62] G. Kruk and CERN. (2021). "ExtJFX," [Online]. Available: https://github.com/extjfx/extjfx (visited on 05/14/2021).

[63] The Apache Software Foundation. (2021). "Commons IO," [Online]. Available: https://commons.apache.org/proper/commons-io/ (visited on 05/14/2021).

[64] G. Mencagli. (2021). "StreamBenchmarks," [Online]. Available: https://github.com/ParaGroup/StreamBenchmarks (visited on 05/17/2021).

[65] Sandec GmbH. (2021). "JPro," [Online]. Available: https://www.jpro.one/ (visited on 05/23/2021).

[66] T. Mikula, J. Martinez, and et al. (2021). "RichTextFX," [Online]. Available: https://github.com/FXMisc/RichTextFX#codearea (visited on 05/23/2021).

A

Code listings

A.1 Weather example - Flink

The Apache Flink Java code for the streaming application example introduced in Section 2.1.1.

```
1  import org.apache.flink.api.common.functions.FilterFunction;
2  import org.apache.flink.api.common.functions.MapFunction;
3  import org.apache.flink.api.java.functions.KeySelector;
4  import org.apache.flink.api.java.tuple.Tuple2;
5  import org.apache.flink.streaming.api.datastream.DataStream;
6  import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
7  import org.apache.flink.streaming.api.functions.sink.SinkFunction;
8  import org.apache.flink.streaming.api.functions.source.SourceFunction;
9  import
   ↪  org.apache.flink.streaming.api.windowing.assigners.TumblingProcessingTimeWindows;
10 import org.apache.flink.streaming.api.windowing.time.Time;
11
12 public class FlinkApplication {
13     private static final String[] CITIES = {"Gothenburg", "Stockholm",
       ↪  "Malmö", "Lund", "Luleå"};
14
15     public static void main(String[] args) {
16         final StreamExecutionEnvironment query =
           ↪  StreamExecutionEnvironment.getExecutionEnvironment();
17
18         DataStream<String> sourceStream = query.addSource(new
           ↪  SourceFunction<>() {
19             @Override
20             public void run(SourceContext<String> ctx) {
21                 ctx.collect(System.currentTimeMillis() + "," + CITIES[(int)
                   ↪  (Math.random() * CITIES.length)] + "," +
22                         Math.random() * 100 + "," + (Math.random() * 50 -
                           ↪  20));
23             }
24
25             @Override
26             public void cancel() {
27             }
28         });
29
30         DataStream<WeatherData> mapStream =
           ↪  sourceStream.map((MapFunction<String, WeatherData>) value -> {
31             String[] s = value.split(",");
```

```
32              return new WeatherData(Long.parseLong(s[0]), s[1],
                ↪  Double.parseDouble(s[2]), Double.parseDouble(s[3]));
33          });
34
35          DataStream<WeatherData> aggregated = mapStream
36                  .keyBy((KeySelector<WeatherData, String>)
                    ↪  WeatherData::getCity)
37                  .window(TumblingProcessingTimeWindows.of(Time.days(1)))
38                  .max("temp")
39                  .setParallelism(1);
40
41          DataStream<Tuple2<Long, Double>> filtered = mapStream
42                  .filter((FilterFunction<WeatherData>) value ->
                    ↪  value.getCity().equalsIgnoreCase(CITIES[0]) &&
                    ↪  value.getTemp() > 21)
43                  .map(new MapFunction<WeatherData, Tuple2<Long, Double>>() {
44                      @Override
45                      public Tuple2<Long, Double> map(WeatherData value) {
46                          return new Tuple2<>(value.timestamp, value.temp);
47                      }
48                  });
49
50          aggregated.addSink(new SinkFunction<>() {
51              @Override
52              public void invoke(WeatherData value, Context context) { // Max
                ↪  temperature reading for each city each day
53                  System.out.println("Max temp for " + value.city + " at " +
                    ↪  value.timestamp + " is " + value.temp);
54              }
55          });
56          filtered.addSink(new SinkFunction<>() {
57              @Override
58              public void invoke(Tuple2<Long, Double> value, Context context) {
                ↪  // Temperature reading for Gothenburg above 21C
59                  System.out.println("Temp for " + CITIES[0] + " at " +
                    ↪  value.f0 + " is " + value.f1);
60              }
61          });
62
63          System.out.println(query.getExecutionPlan());
64      }
65
66      public static class WeatherData {
67          private long timestamp;
68          private double temp, humidity;
69          private String city;
70
71          public WeatherData(long timestamp, String city, double humidity,
            ↪  double temp) {
72              this.timestamp = timestamp;
73              this.city = city;
74              this.humidity = humidity;
75              this.temp = temp;
76          }
77
78          public WeatherData() {
```

```
            this(System.currentTimeMillis(), CITIES[(int) (Math.random() *
            ↪  CITIES.length)],
                    Math.random() * 100, (Math.random() * 50 - 20));
        }

        @Override
        public String toString() {
            return "TempData{" +
                    "timestamp=" + timestamp +
                    ", temp=" + temp +
                    ", humidity=" + humidity +
                    ", city='" + city + '\'' +
                    '}';
        }

        public long getTimestamp() {
            return timestamp;
        }

        public void setTimestamp(long timestamp) {
            this.timestamp = timestamp;
        }

        public double getTemp() {
            return temp;
        }

        public void setTemp(double temp) {
            this.temp = temp;
        }

        public double getHumidity() {
            return humidity;
        }

        public void setHumidity(double humidity) {
            this.humidity = humidity;
        }

        public String getCity() {
            return city;
        }

        public void setCity(String city) {
            this.city = city;
        }
    }
}
```

