Our world is populated by billions of internet-connected devices which are constantly producing events and sending them to organisations, receiving them as high-volume data streams. Many organisations wish to respond to this data with as little delay as possible. Distributed Stream Processing Frameworks (DSPFs) are frameworks that facilitate programming these types of applications, which have to handle enormous volumes of data in real time. These frameworks enable applications to scale horizontally by distributing them over several machines connected in a cluster configuration.

Crucial for the performance of such applications is the distribution strategy that is used to partition data over the cluster nodes. In some DSPFs, like Apache Spark or Flink, this distribution strategy is hardwired into the framework, which can lead to inefficient applications. The other end of the spectrum is offered by Apache Storm, which offers a low-level model wherein programmers can implement distribution strategies on a per-application basis to improve efficiency. However, this model conflates distribution and data processing logic, making it difficult to modify either. As a consequence, today's cluster application developers either have to accept the built-in distribution strategy of a high-level framework or accept the complexity of expressing a distribution strategy in Storm's low-level model.

In this dissertation, we propose Skitter: a novel programming model which uses meta-programming to cleanly disentangle the data processing of a stream processing application from its distribution logic, and to enable the creation of novel distribution strategies in a modular fashion. This enables the expression of distributed stream processing applications in a high-level framework while still retaining the flexibility offered by Storm's low-level model.

We introduce a formal description of Skitter, called featherweightSkitter, and a practical implementation of Skitter as a DSPF, called Skitter.ex. FeatherweightSkitter is used in conjunction with featherweightStorm, a novel formal description of Storm, to prove that any application expressed in Storm can be translated into an equivalent application expressed in featherweightSkitter, showing that the introduced abstractions are "Storm-complete". Skitter.ex is used to show that Skitter enables the implementation of existing distribution strategies from the state of the art in a modular fashion. Our performance evaluation of these strategies shows that they exhibit the expected performance characteristics and that applications written in Skitter.ex obtain throughput rates in the same order of magnitude as Storm.

ISBN 978-94-93461-71-0

VRIJE UNIVERSITEIT BRUSSEL

Modular Distribution Strategies in Stream Processing Applications: a Meta-programming Approach

Mathijs Saey

Modular Distribution Strategies in Stream Processing Applications: a Meta-programming Approach

Mathijs Saey

September 2026

Promotors:

Prof. Dr. Wolfgang De Meuter,
Vrije Universiteit Brussel, Belgium

Prof. Dr. Quentin Stiévenart,
Université du Québec à Montréal, Canada

Modular Distribution Strategies in Stream Processing Applications: a Meta-programming Approach

Mathijs Saey

Dissertation submitted in fulfillment of the requirement for the degree of Doctor of Sciences

September 2026

Promotors:
Prof. Dr. Wolfgang De Meuter, Vrije Universiteit Brussel, Belgium
Prof. Dr. Quentin Stiévenart, Université du Québec à Montréal, Canada

Jury:
Prof. Dr. Ann Nowé, Vrije Universiteit Brussel, Belgium (chair)
Prof. Dr. Bas Ketsman, Vrije Universiteit Brussel, Belgium
Prof. Dr. Pieter Libin, Vrije Universiteit Brussel, Belgium
Prof. Dr. Annette Bieniusa, Rheinland-Pfälzische Technische Universität Kaiserslautern-Landau, Germany
Dr. Pascal Weisenburger, Universität St. Gallen, Switzerland

Vrije Universiteit Brussel
Faculty of Sciences and Bio-engineering Sciences
Department of Computer Science
Software Languages Lab

Printed by
Crazy Copy Center Productions
VUB Pleinlaan 2, 1050 Brussel
Tel: +32 2 629 33 44
crazycopy@vub.be
www.crazycopy.be

ISBN: 978-94-93461-71-0
NUR CODE: 989
THEMA: UMA

Abstract

Our world is populated by billions of internet-connected devices which are constantly producing events and sending them to organisations, receiving them as high-volume data *streams*. Many organisations wish to respond to this data with as little delay as possible. *Distributed Stream Processing Frameworks* (DSPFs) are frameworks that facilitate programming these types of applications, which have to handle enormous volumes of data in real time. These frameworks enable applications to scale horizontally by distributing them over several machines connected in a *cluster* configuration.

Crucial for the performance of such applications is the *distribution strategy* that is used to partition data over the cluster nodes. In some DSPFs, like Apache Spark or Flink, this distribution strategy is hardwired into the framework, which can lead to inefficient applications. The other end of the spectrum is offered by Apache Storm, which offers a low-level model wherein programmers can implement distribution strategies on a per-application basis to improve efficiency. However, this model conflates distribution and data processing logic, making it difficult to modify either. As a consequence, today's cluster application developers either have to accept the built-in distribution strategies of a high-level framework or accept the complexity of expressing a distribution strategy in Storm's low-level model.

In this dissertation, we propose Skitter: a novel programming model which uses meta-programming to cleanly disentangle the data processing of a stream processing application from its distribution logic, and to enable the creation of novel distribution strategies in a modular fashion. This enables the expression of distributed stream processing applications in a high-level framework while still retaining the flexibility offered by Storm's low-level model.

We introduce a formal description of Skitter, called featherweightSkitter, and a practical implementation of Skitter as a DSPF, called Skitter.ex. FeatherweightSkitter is used in conjunction with featherweightStorm, a novel formal description of Storm, to prove that any application expressed in Storm can be translated into an equivalent application expressed in featherweightSkitter, showing that the introduced abstractions are "Storm-complete". Skitter.ex is used to show that Skitter enables the implementation of existing distribution strategies from the state of the art in a modular fashion. Our performance evaluation of these strategies shows that they exhibit the expected performance characteristics and that applications written in Skitter.ex obtain throughput rates in the same order of magnitude as Storm.

Samenvatting

Onze wereld is overladen met toestellen die verbonden zijn met het internet en continu "events" produceren die door organisaties ontvangen worden als *stromen* van data. Veel organisaties willen zo snel mogelijk op deze data reageren. *Distributed Stream Processing Frameworks* (DSPFs) zijn software frameworks die het programmeren van applicaties die enorme volumes aan data verwerken vergemakkelijken. Deze frameworks staan deze applicaties toe om horizontaal te schalen door ze over verschillende machines in een "cluster-configuratie" te distribueren.

Essentieel voor de performantie van deze applicaties is de *distributiestrategie* die gebruikt wordt om de data over de nodes van de cluster te verdelen. In sommige DSPFs, zoals Apache Spark of Flink, zijn deze distributiestrategieën ingebakken in het framework, wat tot inefficiënte applicaties kan leiden. Op het andere eind van het spectrum vind je Apache Storm, dat een "low-level" model aanbiedt waarin programmeurs distributiestrategieën kunnen implementeren per applicatie om zo de applicatie-efficiëntie te verbeteren. Dit model verstrengelt echter de distributie en data logica van een applicatie, wat het moeilijk maakt om beide types logica aan te passen. Daarom moeten cluster programmeurs vandaag ofwel de ingebakken distributiestrategieën van een high-level framework accepteren, of omgaan met de complexiteit van deze strategieën uit te drukken in het low-level model van Storm.

Deze dissertatie introduceert Skitter: een nieuw programmeermodel dat metaprogrammeertechnieken gebruikt om data -en distributielogica in een stroomverwerkingsapplicatie van elkaar te ontwarren, en om het mogelijk te maken om nieuwe distributiestrategieën op een modulaire manier te implementeren. Dit maakt het mogelijk om stroomverwerkingsapplicaties in een high-level framework uit te drukken zonder de flexibiliteit van het low-level model van Storm te verliezen.

We stellen een formele beschrijving van Skitter, featherweightSkitter, en een praktische verwezenlijking van Skitter als een DSPF, Skitter.ex, voor. FeatherweightSkitter wordt samen met featherweightStorm, een nieuwe formele beschrijving van Storm, gebruikt om te bewijzen dat elke applicatie die in Storm uitgedrukt kan worden vertaald kan worden naar een equivalente applicatie in featherweightSkitter, wat aantoont dat de nieuwe abstracties "Storm-complete" zijn. Skitter.ex wordt gebruikt om aan te tonen dat Skitter de implementatie van distributiestrategieën uit de literatuur mogelijk maakt op modulaire wijze. Onze performantie-evaluatie van deze strategieën toont aan dat deze de verwachte performantiekarakteristieken vertonen en dat programma's uitgedrukt in Skitter.ex "throughput" ratio's vertonen in dezelfde grootorde als applicaties uitgedrukt in Storm.

Acknowledgments

A first word of thanks goes out to my promotor team. To Wolfgang De Meuter, for giving me the opportunity to work on this PhD, for supporting me when things took (much) longer than planned, and for the feedback and discussions uncounted which helped me transform my vague ideas into a coherent story. To my co-promotor, Quentin Stiévenart, for the key role he played in shaping the formal parts of this work, and for his valuable feedback on the text as a whole. And to my former co-promotor, Joeri De Koster, for helping shape many of the ideas that would come to underpin this work.

A second word of thanks goes to the members of my jury: Ann Nowé, Bas Ketsman, Pieter Libin, Annette Bieniusa, and Pascal Weisenburger. Their feedback on the initial draft of this text greatly helped me identify open issues and improved the quality of the text as a whole.

Third, I thank my colleagues, past and present, at the Software Language Lab for all the conversations, productive or otherwise, we have had over the past years. Special shout-out to Maarten, for some much needed mutual venting; Ward, for running discussions beyond count (sub 3!), and for starting the CityStrides cult at the lab; Jef, for joining the cult and running discussions as an early pioneer, and for the shared efforts on "Structuur 1"; Sam, for heaps of personalised advice; Bjarno, for all the tech and other talk; Jolien, for somehow managing to stay positive when I was not the best office mate during the final stages of my PhD.

I was fortunate enough to have enough friends by my side throughout my PhD that thanking them all individually becomes difficult. Thank you to the Guys in [redacted]; the Beer, [redacted], and Superheroes gang; the Saint-Antoine crew, the DnD (now pf2e) lab rats; the nowthisispodracing.be team; and to Nas, who was clever enough not to join any poorly named groups. A special thank you to Rani and Kenny; the former for being brave enough to proofread this dissertation while knowing very little about the subject matter (is it "proofread" or "proof-read"?); and the latter for designing the "skitter." logo and cover of this book.

Thank you to my family: to my mom, dad, sister, and brother-in-law, for their unwavering support during the last few years. A special word of thanks to my mom, who can now finally stop worrying (about this particular aspect of my life).

A heartfelt thank you to Yitian, who was there with me during many of the highs and lows of this PhD. A final, special word of thanks goes out to Jana, for supporting me and keeping me sane throughout these last few months.

Then suddenly Merry felt it at last, beyond doubt: a change. Wind was in his face! Light was glimmering. Far, far away, in the South the clouds could be dimly seen as remote grey shapes, rolling up, drifting: morning lay beyond them.

— *J.R.R. Tolkien*

x

Contents

List of Figures

List of Tables

List of Listings

1. Introduction

Over the past decades, our world has started to produce an enormous and ever increasing amount of data. Smartphones became ubiquitous in our day-to-day lives and are constantly producing data which are automatically uploaded to the internet. Sensors, home appliances and "wearables" got connected to the net and became part of the so-called "Internet of Things". Social media sites grew enormous and became a source of a near-constant stream of user interactions. Companies started providing streams of real-time operational data (such as financial transactions). In the meantime, the qualitative demands of end users have grown: they expect software which provides them with up-to-date information with as little delay as possible [Arapakis et al., 2021], as most of the aforementioned data is most valuable close to the time it is generated [Assunção et al., 2018]. For instance, users of a social network may wish to read about trending topics as they emerge, while organizations monitoring sensor data may wish to respond to detected events in seconds. We call these types of applications *reactive Big Data applications*.

Often, the volume and velocity of data processed by these applications is so large that it cannot be processed by a single computer reactively. Instead, these applications are typically deployed over multiple machines connected in a *cluster* configuration. Clusters can comprise from five to hundreds of machines. Writing applications that exploit the massive parallelism offered by these cluster computers in traditional programming languages is hard, as it involves dealing with a great deal of accidental complexity. To tackle this problem, several software construction tools have emerged over the past decade which aim to remove this barrier, thus making cluster programming more accessible to mundane programmers. One group of tools, which is of particular interest for the development of reactive Big Data applications, are *Distributed Stream Processing Frameworks* (DSPFs). The best known are Apache Spark [Zaharia et al., 2013] and Flink [Carbone et al., 2015]. This gives rise to what we will call *Distributed Stream Processing Applications* (DSPAs): applications which process high-volume data streams with low latency by distributing the processing of this incoming data over a cluster.

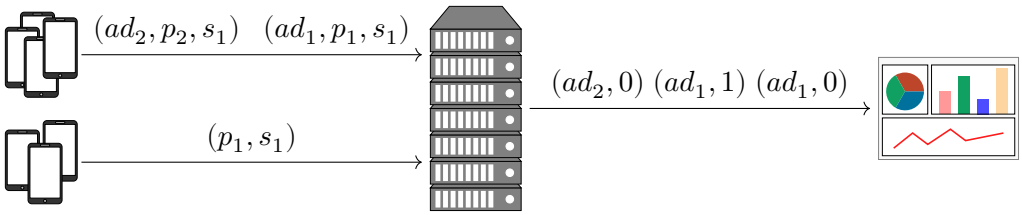

Figure 1.1: Schematic representation of "Adalyze", our example application. ad_i represents an ad, p_i represents a product; s_i represents a (browser) "session". The top stream of data represents clicks on ads, the bottom represents product sales. A sale is caused by an ad if it occurred in the same session. The output stream contains the latest conversion rate of each ad.

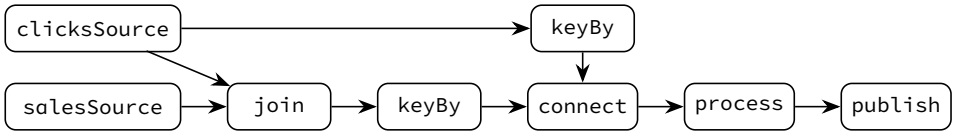

Figure 1.2: DAG of the Apache Flink implementation of Adalyze. The code producing this DAG is shown in Listing 1.1. The various operations in this DAG and its overall structure are discussed in Chapter 2.

example

An example of such an application is "Adalyze", a (hypothetical) DSPA that analyses the effectiveness of advertisements (ads). Adalyze responds to two types of events generated by remote users: *click* events represent users interacting with ads and *sale* events represent users buying a product. Based on these events, Adalyze calculates the *conversion rate* (i.e. the percentage of ad interactions that led to a sale) of each ad, as exemplified in Figure 1.1. Since Adalyze has to be able to continuously process a high volume of incoming ad interaction events, its developers decided to write it in Apache Flink, a popular and featureful DSPF, which enables it to be executed on a cluster.

Supported by these DSPFs, DSPAs are written by transforming data streams through the use of data processing steps, called *operations*. These operations may ingest data from streams and, in turn, emit new data streams, which can then be ingested by other operations. In this way, DSPAs are created by wiring several of these data processing steps into a directed acyclic graph (DAG). Figure 1.2 shows such a DAG for Adalyze, our running example; its code is shown in Listing 1.1.

A DSPF can transparently distribute such an application DAG over the *nodes* (i.e. machines) which constitute the cluster, enabling it to scale with the amount of incoming data. In doing so, the DSPF (and, to some extent, the programmer) is responsible for deciding how the state belonging to each operation is distrib-

```
sales = salesSource();
clicks = clicksSource();

joined = sales
  .join(clicks)
  .where(new SalesKeySelector())
  .equalTo(new ClicksJoinKeySelector())
  .apply(new SalesClicksJoiner());

clicks
  .keyBy(new ClicksRateKeySelector())
  .connect(joined.keyBy(new JoinedKeySelector()))
  .process(new ClickSalesRatioProcessFunction())
  .publish();
```

Listing 1.1: Simplified version of the Adalyze implementation in Apache Flink. Figure 1.2 visualises this code, which we discuss in Chapter 2.

uted over the various cluster nodes, for determining on which nodes the computations for each operation are executed and for deciding which communication occurs between these locations. We call the logic which governs these decisions for each operation the **distribution strategy** of the operation. These distribution strategies are the main topic of this dissertation.

Distribution strategies determine the node on which computations for an operation are executed; they therefore decide how the work to be done for each operation is balanced over the cluster. They also determine which communication occurs between the cluster nodes. Inter-node communication involves passing data over the network, which is extremely expensive [Wu et al., 2018; Liu and Buyya, 2021]. Therefore, **distribution strategies play a key part in determining the performance of a DSPA** [Karimov et al., 2018; Zhou et al., 2019].

Unfortunately, **there is no one-size-fits-all distribution strategy for an operation** that is appropriate for every DSPA. This is the case as applications have non-functional requirements which dictate their performance goals: some require a high level of throughput, while others aim to minimize latency, energy use, or target yet another performance metric. These performance goals may not always match those a particular distribution strategy was optimised to achieve. Moreover, the performance of a distribution strategy is influenced by the properties of the application [Nasir et al., 2016], its input data [Fu et al., 2020; Zhang et al., 2019] and the cluster on which the application is executed [Xu et al., 2018]. Figure 1.3 shows how the properties of a DSPA can impact which distribution strategy offers the "best" performance for the DSPA.

In spite of the impact distribution strategies have on the performance of a DSPA, modern DSPFs, including Spark and Flink, impose a programming model wherein

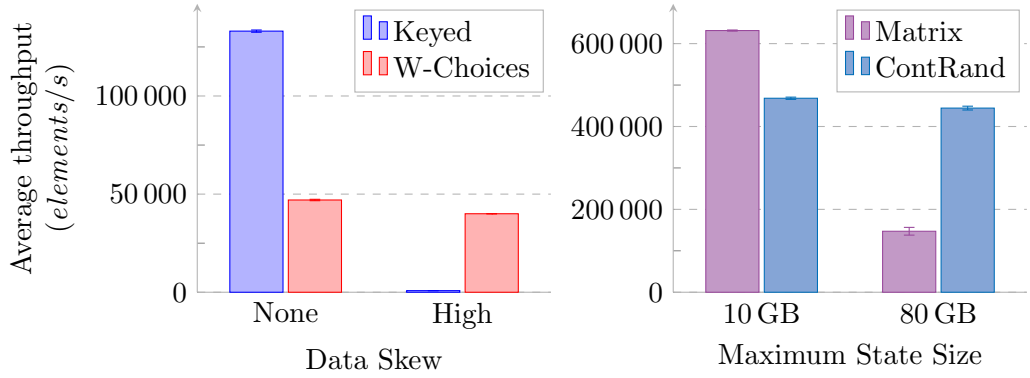

Figure 1.3: Impact of distribution strategies on performance exemplified. Both graphs show the average throughput of two distribution strategies distributing the same operation in two different situations. The left plot shows a stateful operation faced with various levels of data "skew" (discussed in Chapter 2). The right plot shows a join operation joining different amounts of data. The strategies used to distribute the operations (Keyed, W-Choices, Matrix, and ContRand) are discussed in Chapter 2; the details of the experiments used to measure their average throughput are discussed in Chapter 7. We visualise the mean of 30 runs (10 for the 80 GB experiments); error bars represent the 95% confidence interval.

DSPAs are expressed through predefined operations, such as `map`, `join`, or `reduce`, which are hardwired to a fixed, built-in distribution strategy. This fixed dependency between operation and distribution strategy is problematic, as developers cannot modify the distribution strategy of an operation when it is a poor fit for the application and degrades its performance.

example

M.S., the lead developer of Adalyze, sighs in desperation. The Flink implementation of Adalyze (shown in Listing 1.1), which once worked so well, has grinded to a halt. An ad launched by a client went viral, and now the bulk of the data processing work is being handled by a single cluster node, which is struggling to keep up with the influx of data. After some digging, M.S. discovers the culprit: the performance of the `join` operation of Apache Flink seems to degrade when it is faced with heavily skewed data [Karimov et al., 2018].

An alternative model is offered by Apache Storm [Toshniwal et al., 2014], which offers a lower-level programming model that is flexible enough to allow developers to control precisely how each operation in their application is distributed over a cluster. For this reason, Storm is the current "gold standard" for expressing distribution strategies, as it is the only DSPF flexible enough for their implementation [Lin et al., 2015; Nasir et al., 2016; Tran et al., 2018; Zhou et al., 2019; Zhang et al., 2019; Liu et al., 2024; Wu et al., 2025].

```
builder = new TopologyBuilder();

builder.setSpout("sales", new SalesSpout(), 2)
builder.setSpout("clicks", new ClicksSpout(), 2)

builder.setBolt("join-sender", new JoinSendBolt(), 2)
       .localOrShuffleGrouping("clicks")
       .localOrShuffleGrouping("sales")
       .allGrouping("join-sender", "senders_sync")
builder.setBolt("join-joiner", new JoinBolt(), 8)
       .allGrouping("join-sender", "joiners_sync")
       .customGrouping("join-sender", new JoinBicliqueContRandGrouping())

builder.setBolt("rate", new RateBolt(), 2)
       .fieldsGrouping("clicks", "ad-id")
       .fieldsGrouping("join-joiner", "ad-id")
builder.setBolt("publish", new PublishBolt(), 2)
       .localGrouping("rate")
```

Listing 1.2: Simplified version of Adalyze implemented in Apache Storm. Figure 1.4 visualises this code, which we discuss in Chapter 2.

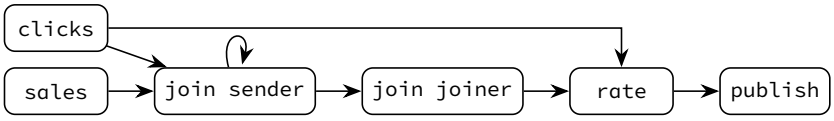

Figure 1.4: Graph of Adalyze implemented in Apache Storm. The code producing this graph is shown in Listing 1.2; it is explained in Chapter 2.

Unfortunately, this flexibility comes at the cost of substantial additional accidental complexity. Moreover, Storm offers no dedicated abstractions for the expression of distribution strategies. Instead, distribution strategies are specified by modifying several distinct locations in the application code. As a result, distribution strategies expressed in Storm are entangled with application logic, not modular, and thus hard to modify.

example

After a grueling engineering effort, M.S. managed to migrate Adalyze to Apache Storm. A few additional weeks of work later, he finished his implementation of the "JoinBiclique ContRand" distribution strategy [Lin et al., 2015], enabling the new and improved Adalyze to handle the most viral of ads. The new implementation of Adalyze is shown in Listing 1.2 and visualised in Figure 1.4.

Unfortunately for M.S., it does not take long before it becomes clear that the Storm implementation of Adalyze is not without its share of issues:

- The distribution logic of the `join` operation is scattered across the definitions of `JoinSendBolt`, `JoinBolt`, and `JoinBicliqueContRandGrouping`

example

(not shown) as well as lines 6–12 and line 16 of Listing 1.2.

- Similarly, the data processing logic of `join` is now scattered over both the `JoinBolt` and `JoinBicliqueContRandGrouping` classes.
- Adalyze's data processing logic is now intertwined with its distribution code: the `JoinBolt` and `JoinBicliqueContRandGrouping` classes and the application definition (i.e. Listing 1.2) contain a mix of both distribution and data processing logic, making it hard to modify either.

M.S.'s fellow developers—data scientists who are not familiar with cluster programming, let alone with the intricacies of Apache Storm—are complaining that his changes made it difficult to adjust Adalyze. They do not understand why the data processing logic is spread over several classes, nor do they appreciate being blamed for breaking Adalyze when they accidentally touch the distribution logic that is now intertwined with the code they need to modify. They much prefer the old Adalyze, and are wondering aloud why a migration was necessary. M.S. quietly weeps.

1.1 Problem Statement

The current state of the art forces developers of DSPAs into a problematic position where they are forced to make a choice between two non-ideal situations:

- They use the **rigid**, high-level programming model offered by DSPFs such as Spark or Flink, but cannot modify the distribution strategies used by their framework of choice when it hampers the performance of their DSPA.
- They use the programming model offered by Storm, thus gaining control over the distribution logic of their application. However, this comes at the cost of having to deal with substantial development **complexity** and having to deal with entanglement between distribution and application logic.

There is need for a programming model for Distributed Stream Processing Frameworks that makes it possible for developers to gain full control over the distribution logic of their application and swap it out when required. In order for this to be possible, the distribution logic of the application should be modular and cleanly separated from the application logic.

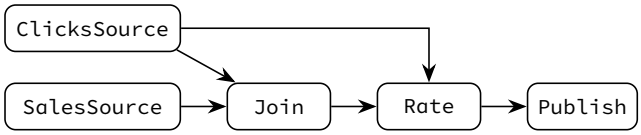

Figure 1.5: DAG of the Skitter.ex implementation of Adalyze. The code producing this DAG is shown in Listing 1.3. The overall structure of this DAG and the nodes contained therein are explored in Chapter 6.

1.2 Research Goal

The goal of this dissertation is to provide a novel programming model that gives developers full control over the distribution logic of their application. Based on our analysis of shortcomings of the current state of the art discussed above, we distil the following requirements for this model:

- Developers should not be limited to the distribution strategies provided by the DSPF, lest it hampers the performance of their applications. Thus, the model should be **extendable**, meaning developers should be able to implement new distribution strategies and use them in their applications.
- The performance requirements of an application are not set in stone and evolve over time. Therefore, distribution strategies should be **modular** and independent from the data processing logic of an application, allowing them to be replaced by a more appropriate distribution strategy when needed.

In order to match the current state of the art, the programming model should also meet the following additional requirements:

- Past work in the distributed stream processing community has introduced several established distribution strategies; the model should be **expressive** enough to support their implementation.
- The main reason developers use DSPFs is to make their applications scale along with the amount of incoming data. Thus, the model should not negatively affect the **performance** of the application. Concretely, it should be possible to implement existing distribution strategies without altering their performance characteristics.

1.3 Approach

In this dissertation, we explore a meta-programming approach to tackle the problem and to satisfy the aforementioned goals. Concretely, we introduce a new

```
1  workflow do
2    node(ClicksSource, as: clicks)
3    clicks.out ~> join.right
4    clicks.out ~> rate.clicks
5
6    node(SalesSource)
7    ~> node(Join, with: BicliqueContRand, as: join)
8    ~> node(Rate, as: rate)
9    ~> node(Publish)
10 end
```

Listing 1.3: Implementation of Adalyze in Skitter.ex. The DAG built by this code is shown in Figure 1.5. Line 7 specifies the application uses a custom distribution strategy for the `Join` operation. We discuss this code, and the definition of distribution strategies, in Chapter 6.

programming model called *Skitter*,[1] which introduces separate abstractions for the expression of data processing logic (i.e. operations) and distribution logic (i.e. strategies). In Skitter, distribution strategies are expressed as meta programs which control how operations are distributed over a cluster. By providing an abstraction for their implementation, Skitter offers a *modular* way to introduce distribution strategies. Moreover, since distribution strategies can be implemented as needed, Skitter offers an open, *extendable* model.

There are two concrete instantiations of the concepts we introduce through Skitter:

FeatherweightSkitter is a formal model which defines the core essence of Skitter as a small-step operational semantics. It is used to evaluate its *expressiveness*.

Skitter.ex is an implementation of Skitter built on top of the Elixir programming language. It is used to evaluate its *modularity* and *performance* properties.

Listing 1.3 shows part of the implementation of Adalyze in Skitter.ex; the equivalent application DAG is shown in Figure 1.5.

1.4 Contributions

The contributions described in this dissertation are as follows:

Survey of Distribution Strategies We investigate distribution strategies used in existing DSPFs, as well as those described in the distributed stream processing literature and discuss their properties.

[1]Named after a character from John C. McCrae's excellent web serial, *Worm*, who uses her powers to coordinate countless entities to set up and execute highly complex schemes.

Taxonomy of DSPFs We provide a taxonomy of the mainstream DSPFs in use today and the extent to which they support the expression of distribution strategies.

FeatherweightStorm We analyse Storm, the current "gold standard" for the implementation of custom distribution strategies, and formalize its behaviour as a small-step operational semantics.

Storm-completeness We introduce and formally define the notion of "Storm-completeness", analogous to relational completeness [Codd, 1972], which serves as a lower bound on the expressiveness DSPFs which aim to enable the expression of distribution strategies should meet.

Skitter Based on our findings, we design a programming model that enables the modular implementation of distribution strategies through the use of meta-programming, as described in Section 1.3.

FeatherweightSkitter We capture the essence of Skitter in the form of a small-step operational semantics. This semantics is a key tool to help us in precisely understanding the exact behaviour of Skitter. We use this semantics to evaluate the expressivity of Skitter by proving that it is "Storm-complete", i.e. that any featherweightStorm application can be expressed as featherweightSkitter application with identical behaviour.

Skitter.ex We provide an implementation of Skitter as a set of three DSLs and a runtime system built on top of Elixir. This implementation serves as a practical tool which enables the concepts introduced by Skitter to be used for the development of real stream processing applications. We use this implementation to evaluate the modularity and performance of Skitter programs.

1.5 Outline

The remainder of this dissertation is structured as follows:

Chapter 2: "Distributed Stream Processing Applications" investigates the state of the art in DSPAs. We discuss how these applications are typically programmed and executed and introduce the terminology used throughout the remainder of this dissertation. Afterwards, we discuss distribution strategies in detail, survey strategies from the state of the art and distil a list of requirements needed to express these distribution strategies in a modular fashion. Finally, we discuss existing DSPFs and the extent to which they meet these requirements.

Chapter 3: "FeatherweightStorm & Storm-completeness" defines the lower bound on the expressiveness DSPFs which aim to enable the expression of distribution strategies should meet. This bound is defined through the notion of "Storm-completeness", which bases this lower bound on Apache Storm, the current "gold standard" for expressing distribution strategies. To obtain a formal definition of Storm-completeness, we introduce featherweightStorm: a formal specification of Apache Storm's programming model and its behaviour at run-time.

Chapter 4: "FeatherweightSkitter" serves as the first introduction to Skitter, the programming model introduced by this dissertation for the modular expression of distribution strategies. In this chapter, we discuss Skitter's essence through the use of featherweightSkitter: a formal specification of Skitter's programming model and its behaviour at run-time. We pay particular attention to the abstractions featherweightSkitter introduces to express distribution strategies and their semantics.

Chapter 5: "FeatherweightSkitter is Storm-complete" formally proves that featherweightSkitter is Storm-complete, showing any featherweightStorm application can be expressed in featherweightSkitter with identical behaviour.

Chapter 6: "Skitter.ex: a DSPF with Modular Distribution Strategies" introduces a practical instantiation of Skitter's programming model, which is realised as a set of domain-specific languages built on top of Elixir. We discuss how it realises the essence of Skitter, and how we extended the model to be a practical tool for expressing real distribution strategies.

Chapter 7: "Experimental Evaluation" evaluates our work by reproducing benchmarks from the literature in both Skitter.ex and Storm and by comparing both implementations and their performance characteristics with one another.

Chapter 8: "Conclusion" concludes this dissertation by presenting our contributions and highlighting possible avenues for future work.

1.6 Supporting Publications

Parts of this dissertation appear in the following publications:

- Mathijs Saey, Joeri De Koster and Wolfgang De Meuter (2025). "Skitter: A Distributed Stream Processing Framework with Pluggable Distribution Strategies". In: *The Art, Science, and Engineering of Programming* 10.1. DOI: 10.22152/PROGRAMMING-JOURNAL.ORG/2025/10/4

This publication discusses Skitter and Skitter.ex. Part of the content of this paper appears in Chapters 6 and 7.

- Mathijs Saey, Quentin Stiévenart, Joeri De Koster, Coen De Roover and Wolfgang De Meuter (2026). "A Storm-complete Stream Processing Formalism with Pluggable Distribution Strategies". Under submission

  This paper, which is currently still under review, discusses featherweightStorm, Storm-completeness and featherweightSkitter. Its contents appear in Chapters 3 to 5.

The following paper discusses an early version of Skitter which did not support the expression of distribution strategies. It is not discussed in this dissertation.

- Mathijs Saey, Joeri De Koster and Wolfgang De Meuter (2018). "Skitter: a DSL for Distributed Reactive Workflows". In: *Proceedings of the 5th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems* (4th Nov. 2018). Boston, MA, USA: ACM, pp. 41–50. DOI: 10.1145/3281278.3281281

Based on the expertise acquired during my research, I contributed to the following other publications:

- Dries Harnie, Mathijs Saey, Alexander E. Vapirev, Jörg Kurt Wegner, Andrey Gedich, Marvin N. Steijaert, Hugo Ceulemans, Roel Wuyts and Wolfgang De Meuter (2017). "Scaling machine learning for target prediction in drug discovery using Apache Spark". In: *Future Generation Computer Systems* 67, pp. 409–417. DOI: 10.1016/J.FUTURE.2016.04.023
- Sophie Mossoux, Mathijs Saey, Stefania Bartolini, Sam Poppe, Frank Canters and Matthieu Kervyn (2016). "Q-LAVHA: A flexible GIS plugin to simulate lava flows". In: *Computers & Geosciences* 97, pp. 98–109. DOI: 10.1016/J.CAGEO.2016.09.003

2. Distributed Stream Processing Applications

DSPFs were introduced to facilitate writing DSPAs and distributing them over a cluster. In this chapter, we discuss these tools and the concepts underpinning them, establishing the background this dissertation builds upon. One of the key concepts we discuss is the notion of a *distribution strategy*: the logic which is responsible for distributing the operations of a DSPA over a cluster. We discuss distribution strategies commonly used by existing DSPFs, as well as those introduced by previous work. Afterwards, we discuss the need for modular distribution strategies by discussing the impact these strategies have on the performance of DSPAs and by discussing why there is no "one-size-fits-all" strategy that can be used for any application. Finally, we discuss the requirements a DSPF should meet to support the modular expression of strategies and the extent to which existing DSPFs support these requirements.

2.1 Structure of Stream Processing Applications

Stream processing applications are typically expressed by programmatically creating a DAG. Each node of this DAG represents a data processing step, called an *operation*, while edges represent data dependencies between these operations. Operations ingest data from their input *streams* and *emit* (i.e. publish) data on their output streams. Operations which do not accept any input are called *sources* while those that do not produce any output are called *sinks*. We discuss how these DAGs are typically created in existing tools in Section 2.3.

example

In Chapter 1, we introduce our running example, Adalyze, a stream processing application which calculates the conversion rate of ads. A potential DAG implementing Adalyze is shown in Figure 2.1.[1] Here, Adalyze is implemented in terms of two sources, `clicks` and `sales`, one sink, `publish`, and two other

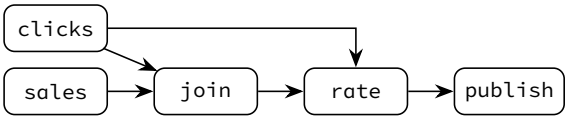

Figure 2.1: Application DAG of Adalyze, a stream processing application calculating ad conversion rates.

example

operations, `join` and `rate`. Both sources are linked to some external system and emit data records corresponding with sales or clicks into the rest of the application. The `join` operation ingests data from both sources: it buffers the clicks and sales that occur; when a sale of a particular product occurs within the same browser session where an ad for that product was clicked, it emits a data record indicating the ad that caused a sale to occur. The `rate` operation ingests data emitted by the `join` operation and the `clicks` source; it maintains two counters for each ad: one counting the number of times the ad has been clicked and another counting the number of times clicks on the ad resulted in a sale. Each time the `rate` operation receives a data record for an ad, it updates the counters associated with the ad, after which it emits an updated conversion rate for the ad. The conversion rate is received by the `publish` operation, which sends it to an external system.

2.2 Distribution Strategies

To execute a stream processing application on a cluster, the computations to be performed for its operations must somehow be distributed over the cluster nodes. This is done by spawning several computational entities, which we call *workers*, for each operation on the nodes of the cluster. Each worker can then receive and process data for its operation independently. Since each operation has its own set of workers, and since an arbitrary amount of workers may be spawned for a single operation, data can be processed in a highly parallel fashion, allowing the application to scale along with the amount of incoming data.

In order to exploit this parallelism, data must be evenly balanced over the various cluster nodes. Obtaining such a balance is not trivial, however, as operations may maintain state, which must be partitioned over the available workers. Moreover, inter-node communication on a cluster is expensive [Wu et al., 2018], as it requires sending data over the network, which means a balance must be found between moving data between cluster nodes and minimizing communication. Overall, the following issues must be addressed when distributing an operation:

[1]This DAG differs from the one shown in Figure 1.2, as the structure of the latter is influenced by the operations offered by the DSPF it is implemented in (Apache Flink).

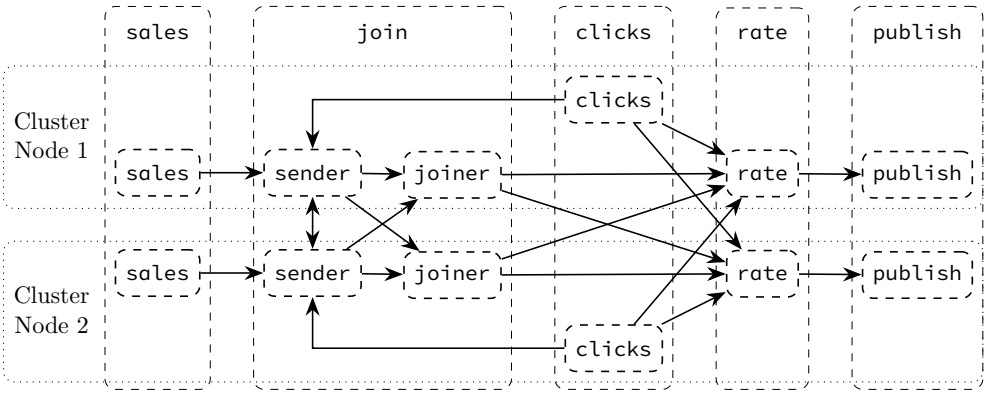

Figure 2.2: Possible run-time view of Adalyze distributed over two cluster nodes. Rows depict cluster nodes; columns depict operations.

- How many workers are created to process data for the operation?
- What is the exact task of each of these workers?
- How is the state of the operation partitioned among these workers?
- Which information is exchanged among the workers?

We call the logic which governs these decisions the *distribution strategy* of an operation. Since these distribution strategies determine how work is distributed over the cluster nodes and which communication occurs between these nodes, distribution strategies are key to the performance of a distributed stream processing application [Karimov et al., 2018].

example

Figure 2.2 visualizes a *potential* deployment of Adalyze over a cluster with two nodes (for the sake of simplicity). Below, we discuss the distribution strategies used to distribute the operations that constitute the application:

`publish` The strategy for `publish` spawns a worker on every cluster node. Since the `publish` operation is stateless, no state is partitioned among these workers. To minimize inter-node communication, a data record that needs to be processed by the `publish` operation is sent to the worker of this operation on the current cluster node, which processes the data record, sending it to an external system.

`clicks` and `sales` These stateless sources are distributed similar to `publish`.

`rate` Similarly to `publish`, a worker is spawned on every cluster node to process data for the `rate` operation. Unlike `publish`, the `rate` operation is stateful, as it needs to maintain two counters for every ad. These counter

pairs are partitioned over both of the operation's workers: when an ad click or sale needs to be processed, the distribution strategy needs to ensure that the event is correctly dispatched to the worker which maintains the counters for this particular ad. The worker which receives the event updates the relevant counter for the ad, after which it calculates the new conversion rate for the ad and emits it.

example

`join` This operation uses the so-called "Join-Biclique ContRand" strategy [Lin et al., 2015], which we discuss in Section 2.2.1. For now, we draw attention to the fact that this distribution strategy uses different worker types (i.e. `sender` and `joiner`) to join its incoming data streams. The `joiner` workers store data records from one input stream and match it with incoming data records from the other stream, while the `sender` workers tag incoming data records with a logical timestamp and coordinate with the joiners and each other to ensure correct behavior.

The discussion above presents one *potential* deployment of Adalyze over a cluster. In reality, several different combinations of distribution strategies may be used to distribute a DSPA. In the next section, we provide a brief survey of these distribution strategies. Following that, in Section 2.2.2, we discuss why none of these strategies can be labeled as the "best", and motivate why a programmer of a DSPA should have the ability to select the most appropriate strategy for each operation in their application.

2.2.1 Distribution Strategies from the State of the Art

Over the past decades, several distribution strategies have been designed. Some were formally introduced in research papers. Many others were used in various DSPAs or offered by DSPFs as a "common sense" approach. In this section, we discuss existing distribution strategies from both categories. We do this to build an intuition on the behavior of distribution strategies, and to introduce key concepts that impact their performance. The behavior of a distribution strategy is typically tied to the properties of the operation which it distributes. Therefore, we split our discussion based on the type of operation they are meant to distribute.

Stateless Operations

Stateless operations typically do not require complex distribution logic, as no coordination is required between the various workers which perform computations for the operation. Therefore, the distribution strategies for these operations are typically "common sense" strategies which are not explicitly described in the scientific literature.

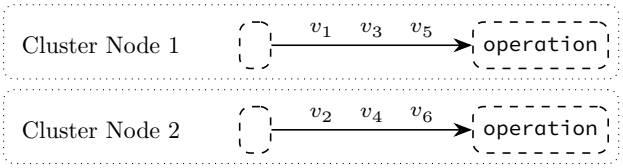

Figure 2.3: Stateless operation distributed over two cluster nodes. Values (denoted as v) are processed on the same cluster node to avoid the overhead of inter-node communication.

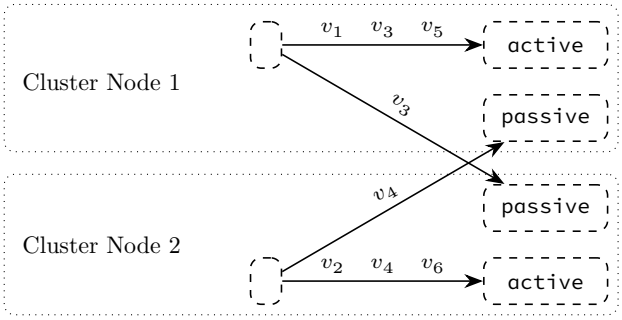

Figure 2.4: Stateless operation distributed over two cluster nodes by Dynamo [Tran et al., 2017; Tran et al., 2018]. Some of the values are duplicated to passive workers, which can be activated when an active worker faces a delay.

To avoid costly inter-node communication, operations processing incoming data typically process the data on a worker which is located on the same cluster node which produced the data, as shown in Figure 2.3. When several of these operations are chained together, DSPFs typically execute them in a single worker to avoid the overhead of scheduling several workers and communicating between them [Hirzel et al., 2014]. Some stateless operations (such as `flatmap`) may produce several data records for an incoming data record, which may lead to an imbalanced distribution of work over the cluster [Li et al., 2021]; when this occurs, a *shuffle* step, which randomly distributes data records over the cluster nodes, may be added after the operation to obtain an even distribution of work for subsequent data processing operations.

While uncommon, examples of distribution strategies from the state of the art which target stateless operations do exist. An example of such a strategy is Dynamo, a distribution strategy which aims to reduce the tail latency of DSPAs. Dynamo aims to achieve this by sending data records to several downstream workers, some of which are designated as "passive", as shown in Figure 2.4. These passive workers receive a configurable fraction of all data records and may be activated by Dynamo's runtime system when it notices an active worker cannot process its

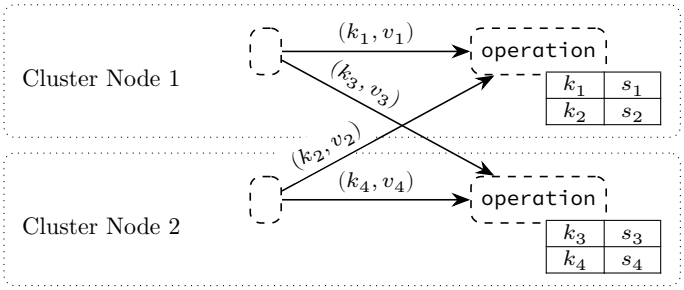

Figure 2.5: Operation with state partitioned over two cluster nodes. Each value, v, is associated with a key, k, based on which the state of the operation is partitioned. Data records are sent to the worker which stores the state associated with their key.

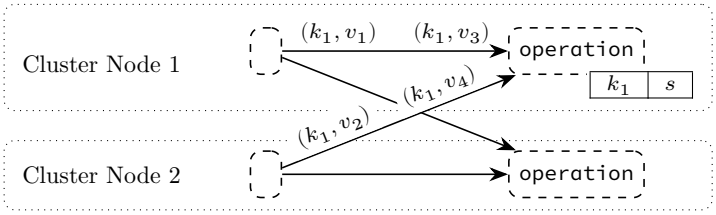

Figure 2.6: Operation with state partitioned over two cluster nodes faced with heavily skewed data. Every incoming data record is associated with key k_1, forcing the worker which maintains the state associated with this key to process all records.

received data records in a timely fashion [Tran et al., 2017; Tran et al., 2018].

Stateful Operations

When a *stateful* operation is distributed over the cluster, it is still the goal to avoid inter-node communication overhead while obtaining a balanced distribution of work over the cluster. Unlike stateless operations, however, strategies distributing stateful operations need to ensure that the computations executing work for the operation can access the appropriate state in order to guarantee correctness. To achieve this, the state of an operation is typically partitioned based on some key; disjoint subsets of the various keys and their associated parts of the state are then distributed over the available workers, after which the distribution strategy must ensure that each incoming data record is sent to the appropriate worker, as shown in Figure 2.5. This is commonly achieved by hashing the key, after which the resulting value is used to select a worker; an alternative approach is to *partition* the key space into *ranges*, which are then used to select a worker.

A common issue that can hamper the performance of key-based operations is the presence of data *skew*: a situation wherein a disproportionate number of data records are associated with a small set of keys. When this occurs, the performance of the "common-sense" approach is hampered, as a small set of workers will need to process the majority of the incoming data records, producing an imbalanced distribution of work over the cluster, a situation shown in Figure 2.6. Since skewed data distributions are very common in real-world workloads [Adamic and Huberman, 2002; Berlinska and Drozdowski, 2018], various distribution strategies have been devised to handle them. Rather than discuss each of these strategies individually, we discuss the techniques many of these strategies have in common and provide an overview of the strategies and the techniques they use in Table 2.1.

Dynamic Mapping A first group of distribution strategies handles data skew by mapping keys over the available workers at run-time based on the properties of the incoming data. Typically, these techniques sample the incoming data records after which the obtained information is used to determine a more balanced distribution of keys [Liu et al., 2018; Fu et al., 2020]. This technique is especially useful in Apache Spark [Zaharia et al., 2013]—a DSPF we discuss in Section 2.3 which processes data in chunks called "microbatches"—as the data obtained from the previous microbatch can be used to improve the distribution of keys over the cluster nodes for the next. However, distribution strategies which change the mapping of keys to workers and migrate existing state live have also been proposed [Shah et al., 2003; Gedik, 2014; Fang et al., 2017].

Splitting Other distribution strategies for operations which face heavily skewed input data split the state of each key over several workers, enabling work for a single key to be performed on several cluster nodes in parallel. Since the partial states need to be re-merged eventually, these distribution strategies can only be used for associative operations. The best-known of these distribution strategies is *Partial Key Grouping* (PKG), which splits the processing of each key over two workers [Nasir et al., 2015], as shown in Figure 2.7.

Selective Splitting & Mapping Splitting the state of a key and re-merging it later is expensive due to the additional communication steps it imposes [Katsipoulakis et al., 2017]. Similarly, maintaining a mapping of keys to cluster nodes can become prohibitively expensive when the key domain is large [Rivetti et al., 2015]. Therefore, it is typically only worth it to pay this cost for the small set of keys which are associated with the majority of incoming data elements [Nasir et al., 2016]. For these reasons, several distribution strategies have emerged which use an online algorithm (e.g. Berinde et al., 2010; Metwally et al., 2005; Manku and Motwani, 2002) to track the

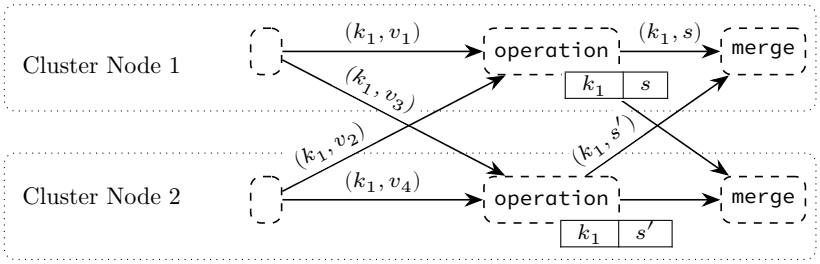

Figure 2.7: Operation with state partitioned over two cluster nodes faced with heavily skewed data distributed by the PKG strategy. The state of each key is split over two workers and merged again later.

most popular keys, which are then distributed differently from the remaining keys.[2] The majority of these strategies use key splitting and dynamically determine the amount of workers the key should be split over [Nasir et al., 2016; Liu et al., 2024; Chen et al., 2021], though approaches which use the aforementioned dynamic mapping technique exist [Rivetti et al., 2015].

Join Operations

To distribute a join operation between two streams, R and S, across a cluster, a distribution strategy must ensure that each data record $r \in R$ is sent to every worker which contains a potentially matching data record $s \in S$. Conversely, any data record from S must also be sent to each worker which contains a potentially matching data record from R. As before, the distribution strategy aims to achieve this with a minimal amount of inter-node communication while still achieving a balanced distribution of work.

Since join operations have a large impact on the performance of a DSPA [Qiu et al., 2019], several strategies have been devised for their distribution. Typically, these distribution strategies are specialized in distributing join operations with a particular type of join predicate:

Low selectivity These are join predicates such as $\neq$ and $<$, for which any potential combination of $r \in R$ and $s \in S$ must be considered. Typically, the strategies which distribute these predicates can be used to distribute any θ-join,[3] as they ensure any combination of data records from R and S is processed as a potential join candidate. These distribution strategies are often based on the *Join-Matrix* strategy, which we discuss below.

[2]Typically, hashing—or, to a lesser extent, PKG—is used for the remaining keys.
[3]i.e. a join which allows for arbitrary join predicates.

Table 2.1: Distribution strategies for operations which maintain key-based state and the techniques they employ to handle data skew.

Strategy		DSPF	Dynamic	Splitting	Selective
PKG	Nasir et al., 2015	Storm		✓	
DKG	Rivetti et al., 2015	Storm	✓		✓
D-Choices	Nasir et al., 2016	Storm		✓	✓
W-Choices	Nasir et al., 2016	Storm		✓	✓
LLFD/Mixed	Fang et al., 2017	Storm	✓		✓
LLFD/MinTable	Fang et al., 2017	Storm	✓		✓
DAGreedy	Pacaci and Özsu, 2018	Storm		✓	✓
SP-Partitioner	Liu et al., 2018	Spark	✓		
ImRP	Fu et al., 2020	Spark	✓		
PStream	Chen et al., 2021	Storm		✓	✓
Pre-filter	Aslam et al., 2021	Storm		✓	✓
St-Stream	Sun et al., 2023	Storm	✓		✓
FlexD	Liu et al., 2024	Storm		✓	✓

High selectivity These are join predicates such as $=$, for which only a limited amount of pairings of $r \in R$ and $s \in S$ must be considered. Distribution strategies targeting these predicates typically target equijoins[4] (e.g. Zhang et al., 2019; Qiu et al., 2019), but specialized strategies for other join predicates have been introduced as well [Yu et al., 2023]. Typically, these strategies aim to ensure that only pairings of $r \in R$ and $s \in S$ which may result in a match, are processed as a potential join candidate. While this enables these strategies to perform less work overall, it usually renders them vulnerable to data skew. Often, some variation of the Join-Biclique strategy (discussed later) is used to distribute these joins. Equijoins may also be distributed by sending data records with the same key to the same worker, through the use of hashing or some of the techniques shown in Table 2.1.

[4] i.e. a θ-join which uses equality as a join predicate.

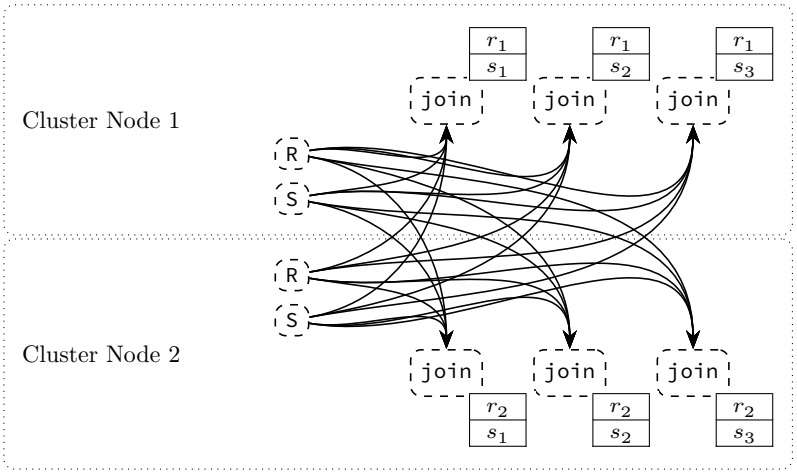

Figure 2.8: Operation joining two streams, R and S, distributed over two cluster nodes using the Join-Matrix distribution strategy. In this example, the strategy partitioned the state over 2×3 workers. The values of R and S are represented by r_i and s_j respectively. r_i and s_j have the same key when $i = j$.

Over the years, many strategies have been introduced to distribute both types of join operations over a cluster. Several of these strategies rely on techniques shared with those used to distribute operations which handle key-based state. For instance, some join strategies treat data records with high-frequency keys differently from others [Qiu et al., 2019; Zhang et al., 2019], or dynamically distribute data over the cluster depending on the overall properties of the incoming data streams [Elseidy et al., 2014; Zhou et al., 2019; Ji et al., 2023]. In the case of join operations, the latter is often paired with the migration of state, in which case the join strategy aims to minimise the cost of the migration [Yu et al., 2023; Wu et al., 2025]. As join operations are often performed on a specific time-based subset of the incoming data called a *window* [Akidau et al., 2015], several distribution strategies rely on the expiration of these windows to avoid or minimise state migration [Wang et al., 2023; Wang et al., 2024].

Table 2.2 provides an overview of the existing join strategies and the techniques they employ. Below, we briefly discuss the Join-Matrix, Biclique and content-sensitive Biclique strategies, as many strategies which target joins extend them.

Join-Matrix This strategy partitions the incoming data records from input streams R and S over a matrix of $m \times n$ workers. When a data record $r \in R$ is received, a random row of the matrix is selected, after which r is replicated n times and sent to each worker in the row. Similarly, when a data record $s \in S$ arrives, s is sent to each of the m workers of a randomly

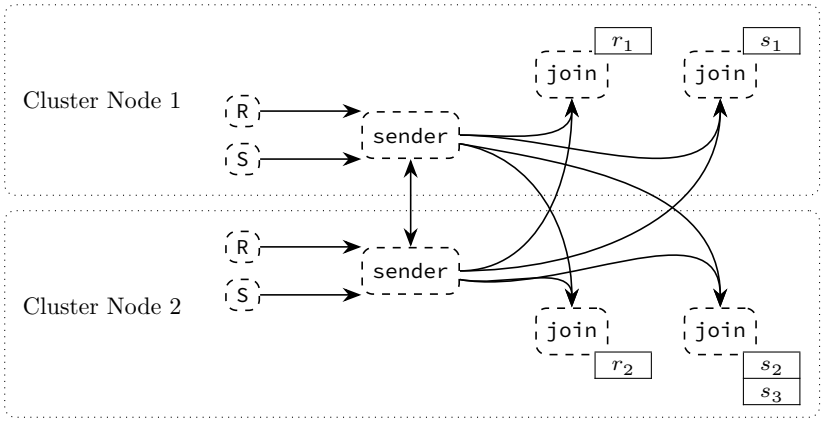

Figure 2.9: Operation joining two streams, R and S, distributed over two cluster nodes using the Join-Biclique distribution strategy. In this example, the strategy partitioned the incoming records over four workers. The two `join` workers on the left form one group of joiners, while the two on the right form the other.

selected column in the matrix. This ensures each potentially matching data record pair (r, s) occurs in exactly one worker in the matrix, as shown in Figure 2.8. All the workers in the matrix may match incoming data records with the stored records as they come in in parallel. Since the row or column for a data record is selected randomly, the Join-Matrix distribution strategy is *content insensitive*, rendering it impervious to data skew. A downside of this strategy is that each data record is stored several times (each $r \in R$ is stored n times, while each $s \in S$ is stored m times), causing a significant amount of memory overhead [Wu et al., 2025]. As mentioned above, the Join-Matrix strategy is typically used to distribute low selectivity joins. Strategies which extend this strategy often rescale the dimensions of the matrix dynamically depending on the properties of the incoming data streams [Elseidy et al., 2014; Wang et al., 2024].

Join-Biclique This strategy was introduced with the intention to store each data record only once [Lin et al., 2015]. It splits the available `join` workers in two groups, one for each incoming data stream. An incoming data record for $r \in R$ is sent to a single worker of its group to be stored and to every worker of the other group to be matched with the previously received data records for S. The inverse is done for any incoming data record $s \in S$, as shown in Figure 2.9. To ensure correctness, the Join-Biclique strategy needs to ensure all workers process the incoming data records in the same order [Lin et al., 2015, §3.2.3]. To achieve this, all incoming data records are first funneled through a set of `sender` workers, which maintain a logical

clock and communicate with one another and the join workers.

Content-sensitive Join-Biclique The Join-Biclique strategy is content insensitive and ensures each data record only needs to be stored once. However, it induces significant communication overhead, as each incoming data record needs to be sent to one worker of its group to be stored and to *every* worker of the other group to be matched. To avoid this overhead, the "Cont-Rand" variant of the Join-Biclique strategy was introduced [Lin et al., 2015]. This variant uses the key of an incoming data record to determine in which workers it will be stored and joined, thus ensuring an incoming data record does not need to be sent to every worker of the other relation to be joined. This comes at the cost of making the strategy content sensitive and thus vulnerable to skew. Several content-sensitive variants of the Join-Biclique strategy have been introduced over the years, which each introduce their own techniques for handling skew [Zhou et al., 2019; Wu et al., 2025].

Summary

Many distribution strategies have been introduced over the past decade. These all aim to distribute a particular operation in a way that balances the work to be done and the data to be stored over the available cluster nodes while keeping the amount of communication as low as possible and preserving the correct behavior of the operation. To achieve this, the distribution strategy reasons about the number of workers to spawn and the job that each of them performs, about the partitioning of state over the workers, and about the communication patterns between them. The performance properties of these distribution strategies are heavily impacted by factors like data skew, which is why there is no single "best" distribution strategy for any particular operation, as we discuss next.

2.2.2 The Need for Modular Distribution Strategies

Distribution strategies have a major impact on the performance of a DSPA, as said at the start of Section 2.2. Therefore, it is important to select the most appropriate strategy for each operation in the application to ensure the application can optimally exploit the parallelism offered by the cluster. Unfortunately, there is no one-size-fits-all distribution strategy for a given operation. Instead, the optimal distribution strategy for an operation is application specific, as we discuss below. Therefore, we argue that distribution strategies should be modular and decoupled from the operation, so that the most optimal distribution strategy can be selected as needed. *This is a central hypothesis of this dissertation.*

There are two key reasons why there is no one-size-fits-all distribution strategy for a given operation that can work in any DSPA:

Table 2.2: Distribution strategies for operations which join streams and the techniques they employ.

Strategy		Type	DSPF	Matrix	Biclique	Dynamic	Migration	Selective
Matrix	Elseidy et al., 2014	θ	Storm	✓		✓	✓	
Biclique	Lin et al., 2015	θ	Storm		✓			
A-DSP	Fang et al., 2021	θ	Storm	✓		✓	✓	
CoStream	Wang et al., 2024	θ^{w}	Flink[z]	✓	✓	✓		
ContRand	Lin et al., 2015	$=$	Storm		✓			
Simois	Zhang et al., 2019	$=$	Storm		✓			✓
FastJoin	Zhou et al., 2019	$=$	Storm		✓	✓	✓	
HyperCube	Qiu et al., 2019	$=$	Storm	✓			✓	✓
NM-Join	Wang et al., 2023	$=^{w}$	Flink[z]			✓		
BS-Join (HS)	Ji et al., 2023	$=^{b}$	Flink			✓		
Ls-Stream	Wu et al., 2025	$=^{w}$	Storm		✓	✓	✓	
Nereus	Yu et al., 2023	Band	Storm		✓	✓	✓	
BS-Join (SL)	Ji et al., 2023	Range[b]	Flink			✓		

[w] Only for windowed join operations

[b] Only for mixed stream-batch join operations

[z] Extended with "Apache ZooKeeper" for additional communication.

1. The notion of "optimal" (i.e. which performance metric to optimise for) is application specific and depends on the nonfunctional requirements of the application and its domain.

2. The performance of a distribution strategy is impacted by many different application specific properties.

We elaborate on each of these reasons below.

"Optimal" is Application Specific

The performance of a DSPA can be measured using many different metrics, such as latency, throughput, sustainable throughput, resource use, tail latency, memory use and others [Shukla et al., 2017; Karimov et al., 2018; van Dongen and Van den Poel, 2020]. The appropriate performance metric to prioritise may depend on the requirements of a given application. Therefore, it is not possible to select the appropriate distribution strategy for an operation without knowing the desired performance characteristics of the application.

The "Dynamo" strategy, discussed in Section 2.2.1, provides an example of this. This distribution strategy duplicates certain data records, which are then processed in parallel [Tran et al., 2018]. This improves the tail latency of applications at the expense of increasing their CPU and communication overhead. This trade off may be worthwhile for applications domains in which tail latency is extremely important, such as online gaming.

Application Properties' Impact on Performance

Even when optimising for a particular performance metric, there is no one-size-fits-all distribution strategy, as the specific properties of the application may influence which strategy is most performant. The following factors can all affect the performance of a strategy:

Operation properties The properties of an operation (such as how computationally expensive it is, or how large its state is), may have a large impact on the performance of a distribution strategy. For instance, the Join-Matrix strategy typically outperforms the Join-Biclique strategy when distributing θ-joins, but this is not the case when the state of the operation grows too large [Lin et al., 2015]. Another example can be found in the "D-Choices" and "W-Choices" distribution strategies: both strategies have near identical behaviour, but the latter strategy is preferred when the state of the operation is not too large [Nasir et al., 2016].

Data properties The properties of the incoming data have a major effect on the performance of a distribution strategy. The best-known example of this is data skew, which greatly influences the design of many distribution strategies, as discussed in Section 2.2.1.

Execution environment The size and properties of the cluster on which a DSPA is executed also affect the choice of distribution strategy. For instance, data skew issues are exacerbated when an application is executed on a larger cluster, which may require a change of strategy [Nasir et al., 2016]. Executing applications on heterogeneous environments also affects the performance of distribution strategies, as certain operations benefit from being executed on machines with specialized hardware (e.g. a GPU [Xu et al., 2018]).

Conclusion

Since there is no one-size-fits-all distribution strategy for an operation, we posit that it should be possible to easily change the strategy of an operation. Moreover, since novel distribution strategies may be introduced as the state of the art advances, we posit that DSPFs should be open and enable the implementation of new strategies as needed.

2.2.3 Requirements for Expressing Modular Distribution Strategies

In order to ensure the most appropriate distribution strategy for an operation can be chosen, DSPFs should make it possible to implement new strategies as needed and make these strategies modular, so they can be swapped out as needed. To enable this, DSPFs need to meet two sets of requirements: first, they should offer the necessary abstractions to enable the implementation of new strategies, second, these abstractions should be offered in such a way that the strategies can be implemented in a modular fashion.

Functionality

At their very core, distribution strategies reason about where (i.e. on which worker) each data record received by an operation is processed. Thus, to express a new strategy, a DSPF needs to reason about workers, the computations performed therein, and the communication between them. Concretely, we identify three key requirements:

Spawning Workers The DSPFs needs to enable programmers to spawn an arbitrary amount of workers.

Arbitrary Stateful Computations The programmer needs to be able to specify exactly which computations are executed on each worker and how they modify the internal state of the worker.

Arbitrary Communication The DSPF needs to enable the programmer to specify to which workers each data record is sent to be processed. This goes beyond expressing the application DAG of the DSPA, as certain strategies (e.g. the Join-Biclique discussed in Section 2.2.1) require arbitrary (i.e. cyclical) communication between workers.

Modularity

The above requirements suffice for the implementation of arbitrary distribution strategies, but do not enable this in a modular fashion. We identify three additional requirements which, when fulfilled, enable this, thus enabling strategies to be swapped out as needed.

Worker-Operation Decoupling Simple distribution strategies typically spawn a set of workers, all of which execute the same logic in parallel. Complex strategies, however, often require the creation of several different worker "types", which perform different computational tasks for a single operation. The Join-Biclique strategy provides an example of this, as shown in Figures 2.2 and 2.9. Thus, to be modular, a distribution strategy must decouple the different worker types from the operation they distribute.

Stratification At run-time, a worker must often execute both distribution and data processing logic. To support modular distribution strategies, a DSPF must separate distribution and data processing logic from each other, even if they are being executed by the same worker.

Reification A distribution strategy reasons about several different concerns, such as spawning workers, sending data between them and deciding which logic to execute in each worker. In order to be modular, all these concerns must be captured by a single abstraction.

2.3 Distributed Stream Processing Frameworks

In the past decade, several DSPFs which use the programming model discussed in Section 2.1 were introduced. In this section, we taxonomise existing DSPFs,

foregoing technical concerns (e.g. how state is persisted, or how partial failures are handled), focussing instead on the programming model they offer and on the extent to which they meet the requirements discussed in Section 2.2.3. A summary of our findings can be found in Table 2.3, at the end of this chapter.

Our taxonomy discusses the following axes:

DAG Creation Modern DSPFs usually offer the so-called **operator** model, wherein DSPAs are created by chaining several operators: generic operations, such as `map`, `filter`, or `reduce`, parameterised with application logic. An imperative alternative to this model involves the manual **wiring** of the application DAG by explicitly extending it with nodes and edges. Many DSPFs also offer a way to declaratively express DSPAs through the use of (some variant of) **SQL** [Olston et al., 2008], which is then mapped down to an application DAG by the DSPFs after a potential optimisation step [Armbrust et al., 2015; Armbrust et al., 2018].

Execution Model Stream processing systems typically process data using the "**continuous operator**" model, which involves processing incoming data one record at a time. In an alternative model, incoming data is collected into a so-called "**microbatch**" until a (time-based) *trigger* occurs, after which the microbatch is processed all at once.

Stateful Computations Every DSPFs offers developers the ability to perform arbitrary computations on some state. However, some DSPFs only offer developers access to the **partial** state of a worker (e.g. the state associated with some key). We distinguish this from DSPFs which enable computations to access the **full** state of the worker they are executed on.

Commmunication We distinguish between DSPFs which support **arbitrary** communication between workers, those that enable sending messages to arbitrary workers with some **limitations** and those that only support **predefined** communication patterns through a limited set of primitives.

Operation-worker Mapping In certain DSPFs, a single operation is always represented by a set of workers which all execute the same logic; we call this a **1:1** mapping. Other DSPFs enable certain primitive operations to be represented by several sets of workers which execute different logic, which we call a **p:n** mapping.

Reification No existing DSPF reifies the notion of a distribution strategy as a concept. However, several DSPFs do introduce primitives which reify specific distribution concepts. Most offer **routing** primitives which tweak how data is routed between workers; several also offer **configuration** primitives

```
sales = salesSource();
clicks = clicksSource();

joined = sales
  .join(clicks)
  .where(new SalesKeySelector())
  .equalTo(new ClicksJoinKeySelector())
  .apply(new SalesClicksJoiner());

clicks
  .keyBy(new ClicksRateKeySelector())
  .connect(joined.keyBy(new JoinedKeySelector()))
  .process(new ClickSalesRatioProcessFunction())
  .publish();
```

Listing 2.1: Essence of the implementation of Adalyze in Apache Flink using the operator model. This implementation has been simplified for the sake of brevity and simplicity. The use of `salesSource`, `clicksSource`, and `publish` abstracts away the real Flink sources and sinks; type specifications and definitions of data processing steps have been omitted. The `join` operator in Flink requires the specification of a window in which data is joined, which has been omitted here.

which enable developers to configure the behaviour of the strategies built into the framework.

Spawning of workers and the stratification of distribution and application logic are not discussed in our taxonomy, as the former is supported by every DSPF, while no DSPFs supports the latter.

2.3.1 Apache Flink

Apache Flink [Carbone et al., 2015] targets stream and batch-based applications in a single framework by utilising a "streaming first" model, where batch-based applications are considered to be stream processing applications executed on a bounded set of data. Flink offers strong support for reasoning about *windows*, logical time-based views of several data records [Akidau et al., 2015].

DAG Creation Applications in Flink can be expressed using operators, or by using one of Flink's relational models[5], which includes SQL [Begoli et al., 2018]. Listing 2.1 shows how Adalyze can be implemented in Apache Flink through the use of the operator model; the application DAG created by this code is shown in Figure 2.10. The first two lines create a source operation for the incoming sale and click events, and parse the incoming data records

[5] `https://nightlies.apache.org/flink/flink-docs-release-2.1/docs/dev/table/overview/` (visited on 21/10/2025)

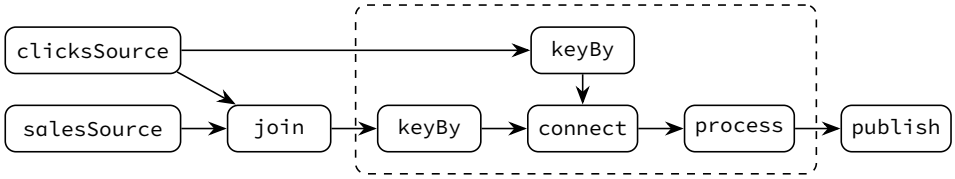

Figure 2.10: DAG of the Apache Flink implementation of Adalyze shown in Listing 2.1. Operations inside the dashed square implement the `rate` data processing step discussed in Section 2.1.

through the use of two `map` operators. Lines 4 to 8 specify how data from both streams is joined. The `rate` logic discussed in Section 2.1 is implemented in lines 10 to 13; the exact functionality of `connect` and `process` is discussed later. The resulting values are published on line 14.

Execution Model Flink utilizes the "continuous operator" model, which processes incoming data records one at a time.

Stateful Computations Flink enables arbitrary, stateful computations through the use of the `process` operator[6], a low-level operator which enables users to express arbitrary stateful computations. While Flink imposes some limitations on the type of state that can be stored, it enables operators to access state scoped to a key, or to a parallel instance of an operator [Carbone et al., 2017]. Line 13 of Listing 2.1 adds such a generic stateful computation to the Adalyze processing pipeline. Since the computation processes keyed input streams, Flink will invoke the computation with the state associated with the received key.

Communication Flink introduces several primitives which modify how data records are sent to workers of the subsequent operation.[7] These include `shuffle` (random distribution), `rebalance` (round-robin redistribution), `broadcast` (send to all), and `rescale` (fan out or fan in). The most flexible primitive, is `partitionCustom`, which can be used to send a data record to any downstream worker. A limitation of `partitionCustom` is that it does not support sending data records to multiple workers; moreover, it only supports communication with downstream workers. While these limitations enable Flink's fault-tolerance mechanism [Carbone et al., 2017], they also make it impossible to implement distribution strategies such as Join-Biclique

[6] `https://nightlies.apache.org/flink/flink-docs-release-2.1/docs/dev/datastream/operators/process_function/` (visited on 21/10/2025)

[7] `https://nightlies.apache.org/flink/flink-docs-release-2.1/docs/dev/datastream/operators/overview/#physical-partitioning` (visited on 21/10/2025)

which requires both sending data to several workers and communication between `sender` workers.

Operation-Worker Mapping Complex operations in Flink (e.g. `join`) may be mapped to several worker sets at run-time.[8] However, Flink does not enable this option for the generic `process` operation, which is always executed on a single set of workers.

Reification Flink reifies several distribution concepts and exposes them as primitives. An example of these are the communication primitives discussed above. Besides these primitives, Flink allows for the *configuration* of operations, which may affect how the operation is distributed. For instance, the `setParallelism` primitive configures how many workers are spawned for a particular operation. Flink also has the notion of "hints", which can be used to modify the behaviour of an operation or its strategy. For instance, a hint can be used to specify which distribution strategy Flink uses for its `join` operation.[9] This shows that Flink acknowledges the need to tweak the distribution logic of an operation. However, this hint only allows the developer to select a built-in strategy for this particular operation; it does not enable them to introduce new strategies or to select one for other operations.

Summary

Apache Flink offers a powerful and expressive high-level programming model for the expression of DSPAs, while still offering developers the required low-level building blocks when needed. However, the restrictions on these low-level building blocks prevent certain distribution strategies from being expressed.

2.3.2 Apache Spark Streaming

Apache Spark Streaming is a DSPF built on top of the batch-based Apache Spark Big Data framework [Zaharia et al., 2012]. Whereas Flink aims to support batch and stream computations in a single engine by treating batch computations as special cases of stream processing applications, Spark Streaming does the opposite and uses the microbatch model to execute stream processing applications. The aim of this approach is to leverage Spark's fault tolerance to reduce the time it takes to recover from partial failures [Zaharia et al., 2013].

[8] `https://flink.apache.org/2015/03/13/peeking-into-apache-flinks-engine-room/` (visited on 24/10/2024)

[9] `https://nightlies.apache.org/flink/flink-docs-release-2.1/docs/dev/table/sql/queries/hints/#join-hints` (visited on 24/10/2025)

DAG Creation Applications in Spark Streaming use the operator-based model, similar to Flink.

Execution Model Spark uses the microbatch model in order to leverage Spark's strong support for batch processing.

Stateful Computations Due to its unique microbatch-based execution model, Spark Streaming has two types of stateful operations: those that perform a stateful transformation within a microbatch, and those that deal with state that is persisted between microbatches.[10] In both cases, Spark only offers developers the ability to access state tied to a particular key.

Communication Similar to Flink, Spark Streaming offers several primitives which can be used to control the distribution of data records over the available downstream workers, such as `repartition` and `coalesce`.[11] Spark Streaming also enables fine-grained control over the distribution of keys over the workers through the use of custom *partitioners*.[12] However, these partitioners can only be used for keyed data streams and can only map a key—and any value for that key in the microbatch—to a single worker.

Operation-Worker Mapping Spark Streaming creates an entirely new execution plan for each microbatch. In this sense, it completely decouples operations from workers. Whenever possible, it will aim to execute subsequent processing steps (in the same microbatch) in the same worker. However, the execution of complex operations, such as joins, which require data redistribution may be spread over several sets of workers. Developers can use communication primitives to ensure subsequent operations are not executed by the same worker, but cannot force a single operation to be executed on several sets of workers.

Reification Like Flink, Spark Streaming exposes several routing constructs (e.g. partitioners). Besides these, Spark Streaming does not offer dedicated primitives for configuring distribution. The parallelism of the data processing pipeline may be configured through the use of the aforementioned communication primitives.

[10] https://spark.apache.org/docs//4.0.1/streaming-programming-guide.html#updatestatebykey-operation (visited on 25/10/2025)

[11] https://spark.apache.org/docs//4.0.1/api/scala/org/apache/spark/api/java/JavaRDD.html (visited on 25/10/2025)

[12] https://spark.apache.org/docs//4.0.1/api/scala/org/apache/spark/Partitioner.html (visited on 25/10/2025)

Summary

Spark Streaming's microbatch model makes it unique compared to other DSPFs. While it offers a high-level programming model, it offers developers little control over the distribution of operations.

2.3.3 Structured Streaming

In 2016, work started on the design of a novel API for Spark Streaming, which would be more declarative and would integrate better in the overall Spark ecosystem. The result of this work was Structured Streaming [Armbrust et al., 2018], a declarative programming model for expressing DSPAs.

DAG Creation Structured Streaming offers two ways to create an application DAG, both of which can be used alongside one another in the same application. It offers a declarative, table-based API [Armbrust et al., 2015], and an operator API.

Execution Model DSPAs expressed using Structured Streaming are typically still executed through the use of microbatches, but the programming model is agnostic to the underlying execution strategy, which enables Structured Streaming to offer an alternative continuous operator execution engine.

Stateful Computations Structured Streaming offers primitives which can perform arbitrary computations on state that is persisted across microbatches.[13] However, as before, the state that can be accessed must be associated with a key. Primitives such as `mapPartition` and `foreachPartition`[14] do enable writing computations that can store state inside a worker, but this state is lost once the current microbatch expires.

Communication Structured Streaming offers similar primitives to Spark to control the distribution of data records over the available partitions and to control the amount of partitions. However, unlike Spark Streaming, Structured Streaming does not allow for the use of custom partitioners.

Operation-Worker Mapping Structured Streaming maps complex operations over multiple workers, but does not enable developers to write operations which are executed on multiple workers.

[13] https://spark.apache.org/docs//4.0.1/api/scala/org/apache/spark/sql/streaming/GroupState.html (visited on 25/10/2025)

[14] https://spark.apache.org/docs//4.0.1/api/java/org/apache/spark/sql/Dataset.html (visited on 25/10/2024)

Reification Structured Streaming reifies routing logic through the communication primitives discussed above. Moreover, similar to Flink, Structured Streaming has the notion of "hints", which are used to change the distribution behaviour of join operations.

Summary

Structured Streaming offers a more declarative programming model over Spark Streaming's microbatch based model. However, it does remove some low-level control from the developer, making it hard to express distribution logic.

2.3.4 Kafka Streams

Apache Kafka [Kreps et al., 2011] is a distributed message system, which is typically used as a "glue" layer used to route messages between various different systems. In practice, many DSPAs source their data from Kafka and write the resulting values back to Kafka, which is then used to route the data to its final destination. In 2016, Kafka was extended to include a lightweight DSPF: Kafka Streams, which can be used to process data sent to Kafka at scale without involving a more traditional, heavyweight DSPF such as Spark or Flink.

DAG Creation Kafka Streams offers an operator-based programming model, similar to Flink and Spark Streaming. More recently, KSQL was introduced, which enables DSPAs to be written in SQL [Jafarpour and Desai, 2019].

Execution Model Kafka Streams uses the continuous operator model.

Stateful Computations Kafka partitions data records over the cluster based on their keys; this same partitioning strategy is used by Kafka Streams. Thus, while Kafka Streams does enable the implementation of arbitrary stateful computations,[15] developers can only access state associated with a key, which may be co-located with the state of other keys.

Communication Kafka Streams does not offer primitives which enable the expression of communication logic. Since all partitioning in Kafka (Streams) is done based on the key of a data record, Kafka Streams will ensure that the necessary repartitioning occurs when a developer uses a primitive operator which may produce a data record with a different key than the incoming data record.

[15] https://kafka.apache.org/documentation/streams/developer-guide/processor-api.html (visited on 26/10/2025)

Operation-Worker Mapping Since the built-in strategies of Kafka Streams can rely on the fact that data records with the same key are always processed on the same cluster node, Kafka Streams does not need any operations which are performed by multiple workers.