A.2 Weather example - Liebre

The Liebre Java code for weather example.

```
import com.codahale.metrics.CsvReporter;
import common.metrics.Metrics;
```

```
3   import common.tuple.BaseRichTuple;
4   import component.operator.Operator;
5   import component.operator.in1.aggregate.BaseTimeBasedSingleWindow;
6   import component.operator.in1.aggregate.TimeBasedSingleWindow;
7   import component.operator.router.RouterOperator;
8   import component.sink.Sink;
9   import component.source.Source;
10  import org.apache.flink.api.java.tuple.Tuple2;
11  import query.LiebreContext;
12  import query.Query;
13
14  import java.nio.file.Paths;
15  import java.util.concurrent.TimeUnit;
16
17  public class TestLiebreWeather {
18      private static final String[] CITIES = {"Gothenburg", "Stockholm",
        ↪  "Malmö", "Lund", "Luleå"};
19
20      public static void main(String[] args) {
21          LiebreContext.setOperatorMetrics(Metrics.dropWizard());
22          LiebreContext.setUserMetrics(Metrics.dropWizard());
23          LiebreContext.setStreamMetrics(Metrics.dropWizard());
24          CsvReporter csvReporter =
            ↪  CsvReporter.forRegistry(Metrics.metricRegistry())
25                  .build(Paths.get("./metrics").toFile());
26          csvReporter.start(1, TimeUnit.SECONDS);
27
28          final Query query = new Query();
29          Source<String> source = query.addBaseSource("src1", () -> {
30              return System.currentTimeMillis() + "," + CITIES[(int)
                ↪  (Math.random() * CITIES.length)] + "," +
31                      Math.random() * 100 + "," + (Math.random() * 50 - 20);
32          });
33
34          Operator<String, WeatherData> map1 = query.addMapOperator("map1",
            ↪  value -> {
35              String[] s = value.split(",");
36              return new WeatherData(Long.parseLong(s[0]), s[1],
                ↪  Double.parseDouble(s[2]), Double.parseDouble(s[3]));
37          });
38
39          RouterOperator<WeatherData> router =
            ↪  query.addRouterOperator("router");
40          Operator<WeatherData, WeatherData> agg1 =
            ↪  query.addAggregateOperator("agg1", new MaxWindow(), 86400,
            ↪  86400);
41          Operator<WeatherData, WeatherData> filter1 =
            ↪  query.addFilterOperator("filter1", weatherData ->
            ↪  weatherData.getCity().equalsIgnoreCase(CITIES[0]) &&
            ↪  weatherData.getTemp() > 21);
42          Operator<WeatherData, Tuple2<Long, Double>> map2 =
            ↪  query.addMapOperator("map2", weatherData -> new
            ↪  Tuple2<>(weatherData.getTimestamp(), weatherData.getTemp()));
43          Sink<WeatherData> sink1 = query.addBaseSink("sink1", weatherData ->
            ↪  System.out.println("Max temp for " + weatherData.city + " at " +
            ↪  weatherData.timestamp + " is " + weatherData.temp));
```

```
44            Sink<Tuple2<Long, Double>> sink2 = query.addBaseSink("sink2", value
              ↪   -> System.out.println("Temp for " + CITIES[0] + " at " + value.f0
              ↪   + " is " + value.f1));
45
46            query.connect(source, map1).connect(map1, router);
47            query.connect(router, agg1).connect(agg1, sink1);
48            query.connect(router, filter1).connect(filter1, map2).connect(map2,
              ↪   sink2);
49
50            query.activate();
51        }
52
53        public static class WeatherData extends BaseRichTuple {
54            private long timestamp;
55            private double temp, humidity;
56            private String city;
57
58            public WeatherData(long timestamp, String city, double humidity,
              ↪   double temp) {
59                super(timestamp, city);
60                this.timestamp = timestamp;
61                this.city = city;
62                this.humidity = humidity;
63                this.temp = temp;
64            }
65
66            @Override
67            public String toString() {
68                return "TempData{" +
69                        "timestamp=" + timestamp +
70                        ", temp=" + temp +
71                        ", humidity=" + humidity +
72                        ", city='" + city + '\'' +
73                        '}';
74            }
75
76            public long getTimestamp() {
77                return timestamp;
78            }
79
80            public void setTimestamp(long timestamp) {
81                this.timestamp = timestamp;
82            }
83
84            public double getTemp() {
85                return temp;
86            }
87
88            public void setTemp(double temp) {
89                this.temp = temp;
90            }
91
92            public double getHumidity() {
93                return humidity;
94            }
95
```

```
96            public void setHumidity(double humidity) {
97                this.humidity = humidity;
98            }
99
100           public String getCity() {
101               return city;
102           }
103
104           public void setCity(String city) {
105               this.city = city;
106           }
107       }
108
109       private static class MaxWindow extends
          ↪   BaseTimeBasedSingleWindow<WeatherData, WeatherData> {
110
111           private WeatherData max = null;
112
113           @Override
114           public void add(WeatherData t) {
115               if (max != null) {
116                   if (max.getTemp() < t.getTemp()) {
117                       max = t;
118                   }
119               } else {
120                   max = t;
121               }
122           }
123
124           @Override
125           public void remove(WeatherData t) {
126               if (max == t) {
127                   max = null;
128               }
129           }
130
131           @Override
132           public WeatherData getAggregatedResult() {
133               return max;
134           }
135
136           @Override
137           public TimeBasedSingleWindow<WeatherData, WeatherData> factory() {
138               return new MaxWindow();
139           }
140       }
141   }
```