Reification Kafka exposes several low-level primitives which enable interaction with the Kafka message broker Kafka Streams relies on, but provides no primitives that directly control the distribution of data.

Summary

Kafka Streams is designed as a simple, lightweight DSPF, which relies on Kafka's partitioning scheme. While this works well in practice, it does not lend itself well to the expression of distribution strategies.

2.3.5 Apache Samza

Apache Samza [Noghabi et al., 2017] is a DSPF which aims to reduce the computational costs of dealing with state in DSPAs and to enable applications to be executed both on historic and real-time data without the need to change application code.

DAG Creation Samza offers an operator-based programming model. It also offers (preliminary) support for expressing DSPAs through the use of SQL.[16]

Execution Model DSPAs written in Samza are executed using the continuous operator model.

Stateful Computations Samza allows developers to express arbitrary stateful computations using its low-level Task API.[17] Using this API, a task can maintain any form of state within its executing worker; however, such state is not managed by Samza, which only manages keyed state.

Communication Samza relies on Kafka [Kreps et al., 2011] for its data routing. As we discussed before, data in Kafka is automatically partitioned based on the key of the data record. Therefore, Samza does not enable sending data to arbitrary workers. Like Flink, Samza does offer a `broadcast` primitive, which sends a data record to all downstream workers.

[16] https://samza.apache.org/learn/documentation/1.8.0/api/samza-sql.html (visited on 26/10/2025)

[17] https://samza.apache.org/learn/documentation/1.8.0/api/low-level-api (visited on 26/10/2025)

Operation-Worker Mapping Samza's `join` operator is executed by several sets of workers. However, operations implemented using Samza's Task API are always executed on a single set of workers.

Reification Similar to Kafka, Samza exposes low-level primitives that allow the developer to control how data records are mapped to Kafka topics. However, as these do not directly control distribution, we do not consider them to be distribution primitives. Samza does expose the previously mentioned `broadcast` primitive, which we classify as a data routing primitive.

Summary

Samza offers a programming model that is similar to Flink, but that offers the developer less control over how data is distributed over the cluster.

2.3.6 Apache Storm

Apache Storm is a DSPF which offers a more low-level model compared to more recent DSPFs. As a trade-off, Storm offers a great deal of flexibility, which is why it remains a popular choice to implement distribution strategies to this day, as evidenced by the amount of strategies implemented using Storm in Tables 2.1 and 2.2. Since Storm is such a popular DSPF for the implementation of distribution strategies, we provide a full formal specification of its behaviour in Chapter 3. In Chapter 5, we use this specification to validate the expressivity of our approach. In Chapter 7, we show Storm makes it difficult to modify and implement distribution strategies.

DAG Creation Unlike other DSPFs, Storm applications are written by explicitly wiring a (possibly cyclic) graph, called a *topology* in Storm, which consists of various sources, called *spouts*, and operations, called *bolts*. Spouts and bolts have to be explicitly connected to each other through the use of *groupings*, which represent edges. An example can be seen in Listing 2.2, which depicts a Storm topology for Adalyze. This code is essentially a textual description of the *run-time* graph shown in Figure 2.2: each worker type shown in Figure 2.2 is added to the application's topology through the use of `setBolt` or `setSpout`. Bolts and spouts are instantiated with an object which encapsulates the logic performed by the bolt or spout, and with a *parallelism hint*, which determines how many parallel copies of these bolts and spouts are spawned at run-time.

```
1  builder = new TopologyBuilder();
2
3  builder.setSpout("sales", new SalesSpout(), 2)
4  builder.setSpout("clicks", new ClicksSpout(), 2)
5
6  builder.setBolt("join-sender", new JoinSendBolt(), 2)
7         .localOrShuffleGrouping("clicks")
8         .localOrShuffleGrouping("sales")
9         .allGrouping("join-sender", "senders_sync")
10 builder.setBolt("join-joiner", new JoinBolt(), 8)
11        .allGrouping("join-sender", "joiners_sync")
12        .customGrouping("join-sender", new JoinBicliqueContRandGrouping())
13
14 builder.setBolt("rate", new RateBolt(), 2)
15        .fieldsGrouping("clicks", "ad-id")
16        .fieldsGrouping("join-joiner", "ad-id")
17 builder.setBolt("publish", new PublishBolt(), 2)
18        .localGrouping("rate")
```

Listing 2.2: Essence of the implementation of Adalyze in Apache Storm. This code is a textual representation of the run-time graph shown in Figure 2.2. Part of the additional nodes required for implementing the Join-Biclique strategy have been omitted in order to make the correspondence with Figure 2.2 more clear.

In more recent versions, Storm has introduced support for expressing applications using operators[18] or SQL[19].

Execution Model Storm uses the continuous operator model.

Stateful Computations Storm places no restrictions on the computations that are expressed inside spouts or bolts. Thus, Storm enables the implementation of arbitrary stateful logic.

Communication Storm groupings represent data dependencies in a Storm topology and also determine which parallel copy of the receiving bolt will process the received data at run-time. Storm defines several built-in groupings (such as the `localOrShuffleGrouping` shown on lines 7 and 8), but also offers a generic interface, `customGrouping`, for the definition of custom groupings, as shown on line 12. The use of this custom grouping places no limitations on the workers a data record can be sent to to be processed. Thus, Storm offers developers complete control over the communication that occurs in their DSPA.

Operation-Worker Mapping A Storm bolt or spout always corresponds to a single set of workers, which will execute the code of the bolt or spout any

[18] https://storm.apache.org/releases/2.8.2/Stream-API.html (visited on 27/10/2025)
[19] https://storm.apache.org/releases/2.8.2/storm-sql.html (visited on 27/10/2025)

time a data record is received.

Reification Storm groupings reify data distribution logic. Besides groupings, Storm introduces primitives such as `setMaxTaskParallelism` which can be used to configure the distributed behaviour of bolts, spouts, or groupings.

Summary

Storm's lower-level model requires application developers to write their DSPAs in a lower-level style, but enables strategy developers full control over the distributed behaviour of their application. Because of this, Storm became a popular choice for the implementation of distribution strategies. However, Storm makes it difficult to implement distribution strategies in a modular fashion, as we show in Chapter 7.

2.3.7 Summary

Table 2.3 shows an overview of the DSPFs we discussed and their classification in our taxonomy. Based on our investigation of the state of the art, we can see that the state of the art seems to be moving towards favouring more declarative approaches for implementing DSPAs. It is also clear that very few DSPFs offer the flexibility required for the implementation of distribution strategies (i.e. arbitrary communication and full access to the state of the worker), and that none meet the modularity requirements we set out in Section 2.2.3.

2.4 Conclusion

In this chapter, we explained how DSPAs are typically structured and how they are distributed over a cluster by a strategy. We investigated these strategies in detail and discussed why they have such a major impact on the performance of a DSPA. We argued that there is no one-size-fits-all strategy that gives the best performance for an operation in every possible situation. Instead, it seems that "best" is application specific, and the performance of a strategy is affected by several application-specific properties. Therefore, we formulated a central hypothesis of our research, namely that the distribution logic of a DSPA should be modular so that a developer can select the most appropriate strategy as needed. Unfortunately, our discussion of the state of the art showed that no existing DSPF meets the requirements to make this possible.

Table 2.3: Taxonomy of existing DSPFs according to the requirements set out in Section 2.2.3.

				Functionality		**Modularity**	
DSPF		**DAG Creation**	**Execution Model**	**Computation**	**Communication**	**O-W Mapping**	**Reification**
Kafka	Kreps et al., 2011[1]	O/S	C	P	P	1:1	-
Spark[2]	Zaharia et al., 2013	O	M	P	L	*p*:*n*	R
Storm	Toshniwal et al., 2014	W/O/S	C	F	A	1:1	R+C
Flink	Carbone et al., 2015	O/S	C	F	L	*p*:*n*	R+C
Samza	Noghabi et al., 2017	O/S	C	F	P	1:1	R
Spark[3]	Armbrust et al., 2018	O/S	M[4]	P	P	*p*:*n*	R+C

DAG Creation: W(Wiring), O(Operator), S(SQL); **Execution**: C(Continuous), M(Microbatch); **Computation**: F(Full), P(Partial); **Communication**: A(Arbitrary), L(Custom, with Limitations), P(Predefined); **O-W Mapping**: 1:1 (one operation maps to one worker), *p*:*n* (primitive operations may map to n workers); **Reification**: R(Routing), C(Configuration).

[1] This work describes the Kafka Messaging System, Stream Processing was added in 2016.

[2] Streaming.

[3] Structured Streaming.

[4] With support for continuous processing.

3. FeatherweightStorm & Storm-completeness

Apache Storm is the current DSPF of choice for expressing novel distribution strategies, as evidenced by the frequency of its use in Tables 2.1 and 2.2. Moreover, it is the only DSPF which fully supports all of the functionality requirements for expressing distribution strategies, as discussed in Section 2.3. We therefore consider Storm the current "gold standard" for developing distribution strategies and believe that any new approach to express distribution strategies needs to be at least as expressive as Storm. In this chapter, we formalise this lower bound on expressiveness by introducing the notion of "*Storm-completeness*". We start with an informal definition of Storm-completeness, after which we move towards a full, formal, definition. To this end, we introduce the first contribution of this dissertation: featherweightStorm, a formalization of Storm which captures the essence of Storm's programming model and its behaviour at run-time. Using featherweightStorm, we precisely define what it means for a programming model to be Storm-complete. In Chapter 5, we use this notion to validate the expressive power of the programming model introduced by this dissertation.

3.1 An Initial, Informal Definition of Storm-completeness

Apache Storm is the de-facto standard for implementing distribution strategies in the current state of the art, as it is the only DSPF powerful enough for their expression (c.f. Sections 2.2.1 and 2.3). *Therefore, any new approach to express distribution strategies needs to be at least as expressive as Storm in order to be able to support the expression of existing distribution strategies.* We call this property *Storm-completeness*, similar to how domain calculus and tuple relational calculus are relationally complete [Codd, 1972].

To be considered Storm-complete, a language needs to meet two criteria:

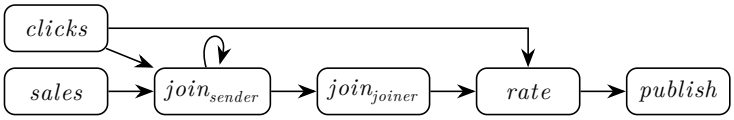

Figure 3.1: Graph of Adalyze expressed in fwStorm. *sales* and *clicks* are spouts, all other nodes are bolts.

Expressive Equivalence Any application that can be expressed in Storm must also be expressible in the language.

Behavioural Equivalence The application expressed in the target language must have identical behaviour to the original Storm application.

The first property ensures that any existing distribution strategy can be expressed. The second ensures that the strategy has the same behaviour at run-time.

3.2 Capturing the Essence of Storm

Showing that a novel language supports the expression of every feature supported by Storm and its host language (Java) is practically infeasible. Instead, we aim to capture the essence of Storm and its behaviour at run-time. To do so, we introduce a novel formalism, featherweightStorm, which precisely captures the essence of Storm, abstracting away any technical details imposed by Storm or its host language. FeatherweightStorm (fwStorm for short) consists of an abstract syntax, which defines how applications can be expressed, and a small-step operational semantics, which defines how these applications are executed.

The names of the concepts introduced by fwStorm match the names of those in Storm. Appendix A contains an overview of the notational conventions we follow, as well as a reference of the various symbols, reduction rules, and auxiliary functions we use to describe fwStorm below.[1]

3.2.1 Abstract Syntax

Figure 3.2 contains the full abstract syntax of fwStorm. A fwStorm application is defined as a *topology*, i.e., a graph. This graph is defined as a pair, (C, G), which contains C, a set of *components* (i.e. nodes), and G, a set of *groupings* (i.e. edges). Each component $c \in C$ represents a data processing step, while each grouping $g \in G$ represents a data dependency between these steps. We

[1] In the digital version of this text, symbols are linked to their definition in Appendix A.

$$\textbf{Topology} ::= (C, G)$$
$$c \in C \subseteq \textbf{Component} ::= s_p \mid b_o$$

$$s_p \in \textbf{Spout} ::= \mathcal{S}_p\langle p, d, \sigma_s\rangle \qquad \sigma_s \in \textbf{SpoutFunction} ::= d \mapsto d, d$$
$$b_o \in \textbf{Bolt} ::= \mathcal{B}_o\langle p, d, \sigma_b\rangle \qquad \sigma_b \in \textbf{BoltFunction} ::= d, d \mapsto d, d$$
$$g \in G \subseteq \textbf{Grouping} ::= \mathcal{G}\langle c, b_o, d, \sigma_g\rangle \qquad \sigma_g \in \textbf{GroupingFunction} ::= d, d, p \mapsto d, \overline{p_i}$$

$$p \in \textbf{Parallelism} \subseteq \mathbb{N}, p_i \in \textbf{TaskIndex} \subseteq \mathbb{N}, \overline{p_i} \in \mathbb{N}^n, d \in \mathbb{D} = \textbf{DataDomain}$$

Figure 3.2: Abstract syntax of fwStorm.

discuss each of these concepts and their representation in fwStorm in detail in the following sections.

$$\textbf{Topology} ::= (C, G)$$

example

In Chapter 1, we introduce Adalyze, a stream processing application which calculates the conversion rate of ads. This application can be expressed as a fwStorm application, (C, G), shown visually in Figure 3.1. This figure shows a fwStorm topology: each node represents a component, $c \in C$, each edge represents a grouping, $g \in G$.

A fwStorm component $c \in C$ is a *spout* (source node) or *bolt* (any other node). Bolts and spouts are represented as three-tuples consisting of a parallelism hint, $p \in \mathbb{N}$, an initial state, $d \in \mathbb{D}$, and a function.

$$c \in C \subseteq \textbf{Component} ::= s_p \mid b_o$$
$$s_p \in \textbf{Spout} ::= \mathcal{S}_p\langle p, d, \sigma_s\rangle$$
$$b_o \in \textbf{Bolt} ::= \mathcal{B}_o\langle p, d, \sigma_b\rangle$$

The parallelism hint of a component determines the number of parallel instances fwStorm will create for the component. The initial state, $d \in \mathbb{D}$, represents the initial state of each of these parallel instances; here, $\mathbb{D}$ is an abstract domain which contains all values processed by a stream processing application.

$$p \in \textbf{Parallelism} \subseteq \mathbb{N}$$
$$d \in \mathbb{D} = \textbf{DataDomain}$$

Both bolts and spouts have a function, which represents the logic performed by the (parallel instances of the) component. A function for a spout (i.e. a source node), σ_s, accepts the current state of the task for the spout, and returns a data record to publish, and a new state for its parallel instance, as we discuss later. A function for a bolt, σ_b, works in much the same way, but accepts two data values:

the current state of the task for the bolt and the data record the bolt needs to process.

$$\sigma_s \in \textbf{SpoutFunction} \coloneqq d \mapsto d, d$$
$$\sigma_b \in \textbf{BoltFunction} \coloneqq d, d \mapsto d, d$$

example

Every node shown in Figure 3.1 is an element of C. In other words, $C = \{clicks, sales, join_{sender}, join_{joiner}, rate, publish\}$. In this set, $clicks$ and $sales$ are spouts, while all other nodes are bolts. The spouts contain a function to generate data records ex nihilo, while the bolts contain a function which determines how received data records are processed.

In this example, the join operation is distributed using the Join-Biclique Cont-Rand strategy, discussed in Chapter 2. Since this strategy requires different parallel instances to perform different work, several bolts are needed to implement the join operation, along with the necessary groupings, $g \in G$, to connect them, which we discuss next.

fwStorm groupings represent data dependencies between components and are defined as a four-tuple consisting of a source component ($c \in C$), which sends the data, and a destination bolt ($b_o \in C$), which receives the data. The receiving component of a grouping is always a bolt, as spouts are sources in the application graph and therefore do not have any incoming data dependencies. The final two elements of a grouping, its initial state (d) and its grouping function (σ_g), are used to determine which parallel instance of the receiving bolt should process the data record. The grouping function is called with the state of the grouping, the data record to be sent, and the parallelism of the receiving bolt, p. Based on this information, the grouping function must return a vector of indices, $\overline{p_i}$ (where $\forall p_i \in \overline{p_i} : p_i \in [0, p[\subseteq \mathbb{N}$), which correspond to the indices of the parallel instances that should receive the data record, and a new state for the grouping.

$$g \in G \subseteq \textbf{Grouping} \coloneqq \mathcal{G}\langle c, b_o, d, \sigma_g \rangle$$
$$\sigma_g \in \textbf{GroupingFunction} \coloneqq d, d, p \mapsto d, \overline{p_i}$$
$$p_i \in \textbf{TaskIndex} \subseteq \mathbb{N}$$
$$\overline{p_i} \in \mathbb{N}^n$$

This definition of a grouping in fwStorm corresponds to a `customGrouping` in Storm, discussed in Section 2.3.6. fwStorm does not define any of Storm's built-in groupings, as these can all be defined in terms of `customGrouping`.

example

The edges shown in Figure 3.1 are elements of G, the set of groupings. Since the join operation requires several parallel instances to perform different work, G contains both groupings between operations (e.g. $\mathcal{G}\langle clicks, rate, d, \sigma_g \rangle$), as well as groupings between components belonging to the same operation (e.g.

$\mathcal{G}\langle join_{sender}, join_{joiner}, d', {\sigma_g}' \rangle$).

The edge between *clicks* and *rate* is represented as $\mathcal{G}\langle clicks, rate, \varnothing, \sigma_g \rangle \in G$. σ_g determines which parallel instances of *rate* should process the received data. Here, $\sigma_g = d_r, d_s, p \mapsto d_s, [\text{hash}(\text{extract}_{\text{ad}}(d_r)) \bmod p]$. This function extracts the identifier of the ad from the incoming data record, d_r, and hashes it modulo the amount of parallel instances (p) to obtain the index of such an instance. This index is wrapped in a vector, as a grouping may send data to several instances. The state of the grouping, d_s, remains unchanged, as it is not used.

example

3.2.2 Operational Semantics

Having defined the syntax of fwStorm, we turn to discussing its execution. At run-time, a fwStorm application is executed by several *tasks* processing data records in parallel. Since we are interested in capturing when these tasks can process data and how data is sent between them, we define the semantics of fwStorm as a small-step operational semantics. These semantics operate on T, the set of all tasks, which represents the current state of the fwStorm application that is being executed.

A task is one of the parallel instances which processes data records for a component; i.e. it is (featherweight) Storm's terminology to denote a worker. Each task, $t \in T$, is associated with the component for which it was created, c, and an index, p_i, which uniquely identifies it among the other tasks created for c; note that if the parallelism hint of c is p, $p_i \in [0, p[\subseteq \mathbb{N}$. A task also contains a value, d, which represents the current state of the task, a mapping Γ, which contains the current state of all outgoing groupings, discussed later, and $\overline{q_s}$, a *message queue*. This queue is represented as a vector containing all the data records to be processed by the task. Tasks for bolts can serve a message (i.e. a data record to be processed) from this queue and process it, while other tasks may enqueue data to the other end of the queue.

$$t \in T \subseteq \mathbf{Task} ::= \mathcal{T}\langle p_i, c, d, \Gamma, \overline{q_s} \rangle$$
$$\overline{q_s} \in \mathbb{D}^n$$

example

At run-time, fwStorm creates several tasks, $t \in T$, for each component $c \in C = \{clicks, sales, join_{sender}, join_{joiner}, rate, publish\}$. Each task contains its component, c, an index which distinguishes it from the other tasks created for c, and a state, grouping state, and message queue, discussed later.

fwStorm groupings determine to which downstream tasks a data record is sent through their grouping function, σ_g. Groupings are stateful: any time σ_g is invoked, it accepts the current state of the grouping. This state is specific to each

upstream task. Therefore, any task $t \in T$ contains Γ: a mapping which maps each outgoing grouping to its current state.

$$\Gamma \subseteq \textbf{GroupingState} \coloneqq g \mapsto d$$

For a component c in application (C, G), the initial state of this mapping is defined by the auxiliary function $init_\Gamma$, which creates this mapping for c by mapping each grouping with c as its source to its initial state as defined in G.

$$init_\Gamma(c) = \{(g, d) \mid g = \mathcal{G}\langle c, \_, d, \_\rangle \in G\}$$

example

The $clicks$ component has two downstream bolts: $rate$ and $join_{sender}$. Therefore, $\left\{g_r = \mathcal{G}\langle clicks, rate, \varnothing, \sigma_{g_r}\rangle, g_j = \mathcal{G}\langle clicks, join_{sender}, d, \sigma_{g_j}\rangle\right\} \subset G$. Every task, $\mathcal{T}\langle \_, c, \_, \Gamma, \_\rangle \in T$ where $c = clicks$ has a grouping state, Γ, which maps g_r and g_j to their current state in t. For each task where $c = clicks$, the initial state of Γ will be $\left[g_r \mapsto \varnothing, g_j \mapsto d\right]$.

Since σ_{g_r} does not modify its state, the state of g_r will always be mapped to $\varnothing$; σ_{g_j} may modify its state, which means that $\Gamma' = \left[g_r \mapsto \varnothing, g_j \mapsto d'\right]$, where Γ' is the value of Γ after t sent one or several data records through g_r and g_j. Here, d' represents the updated state of σ_{g_j}.

Initial State

The initial state of a fwStorm application (C, G) is a set of tasks which contains p tasks for each component, $c \in C$, where p refers to the parallelism hint of c, which may be a spout or bolt: $c = \mathcal{S}_p\langle p, \_, \_\rangle \vee c = \mathcal{B}_o\langle p, \_, \_\rangle$. Each task is created with the initial state defined by its spout or bolt, an empty message queue, and a state for each outgoing grouping of c as defined by $init_\Gamma(c)$:

$$\bigcup \left\{ \left(\bigcup_{i=0}^{p-1} \{\mathcal{T}\langle i, c, d, init_\Gamma(c), \varnothing\rangle\} \right) \mid c = \mathcal{B}_o\langle p, d, \_\rangle \in C \vee c = \mathcal{S}_p\langle p, d, \_\rangle \in C \right\}$$

· example ·

The initial state of Adalyze is defined as T, a set of tasks; one or multiple tasks are present in T for each component $c \in C$. If the parallelism of c is p, there will be p tasks $\in T$ for c.

Reduction Rules

The execution of a fwStorm application, (C, G) is defined by two rules, (SPOUT) and (BOLT), which define how a task for a spout or bolt can generate or process

a data record. In the definition of these rules, the set of components, C, and the set of groupings, G, which define the application, are available globally.

At any point in time, the (SPOUT) rule can be used to obtain a new value from a task for a spout, s_p, which is then sent to all the successors of the spout through the use of auxiliary function *send* (see below). The value to be sent is determined by the function of the spout, σ_s, which is called with the current state of the task, d_s. Based on this state, the function returns a new state, d'_s, and a value to be sent, d_r. The new state, d_s, is stored as the state for the task. Auxiliary function *send* (defined later) is used to send d_r to the successors of s_p; in doing so, it may modify other tasks and the state of any groupings stored inside the task.

(SPOUT)

$$\frac{s_p = \mathcal{S}_p\langle \_, \_, \sigma_s\rangle \qquad d'_s, d_r = \sigma_s(d_s) \qquad T', \Gamma' = send(d_r, T, \Gamma)}{T \uplus \left\{\mathcal{T}\langle p_i, s_p, d_s, \Gamma, \varnothing\rangle\right\} \rightarrow T' \cup \left\{\mathcal{T}\langle p_i, s_p, d'_s, \Gamma', \varnothing\rangle\right\}}$$

example

At any point in time, the (SPOUT) rule can be used on any of the tasks for the *clicks* or *sales* spouts to obtain a new data record to be processed. This rule will then propagate the obtained data record to a task of all components downstream of the spout. The task's state and grouping state may be modified when this rule is used.

The (BOLT) rule can only be reduced when there exists a task for a bolt, b_o, with a value, d_q in its queue. This value is passed as an argument to the bolt's function, σ_b, along with the current state of the task, d_s, to obtain a potentially updated state, d'_s, and a resulting value, d_r. This value is sent to the successors of the bolt. Since bolts may send values to themselves, the sending task, t, is added to the set of tasks, T, when *send* is invoked. If t receives a value, its queue, $\overline{q_s}$, is modified.

(BOLT)

$$\frac{\begin{array}{c} b_o = \mathcal{B}_o\langle \_, \_, \sigma_b\rangle \qquad d'_s, d_r = \sigma_b(d_s, d_q) \\ t = \mathcal{T}\langle p_i, b_o, d'_s, \Gamma, \overline{q_s}\rangle \qquad T' \uplus \left\{\mathcal{T}\langle p_i, b_o, d'_s, \Gamma, \overline{q_s}'\rangle\right\}, \Gamma' = send(d_r, T \cup \{t\}, \Gamma) \end{array}}{T \uplus \left\{\mathcal{T}\langle p_i, b_o, d_s, \Gamma, \overline{q_s} \cdot d_q\rangle\right\} \rightarrow T' \cup \left\{\mathcal{T}\langle p_i, b_o, d'_s, \Gamma', \overline{q_s}'\rangle\right\}}$$

example

$join_{sender}$, $join_{joiner}$, *rate*, or *publish* may receive a data record from an upstream spout or bolt. When this happens, the data record is prepended to the message queue of one of its tasks. In turn, this enables the use of the (BOLT) rule, which can then be used to process the received data record. This may then produce a new data record, which is sent to any downstream bolts. In doing so, the state of the task may be updated, along with its grouping state and message queue.

Auxiliary Functions

(BOLT) and (SPOUT) use auxiliary function $send$ to send a data record, d, to all the successors of the component of the sending task, c. This function may modify any task $t \in T$, as t may need to receive d; and the group state, Γ, of the sending task, as it contains the (mutable) state of each grouping. $send$ does this by calling $send_G$ on all groupings $g \in \Gamma$, i.e. on all the groupings that have c as the sender.

$$send(d, T, \Gamma) = send_G(d, \{g \mid (g, \_) \in \Gamma\}, T, \Gamma)$$

In turn, $send_G$ sends d to each element of G_s (i.e. the set of groupings which have c as the sender), through the use of $send_g$. Each invocation of $send_g$ returns a potentially updated set of tasks and a potentially updated grouping state mapping.

$$\begin{aligned} send_G(\_, \varnothing, T, \Gamma) &= T, \Gamma \\ send_G(d, G_s \uplus \{g\}, T, \Gamma) &= send_G(d, G_s, T', \Gamma') \\ &\text{where} \\ T', \Gamma' &= send_g(d, g, T, \Gamma) \end{aligned}$$

$send_g$ sends d to a single bolt b_o, connected to the sending component, c, through grouping $g \in G$, where $g = \mathcal{G}\langle c, b_o, \_, \sigma_g \rangle$. To do so, it obtains $\overline{p_i}$, the indices of the downstream tasks to which d ought to be sent. These indices are obtained by calling the grouping's function, σ_g, with the current state of the grouping, d, and p, the parallelism of b_o; the current state of the grouping is obtained from Γ. Besides the indices, σ_g returns a new state for the grouping, which is stored inside Γ. The indices are used by $send_t$ to send d to each task with an index in $\overline{p_i}$.

$$\begin{aligned} send_g(d, g, T, \Gamma) &= T', \Gamma[g \mapsto d_s] \\ &\text{where} \\ g &= \mathcal{G}\langle \_, b_o = \mathcal{B}_o\langle p, \_, \_ \rangle, \_, \sigma_g \rangle \\ d_s, \overline{p_i} &= \sigma_g(\Gamma(g), d, p) \\ T' &= send_t(d, b_o, \overline{p_i}, T) \end{aligned}$$

$send_t$ sends d to every task of b_o with an index present in $\overline{p_i}$. This is done by prepending d to the message queue of each of these tasks.

$$\begin{aligned} send_t(\_, \_, \varnothing, T) &= T \\ send_t(d, b_o, p_i \cdot \overline{p_i}, T \uplus t) &= send_t(d, b_o, \overline{p_i}, T \cup t') \\ &\text{where} \\ t &= \mathcal{T}\langle p_i, b_o, d_s, \Gamma, \overline{q_s} \rangle \\ t' &= \mathcal{T}\langle p_i, b_o, d_s, \Gamma, d \cdot \overline{q_s} \rangle \end{aligned}$$

example

When the *clicks* spout produces a data record, d, *send* will be evaluated. This function will evaluate $send_G$ with the set of all groupings for which *clicks* is the upstream, i.e. with $\left\{g = \mathcal{G}\langle clicks, join_{sender}, d, \sigma_{g_j}\rangle, \mathcal{G}\langle clicks, rate, \varnothing, \sigma_{g_r}\rangle\right\}$. This will lead to the application of $send_g$ to each grouping in this set. $send_g$ will apply σ_{g_j} to d, the parallelism of $join_{sender}$, and the current state of g stored in Γ to determine to which tasks of $join_{sender}$ d should be sent. When $send_t$ is evaluated, it prepends d to each task selected by σ_{g_j}. The updated state of the grouping is stored inside Γ by $send_g$.

Execution Sequences

Any fwStorm application has numerous possible executions, as at any point in time, there may be various tasks for which the (SPOUT) or (BOLT) rules can be reduced. Each time one of these rules is reduced, the current state of the fwStorm application, T, is modified, resulting in a new set of tasks, i.e. in a new state. A single execution of a fwStorm application can be represented by a sequence (i.e. a vector) of these states: $\overline{T}$. Each of these states, $T_i \in \overline{T}$, corresponds to the state of the application after the reduction of a rule. We define Θ as the set of all possible executions of a particular fwStorm application.

$$T_i \in \overline{T} \in \Theta$$

example

Let T_0 be the initial state of the fwStorm application representing Adalyze. Given this initial state, the (SPOUT) rule can be used on any of the tasks $\in T_0$ for the *clicks* or *sales* spouts to obtain a new data record to be processed. This rule will then propagate the obtained data record to one or several downstream tasks. Updating the state of the task which produced the data record and updating the state of any tasks which received the data record results in a new state for the application: T_1.

With a data record present in the queue of a task, the (BOLT) rule may be invoked to process the received data record, once again leading to a new state, T_2. This is the beginning of one particular execution sequence of this application.

Instead of invoking the (BOLT) rule on a task $\in T_1$, the (SPOUT) rule could have been used to produce another value for a task representing the *clicks* or *sales* spouts. This would lead to an alternative state, $T_{2'}$, which is part of a different possible execution sequence. Alternatively, another spout task could have been selected from T_0, leading to yet another alternative state: $T_{1'}$. Θ is the set which contains $[T_1, T_2]$, $[T_1, T_{2'}]$, $[T_{1'}]$, and any other possible execution sequence of the application.

Summary

A fwStorm application is a perpetually running group of tasks. Spouts can emit new data records, which are then sent to any bolt connected to the spout. In turn, bolts process the data records in their message queue one by one. The resulting value from processing this data record is then propagated further down the application graph. At any point, several tasks are ready to process values (bolts) or produce values (spouts); an execution sequence represents a particular order in which tasks are selected to process or produce data records.

3.2.3 Other Formal Descriptions of Storm

Previous work has formalised Storm and its behaviour. We briefly discuss these formalisations and compare them to fwStorm below.

- Requeno et al. [2017] model Storm applications using UML diagrams annotated with performance parameters. These UML diagrams can then be converted into stochastic Petri nets, which are used to predict the performance behavior of the application. Compared to fwStorm, this work does not model the functionality expressed in bolts, spouts or groupings, focusing on the information obtained from annotations instead.

- Another approach based on annotated UML diagrams is presented by Marconi et al. [2017]. This work builds on the earlier work from the same authors [Marconi et al., 2016], wherein Storm applications are modelled using constraint linear-time temporal logic. This model is then used to verify if the bolts in the Storm application are able to handle the rate of incoming data records. As before, this work does not model the functionality expressed in a Storm application; instead, it verifies performance-related properties of the application.

- Previous work modelled the processes which make up the Storm runtime system using Communicating Sequential Processes [Zhao et al., 2019]. Unlike fwStorm, this work focuses on verifying the correctness of the Storm runtime system, and does not formalise behaviour associated with the application that is being executed by Storm.

3.3 Storm-completeness

Now that we have formalised the behaviour of Storm, we can precisely define the notion of Storm-completeness in terms of fwStorm.

Recall that, informally, a language, L, is considered Storm-complete if any Storm application can be expressed in L (expressive equivalence) and if the application expressed in L has identical behaviour to the original Storm application (behavioural equivalence).

We define behavioural equivalence in terms of the execution sequences discussed in Section 3.2.2: two applications are behaviourally equivalent if any possible execution sequence they have is equivalent to exactly one other execution sequence in the other application. Here, equivalent means that each task $t \in T_i \in \overline{T} \in \Theta$ has a counterpart in the configuration of the other language.

Definition: Storm-completeness

A language, L, is Storm-complete if and only if, for every possible fwStorm application, (C, G), with potential executions Θ, there exists an equivalent application expressed in L, with potential executions Σ. Any execution $\overline{S} \in \Sigma$ must have exactly one equivalent execution $\overline{T} \in \Theta$, and every execution $\overline{T} \in \Theta$ must have exactly one equivalent execution $\overline{S} \in \Sigma$.

By defining Storm-completeness, we have pinpointed a lower bound on the expressiveness any DSPF which aims to enable expression of distribution strategies must meet. In Chapter 5, we show that our programming model meets this lower bound and that it can express any application that can be expressed in fwStorm. Moreover, featherweightStorm, introduced in this chapter, precisely captures the semantics of a widely-used DSPF, and can be used as a tool to understand its behavior at run-time.

4. FeatherweightSkitter

The main contribution of this dissertation is Skitter: a programming model for writing DSPAs which supports modular, pluggable distribution strategies. In this chapter, we present the core of this programming model through the means of "featherweightSkitter" (fwSkitter): a formal basis of Skitter which captures the essence of its programming model and its behaviour at run-time. In Chapter 6, we present Skitter.ex: a practical instantiation of Skitter.

Skitter is *layered*. The *operation* layer is used to express data processing logic, while the *strategy* layer functions as a meta layer, which controls how and where operation code is executed. The final layer is the *workflow* layer which is used to wire data processing steps and their distribution strategies together into an application DAG. The novelty of Skitter lies in its strict separation of data processing and distribution logic, and in its abstractions for distribution strategies.

As before, Appendix A offers an overview of the notational conventions we use, as well as a complete overview of fwSkitter.[1]

4.1 Abstract Syntax

Figure 4.1 contains the abstract syntax of fwSkitter. Applications in fwSkitter are defined as a DAG consisting of several nodes ($n \in N$), which represent data processing steps. These nodes are connected through *links* ($l \in L$), which represent data dependencies between nodes.

Each node contains both distribution and application logic. The application logic is contained in the operation of the node, Ω, while the distribution logic is contained in the strategy of the node, s. Ω is a collection of named *callbacks*, each callback implements part of the data processing logic of the operation. s is a pair of expressions, which determine how the operation is distributed.

[1] In the digital version of this text, each symbol links to the appropriate entry in this appendix.

$$\begin{aligned}
\textbf{Workflow} &:= (N, L) & s \in \textbf{Strategy} &:= \mathcal{S}\langle e, e\rangle \\
L \subseteq \textbf{Link} &:= \mathcal{L}\langle n, n, i\rangle & \Omega \in \textbf{Operation} &:= cb \mapsto \rho \\
n \in N \subseteq \textbf{Node} &:= \mathcal{N}\langle s, \Omega\rangle & \rho \in \textbf{Callback} &:= d, d \mapsto d, d
\end{aligned}$$

$$\begin{aligned}
e \in \textbf{Expression} := \quad & e; e \mid x \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{let}\ x_s, x_r = \texttt{call}(cb, e, e)\ \texttt{in}\ e \\
\mid \quad & \texttt{null} \mid \texttt{self} \mid \texttt{deployment} \mid \texttt{msg} \mid \texttt{state} \mid \texttt{data} \mid \texttt{port} \\
\mid \quad & \texttt{spawn}(e, e) \mid \texttt{send}(e, e) \mid \texttt{emit}(e) \\
\mid \quad & e[e] \mid \overline{e}^{\,k} \mid \overline{\texttt{send}}(e, e)
\end{aligned}$$

$$i \in \textbf{Port} \subseteq \mathbb{N}, cb \in \textbf{CallbackName}, d \in \mathbb{D} = \textbf{DataDomain}$$
$$x, x_s, x_r \in \textbf{VarName}, k \in \mathbb{N}$$

Figure 4.1: Abstract syntax of fwSkitter.

We discuss these concepts and their representation in fwSkitter in detail in the following sections. The main focus of this discussion is the strategy abstraction, as it is the main novelty introduced by Skitter.

4.1.1 Defining Stream Processing Applications

Similar to the existing DSPFs discussed in Chapter 2, applications in fwSkitter are defined as an application DAG. In fwSkitter, such an application DAG is called a *workflow*. A workflow is a pair, (N, L), where N represents the set of nodes in the application DAG, and L represents the data dependencies between these nodes, here called links.

$$\textbf{Workflow} := (N, L)$$

A node in a fwSkitter application contains both distribution and application logic, separated into two different abstractions. A node is a pair: $\mathcal{N}\langle s, \Omega\rangle$, which consists of a strategy, s, and an operation, Ω. The former defines the distribution logic of the node, while the latter defines the data processing logic. We discuss how operations and strategies are expressed later in this section. fwSkitter does not provide a dedicated abstraction for representing sources or sinks; these are all expressed as nodes.

$$n \in N \subseteq \textbf{Node} := \mathcal{N}\langle s, \Omega\rangle$$

Data dependencies in fwSkitter are defined as *links*. A link is specified as a three-tuple, $\mathcal{L}\langle n_f, n_t, i\rangle$, which contains the node sending the data (n_f), the node receiving the data (n_t) and an index, i, which we refer to as a *port*. This port acts

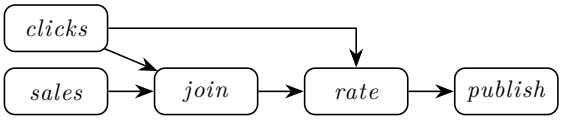

Figure 4.2: Application DAG of Adalyze expressed in fwSkitter.

as an argument index; when a node receives a data record, its strategy can use this port number to identify the origin of the data record while remaining agnostic to the upstream node which emitted the data record.

$$L \subseteq \textbf{Link} \coloneqq \mathcal{L}\langle n, n, i\rangle$$
$$i \in \textbf{Port} \subseteq \mathbb{N}$$

example

In Chapter 1, we introduce our running example, Adalyze. Adalyze can be expressed as fwSkitter workflow, (N, L), shown in Figure 4.2. Each node in this figure corresponds to an element of N, in contrast to fwStorm which requires certain operations to be represented by several nodes, as discussed in Section 3.2.1. Thus, $N = \{clicks, sales, join, rate, publish\}$. Every edge in Figure 4.2 corresponds to a link in L. To ensure $join$ can distinguish incoming events emitted by $clicks$ from those emitted by $sales$, each link has a unique port: $\{\mathcal{L}\langle clicks, join, 0\rangle, \mathcal{L}\langle sales, join, 1\rangle\} \subset L$.

4.1.2 Defining Operations

Operations constitute the data processing logic of a single data processing step (i.e. of a node). This abstraction tackles two key issues:

- In Section 2.2.1, we discuss how the distribution logic of an operation differs based on its properties. For instance, the state of stateful operations is often split based on some key; extracting this key from an incoming data record is data processing logic, which should thus be contained within the operation. However, other types of operations exist, which may not need to extract such a key. Therefore, the functionality that an operation must provide is not fixed but depends on its distribution strategy.
- Operations often need to read or modify state. However, it is the strategy which should specify which state an operation can access and what happens with the (potentially modified) state afterwards, as this is distribution logic.

To tackle the first issue, fwSkitter defines operations as a set of *callbacks*, each of which is associated with a name. An operation definition in fwSkitter is a mapping associating callback names with callback functions (discussed next). An operation

in fwSkitter may contain an arbitrary number of callbacks; moreover, fwSkitter does not impose any required callbacks. Instead, the operation needs to provide the callbacks used by the strategy paired with the operation. This enables the developer of a strategy to specify which parts of logic are data processing logic and which parts are distribution logic. Forcing an operation to implement the callbacks used by the strategy it is paired with does introduce a level of coupling between the data processing logic and distribution logic of the application. We elaborate on this in Section 6.1.2.

$$\Omega \in \textbf{Operation} ::= cb \mapsto \rho$$
$$cb \in \textbf{CallbackName}$$

example

The *rate* operation of Adalyze is stateful. Its state is partitioned by key. Since extracting this key is data processing logic, the strategy associated with this operation specifies that the *rate* operation must define a *key* callback to determine the key of a data record. Additionally, *rate* should also define a *react* callback which will serve to respond to an incoming data record. Thus, $rate = \mathcal{N}\langle s_{rate}, \left[key \mapsto \rho_{key}, react \mapsto \rho_{react}\right]\rangle$.

To tackle the second issue, fwSkitter conceives callbacks as functions which accept the arguments passed to the callback and the state the callback may access. The callback function also returns a potentially updated state and a "regular" return value. Similar to fwStorm, all these data elements are a member of the data domain, $\mathbb{D}$, which contains any value processed by a stream processing application.

$$\rho \in \textbf{Callback} ::= d, d \mapsto d, d$$
$$d \in \mathbb{D} = \textbf{DataDomain}$$

example

The strategy of the *rate* operation calls ρ_{react} with an incoming data record and the state of the key associated with the data record (obtained by invoking ρ_{key}). Based on the data record, ρ_{react} will update the received state, which is returned to the distribution strategy of *rate*. This strategy will also send the value returned by ρ_{react} to nodes downstream of *rate*.

4.1.3 Defining Distribution Strategies

Distribution strategies define how operations are distributed over a cluster. They act as a meta-layer which controls when and where the various callbacks defined by an operation are called. Strategies are implemented by defining several *hooks*, which are invoked in response to predefined events. When invoked, these hooks may call any callback of the operation paired with the distribution strategy to execute data processing logic.

fwSkitter features three hooks: *deploy*, *deliver* and *process*. The `deploy` hook is responsible for creating an initial distribution of the operation over the cluster. The `deliver` hook is responsible for sending a data element sent from a predecessor to a worker of the operation, while the `process` hook is responsible for specifying how such a worker processes a received message.

example

A common strategy for distributing stateful operations such as *rate* is to partition their state based on some key, as discussed in Section 2.2. This strategy can be expressed in terms of the aforementioned hooks as follows:

`deploy` spawns several workers, which will each maintain a part of the state of the operation.

`deliver` ensures that an incoming data record is sent to the worker that maintains the state for the key associated with that data record.

`process` invokes the ρ_{react} callback of the operation with the appropriate state.

An exact definition of each hook is provided later in this section.

A hook in fwSkitter is an expression. A distribution strategy in fwSkitter is a pair of these expressions: $\mathcal{S}\langle e, e\rangle$. The first expression represents the `deploy` hook of the strategy, the second represents the `deliver` hook. The `process` hook is not specified as a part of the strategy pair. Instead, a worker is created with an expression which specifies the `process` hook of that particular worker. This enables a fwSkitter strategy to create multiple workers which each serve a different role, which can be used to express complex distribution strategies, as discussed in Section 2.2.3.

$$s \in \textbf{Strategy} ::= \mathcal{S}\langle e, e\rangle$$

Expressions

fwSkitter introduces several expressions to define strategies. The exact semantics of these expressions is defined in Section 4.2.

$$\begin{aligned} e \in \textbf{Expression} ::= \quad & e; e \mid x \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{let}\ x_s, x_r = \texttt{call}(cb, e, e)\ \texttt{in}\ e \\ \mid \quad & \texttt{null} \mid \texttt{self} \mid \texttt{deployment} \mid \texttt{msg} \mid \texttt{state} \mid \texttt{data} \mid \texttt{port} \\ \mid \quad & \texttt{spawn}(e, e) \mid \texttt{send}(e, e) \mid \texttt{emit}(e) \\ \mid \quad & e[e] \mid \overline{e}^{\,k} \mid \overline{\texttt{send}}(e, e) \end{aligned}$$

$$x, x_s, x_r \in \textbf{VarName}, k \in \mathbb{N}$$

We make a distinction between primitive expressions, syntactic sugar and pseudo-variables. The most important primitives are defined below:

$$e; e' \stackrel{def}{=} \texttt{let}\ x = e\ \texttt{in}\ e' \qquad x \notin FV(e')$$

$$\overline{e}^{\,0} \stackrel{def}{=} \varnothing \qquad \overline{\texttt{send}}(r \cdot \overline{r}, h) \stackrel{def}{=} \texttt{send}(r, h); \overline{\texttt{send}}(\overline{r}, h)$$

$$\overline{e}^{\,k} \stackrel{def}{=} e \cdot \overline{e}^{\,k-1} \qquad \overline{\texttt{send}}(\varnothing, h) \stackrel{def}{=} \texttt{null}$$

Figure 4.3: Abbreviations used in fwSkitter. $v \cdot \overline{v}$ denotes prepending v to $\overline{v}$.

`let` $x_s, x_r =$ `call`(cb, d_s, d_a) `in` e is used to call an operation callback from within the strategy. This expression provides the bridge between the data processing logic defined by the operation and the distribution logic defined by the strategy, as it allows the latter to invoke the former.

`spawn`(h, e) creates a worker with an initial state, h (discussed in Section 4.2), and its `process` hook: e. This expression is reduced when the spawned worker receives a message to process. A `spawn` expression reduces to a reference to the created worker, which can be used to send a message to the worker.

`send`(r, h) sends a message with payload h to the worker with reference r. `send` is used to communicate between workers created by the same strategy.

`emit`(d) publishes a data record, d, from the current node. In contrast to `send`, which is responsible for distribution-level communication (i.e. communication between the workers of a strategy), `emit` is responsible for application-level communication (i.e. communication between nodes in the application DAG). The use of the emit expression will cause the `deliver` hook of all connected downstream nodes to be invoked, as we discuss in Section 4.2.

Besides these primitives, fwSkitter also defines variables (x), constructs for binding them (`let` $x = e$ `in` e), and syntax for indexing vectors ($v[i]$). fwSkitter also defines several expressions in terms of its primitive expressions: $e; e'$ denotes a sequence of expressions, $\overline{e}^{\,k}$ is used to execute expression e k times and gather the results of each expression in a vector, and $\overline{\texttt{send}}(\overline{r}, h)$ is used to send h to every worker with a reference in vector $\overline{r}$. The definition of these expressions is shown in Figure 4.3.

fwSkitter also features a number of pseudovariables which get bound during the reduction of specific hook expressions:

`deployment` is bound when the expression of the `process` or `deliver` hook is reduced. This pseudovariable can be used to access the so-called *deployment data*, which contains the return value of the `deploy` hook.

`data` **and** `port` are bound when the expression of the `deliver` hook is reduced and refer to the data record to deliver and the port the data record was received on, respectively.

`self`, `msg`, and `state` are bound when the expression of the `process` hook is reduced and refer to the reference of the worker which is processing the message, the received message and the state of the worker.

example

The distribution strategy of the *rate* operation is $\mathcal{S}\langle e_{depl}, e_{delv}\rangle$. The hooks of this strategy are defined as follows:

`deploy` is invoked to create an initial distribution of the operation over the cluster; it spawns two workers, one for each cluster node. The references to the created workers are returned as a vector which will be stored as the deployment data of the *rate* node. The workers are spawned with an empty state and process hook e_{proc}, which we define below.

$$e_{depl} = \overline{\texttt{spawn}(\varnothing, e_{proc})}^{\,2}$$

`deliver` is invoked when a predecessor of *rate* emits data; it uses the *key* callback of the operation to obtain the key associated with the incoming data record. It hashes this key and reduces it modulo the amount of workers to select the worker associated with this key from the deployment data. Finally, it sends the data record to the selected worker.

$$\begin{aligned} e_{delv} = {} & \texttt{let } \_, x_k = \texttt{call}(\mathit{key}, \varnothing, \texttt{msg}) \texttt{ in} \\ & \texttt{send}(\texttt{deployment}[\text{hash}(x_k) \bmod |\texttt{deployment}|], \texttt{msg}) \end{aligned}$$

`process` is invoked when a worker for *rate* receives data. It fetches the state associated with this data (through the use of *key*), after which it calls *react* with the received data record and this state. *react* yields an updated state and a return value, the former of which is stored inside the worker, while the latter is emitted, thus causing it to be sent to downstream nodes.

$$\begin{aligned} e_{proc} = {} & \texttt{let } \_, x_k = \texttt{call}(\mathit{key}, \varnothing, \texttt{msg}) \texttt{ in} \\ & \texttt{let } x_s, x_r = \texttt{call}(\mathit{react}, \texttt{state}[x_k], \texttt{msg}) \texttt{ in} \\ & \texttt{emit}(x_r); \texttt{state}\,[x_k \mapsto x_s] \end{aligned}$$

4.1.4 Summary

Figure 4.1 presents the complete abstract syntax of fwSkitter. A fwSkitter application is defined as a workflow, which is a DAG consisting of several nodes (N) and links (L), which represent data dependencies between nodes. Each node ($n = \mathcal{N}\langle s, \Omega\rangle \in N$) is a pair which contains an operation (Ω) and a distribution strategy (s). Operations are a collection of callbacks, which are functions that

map a state and a value to a potentially updated state and a new value. Distribution strategies are defined in terms of three hooks, which determine how the operation is deployed over the cluster (deploy), how a data record produced by a predecessor node is sent to a worker (deliver), and how a worker processes a received message (process). Each hook is an expression (e). In the next section, we discuss the exact behaviour of these constructs.

4.2 Operational Semantics

As with fwStorm, the semantics of fwSkitter is defined as a small-step operational semantics. This semantics operates on "configurations", which represent the state of a fwSkitter application at run-time. Such a configuration is a pair: (Δ, W). Here, Δ represents a mapping between nodes $n \in N$ and their associated deployment data, created by the `deploy` hook of the strategy of the node. W is the set of workers created for the nodes in the application. Each worker, $w = \mathcal{W}\langle r, n, e, h, \overline{q}\rangle \in W$ consists of a reference, r, which uniquely identifies this worker, a node, n, which associates the worker with the node for which the worker was created, and an expression, e, which is the `process` hook to be reduced when the worker processes data. Like fwStorm tasks, the final two values of the worker tuple represent the state of the worker (h) and its message queue ($\overline{q}$). Unlike fwStorm, however, the domain of these values is not the data domain ($\mathbb{D}$); instead, the state of the worker and the messages in the worker queue are part of the *hybrid domain*, $\mathbb{H}$, which can contain both application-level values (i.e. $d \in \mathbb{D}$) or distribution-level values (i.e. worker references or a vector thereof).

$$
\begin{aligned}
\textbf{Configuration} &::= (\Delta, W) \\
\Delta \subseteq \textbf{Deployment} &::= n \mapsto h \\
W \subseteq \textbf{Worker} &::= \mathcal{W}\langle r, n, e, h, \overline{q}\rangle \\
r \in R \subseteq \textbf{WorkerRef} & \\
h \in \mathbb{H} = (\mathbb{D} \cup R \cup R^n) &= \textbf{HybridDomain} \\
\overline{q} \in \mathbb{H}^n &
\end{aligned}
$$

fwSkitter's reduction rules operate on so-called "run-time expressions", which include any expression shown in Figure 4.1, as well as references to workers, $r \in R$, as expressions may reduce to these references before being reduced further.

$$e ::= \ldots \mid r$$

fwSkitter uses evaluation contexts [Felleisen and Hieb, 1992] in its reduction relations to define the order in which subexpressions should be reduced, as shown in

$$
\begin{aligned}
\mathcal{E} ::= \quad & \square \mid \mathcal{E}[e] \mid h[\mathcal{E}] \mid \texttt{let}\ x = \mathcal{E}\ \texttt{in}\ e \\
\mid \quad & \texttt{spawn}(\mathcal{E}, e) \mid \texttt{send}(\mathcal{E}, e) \mid \texttt{send}(r, \mathcal{E}) \mid \texttt{emit}(\mathcal{E}) \\
\mid \quad & \texttt{let}\ x_s, x_r = \texttt{call}(cb, \mathcal{E}, e)\ \texttt{in}\ e \mid \texttt{let}\ x_s, x_r = \texttt{call}(cb, d, \mathcal{E})\ \texttt{in}\ e
\end{aligned}
$$

Figure 4.4: Order in which subexpressions are reduced in fwSkitter.

$$
\begin{aligned}
[h/x]x' &= x' & [h/x]\texttt{self} &= \texttt{self} \\
[h/x]x &= h & [h/x]\texttt{deployment} &= \texttt{deployment} \\
[h/x]e[e'] &= [h/x]e[[h/x]e'] & [h/x]\texttt{msg} &= \texttt{msg} \\
[h/x]\texttt{spawn}(e, e') &= \texttt{spawn}([h/x]e, e') & [h/x]\texttt{state} &= \texttt{state} \\
[h/x]\texttt{send}(e, e') &= \texttt{send}([h/x]e, [h/x]e') & [h/x]\texttt{data} &= \texttt{data} \\
[h/x]\texttt{emit}(e) &= \texttt{emit}([h/x]e) & [h/x]\texttt{port} &= \texttt{port} \\
[h/x]\texttt{null} &= \texttt{null}
\end{aligned}
$$

$$
[h/x]\texttt{let}\ x' = e\ \texttt{in}\ e' \quad = \quad \begin{cases} \texttt{let}\ x' = [h/x]e\ \texttt{in}\ [h/x]e' & x \neq x' \\ \texttt{let}\ x = [h/x]e\ \texttt{in}\ e' & x = x' \end{cases}
$$

$$
[h/x_s]\texttt{let}\ x_s, x_r = \texttt{call}(cb, e, e')\ \texttt{in}\ e'' \quad = \quad \begin{cases} \texttt{let}\ x_s, x_r = \texttt{call}(cb, [h/x]e, [h/x]e')\ \texttt{in}\ [h/x]e'' & x \notin \{x_r, x_s\} \\ \texttt{let}\ x_s, x_r = \texttt{call}(cb, [h/x]e, [h/x]e')\ \texttt{in}\ e'' & x \in \{x_r, x_s\} \end{cases}
$$

Figure 4.5: Substitution rules of fwSkitter.

Figure 4.4. $\mathcal{E}$ denotes an expression with a "hole" (represented as $\square$); $\mathcal{E}[e]$ denotes the expression obtained by replacing the hole with e. Finally, Figure 4.5 defines the substitution rules of fwSkitter; $[h/x]e$ denotes the substitution of h for x in e.

4.2.1 Initial configuration

Unlike fwStorm, fwSkitter does not spawn any workers for nodes automatically. Instead, workers are created by the strategy of a node, which also determines the values to be stored in the deployment data. The initial configuration of fwSkitter is therefore a pair consisting of an empty worker set and an empty deployment mapping:

$$(\varnothing, \varnothing)$$

(DEPLOY ALL) reduces this initial configuration to obtain a configuration that can be further reduced, as we explain in the next section.

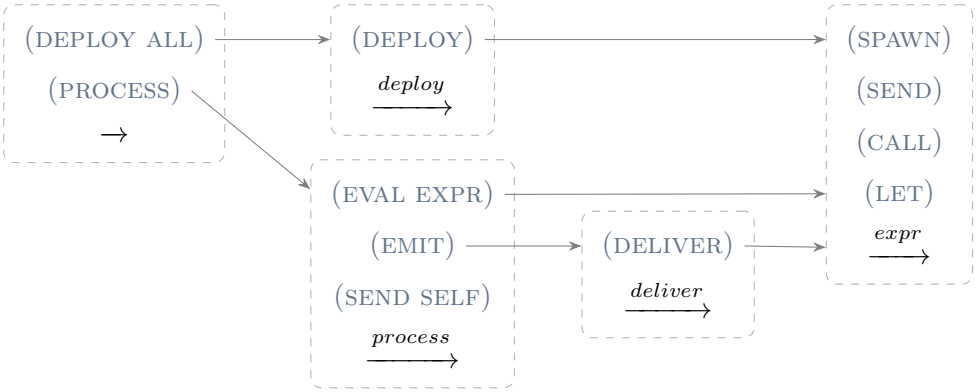

Figure 4.6: Schema of the reduction rules and relations of fwSkitter.

4.2.2 Reduction Rules

There are two global reduction rules: (DEPLOY ALL) and (PROCESS). During the execution of an application, (N, L), the first rule, (DEPLOY ALL), is applied once to create an initial deployment of the application. Afterwards, (PROCESS) is applied repeatedly to process a data record in the queue of a worker. These global rules lead to the application of several other rules, many of which can be applied by both (DEPLOY ALL) and (PROCESS). Figure 4.6 shows an overview of fwSkitter's reduction rules and how they relate to each other. During the application of all rules, N and L are available globally.

Deploying an Application

To deploy an application, (DEPLOY ALL) is applied. This rule deploys every node in turn through $\xrightarrow{deploy}$, the behaviour of which is defined by (DEPLOY). This rule creates an initial set of workers, W, and a deployment, Δ. After the application of (DEPLOY ALL), the deployment of an application, Δ, does not change.

(DEPLOY ALL)

$$\frac{\varnothing, \varnothing, N \xrightarrow{deploy^*} \Delta, W, \varnothing}{(\varnothing, \varnothing) \rightarrow (\Delta, W)}$$

(DEPLOY) deploys a single node by reducing the `deploy` expression of the strategy of the node, e, to a value, h, obtained by traversing $\xrightarrow{expr}$, which we discuss later. Afterwards, h, the value obtained from the reduction of e, is stored as the deployment data for the node, while W_n, the set of workers created during the reduction of e, the `deploy` hook, is added to W.

(DEPLOY)

$$\frac{n = \mathcal{N}\langle \mathcal{S}\langle e, \_\rangle, \_\rangle \qquad e, n, \varnothing \xrightarrow{expr^*} h, n, W_n}{\Delta, W, N_d \uplus \{n\} \xrightarrow{deploy} \Delta\left[n \mapsto h\right], W \cup W_n, N_d}$$

The deployment data of a node, h, can be any value. However, the `deploy` expression often contains several `spawn` subexpressions, which create workers. The references to these workers are then typically returned to be a part of the deployment data so that they can be used by the `deliver` hook later on.

example

When Adalyze is deployed, (DEPLOY) is applied with $n = rate$. This leads to the reduction of its `deploy` hook, $\overline{\texttt{spawn}(\varnothing, e_{proc})}^{\,2}$, which reduces to a vector of worker references: $[r_1, r_2]$. This vector is then stored as the deployment data of $rate$, while the spawned workers are added to W. Later, the `deliver` hook will use these references to message these workers.

Processing Messages

The second global reduction rule, (PROCESS), can only be applied when a worker, $w \in W$ has a message, h_q, in its queue. When this happens, this rule can be applied to reduce the expression stored in the worker (i.e. its `process` hook) to a value through the $\xrightarrow{process}$ relation. The value obtained from this reduction, h, will become the new state of the worker which received the message. Before $\xrightarrow{process}$ is used to reduce the `process` hook associated with the worker, the received message, current state of the worker, deployment data, and the reference to the worker that received the message are substituted into the expression.

(PROCESS)

$$\frac{\begin{array}{c} e = [h_q/\texttt{msg}][h_s/\texttt{state}][\Delta\left(n\right)/\texttt{deployment}][r/\texttt{self}]e_p \\ e, \mathcal{W}\langle r, n, e_p, h_s, \overline{q}\rangle, (\Delta, W) \xrightarrow{process^*} h, \mathcal{W}\langle r, n, e_p, h_s, \overline{q}'\rangle, (\Delta, W') \end{array}}{\left(\Delta, W \uplus \left\{\mathcal{W}\langle r, n, e_p, h_s, \overline{q} \cdot h_q\rangle\right\}\right) \rightarrow \left(\Delta, W' \cup \left\{\mathcal{W}\langle r, n, e_p, h, \overline{q}'\rangle\right\}\right)}$$

example

When a worker of the $rate$ node receives a message (sent by its `deliver` hook, discussed later), its `process` hook, shown below and discussed in Section 4.1.3, is invoked. The received data record and state of the worker is substituted into this expression, after which it is reduced. During the reduction of this expression, the `emit` expression (discussed later) is reduced, which may cause W to be modified. The final value that the `process` expression reduces to (the state, with a new value for x_k) is stored to be used as the state of the worker

for the next time (PROCESS) is applied.

example

$$\begin{aligned}&\texttt{let } \_, x_k = \texttt{call}(key, \varnothing, \texttt{msg}) \texttt{ in}\\ &\quad \texttt{let } x_s, x_r = \texttt{call}(react, \texttt{state}[x_k], \texttt{msg}) \texttt{ in}\\ &\quad\quad \texttt{emit}(x_r); \texttt{state}\,[x_k \mapsto x_s]\end{aligned}$$

Several reduction rules may be applied when processing a message. The first of these rules, (EVAL EXPR) specifies that any expression, e, that can be reduced to another expression, e' through $\xrightarrow{expr}$ can also be reduced to the same e' through $\xrightarrow{process}$.

(EVAL EXPR)

$$\frac{w = \mathcal{W}\langle \_, n, \_, \_, \_\rangle \qquad e, n, W \xrightarrow{expr} e', n, W'}{e, w, (\Delta, W) \xrightarrow{process} e', w, (\Delta, W')}$$

(SEND) (which we discuss below) reduces the `send` expression to send a message to a worker $w \in W$. However, when (PROCESS) is applied, the worker which is processing the received message, w, is not a part of $W : w' \notin W$. Therefore, (SEND SELF) enables a worker to send a message to itself during the application of (PROCESS).

(SEND SELF)

$$\begin{aligned}&\mathcal{E}[\texttt{send}(r, h_q)], \mathcal{W}\langle r, n, e, h_s, \overline{q}\rangle, (\Delta, W)\\ &\xrightarrow{process} \mathcal{E}[\texttt{null}], \mathcal{W}\langle r, n, e, h_s, h_q \cdot \overline{q}\rangle, (\Delta, W)\end{aligned}$$

Emitting Values

While a message is being processed, the `emit` expression can emit a value. This expression causes the value, d, to be forwarded to any node downstream of the current node in the application DAG.

(EMIT) is responsible for sending d to each downstream node. It constructs a set of pairs, N_e, wherein each pair corresponds to an outgoing edge from the node which is emitting d. These pairs contain a downstream node, n_t, and a port, i. (DELIVER) is then applied to each pair $\in N_t$.

(EMIT)

$$\frac{\begin{array}{c} w = \mathcal{W}\langle \_, n, \_, \_, \_\rangle \\ N_e = \{(n_t, i) \mid \mathcal{L}\langle n, n_t, i\rangle \in L\} \qquad d, N_e, (\Delta, W) \xrightarrow{deliver^*} d, \varnothing, (\Delta, W') \end{array}}{\mathcal{E}[\texttt{emit}(d)], w, (\Delta, W) \xrightarrow{process} \mathcal{E}[\texttt{null}], w, (\Delta, W')}$$

A single application of (DELIVER) delivers a data record, d, to a single downstream node, n, by invoking the `deliver` hook of its distribution strategy. Before this is done, the incoming data, d, the port on which the data is received, i, and the deployment data of n are substituted into the expression.

(DELIVER)

$$\frac{n = \mathcal{N}\langle \mathcal{S}\langle \_, e\rangle, \_\rangle \qquad [d/\texttt{data}][i/\texttt{port}][\Delta(n)/\texttt{deployment}]e, n, W \xrightarrow{expr^*} \texttt{null}, n, W'}{d, N_e \uplus \{(n,i)\}, (\Delta, W) \xrightarrow{deliver} d, N_e, (\Delta, W')}$$

example

When a predecessor of *rate* (i.e. *join* or *clicks*) emits a value, d, (EMIT) is applied to deliver d to each downstream node of the emitting node. This eventually causes (DELIVER) to be applied with $n = rate$. This rule evaluates the `deliver` hook of the strategy of *rate*, discussed in Section 4.1 and shown below, substituting d, the port on which *rate* receives d, and the deployment data into the expression.

$$\begin{aligned}&\texttt{let } \_, x_k = \texttt{call}(key, \varnothing, \texttt{msg}) \texttt{ in}\\ &\quad \texttt{send}(\texttt{deployment}[\text{hash}(x_k) \bmod |\texttt{deployment}|], \texttt{msg})\end{aligned}$$

The expression uses the *key* callback to obtain the key associated with the incoming data record and uses this key to fetch a worker reference from the deployment. d is then sent to this reference, modifying W.

Shared Expressions

Several expressions can be used by `deploy`, `deliver` and `process`. The rules which reduce these expressions all define a traversal of the $\xrightarrow{expr}$ relation. We discuss each of these rules below.

`spawn` creates a new worker for a node. A worker is always associated with the node that spawned it and created with an initial state (h) and a process hook (e). While `spawn` may be used in any hook, it is particularly useful to create an initial set of workers in `deploy`.

(SPAWN)

$$\frac{r \text{ fresh}}{\mathcal{E}[\texttt{spawn}(h, e)], n, W \xrightarrow{expr} \mathcal{E}[r], n, W \cup \{\mathcal{W}\langle r, n, e, h, \varnothing\rangle\}}$$

`send` sends a message, h_q, to a worker based on its reference, r. The message is sent by adding it to the queue of the worker. This expression is typically used

by `deliver` to send data to a worker to be processed, but may also be used by `process` to express worker to worker communication, or by `deploy` to ensure a (source) node can start processing once deployment is complete.

(SEND)

$$\mathcal{E}[\texttt{send}(r, h_q)], n, W \uplus \{\mathcal{W}\langle r, n, e, h_s, \overline{q}\rangle\} \xrightarrow{expr} \mathcal{E}[\texttt{null}], n, W \cup \{\mathcal{W}\langle r, n, e, h_s, h_q \cdot \overline{q}\rangle\}$$

`call` calls a callback of an operation from the strategy language. (CALL) fetches the current node, after which the appropriate callback is retrieved from the operation of the node and invoked with the provided arguments and state. The resulting values are substituted into the remaining expression, e.

(CALL)

$$\frac{n = \mathcal{N}\langle \_, \Omega\rangle \qquad \rho = \Omega\,(cb) \qquad d_s', d_r = \rho(d_s, d_a)}{\mathcal{E}[\texttt{let}\ x_s, x_r = \texttt{call}(cb, d_s, d_a)\ \texttt{in}\ e], n, W \xrightarrow{expr} \mathcal{E}[[d_s'/x_s][d_r/x_r]e], n, W}$$

`let` binds a variable x in e by substituting its value into e.

(LET)

$$\mathcal{E}[\texttt{let}\ x = h\ \texttt{in}\ e], n, W \xrightarrow{expr} \mathcal{E}[[h/x]e], n, W$$

4.2.3 Summary

Similar to fwStorm, a fwSkitter application is a perpetually running set of workers. fwSkitter defines two global reduction rules, (DEPLOY ALL) and (PROCESS). The former creates an initial configuration for the application, which contains the set of workers that execute expressions for the application. The second processes a data record in the queue of any worker, similar to the way the (BOLT) rule functions in fwStorm. The other rules are used to precisely capture the behaviour of the various expressions fwSkitter introduces for the expression of distribution strategies.

4.3 Discussion

fwSkitter is a layered model for the expression of DSPAs. In this model, *operations* express data processing logic, while *distribution strategies* express distribution logic. The *workflow* layer is used to tie operations to distribution strategies and combines these into a data processing application.

The unique insight made explicit by fwSkitter is the control it offers over distribution strategies: by offering a dedicated abstraction for the implementation of these

strategies, these can be implemented and swapped out at will. At run-time, these strategies offer the developer fine-grained control over how the computations of their operations are distributed over the various workers and how these communicate with one another. In the next chapter, we show this programming model does not come at the cost of expressivity by proving fwSkitter is Storm-complete.

5. FeatherweightSkitter is Storm-complete

In Chapter 3, we formally define the notion of "Storm-completeness": a lower bound on the expressiveness we have argued any DSPF which aims to enable the expression of distribution strategies should meet. In this chapter, we show that fwSkitter (c.f. Chapter 4) is Storm-complete. Concretely, we prove that any application expressed in fwStorm can be expressed as a fwSkitter application with identical behaviour, showing that our programming model is at least as expressive as the model offered by (featherweight) Storm. We do this in two steps:

- In Section 5.1, we define a function which can transform any fwStorm application into a corresponding fwSkitter application, thus showing fwSkitter is sufficiently expressive to express any fwStorm application.
- In Section 5.2, we prove that any application translated as described in Section 5.1 has the exact same behaviour as the original fwStorm application, thus showing that fwSkitter is Storm-complete.

After showing fwSkitter is Storm-complete, we briefly discuss if fwSkitter applications can be translated to fwStorm.

We remind the reader that Appendix A contains a complete reference of fwStorm, fwSkitter, and the translation between both. This can be especially useful when reading this chapter.[1]

5.1 Translating FeatherweightStorm to FeatherweightSkitter

We define *translate* to translate fwStorm applications into a fwSkitter application which simulates its behaviour. The fwSkitter applications generated by *translate*

[1] As before, symbols links to their definition in this appendix.

still suffer from many of the shortcomings of (featherweight) Storm. The goal of *translate* is not to define how a fwStorm application should be converted into a fwSkitter application; instead, it is to show that fwSkitter can accurately model the run-time behaviour of any fwStorm application (as we prove in Section 5.2). We discuss this and other limitations in Section 5.1.4.

Our translation occurs in two steps. The first step, embodied by *translate*, adds an index, which corresponds with the fwSkitter notion of a port, to each grouping $g \in G$. The second step translates each component and grouping in the application to its fwSkitter counterpart.

$$translate : \mathcal{P}(\textbf{Component}) \times \mathcal{P}(\textbf{Grouping}) \mapsto \mathcal{P}(\textbf{Node}) \times \mathcal{P}(\textbf{Link})$$
$$translate((C, G)) = translate'(C, addport(C, G))$$

translate uses *addport* to add a port to each grouping, after which it applies the second stage, *translate′*. *addport* adds an index to each grouping $g \in G$, ensuring that every grouping with a component c as its destination is associated with a different index $i \in [0, |\{\mathcal{G}\langle \_, c, \_, \_\rangle \in G\}|[\subset \mathbb{N}$. This is done by applying *addport′* on the set of all groupings with destination c for every $c \in C$.

$$addport : \mathcal{P}(\textbf{Component}) \times \mathcal{P}(\textbf{Grouping}) \mapsto \mathcal{P}(\textbf{Grouping} \times \mathbb{N})$$
$$addport(C, G) = \{addport'(\{g \mid \mathcal{G}\langle \_, c, \_, \_\rangle \in G\}, 0) \mid c \in C\}$$
$$addport' : \mathcal{P}(\textbf{Grouping}) \times \mathbb{N} \mapsto \mathcal{P}(\textbf{Grouping} \times \mathbb{N})$$
$$addport'(\varnothing, \_) = \varnothing$$
$$addport'(G_c \cup \{g\}, i) = \{(g, i)\} \cup addport'(G_c, i + 1)$$

– example –
We discuss the fwStorm instantiation of Adalyze in Chapter 3. Its *rate* bolt has incoming groupings from $join_{joiner}$ and from *clicks*. *addport* gives each a unique index: $\left\{(\mathcal{G}\langle clicks, rate, \_, \_\rangle, 0), \left(\mathcal{G}\langle join_{joiner}, rate, \_, \_\rangle, 1\right)\right\} \subseteq G_i$.

The second step of the translation is embodied by *translate′*, which translates each component $c \in C$ to an equivalent fwSkitter node and every indexed grouping to an equivalent fwSkitter link, thus returning a fwSkitter application. The translation of components is handled by $trans_{c \to n}$, while the translation of groupings is handled by $trans_{g \to l}$. To generate a node definition from a component, c, we require information about the outgoing groupings of c. Therefore, *translate′* passes the complete set of indexed groupings to both $trans_{c \to n}$ and $trans_{g \to l}$.

$$translate' : \mathcal{P}(\textbf{Component}) \times \mathcal{P}(\textbf{Grouping} \times \mathbb{N}) \mapsto \mathcal{P}(\textbf{Node}) \times \mathcal{P}(\textbf{Link})$$
$$translate'(C, G_i) = \left(\{trans_{c \to n}(c, G_i) \mid c \in C\}, \left\{trans_{g \to l}(g, i, G_i) \mid (g, i) \in G_i\right\}\right)$$

A fwSkitter link is only responsible for storing a source and destination node and a port. Since *addport* already added a port to each grouping, translating a

fwStorm grouping to a fwSkitter link only requires fetching the translated source and destination nodes and pairing them with the port.

$$trans_{g\to l} : \textbf{Grouping} \times \mathbb{N} \times \mathcal{P}(\textbf{Grouping} \times \mathbb{N}) \mapsto \textbf{Link}$$
$$trans_{g\to l}\big(\mathcal{G}\langle c_f, c_t, \_, \_\rangle, i, G_i\big) = \mathcal{L}\langle trans_{c\to n}\big(c_f, G_i\big), trans_{c\to n}(c_t, G_i), i\rangle$$

Translating components is more difficult, as different logic is needed for bolts and spouts and since the generated fwSkitter nodes must also embed the communication logic fwStorm encodes in groupings. $trans_{c\to n}$ translates spouts and bolts differently, and uses helper functions to handle the communication logic, which is partly shared. We show the full definition of $trans_{c\to n}$ below. The following sections explain how this definition encodes the logic of spouts, bolts, and groupings.

$$trans_{c\to n} : \textbf{Component} \times \mathcal{P}(\textbf{Grouping} \times \mathbb{N}) \mapsto \textbf{Node}$$
$$trans_{c\to n}\big(s_p = \mathcal{S}_p\langle p, d_0, \sigma_s\rangle, G_i\big) = \mathcal{N}\langle\mathcal{S}\langle e_{dp}, \texttt{null}\rangle, [next \mapsto (d_s, \varnothing \mapsto \sigma_s(d_s))]\rangle$$

where

$$e_{pr} = \texttt{let } (x_s, x_\Gamma) = \texttt{state in}$$
$$\texttt{let } x'_s, x_r = \texttt{call}(next, x_s, \varnothing) \texttt{ in}$$
$$\texttt{let } x'_\Gamma, x_{\bar{i}} = emit_\Gamma(x_\Gamma, x_r) \texttt{ in}$$
$$\texttt{send}(\texttt{self}, \texttt{null}); \texttt{emit}((x_r, x_{\bar{i}})); (x'_s, x'_\Gamma)$$
$$e_{dp} = \overline{\texttt{send}\big(\texttt{spawn}\big(\big(d_0, trans_{c\to\Gamma}\big(s_p, G_i\big)\big), e_{pr}\big), \texttt{null}\big)}^{\,p}$$

$$trans_{c\to n}(b_o = \mathcal{B}_o\langle p, d_0, \sigma_b\rangle, G_i) = \mathcal{N}\langle\mathcal{S}\langle e_{dp}, e_{dl}\rangle, [react \mapsto (d_s, d_a \mapsto \sigma_b(d_s, d_a))]\rangle$$

where

$$e_{pr} = \texttt{let } (x_s, x_\Gamma) = \texttt{state in}$$
$$\texttt{let } x'_s, x_r = \texttt{call}(react, x_s, \texttt{msg}) \texttt{ in}$$
$$\texttt{let } x'_\Gamma, x_{\bar{i}} = emit_\Gamma(x_\Gamma, x_r) \texttt{ in}$$
$$\texttt{emit}((x_r, x_{\bar{i}})); (x'_s, x'_\Gamma)$$
$$e_{dp} = \overline{\texttt{spawn}\big(\big(d_0, trans_{c\to\Gamma}(b_o, G_i)\big), e_{pr}\big)}^{\,p}$$
$$e_{dl} = \texttt{let } (x_d, x_{\bar{i}}) = \texttt{data in}$$
$$\overline{\texttt{send}}\big(\texttt{deployment}[x_{\bar{i}}\big(trans_{b_o\to\bar{g}}(b_o, G_i)[\texttt{port}]\big)], x_d\big)$$

5.1.1 Translating Spouts

A spout $\mathcal{S}_p\langle p, d_0, \sigma_s\rangle$ is translated to a node, n, with an operation containing a single callback, $next$. When invoked, this callback applies σ_s, the spout's function.

$$\mathcal{N}\langle\mathcal{S}\langle e_{dp}, \texttt{null}\rangle, [next \mapsto (d_s, \varnothing \mapsto \sigma_s(d_s))]\rangle$$

The strategy of n uses `null` as its `deliver` expression, as source nodes do not receive data. e_{dp}, the `deploy` expression of n, creates p workers with the initial state of the spout, d_0, and sends a message to each to ensure they are ready to be processed by (PROCESS). Ignoring any grouping logic, e_{dp} is defined as follows:

$$\overline{\texttt{send}\big(\texttt{spawn}\big((d_0, \_)\,, e_{pr}\big), \texttt{null}\big)}^{\,p}$$

The `process` hook of the created workers, e_{pr}, calls the *next* callback with the state of the worker, x_s, to obtain the next data record, x_r, which is then published through the use of `emit`. The state returned by *next* is stored as the new state of the worker; `process` also sends itself a message to ensure the worker can be activated by (PROCESS) ad infinitum. Ignoring grouping logic once again, e_{pr} is defined as follows:

$$\begin{array}{l}\texttt{let } (x_s, \_) = \texttt{state in}\\ \quad \texttt{let } x'_s, x_r = \texttt{call}(next, x_s, \varnothing) \texttt{ in}\\ \quad\quad \texttt{let } \_, \_ = \_ \texttt{ in}\\ \quad\quad\quad \texttt{send}(\texttt{self}, \texttt{null}); \texttt{emit}((x_r, \_)); (x'_s, \_)\end{array}$$

5.1.2 Translating Bolts

The operation of a node translated from a bolt, $\mathcal{B}_o\langle p, d_0, \sigma_b\rangle$, contains a single callback, *react*, which applies σ_b:

$$\mathcal{N}\langle \mathcal{S}\langle e_{dp}, e_{dl}\rangle, [react \mapsto (d_s, d_a \mapsto \sigma_b(d_s, d_a))]\rangle$$

The `deploy` expression of the strategy of the generated node spawns p workers with initial state d_0. The references to these workers will be stored inside the deployment data of n. Without the inclusion of grouping logic, the definition of e_{dp} looks as follows:

$$\overline{\texttt{spawn}\big((d_0, \_)\,, e_{pr}\big)}^{\,p}$$

The `process` hook, e_{pr} will call *react* with the received message and the state of the worker when it receives a message. The returned value, x_r, is emitted while the returned state, x_s, is returned to be the new state of the worker. Omitting grouping logic one last time, the definition of e_{pr} follows below:

$$\begin{array}{l}\texttt{let } (x_s, \_) = \texttt{state in}\\ \quad \texttt{let } x'_s, x_r = \texttt{call}(react, x_s, \texttt{msg}) \texttt{ in}\\ \quad\quad \texttt{let } \_, \_ = \_ \texttt{ in}\\ \quad\quad\quad \texttt{emit}((x_r, \_)); (x'_s, \_)\end{array}$$

5.1.3 Translating Groupings

A major difference between fwStorm and `fwSkitter` is how they handle inter-node communication. In fwStorm this is handled through the use of the grouping function, which may be stateful; in `fwSkitter` this is handled through the `deliver` hook, which cannot access any mutable state. In order for our translation to accurately simulate fwStorm's behaviour, we need to express it in `fwSkitter`. This is done by meeting the following requirements:

- Each worker must store the state of the outgoing groupings alongside the state of the operation.
- Before emitting a data record, this state must be used to determine which downstream workers will receive the data record.
- This information must be communicated to the `deliver` hook, as only the downstream strategy can send messages to its workers.