A.3 Weather example generated - Flink

The generated Apache Flink Java code for the weather example used in the evaluation.

```
1   import org.apache.flink.api.common.functions.FilterFunction;
2   import org.apache.flink.api.common.functions.MapFunction;
```

```
3   import org.apache.flink.streaming.api.datastream.DataStream;
4   import org.apache.flink.streaming.api.datastream.DataStreamSink;
5   import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
6   import org.apache.flink.streaming.api.functions.sink.SinkFunction;
7   import org.apache.flink.streaming.api.functions.source.SourceFunction;
8   import org.apache.flink.streaming.api.windowing.time.Time;
9   import org.apache.flink.streaming.api.windowing.windows.TimeWindow;
10
11  public class ApacheFlink1621330594666 {
12    public static void main(String[] args) throws Exception {
13      final StreamExecutionEnvironment env =
      ↪  StreamExecutionEnvironment.getExecutionEnvironment();
14      DataStream<String> source1 =
15          env.addSource(
16              new SourceFunction<String>() {
17                @Override
18                public void run(SourceContext<String> ctx) throws Exception {
19                  ctx.collect(
20                      System.currentTimeMillis()
21                          + ","
22                          + CITIES[(int) (Math.random() * CITIES.length)]
23                          + ","
24                          + Math.random() * 100
25                          + ","
26                          + (Math.random() * 50 - 20));
27                }
28
29                @Override
30                public void cancel() {
31                  //
32                }
33              });
34      DataStream<WeatherData> map1 =
35          source1.map(
36              (MapFunction<String, WeatherData>)
37                  value -> {
38                    String[] s = value.split(",");
39                    return new WeatherData(
40                        Long.parseLong(s[0]),
41                        s[1],
42                        Double.parseDouble(s[2]),
43                        Double.parseDouble(s[3]));
44                  });
45      KeyedStream<WeatherData, String> keyBy =
46          map1.keyBy(
47              (KeySelector<WeatherData, String>)
48                  value -> {
49                    return value.getCity();
50                  });
51      WindowedStream<WeatherData, WeatherData, TimeWindow> op5 =
52          keyBy.window(TumblingProcessingTimeWindows.of(Time.days(1)));
53      DataStream<WeatherData> op6 = op5.max("temp");
54      DataStreamSink<WeatherData> sink10 =
55          op6.addSink(
56              new SinkFunction<>() {
57                @Override
```

```
58                  public void invoke(WeatherData value, Context context) throws
                    ↪  Exception {
59                    System.out.println(
60                        "Max temp for " + value.city + " at " + value.timestamp +
                          ↪  " is " + value.temp);
61                  }
62                });
63        DataStream<WeatherData> op12 =
64            map1.filter(
65                (FilterFunction<WeatherData>)
66                    value -> {
67                      return value.getCity().equalsIgnoreCase(CITIES[0]) &&
                        ↪  value.getTemp() > 21;
68                    });
69        DataStream<Tuple2<Long, Double>> op13 =
70            op12.map(
71                (MapFunction<WeatherData, Tuple2<Long, Double>>)
72                    value -> {
73                      return new Tuple2<>(value.timestamp, value.temp);
74                    });
75        DataStreamSink<Tuple2<Long, Double>> sink14 =
76            op13.addSink(
77                new SinkFunction<>() {
78                  @Override
79                  public void invoke(Tuple2<Long, Double> value, Context context)
                    ↪  throws Exception {
80                    System.out.println("Temp for " + CITIES[0] + " at " +
                      ↪  value.f0 + " is " + value.f1);
81                  }
82                });
83
84        env.execute();
85      }
86    }
```