$trans_{c \to n}$ uses $trans_{c \to \Gamma}$ in the `deploy` hook of both bolts and spouts to ensure the state of each worker contains the operation's state and the state of its outgoing groupings, thus meeting the first requirement. Inside each `deploy` hook shown in the previous two sections, the `spawn` expression is now used as follows:

$$\texttt{spawn}((\_, trans_{c \to \Gamma}(c, G_i)), \_)$$

$trans_{c \to \Gamma}$ maps the outgoing groupings for c to their initial state as defined in G, analogous to fwStorm's $init_{\Gamma}$.

$$trans_{c \to \Gamma} : \textbf{Component} \times \mathcal{P}(\textbf{Grouping} \times \mathbb{N}) \mapsto \mathcal{P}(\textbf{Grouping} \times \mathbb{D})$$
$$trans_{c \to \Gamma}(c, G_i) = \{(g, d) \mid (g = \mathcal{G}\langle c, \_, d, \_\rangle, \_) \in G_i\}$$

example

The *clicks* component has two downstream bolts: $rate$ and $join_{sender}$. Therefore, $\left\{\left(g_r = \mathcal{G}\langle clicks, rate, d_r, \_\rangle, 0\right), \left(g_j = \mathcal{G}\langle clicks, join_{sender}, d_j, \_\rangle, 0\right)\right\} \subset G_i$. Thus, $trans_{c \to \Gamma}(clicks, G_i) = \left[g_r \mapsto d_r, g_j \mapsto d_j\right]$.

When $clicks'$, the translated version of *clicks* has been deployed, it will contain both its own state, as well as the mapping shown above.

To meet the second requirement, we use the groupings' state inside the `process` hook of spouts and bolts to apply the grouping function of each outgoing grouping. This is done in two steps. First, the state of the operation and the state of the various groupings are extracted from the worker's state:

$$\texttt{let } (x_s, x_\Gamma) = \texttt{state in } \ldots$$

Afterwards, after the data record to emit is calculated by the spout or bolt logic (i.e. by calling *next* or *react*, respectively), each grouping stored in x_Γ is applied with its current state and the data record to emit:

$$\ldots \texttt{let } x'_\Gamma, x_{\overline{i}} = emit_\Gamma(x_\Gamma, x_r) \texttt{ in } \ldots$$

This is handled by $emit_\Gamma$, which calls the functions of the appropriate groupings, returning a mapping which maps each grouping to its updated state and another mapping which maps each grouping to its result:

$$emit_\Gamma : \mathcal{P}(\textbf{Grouping} \times \mathbb{D}) \times \mathbb{D} \mapsto \mathcal{P}(\textbf{Grouping} \times \mathbb{D}) \times \mathcal{P}(\textbf{Grouping} \times \mathbb{N}^*)$$
$$emit_\Gamma(\Gamma, d) = \left(\{(g, d_s) \mid (g, (d_s, \_)) \in X\}, \{(g, \overline{i}) \mid (g, (\_, \overline{i})) \in X\}\right)$$
where
$$X = \left\{(g, \sigma_g(d_s, d, p)) \mid (g = \mathcal{G}\langle \_, \mathcal{B}_o\langle p, \_, \_\rangle, \_, \sigma_g\rangle, d_s) \in \Gamma\right\}$$

example

When $clicks'$ emits d, $emit_\Gamma$ will be applied to $\left[g_r \mapsto d_r, g_j \mapsto d_j\right]$ and d. This will lead to the application of the grouping functions of g_r and g_j. These functions will be invoked with d, with the current state of the grouping (i.e. with d_r or d_j), and with the parallelism of the downstream bolt in the original fwStorm application (i.e. with the parallelism of $rate$ and $join_{sender}$).

This leads to a result mapping, $\left[g_r \mapsto \left(d'_r, \overline{i_r}\right), g_j \mapsto \left(d'_j, \overline{i_j}\right)\right]$, which will be split into two individual mappings: one containing the new state of the groupings: $\left[g_r \mapsto d'_r, g_j \mapsto d'_j\right]$, and another with the indices to send d to: $\left[g_r \mapsto \overline{i_r}, g_j \mapsto \overline{i_j}\right]$. The former will be stored in the state of the worker, the latter will be emitted.

The `process` hook of both the translated bolts and spouts emit the result along with the produced value and store the returned grouping state along with the operation's state:

$$\ldots \texttt{emit}((x_r, x_{\overline{i}})); (x'_s, x'_\Gamma)$$

The `deliver` hook of the downstream strategy is invoked with a tuple containing both the data to send along with a mapping which maps each grouping to a vector of indices which correspond to workers. Thus, `deliver` must extract the indices of the correct grouping from this tuple, after which it can use these indices to obtain the appropriate worker references from the deployment data through the use of `deployment`[2]:

$$\texttt{let } (x_d, x_{\overline{i}}) = \texttt{data in } \overline{\texttt{send}}\Big(\texttt{deployment}[x_{\overline{i}}\left(trans_{b_o \to \overline{g}}(b_o, G_i)[\texttt{port}]\right)], x_d\Big)$$

[2]For convenience, indexing a vector, $\overline{v}$, with another vector, $\overline{i}$, returns a vector of elements in $\overline{v}$ whose index is contained in $\overline{i}$.

$trans_{b_o \to \overline{g}}$ gathers the incoming groupings of a bolt into a vector ordered by the port associated with each grouping, thus enabling the appropriate grouping to be found by indexing this vector with `port`.

$$trans_{b_o \to \overline{g}} : \textbf{Component} \times \mathcal{P}(\textbf{Grouping} \times \mathbb{N}) \mapsto \textbf{Grouping}^*$$

$$trans_{b_o \to \overline{g}}(b_o, G_i) = \overline{g}$$

where

$$\forall \{(g, i) \mid (g = \mathcal{G}\langle \_, b_o, \_, \_\rangle, i) \in G_i\} : \overline{g}[i] = g$$

example

The first example of this chapter discusses the two incoming groupings of *rate*: $\left\{\left(g_c = \mathcal{G}\langle clicks, rate, \_, \_\rangle, 0\right), \left(g_j = \mathcal{G}\langle join_{joiner}, rate, \_, \_\rangle, 1\right)\right\} \subseteq G_i$. When applied to *rate*, $trans_{b_o \to \overline{g}}$ encodes these groupings as a vector: $[g_c, g_j]$.

In the previous example, $clicks'$ produced data record d, which was emitted as $\left(d, \left[g_r \mapsto \overline{i_r}, g_j \mapsto \overline{i_j}\right]\right)$. Since $rate'$ is downstream of $clicks'$, this will lead to the reduction of the `deliver` hook of $rate'$:

$$\texttt{let}\ (x_d, x_{\overline{i}}) = \texttt{data}\ \texttt{in}\ \overline{\texttt{send}}\left(\texttt{deployment}[x_{\overline{i}}\left(trans_{b_o \to \overline{g}}(b_o, G_i)[\texttt{port}]\right)], x_d\right)$$

Substituting `data`, $trans_{b_o \to \overline{g}}(b_o, G_i)$ and `port` for their values yields:

$$\texttt{let}\ (x_d, x_{\overline{i}}) = \left(d, \left[g_r \mapsto \overline{i_r}, g_j \mapsto \overline{i_j}\right]\right)\ \texttt{in}\ \overline{\texttt{send}}\left(\texttt{deployment}[x_{\overline{i}}\left([g_c, g_j][0]\right)], x_d\right)$$

g_r and g_c both refer to the grouping linking *clicks* to *rate*, i.e. $g_r = g_c$, which means $x_{\overline{i}}\left(g_c\right) = \overline{i_r}$. Thus, we get:

$$\overline{\texttt{send}}\left(\texttt{deployment}[\overline{i_r}], d\right)$$

Which sends d to those workers whose reference is present in the deployment at an index present in $\overline{i_r}$

5.1.4 Limitations of the Translation

fwSkitter applications generated by *translate* suffer from fwStorm's shortcomings: distribution and application logic are entangled, and workflow nodes are created for each task type. The result of the translation has no qualitative guarantees. But it does demonstrate that there is no Storm program that cannot—at least theoretically—be expressed in Skitter.

The main difference between fwStorm and fwSkitter is how communication between nodes in the application graph is defined. In fwStorm, this is handled through the use of the grouping function, which may be stateful; in fwSkitter, this is handled through the use of the `deliver` hook which cannot access any mutable state. In

practice, this does not cause any issues when implementing distribution strategies, as many distribution strategies do not require stateful delivery logic; when stateful delivery logic is required, an intermediate worker can be used, as we discuss in Section 7.2.3. For the purposes of showing behavioural equivalence between fwStorm and fwSkitter, however, additional workers cause issues, which is why $translate$ encodes fwStorm's behaviour in fwSkitter.

5.2 Behavioural Equivalence of Translated Applications

Now that we have precisely defined how a fwStorm application, (C, G), can be translated into a fwSkitter application, $(N, L) = translate((C, G))$, we formally prove that any such translated application accurately simulates the behaviour of the original fwStorm application. The proof strategy is as follows:

- First, we show that the translated application is structurally equivalent to the original application (Lemmas 1 and 2).
- Second, we show that for every task created for the original application, there is exactly one worker created for the translated application, which we call the corresponding worker of the task (Lemmas 3 to 5).
- Third, we show that these workers behave in an identical fashion to their corresponding tasks when provided with identical input (Lemmas 6 and 7).
- Fourth, we show that communication between the workers is identical to the communication between tasks (Lemmas 8 to 11).
- Fifth and finally, we prove that each execution sequence of the original application has exactly one corresponding execution sequence in the translated application and vice versa. This shows that both applications have the exact same behaviour, which means any fwStorm application can be expressed in fwSkitter with identical behaviour, which shows fwSkitter is Storm-complete.

Figure 5.1 provides an overview of the Lemmas and how they relate to the execution of the fwStorm application and its fwSkitter counterpart.

5.2.1 Structural Equivalence

We begin by showing that (C, G) and $(N, L) = translate((C, G))$ have the same overall structure, i.e. that there is a corresponding node, $n \in N$, for every component, $c \in C$, and a corresponding link, $l \in L$, for every grouping, $g \in G$.

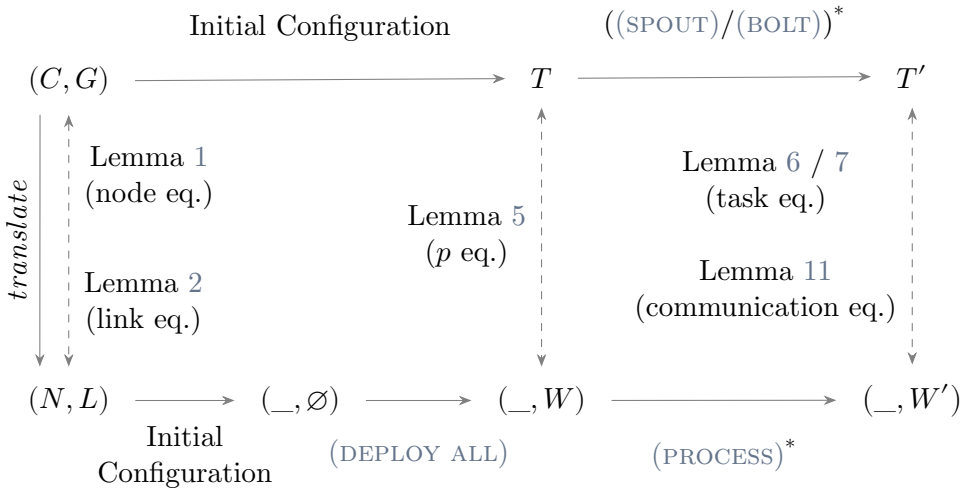

Figure 5.1: Overview of the main Lemmas used in the proof of Storm-completeness. The top row shows the execution of a fwStorm application, (C, G). First, its initial configuration, T, is created. After, (SPOUT) and (BOLT) are repeatedly applied to T, resulting in a new configuration, T'. The bottom row shows the execution of fwSkitter application $(N, L) = translate((C, G))$: from its initial configuration, the application is deployed, resulting in configuration $(\_, W)$. Afterwards, (PROCESS) is repeatedly applied, resulting in a new configuration, $(\_, W')$. The dashed arrows show how the Lemmas show equivalence between these stages: Lemmas 1 and 2 show both applications are structurally equivalent, Lemma 5 shows the equivalence of the created workers and tasks, Lemmas 6, 7 and 11 show this equivalence remains when (SPOUT), (BOLT), and (PROCESS) are applied (repeatedly).

Lemma 1: node equivalence

Let $(N, L) = translate((C, G))$. Each component, $c \in C$ has exactly one corresponding node, $n \in N$, and each node $n \in N$ has exactly one corresponding component $c \in C$.

Proof. *translate* applies *translate′*, which, in turn, applies $trans_{c \to n}$ on every component $c \in C$. Each application creates a single node, n. Every component $c \in C$ is unique, which means their combination of p, d, and σ_s or σ_b is unique. $trans_{c \to n}$ stores each of these elements inside n, making each n unique. Thus, every component, $c \in C$, has exactly one matching node, $n \in N$.

Showing the inverse follows the same reasoning: every $n \in N$ is created by $translate'$, which uses $trans_{c \to n}$ on every $c \in C$, which creates exactly one node for each component. Thus, every $n \in N$ has a corresponding component, i.e. the component from which it was created by $translate'$.

Lemma 2: link equivalence

Let $(N, L) = translate((C, G))$. Each grouping, $g \in G$ has exactly one corresponding link, $l \in L$, and every link $l \in L$ has exactly one corresponding grouping $g \in G$.

Proof. $translate$ uses $addport$ to add a port, i, to each $g \in G$, creating a set of indexed groupings, G_i, where any grouping with the same destination, c_t, is paired with a different port, i. $translate$ applies $translate'$, which applies $trans_{g \to l}$ on every $g = \mathcal{G}\langle \_, c_t, \_, \_ \rangle \in G_i$. Each application of $trans_{g \to l}$ creates a single link, $l = \mathcal{L}\langle \_, n_t, i \rangle$, where $n_t = trans_{c \to n}(c_t, G_i)$.

The combination of n_t and i is always unique, as Lemma 1 guarantees that n_t is unique, while $addport$ ensures each pairing of n_t and i is unique. Thus, l is unique, meaning every grouping, $g \in G$, has exactly one matching link, $l \in L$.

5.2.2 Corresponding Workers

Next, we show that every task created for a component in fwStorm has a corresponding worker in fwSkitter after the application of (DEPLOY ALL). We show this for both spouts and bolts.

Lemma 3: p equivalence (bolts)

For each task, $\mathcal{T}\langle p_i, b_o, \_, \_, \_ \rangle$, created for a bolt $b_o \in C$, there exists a worker $\mathcal{W}\langle r, n, \_, \_, \_ \rangle \in W$ after the application of (DEPLOY ALL).

Proof. Lemma 1 shows there is a node, $n = trans_{c \to n}(b_o, G_i) \in N$ for every bolt $b_o \in C$. Here, we show Lemma 3 holds for any arbitrary bolt, $\mathcal{B}_o\langle p, d, \sigma_b \rangle$. We show this by induction over p, the parallelism of b_o.

$p = 1$ The initial configuration of a fwStorm application (C, G) is a set of tasks, T. Let $T_b \subset T$ be the set of tasks for $b_o = \mathcal{B}_o\langle 1, d, \sigma_b \rangle \in C$. $T_b = \{\mathcal{T}\langle 1, b_o, \_, \_, \_ \rangle\}$. The initial configuration for the fwSkitter application $(N, L) = translate((C, G))$ is $(\varnothing, \varnothing)$. From this configuration, only (DEPLOY ALL) can be applied. To apply this rule, (DEPLOY) must be applied for $n = trans_{c \to n}(b_o, G_i)$. This will lead to the reduction of the spawn expression. The reduction of spawn will create the following set of workers to be added to W: $\{w = \mathcal{W}\langle r, n, \_, \_, \_ \rangle\}$. Since n is unique, w

is unique. Thus, there is a worker in W for every task in T_b. Lemma 3 holds when $p = 1$.

$p = k + 1$ Assume Lemma 3 holds when $p = k$. When $p = k + 1$, T_b will have $k + 1$ tasks for b_o: $T_b = \bigcup_{i=1}^{k+1} \{\mathcal{T}\langle i, b_o, \_, \_, \_\rangle\}$. Applying (DEPLOY) will lead to the reduction of $\overline{\texttt{spawn}(\_, \_)}^{\,k+1}$. The induction hypothesis states that a worker is created for every task in $\bigcup_{i=1}^{k} \{\mathcal{T}\langle i, b_o, \_, \_, \varnothing\rangle\}$. Said otherwise: $\overline{\texttt{spawn}(\_, \_)}^{\,k}$ leads to the creation of W_b, a set of k workers. The remaining expression, $\texttt{spawn}(\_, \_)$, creates a worker: $w = \mathcal{W}\langle r, n, \_, \_, \_\rangle$. (SPAWN) specifies that each worker created by `spawn` has a unique reference, r, making it unique. Thus, $W_b \cup \{\mathcal{W}\langle r, n, \_, \_, \_\rangle\}$ creates a set of $k + 1$ workers, proving the inductive case.

Lemma 4: p equivalence (spouts)

For each task, $\mathcal{T}\langle p_i, s_p, \_, \_, \_\rangle$, created for a spout $s_p \in C$, there exists a worker $\mathcal{W}\langle r, n, \_, \_, \_\rangle \in W$ after the application of (DEPLOY ALL).

Proof. Analogous to the proof of Lemma 3. For spouts where $p = 1$, (DEPLOY ALL) reduces `spawn`, creating worker set: $\{\mathcal{W}\langle r, trans_{c \to n}(s_p, G_i), \_, \_, \_\rangle\}$. When $p = k + 1$, $\overline{\texttt{send}(\texttt{spawn}(\_, \_), \_)}^{\,k}$ creates a set W_s of k workers. The remaining expression creates a worker, $w = \mathcal{W}\langle r, n, \_, \_, \_\rangle$. As before, r is unique, which means $W_s \cup \{w\}$ is a set of $k + 1$ workers, proving the inductive case.

We can now show that each task created for (C, G) has a corresponding worker.

Lemma 5: p equivalence

For each task, $t \in T$, there exists a worker, $w \in W$, after the application of (DEPLOY ALL).

Proof. Lemma 1 shows there is a node $n \in N$ for each $c \in C$. Lemmas 3 and 4 show there is a unique worker for each task created for any $c \in C$. Since each worker stores its node, n, which is unique, there is a corresponding worker, $w \in W$ for every task $t \in T$.

5.2.3 Behavioural Equivalence of Corresponding Workers

Now that we have shown that there is a corresponding worker for every task in a fwStorm application, we show that each individual worker behaves the same as its

corresponding task. As before, we provide a separate proof for spouts and bolts.

In the proof for bolts, we assume that the workers and tasks which process data records for the bolt receive the same inputs. In the final proof of Storm-completeness (shown further below), we show that any execution sequence wherein a task receives a particular set of inputs must have a matching execution sequence wherein its corresponding worker receives the same set of inputs.

Lemma 6: task equivalence (spouts)

After the deployment of (N, L), each application of (SPOUT) for a task $t = \mathcal{T}\langle \_, s_p, d_s, \_, \overline{q_s}\rangle \in T$ has a corresponding application of (PROCESS) for a worker $w = \mathcal{W}\langle r, n, e, (h_s, \_), \overline{q}\rangle \in W$, where $n = trans_{c\to n}(s_p, \_)$. When t sends value d_r to its successors through the use of $send$, w will emit $(d_r, \_)$ through the application of (EMIT). After each application, $d_s = h_s$.

Proof. By induction over the number of times (SPOUT) is applied on t, which we call x.

$x = 1$ The initial state of a task t for a spout $s_p = \mathcal{S}_p\langle p, d_0, \sigma_s\rangle$ is defined as $\mathcal{T}\langle \_, s_p, d_0, \_, \varnothing\rangle$. Lemma 1 shows us there is a corresponding node, n, for s_p, created by $trans_{c\to n}$. The strategy of n has `deploy` expression $\overline{\texttt{send}(\texttt{spawn}((d_0, \_), e), \texttt{null})}^{p}$, which leads to the creation of worker $w = \mathcal{W}\langle r, n, e, (d_0, \_), [\texttt{null}]\rangle$ when (DEPLOY ALL) is applied.

(SPOUT) can be used on t to generate a new data element, d_r, which is sent through the use of $send$. After applying this rule, the new state of t is d'_s, where $d'_s, d_r = \sigma_s(d_0)$.

The message queue of w is [`null`] as $\texttt{send}(\texttt{spawn}((d_0, \_), e), \texttt{null})$ is reduced during the application of (DEPLOY ALL). Therefore, (PROCESS) can be applied on w; this rule will reduce the `process` expression of w:

$$\begin{array}{l}\texttt{let } (x_s, x_\Gamma) = \texttt{state in}\\ \quad \texttt{let } x'_s, x_r = \texttt{call}(next, x_s, \varnothing) \texttt{ in}\\ \quad\quad \texttt{let } x'_\Gamma, x_{\bar{i}} = emit_\Gamma(x_\Gamma, x_r) \texttt{ in}\\ \quad\quad\quad \texttt{send}(\texttt{self}, \texttt{null}); \texttt{emit}((x_r, x_{\bar{i}})); (x'_s, x'_\Gamma)\end{array}$$

Here, `state` gets substituted for $(d_0, \_)$, which means $x_s = d_0$. The callback of $next$ is $d_s, \varnothing \mapsto \sigma_s(d_s)$, therefore $x'_s, x_r = \sigma_s(d_0) = d'_s, d_r$. $x_r = d_r$ is emitted through the use of `emit` alongside $x_{\bar{i}}$, which will lead to the application of (EMIT). $x'_s = d'_s$ is returned alongside x'_Γ to be stored as the state of w, proving the base case of Lemma 6.

$x = k + 1$ The induction hypothesis states that, after k applications of (SPOUT), task $t = \mathcal{T}\langle \_, s_p, d_s, \_, \varnothing \rangle$ has a corresponding worker, w, with matching state: $w = \mathcal{W}\langle r, n, e, (d_s, \_), \overline{q} \rangle$. As before, (SPOUT) can be applied to obtain a new state for t, d'_s, and to emit a data element, d_r, through the use of *send*, where $d'_s, d_r = \sigma_s(d_s)$. The induction hypothesis states that both (PROCESS) and (SPOUT) are applied k times. The former reduces expression `send(self, null)`, which means that, after the k-th application of (PROCESS), `null` will be present in the message queue of w: $\overline{q} = [\texttt{null}]$. Since there is a message present in the message queue of w, (PROCESS) can be used to emit a data element and update the state of w. As before, x'_s will be part of the new state of w, while x_r will be emitted alongside $x_{\bar{i}}$. Since $x'_s, x_r = \sigma_s(d_s) = d'_s, d_r$, Lemma 6 holds for the inductive case.

Lemma 7: task equivalence (bolts)

After the deployment of (N, L), each reduction of (BOLT) for a task $t = \mathcal{T}\langle \_, b_o, d_s, \_, \overline{q_s} \rangle \in T$ has a corresponding reduction of (PROCESS) for a worker $w = \mathcal{W}\langle r, trans_{c \to n}(b_o, \_), e, (h_s, \_), \overline{q} \rangle \in W$ if t and w receive the same messages (i.e. when $\overline{q} = \overline{q_s}$). When t sends value d_r to its successors through the use of *send*, w will emit $(d_r, \_)$ through the application of (EMIT). After each application, $d_s = h_s$.

Proof. Analogous to the proof of Lemma 6, by induction over $|\overline{q}|$, the number of messages t receives.

$|\overline{q}| = 1$ The initial state of a task, t, for a bolt, b_o, is $\mathcal{T}\langle \_, b_o, d_s, \_, \varnothing \rangle$. The state of the corresponding worker, w, is $\mathcal{W}\langle \_, n, e, (d_s, \_), \varnothing \rangle$ after the application of (DEPLOY ALL). (BOLT) can be applied when a single message, m is added to the queue of t. This will apply *send* to emit d_r and update the state of t to d'_s, where $d'_s, d_r = \sigma_b(d_s, m)$. When w receives the same message, (PROCESS) can be used to reduce its `process` expression:

$$
\begin{aligned}
&\texttt{let } (x_s, x_\Gamma) = \texttt{state in} \\
&\quad \texttt{let } x'_s, x_r = \texttt{call}(react, x_s, \texttt{msg}) \texttt{ in} \\
&\quad\quad \texttt{let } x'_\Gamma, x_{\bar{i}} = emit_\Gamma(x_\Gamma, x_r) \texttt{ in} \\
&\quad\quad\quad \texttt{emit}((x_r, x_{\bar{i}})); (x'_s, x'_\Gamma)
\end{aligned}
$$

`state` will be substituted for $(d_s, \_)$, `msg` for m. The results of the call to *react* will be bound to x_s and x_r; x'_s will become part of the new state of the worker, while x_r will be emitted alongside $x_{\bar{i}}$ through the application (EMIT). The *react* callback of n is $d_s, d_a \mapsto \sigma_b(d_s, d_a)$. Therefore, $x'_s, x_r = \sigma_b(d_s, m) = d'_s, d_r$, proving the base case.

$|\overline{q}| = k+1$ According to the induction hypothesis, $h_s = d_s$ after processing k messages. When w and t receive a message, m, (SPOUT) specifies t will send d_r using $send$ while its new state becomes d'_s, where $d'_s, d_r = \sigma_b(d_s, m)$. (PROCESS) will reduce the `process` expression of w, which will change the state of w to $(x'_s, \_)$ and emit x_r alongside $x_{\bar{i}}$. As in the base case, $x'_s, x_r = \sigma_b(d_s, m) = d'_s, d_r$, proving the inductive case.

5.2.4 Communication Equivalence

Now that we have shown that each worker behaves the same as its corresponding task at an individual level, we turn to show that the communication in $translate((C, G))$ matches that of (C, G). We first show that the grouping state stored inside the workers of $translate((C, G))$ matches the grouping state stored inside fwStorm's tasks, and that this state remains the same when the tasks and workers emit identical data records. After, we show that data records sent to components are also sent to their corresponding nodes, and that these records are then internally distributed in the same way, i.e. that a task for a component can only receive a data record if its corresponding worker receives one as well.

Lemma 8: $init_\Gamma(c)$ equivalence

$\forall c \in C : trans_{c\to\Gamma}(c, G_i) = init_\Gamma(c)$

Proof. $addport$ adds a port, i to every grouping $g \in G$. Thus, there is a single (g, i) pair $\in G_i$ for every $g \in G$. $init_\Gamma(c)$ creates a (g, d) pair for each grouping $g \in G$ that has c as its origin. $trans_{c\to\Gamma}$ creates a similar pair for each $(g, i) \in G_i$ where g has c as its origin. Since there is a single (g, i) pair $\in G_i$ for every $g \in G$, $trans_{c\to\Gamma}(c, G_i) = init_\Gamma(c)$.

Lemma 9: Γ equivalence

Let $w = \mathcal{W}\langle \_, \_, \_, (\_, h_\Gamma), \_\rangle \in W$ be the corresponding worker of $t = \mathcal{T}\langle \_, \_, \_, \Gamma, \_\rangle \in T$. After each application of (PROCESS), h_Γ matches Γ of t, after its corresponding application of (SPOUT) or (BOLT). If t is a task for a bolt, this is only true if w and t receive the same messages.

Proof. Analogous to Lemmas 6 and 7, by induction over the number of applications of (SPOUT) or (BOLT) on t and (PROCESS) on w, which we call x.

$x = 1$ Lemmas 6 and 7 show us that any time $send$ is applied for t, there is a corresponding application of (EMIT) for w, which produces the same data record, d. When a task produces a data record, $send_g$ will apply σ_g, the grouping function of each grouping $g \in \{g \mid (g, \_) \in \Gamma\}$, and store the

result in Γ. Similarly, the `process` hook of w will reduce $emit_\Gamma$, which calls σ_g for each grouping $g \in \{g \mid (g, \_) \in h_\Gamma\}$, and store the result in h_Γ. Lemma 8 shows that $h_\Gamma = \Gamma$, which means σ_g will be called for the same set of groupings. Thus, it suffices to show that the call of the grouping function: $\sigma_g(d_s, d, p)$ is identical for w and t for each of these groupings:

- Lemma 8 shows that $h_\Gamma = \Gamma$, which means d_s is identical for every grouping.
- Lemmas 6 and 7 show both w and t produce the same data records, which means d is identical.
- $emit_\Gamma$ and $send_g$ both use the parallelism of the downstream bolt, p, in their application of σ_g.

$x = k + 1$ As in the base case, $send_g$ and $emit_\Gamma$ will both call σ_g for the same set of groupings with the same data record and the same parallelism values. The induction hypothesis states that $\Gamma = h_\Gamma$ after (PROCESS) has been applied k times. Thus, the state passed to each call of σ_g will be identical between w and t. Since all arguments are identical, the same updated state will be returned by σ_g and stored inside both Γ and h_Γ, proving the induction hypothesis.

Lemma 10: $send_g$ equivalence

Each call to auxiliary function $send_g$ for grouping $\mathcal{G}\langle \_, b_o, \_, \_\rangle$ has a corresponding application of (DELIVER) for n, the corresponding node of bolt.

Proof. The proof of Lemma 9 shows us that any time $send$ is called for task $t = \mathcal{T}\langle \_, c, \_, \Gamma, \_\rangle$, there is an equivalent application of (EMIT) for worker w. $send$ will use $send_g$ to send a data record to every grouping $g \in \{g \mid (g, \_) \in \Gamma\}$, i.e. to every grouping where c is the upstream component. (EMIT) will lead to the application of (DELIVER) for each link where n is the upstream node. Lemma 2 shows that each grouping $\mathcal{G}\langle c_f, c_t, \_, \_\rangle \in G$ has a corresponding link $\mathcal{L}\langle trans_{c\to n}(c_f, \_), trans_{c\to n}(c_t, \_), \_\rangle \in L$, which means any grouping where c is the upstream component has a corresponding grouping where n is the upstream node, and thus that any call to $send_g$ has a matching application of (DELIVER).

Lemma 11: communication equivalence

Let T_r be the set of tasks that receive message m, sent by task t through the use of auxiliary function $send$. Each task, $t_r \in T_r$, has a corresponding worker, $w_r \in W$ which will receive m when w, the corresponding worker of t, sends m through the application of (EMIT).

Proof. Lemma 10 shows that any application of $send_g$ has a matching application of (DELIVER). Thus, if downstream component c receives a data record, its corresponding node, n, will receive the same data record. Here, we show that if any task of c receives a data record, its corresponding worker for n will receive the same data record.

(DELIVER) will reduce the following expression:

$$\texttt{let } (x_d, x_{\bar{i}}) = \texttt{data in } \overline{\texttt{send}}\Big(\texttt{deployment}[x_{\bar{i}}\left(trans_{b_o \to \bar{g}}(b_o, G_i)[\texttt{port}]\right)], x_d\Big)$$

Here, `data` is substituted for the emitted value, $(x_r, x_{\bar{i}})$. x_r is equal to the value produced by the upstream node, which is the same value given to *send* by the upstream component, according to Lemmas 6 and 7. This value is sent to each worker whose reference is stored in `deployment` at an index contained in $x_{\bar{i}}\left(trans_{b_o \to \bar{g}}(b_o, G_i)[\texttt{port}]\right)$.

Let g be the grouping which sent data to c. $send_g$ uses the grouping function of g, σ_g, to produce a set of indices, $\overline{p_i}$, which determine which tasks of c will receive data. Lemma 9 shows us that $send_g$ and $emit_\Gamma$ will call σ_g with the same arguments each time, thus showing $x_{\bar{i}}(g) = \overline{p_i}$. We now show $trans_{b_o \to \bar{g}}(b_o, G_i)[\texttt{port}] = g$.

$trans_{b_o \to \bar{g}}$ creates a vector of groupings where each grouping is stored at its index in G_i, the set of indexed groupings created by $addport$. This same set of index groupings is used by $trans_{g \to l}$ to translate groupings to links. Since `port` is substituted for the link's port, $trans_{b_o \to \bar{g}}(b_o, G_i)[\texttt{port}] = g$. This means the appropriate vector of indices is retrieved from $x_{\bar{i}}$, which, in turn, means that when any task for c receives a data record, the corresponding worker for n will receive the same data record.

5.2.5 Storm-completeness

We can now show that $translate((C, G))$ accurately simulates the behaviour of (C, G). We do this by proving that any possible execution sequence for (C, G) has exactly one corresponding execution sequence in $translate((C, G))$ and vice versa.

Theorem: fwSkitter is Storm-complete

Let Θ be the set of all possible executions of fwStorm application (C, G), and Σ be the set of all possible configurations for $(N, L) = translate((C, G))$ after the reduction of (DEPLOY ALL). Any execution $\overline{T} \in \Theta$ has exactly one equivalent execution, $\overline{S} \in \Sigma$, and any $\overline{S} \in \Sigma$ has exactly one equivalent execution, $\overline{T} \in \Theta$.

Proof. By induction over the length of the execution sequence: $|\overline{T}|$.

$|\overline{T}| = 1$ The initial configuration of (C, G) is a set of tasks, T, each with an empty message queue. Lemma 5 shows that there is a corresponding worker for every task, $t \in T$ after the reduction of (DEPLOY ALL).

From this initial configuration, (SPOUT) can be applied on an arbitrary task for a spout to produce a data record. According to Lemma 6, any reduction of this rule has a corresponding reduction of (PROCESS), which selects the corresponding worker of a task to produce the same data record. Lemma 11 shows that the produced data record will be sent to the corresponding worker of each task.

After $translate((C, G))$ has been deployed, only (PROCESS) can be applied to those workers which have a message in their queue. This is only the case for workers translated from spouts; thus, any rule that can be applied here has a corresponding reduction of (SPOUT). Any possible execution for (C, G) thus has exactly one corresponding execution in $translate((C, G))$ proving the base case.

$|\overline{T}| = k + 1$ According to the induction hypothesis, any possible execution sequence of length k in fwStorm has exactly one corresponding execution sequence of length k in fwSkitter and vice versa. In order for the induction hypothesis to hold, there must be a worker, $w \in W_k$ for each task, $t \in T_k$ in the k-th configuration of any execution sequence. As in the base case, any application of (SPOUT) on any of the tasks in T_k has exactly one matching application of (PROCESS) and vice versa.

(BOLT) can be applied on any task $t \in T_k$ which represents a bolt and has a message in its queue to process a single message. According to Lemma 11 and the induction hypothesis, the corresponding worker of t, $w \in W_k$, must also have a message in its queue, which means (PROCESS) can be applied on this worker. For the same reason, any worker with a message in its queue will have a corresponding task with the same message in its queue, enabling the application of (BOLT). Moreover, Lemma 7 states that the application of these rules will lead to the same result in w and t.

Thus, any application of (BOLT) or (SPOUT) has exactly one matching application of (PROCESS), which means there is exactly one execution sequence of length $k+1$ for (C, G) for each execution sequence of that length in $translate$ and vice versa, proving the inductive case.

Section 5.1 defines $translate$, which can translate any fwStorm application into fwSkitter. We have shown that this translation behaves in the exact same way as the original fwStorm application, thus showing that fwSkitter is Storm-complete.

5.3 Translating featherweightSkitter to featherweightStorm

In this chapter, we have shown that any fwStorm application can be expressed in fwSkitter, showing that fwSkitter is at least as expressive as fwStorm. However, we did not discuss if every fwSkitter application can be expressed in fwSkitter, i.e. if fwSkitter is more expressive than fwStorm. Here, we briefly discuss how fwSkitter applications may be translated to fwStorm, and if this is possible for every possible fwSkitter application.

A direct translation from fwSkitter to fwStorm is not straightforward, as a single node in an fwSkitter application may spawn several different workers which reduce different process hooks and communicate with other workers created by the same strategy in an arbitrary fashion. This is not possible in fwStorm, where tasks can only execute the bolt or spout function of their components. Thus, any translation between fwSkitter and fwStorm would require changing the structure of the original fwSkitter application.

If we do not worry about preserving the structure of the original application, but only care about producing a fwStorm application which produces the same execution sequences as the original fwSkitter application, we could still not express every fwSkitter application in fwStorm. This is the case because fwSkitter boasts features which cannot be expressed in fwStorm. For instance, the spawn expression of fwSkitter enables workers to be created after the deployment of an application. This is not possible in fwStorm, which does not allow the amount of tasks created for an application to be changed after the start of the application. Thus, certain fwSkitter applications cannot be expressed in fwStorm if every worker created for the original fwSkitter application must have a matching task in the translated fwStorm application.

It may be possible to express any possible fwSkitter application in fwStorm, if we consider translations wherein a single fwStorm task simulates several fwSkitter workers. However, we do not consider these translations in this discussion, as the main goal of both fwStorm and fwSkitter is to capture how DSPAs are mapped onto tasks and workers and the communication patterns between them.

5.4 Conclusion

One of the requirements we set out for our programming model in Chapter 1 was that it should be *expressive* enough to support the implementation of distribution strategies from the state of the art. The majority of these strategies are implemented in Apache Storm, thanks to its flexibility and low-level nature. In this chapter, we showed that fwSkitter, the formal instantiation of Skitter, is Storm-

complete, and thus, that it is at least as expressive as (featherweight) Storm. This means that any distribution strategy expressed in (featherweight) Storm can be expressed in (featherweight) Skitter.

We have shown that fwSkitter is Storm-complete by defining a translation from fwStorm to fwSkitter, and by proving that this translation precisely and correctly simulates the behaviour of the original fwStorm application.

6. Skitter.ex: a Practical DSPF with Modular Distribution Strategies

After having presented featherweightSkitter, which distils formal essence of Skitter, in Chapter 4, we now turn towards the design of a practical DSPF for the expression of modular, pluggable distribution strategies. To this end, we present "Skitter.ex": a practical instantiation of Skitter built as a domain-specific language (DSL) on top of Elixir.[1] Through Skitter.ex, we show how the concepts introduced by Skitter can be used to write real DSPAs with support for modular, pluggable distribution strategies. Skitter.ex is open-source, extensively documented,[2] and available online.[3]

Our discussion of Skitter.ex consists of four parts:

- Section 6.1 describes the core abstractions introduced by Skitter (*workflows*, *operations*, and *distribution strategies*) and their embodiment in Skitter.ex. These abstractions correspond to the abstractions introduced by fwSkitter in Chapter 4.
- Section 6.2 describes the features Skitter.ex builds on top of these core abstractions to make it a practical tool for the expression of DSPAs.
- Section 6.3 discusses the implementation of Skitter.ex as a DSL with an accompanying runtime system in Elixir.
- Section 6.4 discusses how Skitter.ex can be extended to be resilient to failures of one or multiple cluster nodes.

Appendix B contains a brief primer on Elixir, and an overview of the functions from its standard library we use, for readers unfamiliar with the language.