B

SPE JSON file

The JSON file containing information about the Liebre SPE.

```
1  {
2    "name": "Liebre",
3    "operators": {
4      "sources": [
5        "BaseSource",
6        "TextFileSource"
7      ],
8      "sinks": [
9        "BaseSink",
10       "TextFileSink"
11     ],
12     "regular": [
13       "BaseOperator1In",
14       "BaseOperator2In",
15       "Map",
16       "FlatMap",
17       "Filter",
18       "Router",
19       "Aggregate",
20       "Join"
21     ]
22   },
23   "links": {
24     "addBaseSource": "source:BaseSource",
25     "addMapOperator": "op:Map",
26     "addFlatMapOperator": "op:FlatMap",
27     "addFilterOperator": "op:Filter",
28     "addOperator": "op:BaseOperator1In",
29     "addOperator2n": "op:BaseOperator2In",
30     "addRouterOperator": "op:Router",
31     "addJoinOperator": "op:Join",
32     "addAggregateOperator": "op:Aggregate",
33     "addTextFileSource": "source:TextFileSource",
34     "addBaseSink": "sink:BaseSink",
35     "addTextFileSink": "sink:TextFileSink"
36   },
37   "imports": {
38     "base": [
39       "query.Query",
40       "component.operator.Operator"
41     ],
42     "BaseOperator1In": [
43       "component.operator.in1.BaseOperator1In"
```

```
44        ],
45        "BaseOperator2In": [
46          "component.operator.in2.BaseOperator2In"
47        ],
48        "Map": [
49          "component.operator.in1.map.MapFunction"
50        ],
51        "FlatMap": [
52          "component.operator.in1.map.FlatMapFunction"
53        ],
54        "Filter": [
55          "component.operator.in1.filter.FilterFunction"
56        ],
57        "Join": [
58          "component.operator.in2.join.JoinFunction"
59        ],
60        "Router": [
61          "component.operator.router.RouterOperator"
62        ],
63        "Aggregate": [
64          "component.operator.in1.aggregate.BaseTimeBasedSingleWindow",
65          "component.operator.in1.aggregate.TimeBasedSingleWindow"
66        ],
67        "BaseSource": [
68          "component.source.Source"
69        ],
70        "TextFileSource": [
71          "component.source.Source"
72        ],
73        "BaseSink": [
74          "component.sink.Sink"
75        ],
76        "TextFileSink": [
77          "component.sink.Sink"
78        ]
79      },
80      "definition": {
81        "base": [
82          "Query q = new Query();"
83        ],
84        "BaseOperator1In": {
85          "before": "Operator<@IN1, @OUT1> @ID = new BaseOperator1In<@IN1,
          ↪  @OUT1>(\"@ID\") {\n@Override\npublic List<@OUT1>
          ↪  processTupleIn1(@IN1 tuple) {",
86          "middle": "return null;",
87          "after": "}\n};",
88          "placeholders": {
89            "input": [
90              "@IN1"
91            ],
92            "output": [
93              "@OUT1"
94            ],
95            "identifier": "@ID"
96          }
97        },
```

```
98          "BaseOperator2In": {
99            "before": "BaseOperator2In<@IN1, @IN2, @OUT1> @ID = new
            ↪  BaseOperator2In<@IN1, @IN2, @OUT1>(\"@ID\") {",
100           "middle": "@Override\npublic List<@OUT1> processTupleIn1(@IN1 tuple)
            ↪  {\nreturn null;\n}\n\n@Override\npublic List<@OUT1>
            ↪  processTupleIn2(@IN2 tuple) {\nreturn null;\n}",
101           "after": "});",
102           "placeholders": {
103             "input": [
104               "@IN1",
105               "@IN2"
106             ],
107             "output": [
108               "@OUT1"
109             ],
110             "identifier": "@ID"
111           }
112         },
113         "Map": {
114           "before": "Operator<@IN1, @OUT1> @ID = q.addMapOperator(\"@ID\",
            ↪  (MapFunction<@IN1, @OUT1>) tuple -> {",
115           "middle": "return null;",
116           "after": "});",
117           "placeholders": {
118             "input": [
119               "@IN1"
120             ],