[1] https://elixir-lang.org/ (visited on 11/03/2026)
[2] https://hexdocs.pm/skitter/0.8.0 (visited on 14/05/2026)
[3] https://github.com/mathsaey/skitter (visited on 17/03/2026)

```
workflow do
  node(ClicksSource, as: clicks)
  clicks.out ~> join.right
  clicks.out ~> rate.clicks

  node(SalesSource)
  ~> node(Join, with: BicliqueContRand, as: join)
  ~> node(Rate, as: rate)
  ~> node(Publish)
end
```

Listing 6.1: Implementation of Adalyze in Skitter.ex. The corresponding DAG is shown in Figure 6.1. Constructs of Skitter.ex are marked using bold.

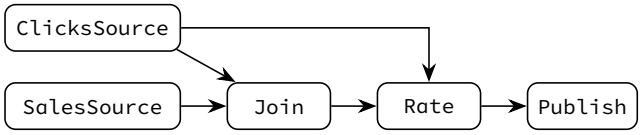

Figure 6.1: DAG of the Skitter.ex implementation of Adalyze shown in Listing 6.1.

6.1 Three Core Abstractions

In Skitter.ex, DSPAs are implemented using three main abstractions: workflows, operations, and distribution strategies. An operation embodies the data processing logic of an application, while strategies determine how an operation is distributed over a cluster. Workflows combine operations with strategies and wire these into a DAG to create a stream processing application. We discuss each of these abstractions in the following sections.

6.1.1 Workflows

A Skitter.ex application is defined as a workflow: a DAG consisting of several *nodes*. Each node represents a data processing step which embeds an operation (i.e. the data processing logic of the step) and its distribution strategy (i.e. the distribution logic of the data processing step). Nodes are connected through *links*, which wire the inputs and outputs of the nodes to each other. In this way, Skitter.ex enables stream processing applications to be built by combining several operations into a DAG, each paired with its own distribution strategy.

The main novelty of Skitter is its ability to pair operations with strategies. This is enabled by the **`with:`** operator, which can be used inside a **`node`** definition to specify which strategy is used to distribute the operation of that node. In many cases, developers may not want to tweak the strategy of an operation. Therefore, operations may specify a default strategy, which is used when **`with:`** is omitted.

example

Listing 6.1 implements Adalyze, our running example introduced in Chapter 1, as a Skitter.ex workflow. This workflow is a textual representation of the application DAG shown in Figure 6.1. The various sources and operations which implement the data processing steps of the application are specified using the `node` operator, while data dependencies between the nodes are expressed through the use of the link operator, `~>`. When needed, nodes can be named using `as:`, as shown on lines 2, 7, and 8. A node can override the default distribution strategy of its operation through the use of `with:`, shown on line 7. The link operator can be used to link a specific output of a node to the input of another node, as on lines 3 and 4, or it can be used to "chain" nodes, as shown on lines 7 to 9; when this is done, the first output of a node is linked to the first input of the next node in the chain.

6.1.2 Operations

A Skitter.ex operation is a data processing step which can be embedded in a workflow. At run-time, an operation is always distributed by a distribution strategy. Operation definitions consist of meta-information, which is used to embed the operation inside a workflow, and of various *callbacks*, which implement the data processing logic provided by the operation. The meta-information specifies the inputs and outputs accepted and returned by the operation (called *ports*), as well as the strategy to use when none is provided in the workflow (using `with:`).

example

The `Rate` operation calculates the conversion rate of incoming ads. It does this by responding to click and sale events. Listing 6.2 shows the `Rate` operation used by the workflow in Listing 6.1. The first line contains the meta-information of the operation which specifies that it accepts two inputs streams: `clicks` and `sales`, and that it publishes a single output stream, `rate`. It also specifies that `KeyedState` is used to distribute this operation if no other strategy is provided.

Required Callbacks

In Section 4.1.2, we discuss how the distribution strategy of an operation may require it to implement specific pieces of data processing logic based on its properties. For this reason, the strategy of an operation determines which callbacks the operation needs to implement. In the most simple case, a strategy only requires an operation to implement a single callback which specifies how it responds to incoming data records (typically named `react`). However, the strategy may require several strategy-specific callbacks to be present, as exemplified next.

example

The `Rate` operation is inherently stateful, as it needs to store two counters for each ad: one which counts the number of times the ad has been clicked, and another counting the number of clicks on the ad which resulted in a sale. The

default strategy of `Rate`, `KeyedState`, partitions this state based on a key (the id associated with the ad), and needs to ensure each incoming data record is processed by the worker which maintains the state associated with this key, as discussed in Section 2.2.1.

example

To keep `KeyedState` agnostic to the key extraction logic, which is operation-specific, it requires any operation paired with it to implement a `key` callback, which is responsible for extracting the key of an incoming data record. This callback is defined on lines 4 to 6: it obtains the key of an incoming data record by extracting its `ad_id`. Besides this, `KeyedState` also requires the implementation of a `react` callback, which is called to process an incoming data record together with the state associated with the key of that data record, as we discuss later.

Operations are only compatible with those distribution strategies which call (a subset of) the callbacks the operation implements. This introduces a level of coupling between strategy and operation. This is unavoidable, as distribution strategies typically place certain expectations on the operations they distribute, as shown in Section 2.2. Therefore, Skitter relies on strategy developers to design an appropriate interface for operations which can be used by several strategies. Typically, strategies which distribute the same type of operations (e.g. strategies which distribute joins, or those which distribute stateful operations) impose the same required callbacks on their operations, enabling them to be changed in a plug & play fashion (as we show in Chapter 7). More specialised strategies may require additional callbacks to be implemented (e.g. the PKG strategy discussed in Section 2.2.1 would require a callback which merges the states of split keys). These additional callbacks are ignored by strategies which do not require them.

Callback Definition

Managing the state of an operation is the responsibility of a distribution strategy, as it determines where its computations may be executed, and thus which state these computations can access. However, an operation often needs to access and update its state. Similarly, an operation should have the ability to publish data records to its successors, but a strategy should also be able to control how this happens (e.g. to execute a computation multiple times in parallel, as in Dynamo [Tran et al., 2017; Tran et al., 2018], discussed in Section 2.2.1). To reconcile these concerns, we conceive callbacks as functions which accept a state along with their regular arguments and which return a potentially updated state, emitted data and a return value.

In other words, a callback is a function: $state, args \mapsto state', retval, emit$.

In Skitter.ex, callbacks are conceived as functions which can implicitly access their

```
1  defoperation Rate, in: [sales, clicks], out: rate, strategy: KeyedState do
2    initial_state {0, 0}
3
4    defcb key(data) do
5      data.ad_id
6    end
7
8    defcb react(data) do
9      {clicks, sales} = state()
10     {new_clicks, new_sales} =
11       case port_of(data) do
12         :sales -> {clicks, sales + 1}
13         :clicks -> {clicks + 1, sales}
14       end
15     state <~ {new_clicks, new_sales}
16     {data.ad_id, new_sales / new_clicks} ~> rate
17   end
18 end
```

Listing 6.2: Definition of the `Rate` operation.

state through the use of **`state`**, update their state through the use of **`<~`** and emit values through the use of **`~>`** (emitting a single value) and **`~>>`** (emitting every value in a collection). When a strategy calls a callback, it provides it with this state and receives the (potentially updated) state along with any published and returned values on callback completion.

example

The `Rate` operation maintains two counters for each ad: one counting the number of times the ad has been clicked, and another counting the number of times clicks on the ad resulted in a sale. These are stored as a tuple containing two counters, each starting at 0, as defined on line 2 of Listing 6.2.

When `Rate` receives a click or sale event, its strategy is responsible for ensuring the `react` callback is called with the appropriate state. `react` will then increment the appropriate counter (determined by whether a sale or click event occurred), calculate the current conversion rate, and publish this information to downstream operations.

`react` is shown on lines 8 to 17. This callback retrieves the current counters by using the **`state`** operator (line 9); afterwards it updates either the `clicks` or `sales` counter based on the port on which the data record was received (lines 10 to 14). The name of this port is obtained using **`port_of`**, which we discuss in Section 6.2.3. The updated counters are then stored as the new state to be returned to the strategy through the use of **`<~`**. Afterwards, the new conversion rate is calculated and emitted as a tuple along with `ad_id` using **`~>`**. The callback DSL ensures the emitted values, updated state and return value are all returned to the strategy, which we discuss next.

6.1.3 Distribution Strategies

We introduce distribution strategies as a means for expert meta-programmers to specify how an operation is distributed over a cluster. A Skitter.ex distribution strategy is a meta-program which determines how and where the various computations for an operation are executed. A strategy is defined as a set of *hooks*, which are called by the Skitter.ex runtime system in response to predefined events. These hooks deal with *workers*, which are computational entities based on the actor model [Agha, 1985; De Koster et al., 2016]. Each worker maintains a state and can perform computations for the operation and its strategy.

The various hooks are responsible for creating these workers, determining which computations they execute and communicating between them. Strategies are created by implementing the `deploy`, `deliver`, and `process` hooks:

`deploy` creates an initial deployment of the operation over the cluster, typically by spawning several workers.

`deliver` sends data records emitted by an upstream operation to a worker to be processed.

`process` processes a single data record received by a worker of the operation.

example

Listing 6.3 shows the definition of the `KeyedState` strategy, used to distribute the `Rate` operation. The workings of this strategy are visualised in Figure 6.2.

`KeyedState` partitions the state of an operation based on a set of keys: inside the `deploy` hook (lines 2 to 6), a worker is created for each cluster node; each worker maintains the state of several keys. When an upstream operation (`Join` or `ClicksSource` in the case of Adalyze) emits a value, the `deliver` hook (lines 8 to 14) of `KeyedState` is invoked. Inside this hook, the key of the emitted data record is extracted and hashed. Based on this hash, the data record is sent to the appropriate worker.

When a worker receives a data record, the `process` hook (lines 16 to 22) is invoked. This hook fetches the state associated with the received data record from the state of the worker and calls the `react` callback of the operation with this state. The state and emitted data returned by `react` will be used as the new state for the received key and be sent to downstream operations, respectively.

We discuss each of these hooks in detail in the following sections. Table 6.1 provides a summary of the hooks, the events that trigger them, and their purpose; Table 6.2 shows an overview of the primitives available for use inside strategies.

Table 6.1: Overview of Skitter.ex's hooks.

Hook	Signature	Event	Description
`deploy`	*args*[1] → *dep*[2]	Application start	Distribute operation
`deliver`	*data* → _	Upstream emits *data*	Send *data* to worker
`process`	*msg*, *state*, *role* → *state'*	Worker receives *msg*	Process *msg*, update *state*

[1] Arguments passed to the node in the workflow
[2] Deployment data

Deployment Over a Cluster: `deploy`

The `deploy` hook is responsible for distributing an operation over the cluster nodes. When a Skitter.ex application is started, the runtime system calls the `deploy` hook of the distribution strategy of each operation in the application. The `deploy` hook is responsible for creating the resources required for the operation to respond to incoming data records. These resources include references to workers, which will perform work for the operation, and any constants maintained by the operation. The result of the `deploy` hook is stored by the runtime system, which makes it available as the so-called *deployment data*. This data is automatically copied to every node in the cluster and can be accessed by the `deliver` and `process` hooks. Once deployed, a strategy cannot modify its deployment data. Typically, the `deploy` hook creates a set of workers, after which references to these workers are returned to be stored inside the deployment data.

example

The `deploy` hook of `KeyedState` creates one worker for each cluster node (line 3). This is done by using `Remote.on_all_workers`, which calls a function on every node in the cluster, and `local_worker`, which creates a worker with an initial state (an empty map), and a *role* (which we discuss later). After, the result of `Remote.on_all_workers` is transformed into a tuple of worker references (for faster indexing), to be stored in the deployment data.

Delivering Data to Workers: `deliver`

Skitter.ex's workers perform computations and manage state for a distribution strategy. The `deliver` hook is responsible for selecting which worker a data record is sent to. When an operation's strategy emits data, the `deliver` hook of the strategy of each of its downstream operations is called. This hook must then make sure that the received data record is sent to a worker, where it can

```
1  defstrategy KeyedState do
2    defhook deploy(args) do
3      Remote.on_all_workers(fn -> local_worker(Map.new(), :bucket) end)
4      |> Enum.map(fn {remote, worker} -> worker end)
5      |> List.to_tuple()
6    end
7
8    defhook deliver(data) do
9      key = call(:key, args: [data]).result
10     buckets = deployment()
11     idx = rem(Murmur.hash_x86_32(key), tuple_size(buckets))
12     worker = elem(buckets, idx)
13     send(worker, data)
14   end
15
16   defhook process(data, state_map, :bucket) do
17     key = call(:key, args: [data]).result
18     state = Map.get(state_map, key, initial_state())
19     res = call(:react, state: state, args: [data])
20     emit(res.emit)
21     Map.put(state_map, key, res.state)
22   end
23 end
```

Listing 6.3: Definition of the `KeyedState` distribution strategy. The behaviour of this strategy is shown visually in Figure 6.2.

be processed. Since a node in a Skitter.ex workflow may select any distribution strategy for an operation in their application, the `deliver` hook cannot make any assumption about the current location of the data to be sent. Typically, the `deliver` hook selects a worker from the deployment data to send the data to.

example

When `Join` or `ClicksSource` emits a value, the `deliver` hook of `KeyedState` is called. This hook uses the **`call`** primitive to call the `key` callback of its operation to obtain the key associated with the incoming data record (line 9). Afterwards, it fetches the worker references created by the `deploy` hook through the use of the **`deployment`** primitive (line 10). The key is then hashed and reduced modulo the amount of workers to obtain an index (line 11), which is then used to obtain a reference to a worker (line 12). The **`send`** primitive is then used to the data to be delivered to the selected worker (line 13).

Processing Data: `process`

Skitter.ex's **`send`** primitive can be used to send a message to a worker. Messages sent to a worker are stored in a queue and are processed sequentially in the order of arrival. When a worker is ready to process a message, it calls the `process` hook of the strategy that spawned it with the message, the current state of the

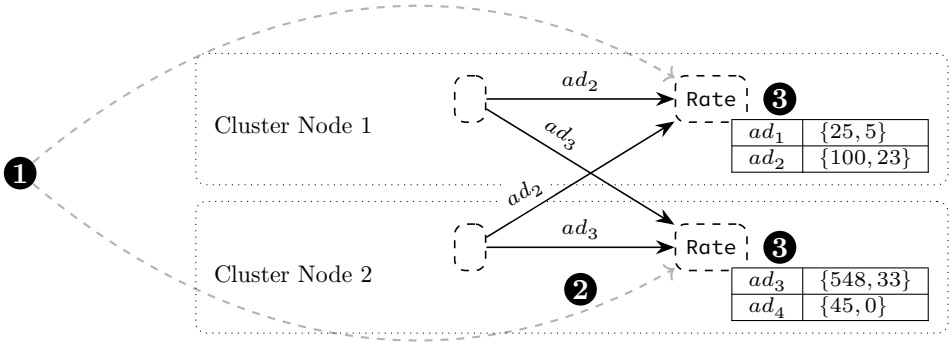

Figure 6.2: Runtime view of `Rate` distributed over two cluster nodes by `KeyedState`. The `Rate` workers are spawned by the `deploy` hook (❶), the ads are sent to a particular worker based on their key by the `deliver` hook (❷). The `process` hook (❸) is invoked when any of these workers receives a data record.

worker, and the worker's role (discussed later). The value returned by `process` is used as the new state of the worker. Typically, this hook invokes a callback of the operation to process the received data with the appropriate state.

example

The `process` hook of `KeyedState` calls the `key` callback of its operation to obtain the key associated with the incoming data record (line 17). This key is then used to fetch the state of this particular key from the worker's state, a map. When the worker's state does not contain a value for the key, the initial state of the operation is used instead (line 18). Next, the `react` callback of the operation is called with the received message and the obtained state to process the incoming data record (line 19). The value emitted by the operation is emitted through the use of the **`emit`** primitive (line 20), which causes the Skitter.ex runtime system to call the `deliver` hook of any downstream operations. Finally, the updated state returned from calling `react` is inserted into the state map and returned as the result of the `process` hook, causing it to be stored as the new state of the worker.

Expressions Available Inside Hooks

The `deploy`, `deliver` and `process` hooks discussed above and summarised in Table 6.1 enable developers to hook into Skitter.ex's runtime system to control how operations are distributed over the cluster. Besides these three hooks, Skitter.ex defines several primitives to be used for programming the distribution logic. Table 6.2 lists the most important ones. Most of these primitives were mentioned

Table 6.2: Overview of the most important Skitter.ex primitives available inside hooks. Operation refers to the operation paired with the distribution strategy.

Primitive	Description
`call(cb, state: state, args: args)`	Call an operation callback
`initial_state()`	Fetch initial state of the operation
`deployment()`	Access deployment data[1]
`local_worker(state, role)`	Create worker on local cluster node with initial *state* and *role*
`remote_worker(state, role)`	Create worker on a remote cluster node with initial *state* and *role*
`send(ref, msg)`	Send *msg* to worker with reference *ref*
`emit(data)`	Publish to downstream operations
`self()`	Get reference to current worker
`Remote.on(r, f)`	Execute f on cluster node r
`Remote.on_all_workers(f)`	Execute f on every cluster node[2]
`Remote.self()`	Get name of current cluster node

[1] Not available inside the `deploy` hook
[2] On every worker (i.e. non coordinator) cluster node

in the discussion above; here, we take the time to highlight two non-obvious features which enable the expression of complex distribution strategies.

Worker Roles In Section 2.2.3, we discuss how complex strategies often require the creation of several different worker "types", which perform different computational tasks for an operation. In Skitter.ex, we accomplish this with *roles*: when a worker is created through the use of `local_worker` or `remote_worker`, it is provided with a role, which is a name (i.e. an Elixir atom) that defines the computational task it will perform for the strategy. This role is provided as an argument to the `process` hook by the Skitter.ex runtime system, which enables developers to use pattern matching to write different implementations for the `process` hook depending on the role of the worker. Listing 6.7, discussed later in this chapter, provides an example of a distribution strategy which uses these roles.

Two Communication Levels Skitter.ex distinguishes between communication at the strategy level and communication at the application level, adhering to the principle of *stratification* [Bracha and Ungar, 2004]. The former relates to communication that occurs between workers of the same strategy, while the latter relates to the data dependencies specified in the application's workflow. Skitter.ex handles intra-strategy communication through the use of `send`, which requires a reference to a worker, while application-level communication is handled through the use of `emit`, which causes the runtime system to call the appropriate `deliver` hooks. By separating these two types of communication, Skitter.ex ensures strategies remain agnostic to each other, thus ensuring an appropriate distribution strategy can be selected for each node in the application.

6.2 Practical Skitter.ex

The constructs defined in the previous sections are the core abstractions of Skitter.ex and suffice to write DSPAs which support modular, pluggable distribution strategies. With these core abstractions defined, we now turn our attention to the constructs Skitter.ex introduces which serve to make it a practical tool for the expression of DSPAs. These constructs were introduced to address shortcomings compared to the state of the art we encountered when writing real DSPAs in Skitter.ex. They range from syntactic sugar introduced to make it possible to write DSPAs in the more high-level operator style, to an inheritance system used to share code between related distribution strategies.

6.2.1 Operator-style Workflow Definitions

In Section 2.3, we discuss how modern DSPFs typically offer the "operator" model, wherein DSPAs are expressed by chaining several predefined, generic operations such as `map`, `filter` or `reduce`. While convenient, these operators do not fit well within Skitter.ex's open model. Moreover, this model can be restrictive when an application does not fit into its constraints. An example of this can be found in Listing 2.1, wherein Adalyze's `Rate` operation can only be expressed through the use of 4 separate operators. For this reason, Skitter.ex's workflow language utilizes a style more akin to the "wiring" model, where DSPAs are created by linking generic nodes into an application DAG.

Nevertheless, the operator model can be useful for the definition of the parts of a DSPA that fit within its constraints. For this reason, we introduce three concepts which enable DSPAs to be implemented in the operator style.

```
workflow do
  tcp_source("localhost", 8129, with: CustomStrategy)
  ~> flat_map(&String.split/1)
  ~> keyed_reduce(
      fn word -> word end,
      fn word, count -> {count + 1, {word, count + 1}} end,
      0
    )
  ~> print()
end
```

Listing 6.4: Operator-style definition of a word count application. `tcp_source`, `flat_map`, `keyed_reduce`, and `print` are syntactic sugar introduced by Skitter.ex. Listing 6.5 shows a desugared version of this workflow definition.

```
workflow do
  node(
    Skitter.BIO.TCPSource,
    args: [address: "localhost", port: 8129],
    with: CustomStrategy
  )
  ~> node(Skitter.BIO.FlatMap, args: &String.split/1)
  ~> node(Skitter.BIO.KeyedReduce, args: {
      fn word -> word end,
      fn word, count -> {count + 1, {word, count + 1}} end,
      0
    })
  ~> node(Skitter.BIO.Print)
end
```

Listing 6.5: Desugared version of Listing 6.4. Operation names prefixed with `Skitter.BIO` are operations predefined by Skitter.ex.

- Nodes in a Skitter.ex workflow can accept arguments through the use of `args:`. These arguments are passed to the `deploy` hook of the strategy of the node, which may then pass them to the operation.
- We provide Skitter.ex with a standard library of several operations, such as `Map`, `FlatMap`, `Filter`, and `KeyedReduce`, which behave like these well-known operators. These operators are implemented using the operation DSL discussed in Section 6.1.
- We introduce syntactic sugar for defining nodes which use these operations and enable developers to introduce similar shorthands for the operations in their application.

Listing 6.4 shows an example of an operator-style workflow definition in Skitter.ex. This workflow defines a "WordCount" application, which is the "hello world" of DSPFs; it counts the number of times it has seen each incoming word. Incoming

```
1  defstrategy PKG, extends: Split do
2    defhook forwarder_state(buckets) do
3      List.duplicate(0, tuple_size(buckets)) |> List.to_tuple()
4    end
5
6    defhook forward(data, buckets, key, state) do
7      idx_1 = rem(Murmur.hash_x86_128(key, 0), tuple_size(state))
8      idx_2 = rem(Murmur.hash_x86_128(key, 1), tuple_size(state))
9      idx = if(elem(state, idx_1) < elem(state, idx_2), do: idx_1, else: idx_2)
10
11     put_elem(state, idx, elem(state, idx) + 1)
12     send(elem(buckets, idx), data)
13   end
14 end
```

Listing 6.6: PKG implemented by extending `Split` (shown in Listing 6.7). Figure 6.3 shows a run-time view of the workings of this strategy.

sentences are received on a tcp socket (line 2), after which they are split into individual words (line 3), which are then counted (lines 4 to 8) and printed (line 9). The `keyed_reduce` operator accepts three arguments: the first extracts a key from an incoming data record, the second updates the state associated with a key and determines which value to emit afterwards, the third determines the initial state for a key. Listing 6.5 shows a desugared version of this workflow.

By enabling both styles to be used as needed, we enable developers to write their application in the more declarative, high-level style offered by the operator model, while still offering them the flexibility to define custom operations, distribution strategies, and even operators, as needed.

6.2.2 Sharing Distribution Logic using Object-oriented Abstractions

In Section 2.2.1, we discuss how several distribution strategies share techniques. For instance, the PKG [Nasir et al., 2015], W-Choices, and D-Choices [Nasir et al., 2016] strategies all split the state of a key over several workers, differing only in how they select the workers over which a key is split. Implementations of these strategies often have a high degree of similarity, differing only in a small part of their execution logic (e.g. the worker selection logic in the case of the aforementioned strategies).

In order to enable the reuse of this shared logic, strategies can be extended in an object-oriented fashion: extensions inherit all the hooks the strategy implements and late binding enables the implementation of abstract strategies which implement all or some of the three hooks and delegate the implementation of the unique features of each strategy to their children.

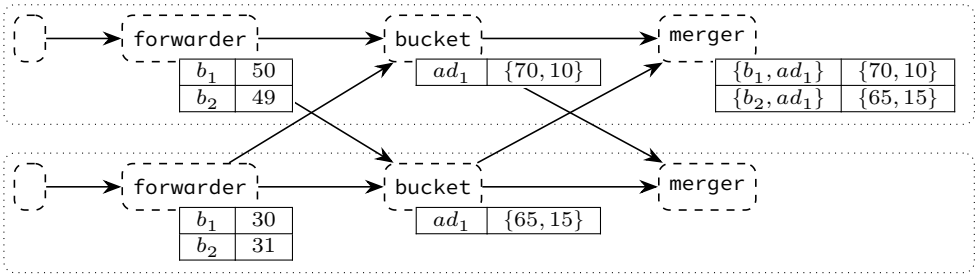

Figure 6.3: Workings of the PKG strategy implemented in Skitter.ex. Data is forwarded to a `forwarder` worker on the local cluster node, which selects a `bucket` to send data to. In the bucket, the computation of the operation is executed on the part of the state stored in this worker. The updated state is sent to the `merger`, which merges the partial states and emits the final result.

example

Listing 6.6 shows the implementation of the `PKG` strategy (discussed in Section 2.2.1) as an extension to an abstract `Split` strategy (Listing 6.7) which can be used to implement strategies which split the state of a key over several workers. The strategy works in four steps, visualised in Figure 6.3.

1. The `deliver` hook sends a received data record to a `forwarder` worker on the current cluster node.
2. The `forwarder` decides on the `bucket` which will process the received data record. In the case of PKG, each key is managed by two `bucket` workers (obtained through hashing); the strategy aims to send the data record to the least loaded of these two workers, which is done by keeping track of the amount of data records each worker has received from this `forwarder` and sending the data to the least loaded one [Nasir et al., 2015].
3. The `bucket` processes the data record using the `react` callback, as before. Afterwards, it sends the state of the key to the `merger`.
4. The `merger` merges all partial states, after which it emits the result of the operation. The partial states are merged through the use of the `merge` callback, which must be defined by the operation. An implementation of `Rate` which includes this callback can be found in Listing C.2.

The second step contains the forwarding logic which is unique to each strategy that extends `Split`. Therefore, `Split` implements all three hooks, and calls the `forward` hook of its children to handle the actual forwarding. It also requires its children to determine the initial state of a `forwarder` worker, which is done through the implementation of the `forwarder_state` hook. These hooks are

```
1  defstrategy Split do
2    defhook deploy(_) do
3      buckets = # Create buckets, store references as tuple
4      mergers = # Create mergers
5
6      forwarder_state = strategy().forwarder_state(context(), buckets)
7      forwarders = # Create forwarders with forwarder_state
8
9      {forwarders, buckets, mergers}
10   end
11
12   defhook deliver(data) do
13     # Send data to forwarder on the current node
14   end
15
16   defhook process(data, state, :forwarder) do
17     {_, buckets, _} = deployment()
18     key = call(:key, args: [data]).result
19     strategy().forward(context(), data, buckets, key, state)
20   end
21
22   defhook process(data, state_map, :bucket) do
23     # Call react with appropriate state, forward state to merger
24   end
25
26   defhook process({key, bucket, state}, state_map, :merger) do
27     # Merge partial states, emit result
28   end
29 end
```

Listing 6.7: Shortened version of the `Split` strategy used as a base to implement PKG in Listing 6.6. The full version of this listing can be found in Listing C.1.

example

called on line 6 and 19 of Listing 6.7; the **`strategy`** primitive is used to obtain the current strategy, after which a hook is called on it. Each hook accepts a **`context`**, which is used by the strategy DSL to implement the primitives defined in Table 6.2. This context needs to be passed explicitly when a hook is called.

6.2.3 Tracking Metadata

DSPFs often track additional information about the data records they process. For instance, Apache Flink tags each data record with a timestamp, which it uses to offer its strong support for reasoning about windows [Akidau et al., 2015], while Apache Storm attaches a unique id to data records to power its tuple acknowledgement mechanic.

While these frameworks can track this additional information internally, doing so is more difficult in Skitter.ex due to its open nature. We therefore introduce a

```
1 defoperation AddProcessTime, in: _, out: _ do
2   defcb react(data) do
3     with_meta(data, process_time: System.monotonic_time(:millisecond)) ~> _
4   end
5 end
```

Listing 6.8: Operation which adds the process time to the outgoing token.

lightweight mechanism to track this type of meta-information in a generic way.

When a data record is emitted by an operation, it is transparently wrapped in a *token*, which contains the data record along with an (initially empty) set of meta-information. Strategies and operations can read the information contained in this token and add data to it as needed. In this way, operations and strategies can attach meta-information such as timestamps to a data record, which can then be used by other operations as needed. An example of this is shown in Listing 6.8, which shows an operation that uses this mechanism to add the current process time (i.e. the system time of the cluster node processing the data record) to any data record it receives.

Skitter.ex's runtime system also uses these tokens to track additional information. For instance, the runtime system tracks which port a data record was received on and stores this information inside the token. This information is then used by the `port_of` primitive shown on line 9 of Listing 6.9.

6.2.4 Managing Complex State in Operations

One of the central design aspects of operations is maintaining state. In the examples discussed thus far, this was rather trivial. However, in reality, the state of an operation may grow to be complex.

Elixir applications typically use *structs* (c.f. Appendix B) to represent compound data types. Structs are key-value collections with a predefined set of keys and default values. We leverage these structs and offer several shorthands for using them as the state of an operation. We discuss these based on their uses in Listing 6.9, which shows an alternative implementation of the `Rate` operation.

- `state_struct` (line 2) defines the initial state of an operation as a struct. In the example, we specify that this struct has fields `:clicks` and `:sales` and that both default to `0`.
- The `~f` *sigil* (c.f. Appendix B) can be used to read the current value of a field of the state struct, as shown on lines 10, 11, and 14.
- `<~` can be used to update a single field of the state struct of an operation,

```
1  defoperation Rate, in: [sales, clicks], out: rate, strategy: KeyedState do
2    state_struct clicks: 0, sales: 0
3
4    defcb key(data) do
5      data.ad_id
6    end
7
8    defcb react(data) do
9      case port_of(data) do
10       :sales -> sales <~ ~f{sales} + 1
11       :clicks -> clicks <~ ~f{clicks} + 1
12     end
13
14     {data.ad_id, ~f{sales} / ~f{clicks}} ~> rate
15   end
16 end
```

Listing 6.9: Definition of the `Rate` operation using syntactic sugar for accessing and updating state fields. `~f{<name>}` is shorthand for reading field `name` of the state, while `<name> <~` is shorthand for writing to that field.

as shown on lines 10 and 11.

Finally, `initial_state` and `state_struct` are syntactic sugar for defining a callback named `initial_state`. When needed, this callback can be created by hand to dynamically determine the initial state of an operation. This is shown at lines 4 to 6 of Listing 6.10, where the initial state of the operation is dynamically determined based on its so-called configuration (discussed next).

6.2.5 Operation Configuration

Operations are often parametrised with immutable information. For instance, the `KeyedReduce` operator used in Listing 6.5 is parametrised with a function to extract keys from data records, a function to update the state for a key, and an initial state for each key. This information is immutable and distinct from the mutable state the operation operates on.

We provide an abstraction which can be used to capture this immutable configuration data, which we call the *configuration*. A callback can be invoked with configuration data, which can then be accessed inside the callback through the use of the `config` primitive. Typically, distribution strategies call a `conf` callback at deployment time, and store its result inside the deployment data. This result is then used as the configuration for future callback invocations.

An example of this can be found in Listing 6.10. This listing shows the definition of a `KeyedReduce` operator, which is parametrised with a key function, a reduce

```
1  defoperation KeyedReduce, in: _, out: _, do
2    defcb conf(args), do: args
3
4    defcb initial_state do
5      {_, _, initial} = config()
6      initial
7    end
8
9    defcb key(val) do
10     {key_fun, _, _} = config()
11     key_fun.(val)
12   end
13
14   defcb react(val) do
15     {_, reduce_fun, _} = config()
16     {new_state, emit} = reduce_fun.(val, state())
17     state <~ new_state
18     emit ~> _
19   end
20 end
```

Listing 6.10: Definition of the `KeyedReduce` operator used in Listing 6.5 as a Skitter.ex operation. The return value of the `conf` callback (a three tuple containing the key and reduce functions as well as an initial state) is stored as the configuration of the operation and used by later callbacks.

function, and an initial state. The strategy passes these parameters to the `conf` callback, which returns them unmodified. The result is then used at lines 5, 10, and 15 to access the initial state of a key, the function to extract a key, or the reduce function, respectively.

6.3 Implementation in Elixir

Skitter.ex is implemented in Elixir. We chose this language due to its strong support for meta-programming through the use of macros and due to its underlying VM known as BEAM, which is uniquely suited to building distributed systems. Skitter.ex leverages Elixir's support for meta-programming to implement all three of its DSLs and leverages the BEAM as the foundation for its runtime system. We discuss the implementation of these DSLs, runtime system and supporting tools in the following sections.

6.3.1 Skitter.ex's Domain-specific Languages

Skitter.ex features DSL constructs for defining workflows, operations, and distribution strategies. Each of these DSLs is defined through a set of Elixir macros.

The Workflow DSL

The workflow DSL transforms workflow definitions (such as those seen in Listings 6.1 and 6.4) to a data structure which represents the application DAG of the DSPA. It mainly does this through two abstractions: `node` and `~>`, which represents nodes and links, respectively. Some syntactic sugar, such as the node "chaining" and the operators, have been added to make workflow definitions more concise and declarative. Finally, workflows can be nested to enable their reuse, in which case the outermost workflow is flattened before it is deployed.

The Operation DSL

Operation definitions are transformed into Elixir modules (c.f. Appendix B). Each of these modules contains several functions which track meta-information (such as the default strategy and the in- and out ports of the operation), as well as one Elixir function for each callback defined by the operation.

A callback is compiled down to an Elixir function, which accepts any arguments accepted by the callback, a state, and a configuration. Any argument accepted by the callback is provided as a token, which is unwrapped by the function.

The callback DSL needs to track two forms of state: the actual state of the callback, as well as any values that the callback emits. This is not trivial, as Elixir is a functional language which does not allow mutating variables. To work around this, the callback DSL uses hidden bindings which track these values and return them along with the return value of the callback's body as the return value of the callback; this essentially implements a state monad through the use of macros. Any use of `<~`, `~>`, or `~>>` corresponds to rebinding one of these hidden bindings. Since Elixir does not support mutation, changes to these bindings inside of control flow constructs are not visible outside of the scope of these constructs. Therefore, any control flow construct that occurs inside the body of a callback is modified to return the current value of the state and emitted values along with the original return value of the branch of the control flow construct that was executed.

example

The desugared version of the `react` callback shown in Listing 6.9 is shown in Listing 6.11. This function accepts a `state` and a `config`; the `data` argument accepted by the callback is deconstructed to separate the argument from its accompanying meta-information, stored in a token.

Variables are used to track the state and emitted values throughout the body of the function. The various `result` variables have been given different names to make their use clear. To ensure updates to the state are visible outside of the *`case`* control flow construct, the body of each of its branches is modified to return both the result of this branch (named `inner_result`) and the current

```
def react(state, config, data_token = %Token{value: data}) do
  emit = []
  body_result = (
    {case_result, state} = case data_token.port do
      :sales ->
        inner_result = (state = %{state | sales: state.sales + 1})
        {inner_result, state}
      :clicks ->
        inner_result = (state = %{state | clicks: state.clicks + 1})
        {inner_result, state}
    end
    case_result
    emit = Keyword.put(emit, :rate, [{data.ad_id, state.sales / state.clicks}])
  )
  %Result{result: body_result, state: state, emit: emit}
end
```

Listing 6.11: Desugared version of the `react` callback shown in Listing 6.9.

– example –

value of the `state` binding.

The result of the original body is captured by the `body_result` variable, which is returned as the callback's result along with the final emitted values and state.

The Strategy DSL

Similar to operation definitions, distribution strategies written in the strategy DSL are transformed into Elixir modules. The main concern of this transformation is to implement Skitter.ex's inheritance mechanism. Each strategy keeps track of every hook it implements and inherits. When a strategy extends another strategy, the child creates a hook for every hook the parent implements that it lacks, which calls the original parent hook. The child then stores that the parent strategy is the original implementer of the hook, so that any of its children (direct or otherwise) can directly call the original parent's hook.

Similar to callbacks, hooks accept their original arguments, along with one additional hidden argument: the so-called `context`. This context stores run-time information, such as the current strategy, operation, and reference to the deployment data. This information is then accessed and used by the various primitives listed in Table 6.2. For instance, the **`call`** primitive uses the `context` to identify the appropriate operation and call its callback.

6.3.2 Skitter.ex's Runtime System

Skitter.ex's runtime system enables the distributed execution of stream processing applications. To do this, it needs to deploy applications over a cluster, call the

appropriate hooks in response to events and provide workers which can be used to execute computations for the application.

Deploying Applications

In order to distribute an application over a cluster, we leverage BEAM's support for distributed execution. A Skitter.ex application can be executed on a cluster by spawning several connected BEAM instances on the cluster nodes, typically one for each node. We provide the required tooling to spawn these instances and connect them with one another. We designate one of the spawned BEAM instances as the master node, which is responsible for distributing the work to be done over the remaining nodes. The remaining nodes, called worker nodes, are responsible for hosting the workers which will perform the actual computations for the DSPA.

In order to facilitate the development of DSPAs, Skitter.ex can also be started in its so-called local mode, wherein DSPAs are executed on a single BEAM instance, which acts as both a worker and master node. Thanks to BEAM's robust support for distributed execution, we can also simulate a full distributed setup with a master node and several workers on a single machine which greatly facilitates debugging Skitter.ex applications.

Executing Applications & Workers

Once started, the Skitter.ex runtime system is responsible for starting the DSPA; this is done by calling the `deploy` hook of every strategy in the application. While deployment is ongoing, we ensure workers do not process incoming messages, buffering them until deployment is completed to avoid race conditions.

We built Skitter.ex's workers on top of Elixir's actors. When such an actor receives a message, it activates the runtime system, which ensures the `process` hook of the appropriate strategy is called with the appropriate arguments. In turn, this hook may call the primitives listed in Table 6.2, which are provided by the runtime system. Of particular interest is the **`emit`** primitive, which will cause the runtime system to invoke the `deliver` hooks of the appropriate downstream strategies.

6.3.3 Supporting Tools

Besides the various DSLs and its runtime system, we also extended Skitter.ex with the various built-in generic operations discussed in Section 6.2.1 and the distribution strategies to support them. Due to Skitter.ex's flexible, open nature, implementing these operations and strategies did not require any changes to the

implementation of Skitter.ex; instead, they are defined using the DSLs discussed in Section 6.1.

Finally, we provide several tools to aid developers, such as a host of functions which transform workflows into DOT[4] graphs, and a code generator which aids developers in setting up a new Skitter.ex application.

6.4 Skitter.ex vs. Partial Failures

While executing DSPAs on a cluster enables them to scale horizontally along with the amount of computational resources at their disposal, it also renders them vulnerable to so-called *partial failures*, wherein a subset of the cluster nodes fail in some way. The chance of such a failure occurring increases as the amount of nodes in the cluster increases [Schroeder and Gibson, 2010]. A key feature of many DSPFs is thus to ensure that the application can remain online and produce correct results in these scenarios.

The design of Skitter.ex currently does not take resilience to partial failures into account. Nevertheless, preliminary work shows that it is possible to apply existing fault tolerance techniques to Skitter.ex without impacting its ability to express custom, pluggable distribution strategies. We describe this work, which was performed in collaboration with Tim Ameryckx in the context of his Master's Thesis and guided by the author of this dissertation [Ameryckx, 2025], in this section.

Several mechanisms exist for dealing with partial failure, such as log-based recovery [Elnozahy et al., 2002], replication [Liu et al., 2017], or the partial recomputation model used by Spark Streaming [Zaharia et al., 2013]. In this work, we use *Asynchronous Barrier Snapshotting* (ABS) [Carbone et al., 2017], a checkpointing mechanism introduced by Apache Flink. This approach was chosen as checkpointing is the most common technique used by DSPFs, while ABS in particular is a fast, reliable, and flexible [van Dongen and Van den Poel, 2021; Vogel et al., 2024] approach. We describe the intuition behind ABS in Section 6.4.1, and its implementation in Skitter.ex in Section 6.4.2.

6.4.1 Asynchronous Barrier Snapshotting

Asynchronous Barrier Snapshotting (ABS) is a technique based on the Chandy-Lamport algorithm [Chandy and Lamport, 1985] to create global checkpoints of a running DSPA without stopping the processing of incoming data records [Carbone et al., 2017]. The idea behind ABS is to inject meta-events called *barriers* in between regular data records and propagate them through the run-time graph

[4] https://www.graphviz.org/doc/info/lang.html (visited on 21/03/2026)

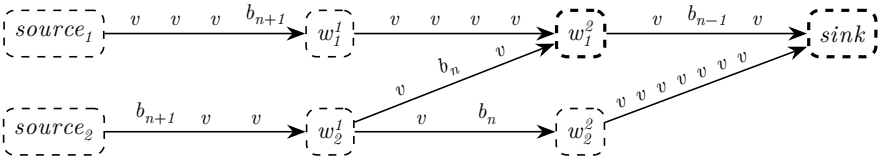

Figure 6.4: ABS barriers propagating through a DSPA. Barriers are denoted as b_x, while values are denoted as v. Several barriers are being processed by the application concurrently. Workers marked with a thick line cannot process all their input streams as they are waiting for barriers to arrive: w_1^2 cannot process inputs from w_1^1 until it has processed barrier b_n coming from w_1^2, $sink$ cannot process values coming from w_2^2 until it has processed b_{n-1}. Once $sink$ has processed b_{n-1}, checkpoint $n-1$ will be complete.

of the DSPA. When a worker receives the barrier, it creates a checkpoint of its internal state after which it propagates the barrier to any downstream workers. A checkpoint is complete once a barrier has travelled through the entire run-time graph of the application.

Special consideration needs to be taken when a barrier passes a worker which has multiple upstream workers. When this is the case, the worker must cease to process any data from upstreams for which it has already received a barrier until it has received the barrier of every upstream worker.[5] Figure 6.4 shows how barriers can propagate through a DSPA to make a complete checkpoint.

When a partial failure occurs, the DSPFs can now roll back to the latest complete checkpoint and restart execution from that point. Doing this correctly places several requirements on the sources, sinks, and other operations in the DSPAs, which we discuss below.

Deterministic Execution Since a failure may lead to arbitrary computations being re-executed, computations should be deterministic to ensure the behaviour of the DSPAs remains unchanged before and after the failure.

Replayable Sources Sources need to have the ability to replay the data records they emitted into the remainder of the application after a barrier was injected into the application. In practice, DSPAs often source their data from some external system (such as Apache Kafka [Kreps et al., 2011]) which enables replaying data records from a given offset.

External Consistency Sinks in DSPAs are typically responsible for writing the resulting data records to some external system. When the DSPAs process

[5] Flink allows this requirement to be relaxed for applications which value low latency over consistency.

```
defstrategy Split do
  # deploy, deliver, and process remain unchanged

  defhook dag do
    ~> forwarder ~> bucket ~>
  end

  defhook redeploy(refs) do
    forwarders = respawn(refs, :forwarder)
    buckets = respawn(refs, :bucket)
    mergers = respawn(refs, :merger)
    {forwarders, buckets, mergers}
  end
end
```

Listing 6.12: Implementation of the ABS hooks for the `Split` strategy. The `dag` hook specifies the communication that occurs within the strategy; an empty left-hand side refers to the input received from the `deliver` hook, while an empty right-hand side refers to emitting data.

the same data several times, the same write may occur several times, a situation known as the output commit problem [Elnozahy et al., 2002]. Thus, operations need to be able to perform these writes without causing inconsistency. There are two typical solutions to this problem:

- The write is idempotent and can thus be executed several times without issue (e.g. writing the same value to a database).
- The system that the data is written to enables transactional writes; a set of writes is then only committed once a checkpoint has been created.

6.4.2 Asynchronous Barrier Snapshotting in Skitter.ex

We successfully applied ABS to Skitter.ex, which enables it to handle the failure of one or several worker nodes of the cluster.[6] This required some modifications to the strategy DSL, but did not require any changes which break the modular nature of Skitter's distribution strategies. We discuss these changes, and the most important changes to the Skitter.ex runtime system, in the following sections. Listing 6.12 shows an example of the abstract `Split` strategy with the additional hooks required for our preliminary implementation of ABS.

[6]Failure of the master (i.e. coordinator) node is not handled, as the state maintained by this node falls outside of the state captured by ABS. Other techniques (typically based on replication) are typically used for this. Implementing these is a technical challenge rather than a scientific one.

Capturing Communication Patterns

Typically, the programming model offered by a DSPF is structured in such a way that the DSPF can statically determine all possible communication paths. In other words, the DSPF is aware of all the downstream workers to which a worker may send data. This information is necessary to correctly propagate barriers throughout the run-time graph. Skitter.ex's strategy DSL enables developers to dynamically determine to which worker a message is sent. This makes it more difficult to apply ABS, as it is not immediately clear to which workers a barrier should propagate.

Our implementation of ABS tackles this issue by making three changes to the strategy DSL which provide the runtime with additional information about the inter-worker communication:

Static Worker Set Workers may not be spawned after the application has finished deployment. In other words, `local_worker` and `remote_worker` may only be used inside the `deploy` hook.

Predefined Communication Each strategy implements a new `dag` hook, which specifies how the various workers in the strategy communicate with each other. This hook specifies a DAG (using the syntax introduces by the workflow DSL) which specifies how the workers communicate at the role level. An implementation of this hook can be found in Listing 6.12.

No Cycles Workers are not allowed to communicate with other workers which have the same role; this includes workers communicating with themselves. This is a technical limitation of the current implementation, and not inherent to ABS.[7]

An alternative approach, which we did not explore as of yet, would be to create an additional hook which is responsible for propagating barriers throughout the strategy. Such an approach would place more burden on the strategy developer, but would support the current dynamic nature of worker communication.

Additional Hooks

Besides the `dag` hook, a strategy needs to specify how it is deployed from a checkpoint. This is done in the `redeploy` hook, which is triggered by the runtime system when it is attempting to redeploy a DSPA from a checkpoint. This hook is called with a reference to a checkpoint[8] scoped to the current operation. This

[7] Which can handle cycles [Carbone et al., 2017].

[8] The actual checkpoint data is distributed over the cluster and can thus not be accessed directly.

reference can be used by the `respawn` primitive, which will respawn all the workers with a given role with the state contained in the checkpoint.

By default, our implementation of ABS will store the state of each worker directly inside the checkpoint. However, in some situations, developers may wish to modify or minimise this state before the checkpoint is persisted. This can be done in the optional `make_checkpoint` hook, which is called with the worker's state and its role, and determines which data gets persisted inside the checkpoint. An optional hook, `restore_checkpoint`, forms the counterpart of `make_checkpoint` and is used to transform the checkpoint back into a state that can be used by a newly respawned worker.

Implementation

Besides the creation of the necessary infrastructure to store and retrieve checkpoints on a cluster, the following changes were required to integrate ABS into Skitter.ex. These changes were integrated into the runtime system of Skitter.ex, although an implementation on top of the metadata and inheritance systems discussed in Section 6.2 is also possible.

Source Nodes were modified to enable the runtime system to use them to inject barriers into the application.

Metadata discussed in Section 6.2.3 was extended to store the latest injected barrier, which workers use to properly align their input streams.

Workers were modified to buffer data on an input stream when required for stream alignment.

Barrier Propagation is handled by the runtime system based on the information contained within the `dag` hook.

6.5 Conclusion

Skitter.ex is a technical instantiation of the ideas introduced by the Skitter programming model. It features three abstractions: *workflows*, *operations*, and *distribution strategies*, which serve to express DSPAs, data processing logic in these applications, and their distribution logic, respectively. Strategies specify which *callbacks* an operation needs to implement, and are themselves composed of several *hooks*, which are called by the Skitter.ex runtime system in response to predefined events.

We combine these core abstractions with conveniences inspired by modern programming language concepts and DSPFs, such as enabling the use of operators

inside the workflow language and an inheritance system for strategies, to offer a flexible yet practical DSPF that enables developers to express their DSPAs using high-level abstractions while still offering them full control over the distribution logic of their applications. These abstractions were built on top of Elixir, which served as a flexible platform for the creation of DSLs while also offering powerful support for distributed programming.

While standard Skitter.ex is currently not resilient to potential partial failures, preliminary work done in the context of a Master's Thesis shows that techniques from existing DSPFs can be applied without impacting its support for the modular expression of distribution strategies.

7. Experimental Evaluation

In this chapter, we evaluate the expressiveness, modularity and performance characteristics of Skitter based on its current implementation in Elixir. We do this by implementing well-known distribution strategies published in the Big Data stream processing community and by reproducing their published benchmarks in Skitter.ex. Based on these benchmarks, we aim to answer the following questions:

Q1 How modular are distribution strategies expressed in Skitter.ex compared to the state of the art (i.e. Storm)?

Q2 Do distribution strategies implemented in Skitter.ex exhibit the same performance characteristics as those implemented in Storm?

Q3 How much overhead do Skitter.ex's language abstractions introduce?

Q4 Can the performance of a DSPA be improved by selecting an alternative distribution strategy?

Additionally, we discuss the implementation of several distribution strategies from contemporary DSPFs on top of Skitter.ex.

7.1 Experimental Setup

We answer our research questions by reproducing the benchmarks described by Lin et al. [2015] and Nasir et al. [2016]. The former work introduces distribution strategies for `join` operations, while the latter introduces distribution strategies for key-based `reduce` operations. We selected these works for several reasons:

- `join` and `reduce` operations are commonly used in DSPAs and can have a major impact on their performance [Karimov et al., 2018].
- Both works compare several different distribution strategies for the same operation with each other in a single experiment.

- The strategies benchmarked in these works serve as the foundation for many other strategies, as discussed in Section 2.2.1.
- These benchmarks operate on data that is freely available or that can be generated synthetically.

In the following sections, we discuss the benchmarks, strategies used therein, and the technical setup used for our experiments. An artefact containing the full source code and associated scripts used for the experiments described below is available online [Saey, 2025].

7.1.1 Benchmarks

Both benchmarks we implement compare the average throughput of various distribution strategies distributing a particular operation. The *WordCount* benchmark compares strategies for distributing key-based state, and the *Join* benchmarks compare different strategies for distributing joins. Table 7.1, found at the end of this section, summarizes the benchmarks, the strategies contained therein, and the labels we use to refer to them when discussing results.

WordCount

Nasir et al. [2016] introduce distribution strategies for key-based stateful operations geared towards handling highly skewed workloads. They introduce the *D-choices* (DC) and *W-Choices* (WC) strategies, and compare them to *Partial Key Grouping* (PKG), introduced by the same authors in earlier work [Nasir et al., 2015], and to the *Key Grouping* (KG) and *Shuffle Grouping* (SG) strategies. We provide a brief summary of each strategy below; a more elaborate discussion of strategies for managing key-based state can be found in Section 2.2.1.

Key Grouping refers to dividing the keys over the available workers through the use of hashing. Each worker manages the state of several keys; each key is managed by one worker.

Partial Key Grouping splits the processing of data records for each key over two workers. Hashing is used to select the two workers for each key; each upstream worker sends an outgoing data record to the worker to which it has sent the least amount of data records so far.

W-Choices uses the space-saving algorithm [Metwally et al., 2005; Berinde et al., 2010] to track the most frequently occurring keys and distributes these differently from the other keys. Data records for frequently occurring keys are split over all the workers to be processed; others are handled by PKG.

D-Choices functions similarly to W-Choices, but dynamically determines the number of workers (called d) over which data records for frequently occurring keys should be distributed.

Shuffle Grouping refers to randomly distributing data records over the available workers, not taking the key of the data record into account. This strategy is used as a comparison point, as it achieves a uniform distribution of work over the cluster, regardless of skew.

Nasir et al. investigate if their strategies attain an even distribution of work over the cluster. This is done by evaluating these strategies in the context of an application which counts random words drawn from a synthetically generated, highly skewed dataset. The dataset contains 10^4 words, distributed according to a Zipf distribution with an exponent $z \in \{1.4, 1.7, 2.0\}$; higher values for z indicate a higher level of skew. The application does not perform any operations on the aggregated state. Instead, work is simulated by adding a fixed delay (of 1ms) to the processing of each message. The W-Choices, D-Choices and PKG strategies all split the processing of data for a single key over several cluster nodes. However, since the authors are only interested in the even distribution of work over the cluster, partial results are never merged again, as they would be in a real application.

Join

Lin et al. [2015] introduce the *Join-Biclique* (JB) strategy and its *Join-Biclique ContRand* (CR) variant for the distribution of `join` operations and compares these to the *Join-Matrix* (JM) strategy [Elseidy et al., 2014]. Both strategies form the foundation of various strategies used to distribute `join` operations, c.f. Section 2.2.1. We discuss all three strategies below. In this discussion, the streams to be joined are called R and S; individual data records of these streams are referred to as r_1, r_2, s_1, s_2, and so on.

Join-Matrix groups the workers which will perform the `join` in a matrix, as shown in Figure 7.1. Data records of R are sent to each worker of a randomly selected row of this matrix, while data records of S are sent to each worker of a randomly selected column of the matrix. This ensures that any potential pairing of records, $(r_i, s_j) \in R \times S$, occurs in exactly one worker. A drawback of this approach is that each data record has to be stored multiple times.

Join-Biclique ensures that each data record is stored only once by grouping the available workers into those which will store data records of S and those which will store data records of R. Each data record is then sent to one

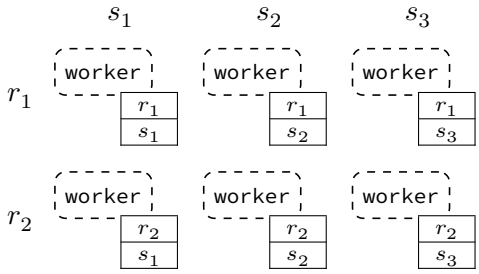

Figure 7.1: Data distribution of the Join-Matrix strategy. Elements of R are sent to each worker in a random row, elements of S are sent to each worker in a random column.

worker of its own group to be stored and to all workers of the other group to be matched. To guarantee correctness, each of these workers must process incoming data records in the same order, which requires additional synchronisation logic with additional overhead.

Join-Biclique ContRand further subdivides the groups introduced by Join-Biclique into several subgroups. Each subgroup is responsible for storing data records associated with a specific key (determined through hashing). Each incoming data record is sent to one worker of its subgroup to be stored and to all the workers of the appropriate subgroup of the other stream to be matched. Thus, ContRand ensures a data record does not need to be processed by all the workers of the other stream, at the cost of making the strategy content-sensitive and thus vulnerable to data skew.

Lin et al. compare these distribution strategies by implementing and comparing two `join` queries present in the TPC-H benchmark suite.[1] Specifically, they use the `join` operations present in queries five and seven. For query five, this is Region $\bowtie$ Nation $\bowtie$ Supplier $\bowtie$ LineItems; Nation $\bowtie$ Supplier $\bowtie$ LineItems is used for query seven. The input for this experiment is generated using the `dbgen` tool, part of TPC-H. Before performing the experiments, the generated data is processed to only include the values used by the benchmark. Input data is fed to the application at a fixed rate, which differs for each input stream.

7.1.2 Technical Setup

We implement the distribution strategies and benchmarks discussed above in Skitter.ex, Storm, and plain Elixir, and use the implementations to run the experiments

[1] https://www.tpc.org/tpch/ (visited on 27/04/2026)

Table 7.1: Overview of the benchmarks and strategies used in the evaluation.

Benchmark	Strategy		Label
WordCount Nasir et al., 2016	Key Grouping		KG
	Shuffle Grouping		SG
	Partial Key Grouping	Nasir et al., 2015	PKG
	W-Choices	Nasir et al., 2016	WC
	D-Choices	Nasir et al., 2016	DC
Join Lin et al., 2015	Join-Matrix	Elseidy et al., 2014	JM
	Join-Biclique	Lin et al., 2015	JB
	Join-Biclique ContRand	Lin et al., 2015	CR

described in the original work. For both benchmarks, we adjust the amount of created workers (tasks for Storm) to fit the cluster environment on which we evaluate our work (discussed below). For the WordCount experiment, we create one worker for each hardware thread of the worker nodes of the cluster, leading to a total of 80 workers. For the `join` benchmarks, we aim to spawn an equal amount of workers on every cluster node, while still obtaining a reasonable distribution of work for the Join-Matrix strategy. For this reason, we spawn 20 workers for each join operation (20×2 for query five, 20×3 for query seven).

We consider the average throughput of the applications under test for our performance evaluation. This is calculated by measuring the elapsed time between the first and final data record and dividing this time by the total amount of processed data records. We run each experiment 30 times, unless noted otherwise, and visualise the mean of all runs. Error bars represent the 95% confidence interval. For the `join` benchmarks, input is provided to the application at a fixed rate. A limitation of this setup is that this fixed input rate may not be sufficient to saturate the application and effectively serve as a bottleneck, in which case the measured throughput rate may be lower than the achievable maximum throughput rate. We work around this limitation by experimentally determining the highest input rate for which the benchmark consistently finishes within a reasonable amount of time (ten minutes for the 10 GB experiments, one hour for 80 GB) after receiving all inputs.

Programmers using Storm typically programmatically acknowledge emitted data records. This is done to enable Storm's fault tolerance mechanism, which ensures each data record is processed at least once. Since Skitter.ex does not provide a

```
def word_count(skew, strategy) do
  zipf_words = Zipf.gen_word_distribution(skew, @alphabet_size, @string_size)

  workflow do
    node(ZipfSource, args: [elements: @input_amount, word_probs: zipf_words])
    ~> node(WordCount, with: strategy)
  end
end
```

Listing 7.1: WordCount benchmark implemented as a Skitter.ex workflow. `zipf_words` contains a statistical distribution of words, used by `ZipfSource` to generate input data. Names prefixed with `@` denote compile-time constants.

fault tolerance mechanism,[2] our Storm implementation of the benchmarks does not acknowledge emitted data records, ensuring this discrepancy does not prevent a fair comparison between both frameworks.

Our performance benchmarks are performed on a cluster of ten worker nodes and one master node. Each node is equipped with a 4-core Intel® E5–1620 Xeon® CPU with 8 hardware threads running at 3.50 GHz. Each machine has 32 GB of 2133 MHz DDR4 RAM, and is configured with 32 GB of swap space. The cluster nodes are connected with a 10 Gigabit Ethernet connection and run Ubuntu 22.04.4. We compiled and ran our evaluation with Elixir 1.17.2 on Erlang/OTP 27.0.1 and with Storm 2.6.3 on OpenJDK 21.0.3.

7.2 Implementation in Skitter.ex

Before discussing the results of our evaluation, we reflect on how using Skitter.ex influenced the implementation of the benchmarks, and of the operations and distribution strategies contained therein. We do not discuss the full implementation of the benchmarks here and simplify the code examples somewhat for ease of reading; readers interested in the full code of our experiments are referred to the aforementioned artefact [Saey, 2025].

7.2.1 Distribution Strategies as a Language Construct

The main novelty of Skitter is its reification of distribution strategies as a language construct. We use this in our implementation of all benchmarks in Skitter.ex. As an example, the WordCount benchmark in Skitter.ex (shown in Listing 7.1) is an Elixir function which is parametrised with the benchmark parameters (the level of skew and the strategy to use) and returns a workflow. This stands in contrast with the

[2]The version of Skitter which uses ABS (c.f. Section 6.4.1) is not discussed in this chapter.

```
1  defoperation WordCount, in: words, out: current do
2    initial_state 0
3
4    defcb key(word), do: word
5
6    defcb react(word) do
7      state <~ (state() + 1)
8      {word, state()} ~> current
9    end
10
11   defcb merge(word, to_merge) do
12     state <~ state() + to_merge
13     {word, state()} ~> current
14   end
15 end
```

Listing 7.2: WordCount operation with support for split keys.

implementation in Storm, which requires a separate topology definition for each strategy. We discuss the effect this has on the modularity of the application as a whole in Section 7.3.1.

7.2.2 Callback Interface of Operations

When writing a distribution strategy, a developer must decide on the interface between strategy and operation, i.e. on the callbacks the strategy requires the operation to implement. It is important that this interface remains generic enough so that different strategies can be used to distribute an operation. Typically, this interface ends up resembling the interface existing DSPFs such as Spark or Flink offer through their operator API (c.f. Section 2.3). The main difference between Skitter and these DSPFs is that Skitter enables an operation to be distributed by any strategy that uses the same interface, or a subset thereof.

An example of this can be found in the implementation of the `WordCount` operation, shown in Listing 7.2. This operation implements both the `key` and `react` callbacks, enabling it to be distributed by the `KeyedState` strategy discussed in Section 6.1.3, but also implements the `merge` callback, which enables it to be distributed by any strategy that splits the processing for a particular key over several workers, such as PKG, W-Choices, and D-Choices.

Table 7.2 shows an overview of the callbacks each operation in our benchmark implements and the various strategies that use them. All Join strategies rely on the same callback interface: a `key` callback which extracts the key of incoming data records and a `match` callback which merges two data records. All WordCount strategies use `react` and `key` as discussed above, while the `merge` callback is only used by those strategies which split the state of a single key over multiple workers.

Table 7.2: Operation callbacks and the strategies that use them.

Operation	Callback	Strategies
WordCount	`key(word)`	KG PKG WC DC
	`react(word)`	SG KG PKG WC DC
	`merge(key, partial_state)`	PKG WC DC
Join	`key(value)`	JM JB CR
	`match(left, right)`	JM JB CR

7.2.3 Stateful Deliver Logic

The PKG, D-Choices, and W-Choices distribution strategies all require stateful logic to decide which worker will process a data record. In Storm, groupings can read and modify a state which is stored inside the task of the sending component, as we discuss in Chapter 3. This is different from `Skitter`, which only allows state to be stored inside a worker and which prohibits the `deliver` hook from accessing this state, as we discuss in Section 5.1.4.

Typically, this is not an issue, as the `deliver` logic of many distribution strategies is stateless (e.g. Key Grouping, Shuffle Grouping, Join-Matrix, Join-Biclique, Join-Biclique ContRand). To express stateful `deliver` logic, an intermediate worker can be used. One of these intermediate workers is spawned on each cluster node; the `deliver` hook is then responsible for sending the data record to be processed to the worker on the current node. The `process` hook of the forwarder can then execute the required stateful logic and send the data record to an appropriate worker. We exemplify this in Listing 7.3, which showcases part of the `KeyedState.Forwarded` strategy which is used as a base for the PKG, W-Choices and D-Choices strategies. This strategy spawns an intermediate worker, here called the *forwarder*, which executes the stateful `deliver` logic.

7.2.4 Complex Worker Interaction

The Join-Biclique strategy and its ContRand variant require complicated worker interaction to ensure the various workers responsible for joining data records do not emit duplicate values. Concretely, a set of *sender* workers needs to maintain a logical clock which needs to be synchronized and sent to all join workers at a regular interval. As in the previous section, we tackle this issue by introducing different worker roles. Our implementation of the Join-Biclique strategy uses two worker types: *senders* and *joiners*. The first is responsible for sending data to joiners and for maintaining a logical clock which is sent to the joiners and other

```
defstrategy KeyedState.Forwarded do
  defhook deploy(args) do
    # Create other workers
    state = strategy().forwarder_state(context())

    forwarders =
      Remote.on_all_workers(fn -> local_worker(state, :forwarder) end)
      |> Map.new()

    # Return forwarders to be part of deployment
  end

  defhook deliver(data) do
    forwarders = # Fetch forwarders from deployment
    send(forwarders[Remote.self()], data)
  end

  defhook process(data, state, :forwarder) do
    # Execute stateful deliver logic
  end

  defhook process(data, state, ...) do
    # Execute non-forwarder logic
  end
end
```

Listing 7.3: Using forwarder workers to implement stateful deliver logic. The `forwarder_state` hook is implemented by child strategies which extend this strategy, as explained in Section 6.2.2. Listing C.1 shows an example of a strategy which uses this logic.

senders at regular intervals. The second is responsible for joining the received data records, when this is permitted by Join-Biclique's synchronization logic.

7.3 Results

We now answer the four research questions set out at the start of this chapter by setting up various experiments based on the benchmarks described in Section 7.1.

7.3.1 Modularity of Distribution Strategies

To examine the modularity of distribution strategies in Skitter.ex, we measure which types of code a programmer must adjust to change the distribution strategy of an existing application and compare it to the types of code that need to be changed in the current state of the art, i.e. to Storm. This is done by implementing the experiments using a basic distribution strategy (Key Grouping for the Word-

Table 7.3: Lines of code added or modified to change distribution strategy from Key Grouping to Shuffle Grouping, Partial Key Grouping, W-Choices, and D-Choices (WordCount) or from Join-Matrix to Join-Biclique and Join-Biclique ContRand.

		Strategy	**Storm**					**Skitter**				
			Topology	**Component**	**Grouping**	**Other**	**Total**	**Workflow**	**Operation**	**Strategy**	**Other**	**Total**
WordCount		SG	1	0	0	0	1	1	0	8	0	9
		PKG	1	0	0	0	1	1	0	46	0	47
		WC	1	0	29	91	121	1	0	71	25	97
		DC	1	0	59	91	151	1	0	107	25	133
		PKG†	4	38	0	0	42	1	4	65	0	70
		WC†	4	38	29	91	162	1	4	90	25	120
		DC†	4	38	59	91	192	1	4	126	25	156
Join	Q5	JB	29	162	46	0	237	3	0	119	0	122
		CR	29	162	61	0	252	3	0	134	0	137
	Q7	JB	22	162	46	0	230	2	0	119	0	121
		CR	22	162	61	0	245	2	0	134	0	136

† With key merge logic.

Count experiment, Join-Matrix for the `join` experiment), after which we modify the code to use a different strategy and count the lines of code that were added or modified between both versions. Since we only change the distribution strategy, the locations that were changed indicate which code a programmer must adjust to change distribution strategies. We categorize the changed lines according to the abstractions offered by the DSPF (topology, component and grouping in Storm; workflow, operation and strategy in Skitter.ex), and the "Other" category, used to capture code expressed in the base language. The results of this experiment are shown in Table 7.3. Each row in this table represents a strategy and every column represents the lines of code changed for each category in both Storm and Skitter.ex compared to the implementation of the basic strategy. The results shown in this table are discussed per benchmark below.

WordCount

The first two lines of Table 7.3 show that very few changes are needed to use the PKG or Shuffle Grouping strategies in Storm. This is the case because Storm offers a built-in grouping for both of these strategies. Thus, this strategy can be used in Storm by making minor changes to the topology. Skitter.ex does not include an implementation of these strategies by default, so changes are needed in two places: the strategy needs to be defined, and the workflow needs to be adjusted to use the new strategy (through the use of `with:`, described in Chapter 6).

The W-Choices and D-Choices strategies are not provided by Storm but can be defined as a grouping; the Skitter.ex implementation of these strategies are expressed as a strategy used by the workflow, similar to how this was done for PKG. Table 7.3 shows that code in the "Other" category was added to implement these strategies in both Storm and Skitter.ex. This code contains an implementation of the "Space-saving" algorithm [Metwally et al., 2005; Berinde et al., 2010], used by the W-Choices and D-Choices strategies.

The WordCount benchmark is only concerned with how the various strategies balance work over the cluster. Therefore, it does not merge the partial results for each key generated by the PKG, D-Choices and W-Choices strategies. This merge logic would be present in a realistic application, however. The PKG, W-Choices and D-Choices rows marked with † compare an implementation of these strategies with this merge logic included compared to the base Key Grouping implementation (which does not split keys and therefore does not need this logic). In Storm, this results in the creation of an additional Component (a `MergeBolt`) and extra changes to the topology, as the merge bolt and the connections to this bolt need to be encoded in the topology definition. In Skitter.ex, these changes are captured in the strategy definition, meaning no additional changes to the workflow are needed. Merging partial states associated with a key is application-

specific behaviour, however, which is therefore encoded as a callback inside of the `WordCount` operation, as we show in Listing 7.2. The changes to the operation are reflected in the PKG†, WC† and DC† rows of Table 7.3. Storing this logic inside the operation allows the reuse of the W-Choices, D-Choices and PKG strategies which stands in contrast to the `MergeBolt` bolt in Storm, which contains both distribution logic and application-specific behaviour. The updated `WordCount` operation in the Skitter.ex implementation is also still usable by the Key Grouping strategy, which simply does not call the added callback.

Join

The Join benchmark consists of two queries: query five and query seven from the TPC-H benchmark. Table 7.3 shows the changes required to implement the Join-Biclique strategy and its ContRand variant for each query. Implementing these strategies proves to be troublesome in Storm, where they require the creation of a grouping, several bolts, and spouts. All of these need to be wired together and to other components in the application's topology. In Skitter.ex, on the other hand, changes are contained to the strategy and workflow languages, as the strategy language is sufficiently expressive to capture the definition of both strategies without the need to change the workflow. Therefore, the only change made to the workflow is the strategy passed to `with:`; three and two lines are modified for queries five and seven, respectively, as the former uses three joins, while the latter uses two.[3]

Conclusion

The grouping abstraction provided by Storm suffices to express simple distribution strategies which determine how data is divided over several tasks. However, they are insufficient for expressing more complex distribution strategies. Thus, implementing complex distribution strategies in Storm requires changing the topology, components, and groupings of the application. In contrast, Skitter.ex manages to capture all the distribution logic inside its strategy abstraction.

7.3.2 Performance of Distribution Strategies

To evaluate whether or not distribution strategies maintain their performance when implemented in Skitter.ex, we run the benchmarks implemented in both Skitter.ex and Storm and compare the results with each other. In this experiment,

[3]We do not parametrize the workflow with the strategy for this experiment (as in Listing 7.1), as this may not accurately represent how a developer would change a strategy in a real application.

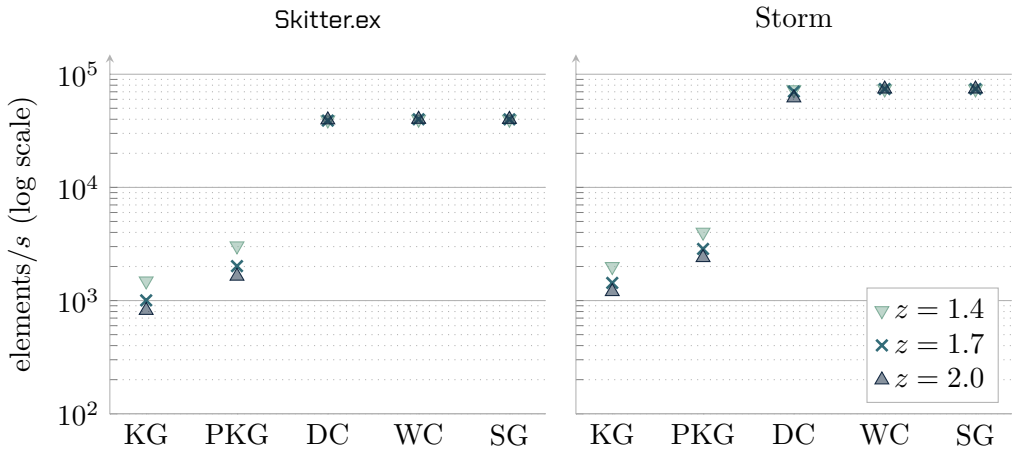

Figure 7.2: Comparison of the average throughput of the WordCount strategies (Key Grouping, Partial Key Grouping, D-Choices, W-Choices and Shuffle Grouping) written in Storm and Skitter.ex. Higher z-values indicate higher levels of skew.

we mainly verify that Skitter.ex does not change the performance properties of a distribution strategy. Therefore, our discussion focuses mainly on the relative performance of the various distribution strategies compared to one another.

WordCount

Figure 7.2 compares the average throughput of the various strategies in Skitter.ex with those obtained by Storm. Both implementations have similar behaviour: the D-Choices, W-Choices and Shuffle Grouping strategies remain largely unaffected by skew, while the Key Grouping and PKG strategies become slower as the skew level (z) increases. The rate at which these strategies slow down seems to be similar in both implementations. Additionally, the average throughput obtained by both DSPFs seems to fall within the same order of magnitude.

Join

Figure 7.3 compares the average throughput of the join strategies when used to execute the queries discussed in Section 7.1 on 10 GB of data. The behaviour of the strategies is similar in both Storm and Skitter.ex: the Join-Matrix strategy outperforms both variants of Join-Biclique, while the Join-Biclique strategy is significantly outperformed by both Join-Matrix and Join-Biclique ContRand.

Figure 7.3 shows that Skitter.ex's Join-Biclique ContRand strategy significantly outperforms its Storm counterpart while the inverse is true for the Join-Biclique

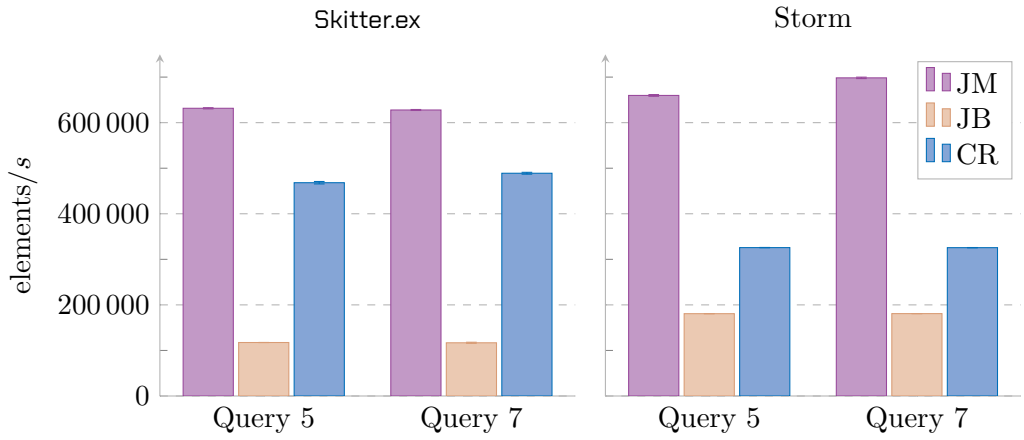

Figure 7.3: Comparison of the average throughput of the join strategies (Join-Matrix, Join-Biclique and Join-Biclique ContRand) for queries five and seven written in Skitter.ex and Storm.

strategy. Unfortunately, further investigation did not reveal an immediate cause for this discrepancy; we hypothesize it is related to implementation differences between both DSPFs and by the different virtual machines underpinning them.

As before, the average throughput of both DSPFs falls within the same order of magnitude.

Conclusion

While there are discrepancies between Storm and Skitter.ex, distribution strategies implemented in Skitter.ex show the same relative performance as those expressed in Storm. Moreover, the benchmarks expressed in Skitter.ex exhibit average throughput rates in the same order of magnitude as those written in Storm. Thus, we conclude Skitter.ex enables the modular implementation of distribution strategies without affecting their performance characteristics.

7.3.3 Overhead Introduced by Skitter.ex

After showing distribution strategies implemented in Skitter.ex exhibit the expected performance characteristics, we measure the computational overhead of Skitter.ex's abstractions. We do this by implementing an ad-hoc version of the experiments described in Section 7.1 in "plain" Elixir, and comparing the performance of this ad-hoc implementation with the Skitter implementation.

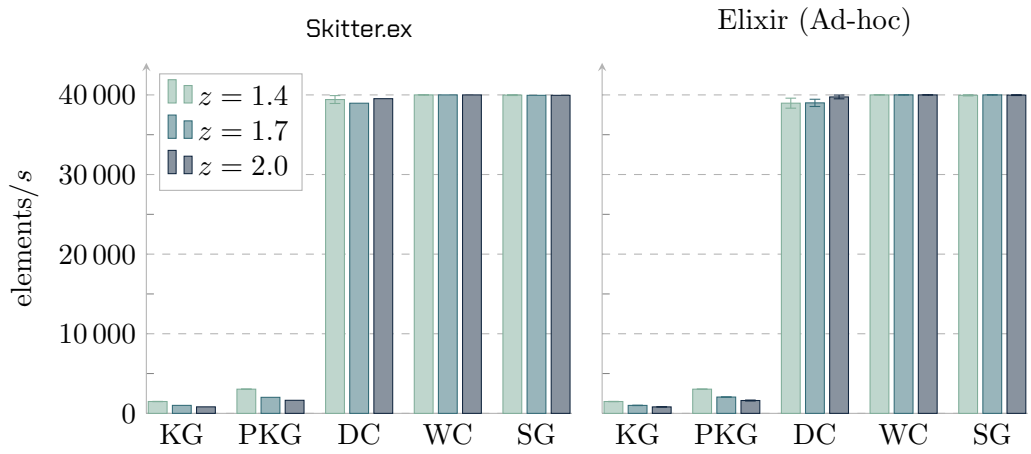

Figure 7.4: Comparison of the average throughput of the WordCount strategies (Key Grouping, Partial Key Grouping, D-Choices, W-Choices and Shuffle Grouping) written in Skitter.ex and Elixir. Figure 7.6 shows the difference between both as a percentage of the overall average throughput.

WordCount

Figure 7.4 compares the average throughput of the Skitter.ex implementation of the WordCount benchmark with an ad-hoc implementation in Elixir; the left side of Figure 7.6 shows the difference between both. Both applications exhibit similar average throughput and Skitter.ex outperforms the ad-hoc implementation in some cases, hinting that the abstractions introduced by Skitter do not affect the performance of this benchmark.

Join

Figure 7.5 compares the average throughput of the Skitter.ex Join benchmark with the ad-hoc implementation in Elixir; the right side of Figure 7.6 shows the relative difference between both as a percentage. The results show that the ad-hoc implementation outperforms Skitter.ex in every case, up to a worst-case difference of 30 %. We hypothesise that these differences are related to the meta-information the Skitter.ex runtime system attaches to each data element, discussed in Section 6.2.3. This additional information to track is especially impactful for communication and storage-heavy applications, both of which apply to the Join benchmark. This disproportionately affects the Join-Biclique and Join-Biclique ContRand strategies, which have several additional communication steps that occur before data records are stored in a join table.

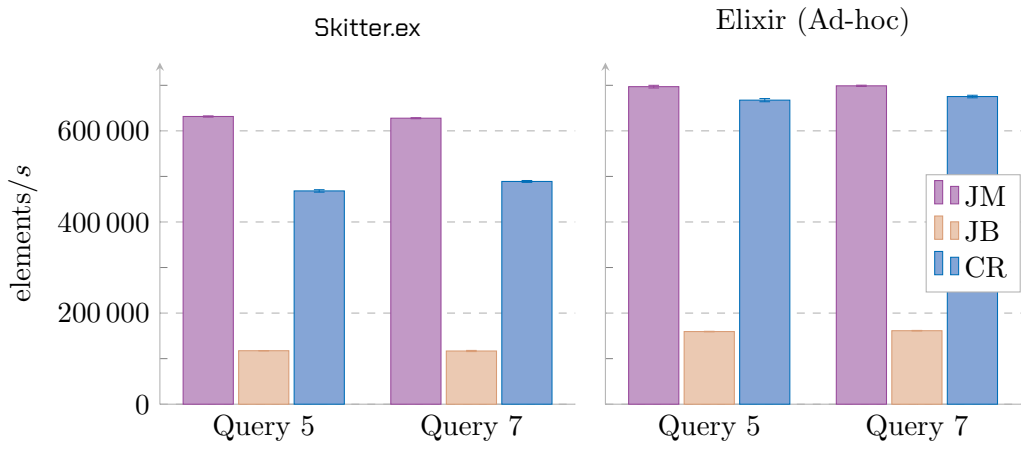

Figure 7.5: Comparison of the average throughput of the Join strategies (Join-Matrix, Join-Biclique and Join-Biclique ContRand) written in Skitter.ex and Elixir. Figure 7.6 shows the difference between both as a percentage of the overall average throughput.