121             "output": [
122               "@OUT1"
123             ],
124             "identifier": "@ID"
125           }
126         },
127         "FlatMap": {
128           "before": "Operator<@IN1, @OUT1> @ID = q.addFlatMapOperator(\"@ID\",
            ↪  (FlatMapFunction<@IN1, @OUT1>) tuple -> {",
129           "middle": "List<@OUT1> list = new LinkedList<>();\nreturn list;",
130           "after": "});",
131           "placeholders": {
132             "input": [
133               "@IN1"
134             ],
135             "output": [
136               "@OUT1"
137             ],
138             "identifier": "@ID"
139           }
140         },
141         "Filter": {
142           "before": "Operator<@IN1, @IN1> @ID = q.addFilterOperator(\"@ID\",
            ↪  tuple -> {",
143           "middle": "return false;",
144           "after": "});",
145           "placeholders": {
146             "input": [
147               "@IN1"
```

```
148             ],
149             "output": [
150             ],
151             "identifier": "@ID"
152           }
153         },
154         "Join": {
155           "before": "Operator2In<@IN1, @IN1, @IN1> @ID =
            ↪  query.addJoinOperator(\"@ID\", (richTuple, richTuple2) -> {",
156           "middle": "return null;\n},\n60 // window size",
157           "after": ");",
158           "placeholders": {
159             "input": [
160               "@IN1"
161             ],
162             "output": [
163             ],
164             "identifier": "@ID"
165           }
166         },
167         "Router": {
168           "before": "Operator<@IN1, @IN1> @ID = q.addRouterOperator(\"@ID\");",
169           "middle": "",
170           "after": "",
171           "placeholders": {
172             "input": [
173               "@IN1"
174             ],
175             "output": [
176             ],
177             "identifier": "@ID"
178           }
179         },
180         "Aggregate": {
181           "before": "Operator<@IN1, @OUT1> @ID = q.addAggregateOperator(\"@ID\",
            ↪  ",
182           "middle": "null, // BaseTimeBasedSingleWindow\n60, // window size\n20,
            ↪  // window slide",
183           "after": ");",
184           "placeholders": {
185             "input": [
186               "@IN1"
187             ],
188             "output": [
189               "@OUT1"
190             ],
191             "identifier": "@ID"
192           }
193         },
194         "BaseSink": {
195           "before": "Sink<@OUT1> @ID = q.addBaseSink(\"@ID\", tuple -> {",
196           "middle": "//",
197           "after": "});",
198           "placeholders": {
199             "input": [
200             ],
```

```
201         "output": [
202           "@OUT1"
203         ],
204         "identifier": "@ID"
205       }
206     },
207     "TextFileSink": {
208       "before": "Sink<@OUT1> @ID = q.addTextFileSink(\"@ID\",",
209       "middle": "\"file.txt\" // path\n\"true // autoFlush\"",
210       "after": ");",
211       "placeholders": {
212         "input": [
213         ],
214         "output": [
215           "@OUT1"
216         ],
217         "identifier": "@ID"
218       }
219     },
220     "BaseSource": {
221       "before": "Source<@OUT1> @ID = q.addBaseSource(\"@ID\", () -> {",
222       "middle": "return null;",
223       "after": "});",
224       "placeholders": {
225         "input": [
226         ],
227         "output": [
228           "@OUT1"
229         ],
230         "identifier": "@ID"
231       }
232     },
233     "TextFileSource": {
234       "before": "Source<String> @ID = q.addTextFileSource(\"@ID\",",
235       "middle": "\"file.txt\" // path",
236       "after": ");",
237       "placeholders": {
238         "input": [
239         ],
240         "output": [
241         ],
242         "identifier": "@ID"
243       }
244     }
245   }
246 }
```