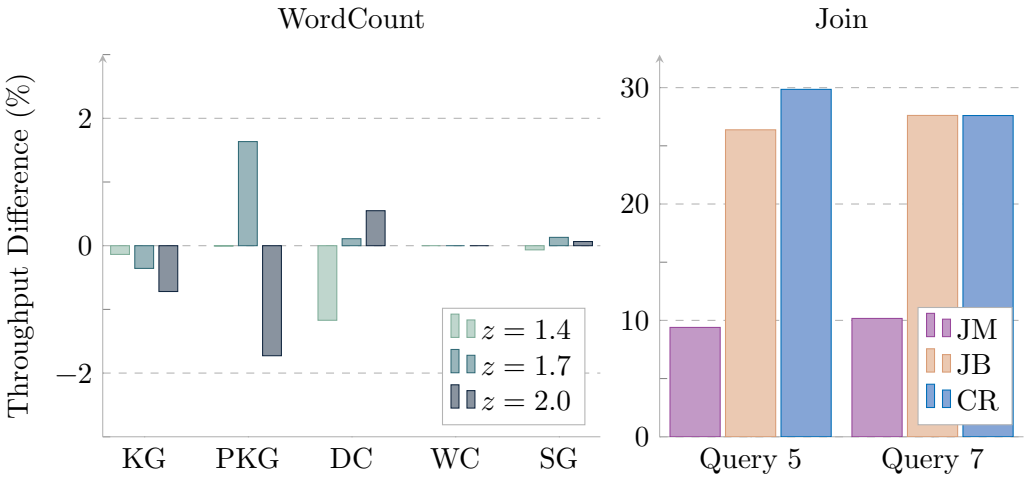

Figure 7.6: Relative difference in average throughput between the Skitter.ex and Elixir implementations of the WordCount (left) and Join (right) benchmarks. This shows the difference in average throughput between the Skitter.ex and Ad-hoc implementations, calculated as (Ad-hoc − Skitter.ex)/Ad-hoc.

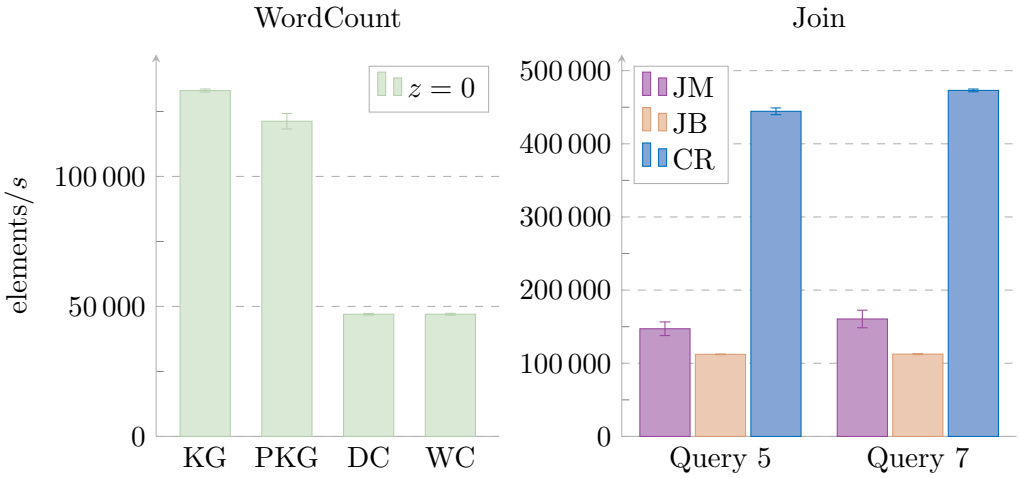

Figure 7.7: Average throughput of the modified WordCount (left) and Join (right) benchmarks. The input of the WordCount benchmark has been modified to no longer be skewed (i.e. $z = 0$); the Join benchmark has been modified to operate on a larger amount of input data (80 GB instead of 10 GB).

Conclusion

In the best case, Skitter.ex applications can exhibit performance characteristics which match or even (slightly) exceed those of handwritten implementations. However, in certain cases, the additional information tracked by the Skitter.ex runtime system may have an effect on the performance of the application.

7.3.4 Impact of Distribution Strategies on Performance

In both benchmarks, a single distribution strategy always outperformed the others: D-Choices was always the superior choice for the WordCount benchmarks, while Join-Matrix always boasted the highest average throughput for the Join benchmarks. In order to show that this is not always the case, and that the performance of an application can be improved by modifying its distribution strategies, we present a modified version of each application where a different distribution strategy attains the highest throughput.

WordCount

The left part of Figure 7.7 shows the average throughput of the WordCount benchmark if the words follow a uniform distribution and if the states of split keys are

merged, which is typically the case in a real application. In this benchmark, the Key Grouping strategy, which was the slowest by a large margin in the previous experiments, outperforms the PKG strategy while also outperforming both the D-Choices and W-Choices strategies by a significant margin.

Join

We change the scale of the Join benchmark to process 80 GB of data and show the results of this experiment on the right side of Figure 7.7. Due to the long duration of this experiment, we run this experiment 10 times instead of the 30 runs used for other experiments. In this scenario, the Join-Matrix strategy's higher memory use causes it to run out of memory on some nodes and use swap space, causing its average throughput to drop. The Join-Biclique ContRand strategy is less affected by the increase of received data and significantly outperforms both other strategies in this situation.

Conclusion

Our experiments show that the most performant distribution strategy for an operation is dependent on the properties of the application, and that changing the strategy can improve the performance of a stream processing application.

7.3.5 Summary

After investigating the results of our experiments, we can now answer the research questions we set out to answer at the start of this chapter.

Q1 Skitter.ex enables the expression of distribution strategies in a modular fashion by capturing this logic inside of its strategy abstraction.

Q2 Distribution strategies implemented in Skitter.ex exhibit the expected performance characteristics and have average throughput in the same order of magnitude as those implemented in Storm.

Q3 Skitter.ex's abstractions may situationally introduce overhead compared to a handwritten implementation of the same application.

Q4 Changing the distribution strategy of a DSPA can drastically change its performance properties.

7.4 Kafka Streams, Structured Streaming & Flink in Skitter.ex

As an additional experiment, we implement several strategies of DSPFs commonly used in the industry on top of Skitter.ex, to wit: Kafka Streams, Spark Structured Streaming [Armbrust et al., 2018], and Apache Flink [Carbone et al., 2015]. The goal of this is twofold:

- First and foremost, it allows us to verify Skitter.ex is expressive enough to implement these strategies.
- Second, it enables us to isolate these strategies from their technical embodiment in their respective DSPFs. This makes it possible to study them in isolation and to compare them with each other on common ground.

Concretely, we implement the strategies Kafka Streams, Apache Flink, and Spark Structured Streaming use to distribute their windowed join and keyed state operations. The work described in this section was performed in collaboration with Salaar Ahmed Chaudhary, in the context of his Master's Thesis, which was guided by the PhD Candidate [Chaudhary, 2025]. In this discussion, we mainly focus on the insights the experiment gave us which relate to the design and expressivity of Skitter.ex.

7.4.1 Split Operators in Skitter.ex

We consider the operator-based API offered by all three DSPFs. The first operators we consider are `keyBy` and `reduce`, which correspond to `keyBy` and `reduce` in Flink, `groupByKey` and `reduce` in Kafka, and `groupBy` and `agg` in Structured Streaming. Together, these two operators are used to implement operations which manage keyed state.

Having two operators cooperate to realise one logic step is natural in these DSPFs, as the (distribution) logic of each operator is wired into each framework. This same pattern is atypical in Skitter.ex, however, as it strictly associates one distribution strategy with one operation. This means that an operation can typically not make assumptions about its predecessor, like the `reduce` operator does in these frameworks. This is the reason why Skitter.ex's `keyed_reduce` operator, shown in Listing 6.4, accepts both `keyBy` and `reduce` logic instead of splitting them over two operators. Nevertheless, to remain faithful to the design of the DSPFs we emulate, our experiment provides an implementation of both operators, effectively splitting the implementation of the various distribution strategies into two

```
1  def wordcount(key_strat, reduce_strat) do
2    workflow do
3      node(Source)
4      ~> node(KeyBy, with: key_strat, args: [&Map.get(&1, :key)])
5      ~> node(Reduce, with: reduce_strat, args: [
6           fn word, count -> {count + 1, {word, count + 1}} end,
7           0
8         ]
9      ~> node(Output)
10   end
11 end
```

Listing 7.4: WordCount workflow which uses the `keyBy` and `reduce` operators. The strategies used by the operators are provided as an argument to the function which wraps the workflow.

separate pieces. An example workflow which uses both operators is shown in Listing 7.4. The definition of the `KeyBy` and `Reduce` operations, which are used by the strategies simulating Kafka, Flink, and Structured Streaming, can be found in Listings C.3 and C.4, in Appendix C.

7.4.2 Microbatches in Skitter.ex

The implementation of the distribution strategy used by Kafka and Flink is quite similar to the `KeyedState` strategy discussed earlier in this disseration (e.g. in Listing 6.3). We therefore only discuss the implementation of the strategy used by Structured Streaming's reduce operator.

Structured Streaming differs significantly from other DSPFs as it processes data in so-called "microbatches", as discussed in Section 2.3. Using microbatches allows for the reduce operation to happen in two stages: a partial aggregation on the `map` (i.e. sender) side, and a final aggregation on the `reduce` (receiver) side. The first stage creates a partial result for each key present in the current microbatch on every worker. Each of these partial results are then sent to a reducer, which will calculate the final, complete result for each key. By performing part of the aggregation upfront, less data records need to be sent over the wire.

Listing 7.5 contains the distribution logic of the first stage of the `Reduce` operator. The `deliver` hook of this strategy (lines 4–7) forwards the received data records (as these strategies assume they operate on microbatches) to a worker on the current cluster node called the mapper.

The `process` hook of the mapper (lines 9–25) performs a partial aggregation on all the data records in the received microbatch (lines 12–16) by calling the `react` callback on every data record with the current state of the key associated with this data record; if no state is associated with the key, the initial state passed to the

```
1  defstrategy StructuredStreams.Reduce do
2    # deploy omitted.
3
4    defhook deliver(batch) do
5      {mappers, _reducers, _config} = deployment()
6      send(mappers[Remote.self()], {:process_batch, batch})
7    end
8
9    defhook process({:process_batch, batch}, _, :map) do
10     {_mappers, reducers, config} = deployment()
11
12     partial_results = Enum.reduce(batch, Map.new(), fn {key, el}, acc ->
13       current = Map.get_lazy(acc, key, fn -> initial_state(config) end)
14       res = call(:react, state: current, config: config, args: [el])
15       Map.put(acc, key, res.state)
16     end)
17
18     Enum.each(partial_results, fn {key, aggregated} ->
19       idx = rem(Murmur.hash_x86_128(key), tuple_size(reducers))
20       reducer_pid = elem(reducers, idx)
21       send(reducer_pid, {:partial, key, aggregated})
22     end)
23
24     reducers |> Tuple.to_list() |> Enum.each(&send(&1, :mapper_done)
25   end
26
27   # process of reducer omitted
28 end
```

Listing 7.5: First stage of Structured Streaming's `reduce` strategy. Listing C.5 shows the complete implementation of this strategy.

operation is used. Each partial result is then sent to an appropriate downstream worker (lines 18–22), called the reducer. Which reducer to select is determined by hashing the key of the partial result. Finally, the mapper worker informs every reducer that it has completed its work for the current microbatch (line 24).

Listing 7.6 contains the distribution logic of the second stage of the `Reduce` operator. Any time a reduce worker receives a partial state for a key, it will update its current state for that key (lines 4–10). As before, this is done by calling `react` with the current state associated with the key, using the initial state if no state is present. Alternatively, the reduce worker may receive a message that a mapper is done (lines 12–21). When this is the case, the reducer may either have received this message from every upstream mapper, in which case it will emit its complete state (i.e. the state of every key it manages) as a complete microbatch (lines 15–17), or it may still be waiting to receive this message from additional mappers, in which case it updates its state (line 19).

Expressing the distribution strategy Structured Streaming uses for its `reduce`

```
defstrategy StructuredStreams.Reduce do
  # deploy, deliver, process of mapper omitted.

  defhook process({:partial, key, partial}, {state, done_count}, :reduce) do
    {_mappers, _reducers, config} = deployment()
    current = Map.get_lazy(state, key, fn -> initial_state(config) end)
    res = call(:react, state: current, config: config, args: [part_val])
    new_state = Map.put(state, key, res.state)
    {new_state, done_count}
  end

  defhook process(:mapper_done, {state, mappers_done}, :reduce) do
    {mappers, _reducers, _config} = deployment()

    if mappers_done + 1 == map_size(mappers) do
      emit(to_port(0, state))
      {Map.new(), 0}
    else
      {state, mappers_done + 1}
    end
  end
end
```

Listing 7.6: Second stage of Structured Streaming's `reduce` strategy. Listing C.5 shows the complete implementation of this strategy.

operator in Skitter.ex exposes the essential logic of its microbatching approach and shows how this approach influences its communication and state management. It also shows an example of a more complex distribution strategy expressed in Skitter.ex. Even more complex is the implementation of the various join strategies which we omit from our discussion here due to their inherent complexity.

7.4.3 Comparison of Strategies

We evaluate the implementation of the strategies from existing DSPFs on top of Skitter.ex by adapting the "Open Stream Processing" benchmark introduced by van Dongen and Van den Poel [2020]. This benchmark provides two synthetic data streams: "flow" and "speed", which represent traffic measurements. The former captures the number of vehicles that pass a given location, while the latter represents the speed of these vehicles. We measure the end-to-end latency of two workloads, each designed to test a particular strategy implementation:

Keyed Reduce Uses the "flow" stream to count the amount of vehicles that pass by a location.

Windowed Join Merges both streams by key within a fixed window.

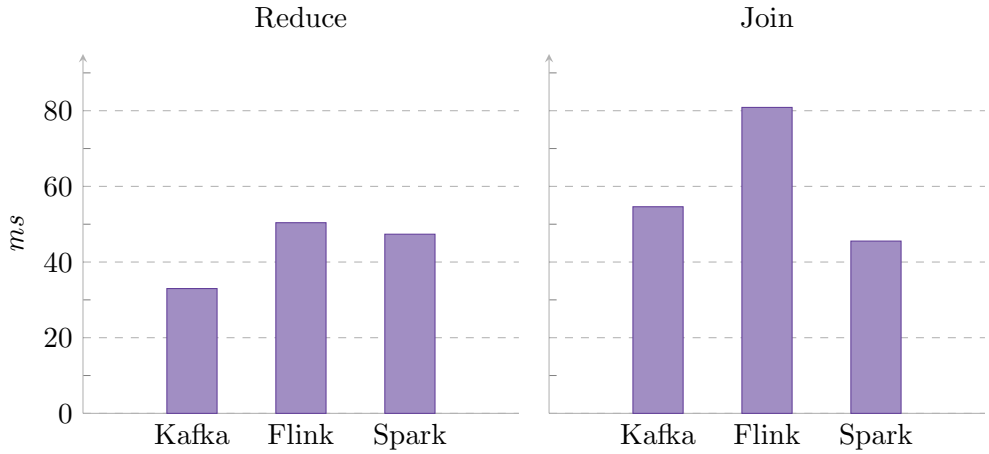

Figure 7.8: Median latency of the DSPF operators implemented in Skitter.ex. Visualisation of the results reported by Chaudhary [2025]. Spark refers to Spark Structured Streaming.

The left side of Figure 7.8 reports the median latency obtained by the benchmark of the `reduce` operator. Here, Kafka is significantly faster than Flink and (Spark) Structured Streaming; this is the case as Kafka does not redistribute data records over the cluster. Spark and Flink do, which introduces a significant amount of latency. Compared to the results described by van Dongen and Van den Poel, Flink's strategies perform significantly worse in our benchmarks. We hypothesise that Flink's optimisations such as chaining [Hirzel et al., 2014], which are not present in Skitter.ex, significantly lower its latency.

The right side of Figure 7.8 reports the median latency obtained by the (windowed) join benchmarks. Here, Spark Structured Streaming reports the lowest latency, as its microbatching approach maps well to operations such as join which inherently require data records to be buffered; this aligns with the findings of van Dongen and Van den Poel. The eager model used by Kafka, which compares incoming data records with previously buffered data records as they arrive provides a lower latency than Flink's model, which only processes elements in a window when it closes.

7.5 Conclusion

In this chapter, we investigated the modularity and performance characteristics of Skitter through its implementation, Skitter.ex. We did this by implementing distribution strategies from both the literature as well as from currently popular

DSPFs in Skitter.ex. Based on the implementations of these distribution strategies, we found the following answers to our research questions:

Q1 Distribution strategies expressed in Skitter.ex are modular, especially when compared to those implemented in Storm.

Q2 These strategies perform similarly to their implementation in Storm, both in terms of relative performance (how do the strategies relate to one another?) and absolute performance (what is the average throughput of Skitter.ex compared to Storm?).

Q3 Skitter.ex may introduce overhead compared to a handwritten implementation of these strategies in plain Elixir.

Q4 Changing a distribution strategy of a DSPA can drastically improve its performance.

8. Conclusion

This dissertation discusses the implementation and integration of distribution strategies in stream processing applications. We introduce a programming model which uses meta-programming to enable the expression of these strategies in a modular fashion. In this final chapter, we provide a bird's-eye view of the presented work, restate our contributions, discuss limitations and avenues for improvement, and wrap up with a look towards the future.

8.1 Summary

We started this dissertation by identifying *Distributed Stream Processing Applications* (DSPAs), and their distribution over a cluster. Such an application consists of several operations, which are distributed by a *distribution strategy*, the central topic of this dissertation. These strategies create workers, decide how these workers communicate, and decide which state they manage. Since inter-node communication is expensive, and since the location of data determines where work can be done, distribution strategies have a major impact on the performance of a DSPA.

We investigated distribution strategies from the state of the art and found that their performance is heavily impacted by application properties such as the properties of the incoming data, execution environment, or operations they distribute. Based on this, we formulated the central hypothesis of this dissertation: *distribution strategies should be modular and decoupled from their operations, so that the most optimal distribution strategy can be selected as needed without having to rewrite the application.*

An investigation of the currently existing *Distributed Stream Processing Frameworks* (DSPFs) revealed that no existing framework enables the modular expression of distribution strategies and that only one, Apache Storm, is flexible enough for their implementation. As a first contribution, we presented **featherweight-Storm**, a formal description of Storm which exactly captures its semantics. Based

on this description, we introduced the notion of **Storm-completeness**, which is a bar we argue any DSPF which aims to enable the implementation of distribution strategies should meet. A DSPF is Storm-complete if it is at least as expressive as (featherweight) Storm.

The key contribution we present in this dissertation is **Skitter**: a programming model which uses meta-programming to enable the modular implementation and deployment of distribution strategies. Skitter consists of three separate abstractions: the *operation* language which enables the expression of data processing logic, the *strategy* language which serves to implement distribution strategies, and the *workflow* language which combines operations into an application and ties each to a strategy.

We introduced two concrete instantiations of Skitter. The first, **featherweight-Skitter**, is a formal small-step semantics which captures the essence of the three abstractions mentioned above and their behaviour at run-time. The second, **Skitter.ex**, is a practical implementation of Skitter as three DSLs, one for each abstraction, and an accompanying runtime system built on top of Elixir. Each instantiation of Skitter is used as a research vehicle for a separate validation:

- We use featherweightSkitter to formally show that the introduced abstractions are Storm-complete. This proves that featherweightSkitter is at least as expressive as featherweightStorm, and thus, that any distribution strategy expressed in Storm can also be expressed in Skitter.

- Skitter.ex is used as a research vehicle for the experimental validation, wherein we implement real distribution strategies from the state of the art in both Storm and Skitter.ex, and compare their modularity and performance characteristics. We show that strategies expressed in Skitter.ex are modular and that their performance characteristics match those of Storm. Additionally, we show that strategies from existing DSPFs can be implemented on top of Skitter.ex and that the performance of a DSPA can be drastically improved by swapping out its distribution strategy.

8.2 Contributions

This dissertation introduced the following contributions:

Survey of Distribution Strategies We discuss the distribution strategies introduced in the state of the art as well as those used by existing DSPFs and provide a taxonomy of strategies used to distribute key-based stateful operations and joins.

Taxonomy of DSPFs We investigate contemporary DSPFs, provide an overview of the abstractions they offer and the extent to which they support the modular expression of distribution strategies.

FeatherweightStorm We introduce a formal description of Storm, the DSPF which is currently used to express distribution strategies in the state of the art. This description can be used to better understand the behaviour of Storm and to formalise existing distribution strategies written therein. To the best of our knowledge, this is the first formal description of the behaviour of a Storm application at run-time.

Storm-completeness Based on featherweightStorm, we formalise a lower bound on the expressiveness any DSPF which aims to enable the implementation of distribution strategies should meet, and named it "Storm-completeness", analogous to relational completeness [Codd, 1972].

Skitter To enable the modular expression of distribution strategies in DSPFs, we introduce a programming model for DSPFs called Skitter. Through Skitter's strategy language, we offer a meta-programming approach for the implementation of distribution strategies. This language controls the distributed execution of Skitter's operations; Skitter's workflow abstraction combines operations and strategies into a DSPA. We introduce two instantiations of Skitter, discussed next.

FeatherweightSkitter This instantiation of Skitter provides a formal description of the core language abstractions we introduce and their behaviour at run-time. It is intended to aid researchers in formally describing their distribution strategies and in understanding the behaviour of Skitter. It also serves as a useful tool to validate the expressive power of Skitter; this is done by proving that featherweightSkitter is Storm-complete.

Skitter.ex We provide an implementation of the language abstractions introduced by Skitter as a DSPF built on top of Elixir. This DSPF serves as a practical tool researchers can use to implement distribution strategies and experiment with them. It also serves to enable our quantitative evaluation of the modularity and performance of Skitter.

8.3 Limitations and Future Work

Skitter offers a practical model for the implementation and deployment of distribution strategies. However, in our focus on expressing these strategies, other concerns typically managed by a DSPF were left by the wayside. Resolving some of these gaps is an engineering problem; tackling others require additional research. We discuss both types of limitations in this section.

8.3.1 Technical Limitations of Skitter.ex

Building a DSPF from scratch is a significant undertaking. Therefore, Skitter.ex foregoes several features not directly related to expressing distribution strategies which are offered by many DSPFs used in production. Many of these features could be added to Skitter.ex without running into fundamental issues. We discuss the features for which we conjecture this to be the case below.

Monitoring Since DSPAs are intended to keep processing incoming data records ad infinitum, monitoring their health and overall status is useful. Typically, DSPFs offer tools such as dashboards to enable developers to do so. Skitter.ex offers developers access to so-called telemetry data which can be used to build these tools, but does not provide such a tool out of the box.

Backpressure Modern DSPFs typically provide a feedback mechanism, called backpressure, which ensures slow downstream operations don't get overwhelmed with data produced by faster upstream operations. Adding such a mechanism to Skitter.ex would make it more usable for production workloads.

Library of Operators DSPFs typically ship with a large set of built-in operators and abstractions (such as windowing abstractions) for building data processing pipelines. In contrast, Skitter.ex defines a limited amount of built-in strategies, operations and operators, which were added as needed. Fortunately, this is not a major limitation for Skitter.ex, as its open nature enables developers to add these to Skitter.ex without modifying its implementation.

Declarative API In Section 2.3, we discuss how modern DSPFs offer a SQL-like language to decoratively express DSPAs. It would be interesting to add such an interface to Skitter.ex, as an alternative to its workflow language.

Callback Interface Enforcement Skitter.ex does not ensure an operation implements all of the callbacks expected by the strategy it is paired with. Instead, when this is not the case, an error occurs at run-time. Enforcing this earlier would be useful to avoid potential operation-strategy mismatches.

8.3.2 Avenues for Future Research

While certain limitations can be tackled by putting in engineering effort, other limitations can only be solved after more fundamental, scientific issues have been tackled. We identify two limitations for which we believe this to be the case.

Scheduling

DSPFs typically use a *scheduler* to determine on which cluster node a particular worker is created. Such a scheduler can take concerns such as the hardware characteristics or current resource usage of the cluster into account when placing the worker to improve performance [Peng et al., 2015; Xu et al., 2018].

Skitter.ex does not feature such a scheduler and instead places workers created by `remote_worker` on a random node. To work around this, Skitter.ex enables strategy developers to explicitly place workers by combining functions such as `Remote.on_all_workers` with `local_worker`.

It would be interesting to design a scheduler for Skitter.ex and investigate how this design is influenced by its support for modular distribution strategies.

Fault Tolerance

Executing a DSPA on a cluster enables it to scale horizontally but also renders it vulnerable to *partial failures*, a situation where a subset of the cluster nodes fail. In Section 6.4, we discuss how we applied ABS, the fault tolerance algorithm used by Flink [Carbone et al., 2017], to Skitter.ex to make it resilient to partial failures. It would be interesting to expand on this work and to see how many of the limitations discussed there we can forego.

Additionally, several approaches exist for making a DSPF resilient to partial failures, each with their own strengths and drawbacks [Hwang et al., 2005; Zaharia et al., 2013; Carbone et al., 2017]. It would be interesting to expand our approach and to make these fault tolerance mechanisms pluggable and programmable, similar to how Skitter does this for distribution strategies. A major difference between the expression of fault tolerance and the distribution strategies discussed in this dissertation, is that strategies operate on the level of a single operation while a fault tolerance strategy would need to operate on the DSPA as a whole.

8.4 Closing Remarks

Modern DSPFs aim to hide the inherent complexity of distribution strategies from developers, offering them a streamlined, high-level development experience instead. While this has made these tools more accessible to mundane programmers, it also comes at a cost: developers are no longer able to control the distribution logic of their applications, which is problematic when the built-in strategies of their DSPF of choice hamper the performance of their applications.

In the meantime, researchers in the distributed stream processing community

have introduced several novel strategies and published these in the literature. These strategies are almost exclusively developed in Apache Storm, as more recent DSPFs do not allow for their expression.

We hope that the contributions introduced in this dissertation help to bridge this gap between theory and practice. By showing that it is possible to allow for the expression of distribution strategies without imposing their complexity on non-expert programmers, and without sacrificing performance, we show that it is possible to get the best of both worlds, i.e. that it is possible to have a streamlined, high-level development experience while still enabling expert programmers to implement and use tailor-made distribution strategies when required.

For practitioners, we hope that our contribution of an *open* DSPF opens the door to new possibilities. At a basic level, our contributions can facilitate the development of novel distribution strategies or serve as a research vehicle for the comparison of existing strategies previously implemented in different frameworks. Beyond these basic possibilities, similar work—which also decouples computations from the details of their execution plan—has shown that it is possible to automatically determine the most appropriate strategy for an application [Ragan-Kelley et al., 2018]; we hope the work introduced in this dissertation can help enable this type of work for DSPFs in the future.

example

After taking a gamble on a research prototype he found online called "Skitter", M.S. cautiously enters the office. His fellow developers are not completely sold on this new tool yet, but are happy enough now that they can write their data analysis code in their own little corner of Adalyze's codebase. For his own part, M.S. is pleased enough: implementing JoinBiclique ContRand was not trivial by any means, but he is confident enough that he can reuse the implementation should this be needed in the future. What's more, he has a hunch that future performance bottlenecks in Adalyze won't be as hard to resolve as they were in the two previous versions.

For the first time in a long while, M.S. is content. All is well with the world.

A. Formal Notation, Syntax, Rules, and Functions

This appendix contains a complete overview of the formalisms introduced in Chapters 3 and 4, and of the translation between both discussed in Chapter 5.

Notational Conventions

Values We denote values using lowercase Latin letters (e.g. v).

Vectors Vectors are typeset using overlined letters (e.g. $\overline{v}$). The empty set notation ($\varnothing$) is used to denote the empty vector; the length of a vector, $\overline{v}$, is denoted as $|\overline{v}|$. A vector containing value v is denoted as $[v]$. Prepending or appending element v to vector $\overline{v}$ is denoted as $v \cdot \overline{v}$ or $\overline{v} \cdot v$, respectively.

Functions We denote functions which are part of the definition of an application using lowercase Greek letters (e.g. λ).

Mappings We denote mappings (i.e. sets of key, value pairs) with uppercase Greek letters (e.g. Π). Adding element e with key k to a mapping is denoted as $\Pi\,[k \mapsto v]$. Fetching an element from a mapping based on key k is denoted as a function call, e.g. $\Pi\,(k)$.

Tuples We typeset tuples using the standard (v_1, v_2) notation. Some tuples are typeset using the "constructor" notation, with a preceding calligraphic letter and angular brackets (e.g. $\mathcal{C}\langle v_1, v_2 \rangle$). This is done to make certain frequently occurring constructs visually distinct from each other.

Sets We make a distinction between several different types of sets:

- "Normal" sets, representing part of an application or its state at runtime, are represented using uppercase Latin letters (e.g. S).

- Sets of sequences are denoted as uppercase Greek letters (e.g. Σ).
- "Universal" sets, which represent any possible value of a certain type, are represented by the name of the type, typeset in bold (e.g. **Type**).
- Domains, which are typeset using "blackboard bold" notation, e.g. $\mathbb{N}$.

Shared Domains

The *data* domain, which represents any value a DSPA may need to process, is used by both fwStorm and fwSkitter, and is represented as $\mathbb{D}$.

Element		Domain		Name	Description
d, d_s, d_r	$\in$	$\mathbb{D}$	$=$	**DataDomain**	Any application data. d_s denotes state, d_r a result.
$\overline{d}$	$\in$	$\mathbb{D}^n$			Vector of data domain values

FeatherweightStorm

Abstract Syntax

A fwStorm application is defined as a *topology*: a two-tuple consisting of a set of components (vertices) and a set of groupings (edges). Components can be *spouts* (source nodes) or *bolts* (non-source nodes). All components are defined with a parallelism, which defines the maximum amount of tasks that are spawned for the component at run-time, a function, which determines what it does when it receives a data record, and an initial state. Groupings are defined with a "from" and "to" component, an initial state, and a grouping function, which determines to which tasks data records sent along the grouping will be sent.

Element		Set		Name		Form
				Topology	$::=$	(C, G)
c	$\in$	C	$\subseteq$	**Component**	$::=$	$s_p \mid b_o$
s_p			$\in$	**Spout**	$::=$	$\mathcal{S}_p\langle p, d, \sigma_s\rangle$

Continued on next page

(Continued)

Element		Set		Name		Form
b_o			$\in$	**Bolt**	$::=$	$\mathcal{B}_o\langle p, d, \sigma_b\rangle$
g	$\in$	G	$\subseteq$	**Grouping**	$::=$	$\mathcal{G}\langle c, b_o, d, \sigma_g\rangle$
σ_s			$\in$	**SpoutFunction**	$::=$	$d \mapsto d, d$
σ_b			$\in$	**BoltFunction**	$::=$	$d, d \mapsto d, d$
σ_g			$\in$	**GroupingFunction**	$::=$	$d, d, p \mapsto d, \overline{p_i}$
p			$\in$	**Parallelism**	$\subseteq$	$\mathbb{N}$
p_i			$\in$	**TaskIndex**	$\subseteq$	$\mathbb{N}$
$\overline{p_i}$					$\in$	$\mathbb{N}^n$

Semantic Entities

At run-time, a fwStorm application is represented by a set of tasks. Each task is associated with a component, and maintains a state for that component and states for each grouping which has that component as its origin. A task also contains a queue of values to be processed.

Element		Set		Name		Form
t	$\in$	T	$\subseteq$	**Task**	$::=$	$\mathcal{T}\langle p_i, c, d, \Gamma, \overline{q_s}\rangle$
Γ			$\subseteq$	**GroupingState**	$::=$	$g \mapsto d$
$\overline{q_s}$					$\in$	$\mathbb{D}^n$

Initial Configuration

The initial configuration of a fwStorm application contains p tasks for each component $c \in C$, where p is the parallelism hint of c. Each task is created with an index, its component, an initial state, a message queue, and an initial state for each grouping, as defined by $init_\Gamma$.

$$\bigcup \left\{ \left(\bigcup_{i=0}^{p-1} \{\mathcal{T}\langle i, c, d, init_\Gamma(c), \varnothing\rangle\} \right) \mid c = \mathcal{B}_o\langle p, d, \_\rangle \in C \vee c = \mathcal{S}_p\langle p, d, \_\rangle \in C \right\}$$

Reduction Rules

fwStorm defines two reduction rules. The set of components, C, and the set of groupings, G, are known statically in both rules. The first rule defines how a task for a spout can be used to produce a new data record at any time.

(SPOUT)

$$\frac{s_p = \mathcal{S}_p\langle \_, \_, \sigma_s\rangle \qquad d'_s, d_r = \sigma_s(d_s) \qquad T', \Gamma' = send(d_r, T, \Gamma)}{T \uplus \left\{\mathcal{T}\langle p_i, s_p, d_s, \Gamma, \varnothing\rangle\right\} \rightarrow T' \cup \left\{\mathcal{T}\langle p_i, s_p, d'_s, \Gamma', \varnothing\rangle\right\}}$$

The second defines how a task for a bolt with at least one data record in its queue may be activated to process the data record.

(BOLT)

$$\frac{\begin{array}{c} b_o = \mathcal{B}_o\langle \_, \_, \sigma_b\rangle \qquad d'_s, d_r = \sigma_b(d_s, d_q) \\ t = \mathcal{T}\langle p_i, b_o, d'_s, \Gamma, \overline{q_s}\rangle \qquad T' \uplus \left\{\mathcal{T}\langle p_i, b_o, d'_s, \Gamma, \overline{q_s}'\rangle\right\}, \Gamma' = send(d_r, T \cup \{t\}, \Gamma) \end{array}}{T \uplus \left\{\mathcal{T}\langle p_i, b_o, d_s, \Gamma, \overline{q_s} \cdot d_q\rangle\right\} \rightarrow T' \cup \left\{\mathcal{T}\langle p_i, b_o, d'_s, \Gamma', \overline{q_s}'\rangle\right\}}$$

Auxiliary Functions

$init_\Gamma$ defines the initial state for each task of component $c \in C$ by mapping each outgoing grouping of c to its initial state.

$$init_\Gamma(c) = \{(g, d) \mid g = \mathcal{G}\langle c, \_, d, \_\rangle \in G\}$$

The various $send$ functions are responsible for sending an outgoing data record, d, to tasks of the bolts downstream of the sending component.

$$send(d, T, \Gamma) = send_G(d, \{g \mid (g, \_) \in \Gamma\}, T, \Gamma)$$

$$\begin{aligned} send_G(\_, \varnothing, T, \Gamma) &= T, \Gamma \\ send_G(d, G_s \uplus \{g\}, T, \Gamma) &= send_G(d, G_s, T', \Gamma') \\ &\quad \text{where} \\ &\qquad T', \Gamma' = send_g(d, g, T, \Gamma) \end{aligned}$$

$$\begin{aligned} send_g(d, g, T, \Gamma) &= T', \Gamma[g \mapsto d_s] \\ &\quad \text{where} \\ &\qquad g = \mathcal{G}\langle \_, b_o = \mathcal{B}_o\langle p, \_, \_\rangle, \_, \sigma_g\rangle \\ &\qquad d_s, \overline{p_i} = \sigma_g(\Gamma(g), d, p) \\ &\qquad T' = send_t(d, b_o, \overline{p_i}, T) \end{aligned}$$

$$
\begin{aligned}
send_t(\_, \_, \varnothing, T) &= T \\
send_t(d, b_o, p_i \cdot \overline{p_i}, T \uplus t) &= send_t(d, b_o, \overline{p_i}, T \cup t') \\
&\text{where} \\
t &= \mathcal{T}\langle p_i, b_o, d_s, \Gamma, \overline{q_s}\rangle \\
t' &= \mathcal{T}\langle p_i, b_o, d_s, \Gamma, d \cdot \overline{q_s}\rangle
\end{aligned}
$$

Execution Sequences

A potential execution of a fwStorm application is defined as a sequence of task sets: $\overline{T}$. The set of all possible executions for an application is named Θ.

Configuration		Sequence		All Possible
T_i	$\in$	$\overline{T}$	$\in$	Θ

FeatherweightSkitter

Abstract Syntax

A fwSkitter application is defined as a DAG: a two-tuple consisting of nodes (data processing steps) and links (data dependencies). Any node consists of an operation (data processing logic) and a strategy (distribution logic). Distribution strategies are defined as a two-tuple of expressions, which we discuss later.

Element		Set		Name		Form
				Workflow	::=	(N, L)
		L	$\subseteq$	**Link**	::=	$\mathcal{L}\langle n, n, i\rangle$
n	$\in$	N	$\subseteq$	**Node**	::=	$\mathcal{N}\langle s, \Omega\rangle$
s			$\in$	**Strategy**	::=	$\mathcal{S}\langle e, e\rangle$
Ω			$\in$	**Operation**	::=	$cb \mapsto \rho$
ρ			$\in$	**Callback**	::=	$d, d \mapsto d, d$

Continued on next page

(Continued)

Element	Set	Name		Form
i	$\in$	**Port**	$\subseteq$	$\mathbb{N}$
cb	$\in$	**CallbackName**		
e	$\in$	**Expression**		

Expressions, Variables & Substitutions

fwSkitter formalises the various expressions that can be used to introduce a distribution strategy. Here, we separate these expressions into three groups: pseudovariables, normal expressions and syntactic sugar.

Pseudovariables can occur in any expression and are bound by a reduction rule. The table below summarises the meaning of each pseudovariable and the hooks they are used in.

Pseudovariable	Meaning	Hooks
`deployment`	deployment data of current node	process, deliver
`self`	reference to current worker	process
`msg`	message to process	process
`state`	state of the current worker	process
`data`	data to deliver	deliver
`port`	port receiving data	deliver

We use the following normal expressions in fwSkitter. The exact behaviour of these expressions is defined in Chapter 4. We summarise their intended use here.

Expression	Meaning
`null`	empty value
x	variable name

Continued on next page

(Continued)

Expression	Meaning
$h[h]$	get value at index from vector
$\texttt{let}\ x = e\ \texttt{in}\ e'$	bind variable in expression
$\texttt{let}\ x_s, x_r = \texttt{call}(cb, d_s, d_a)\ \texttt{in}\ e$	call cb, bind resulting value and state
$\texttt{spawn}(h, e)$	create worker with initial state h
$\texttt{send}(r, h)$	send h to worker at reference r
$\texttt{emit}(d)$	publish h to downstream nodes

The following expressions are all defined in terms of other, existing expressions.

Expression		Expanded		Meaning
$e; e'$	$\stackrel{def}{=}$	$\texttt{let}\ x = e\ \texttt{in}\ e'$	$x \notin FV(e')$	Perform e' after e
$\overline{e}^{\,0}$	$\stackrel{def}{=}$	$\varnothing$		Execute e n times.
$\overline{e}^{\,k}$	$\stackrel{def}{=}$	$e \cdot \overline{e}^{\,n-1}$		
$\overline{\texttt{send}}(\varnothing, h)$	$\stackrel{def}{=}$	$\texttt{null}$		Send h to every worker in $\overline{r}$
$\overline{\texttt{send}}(r \cdot \overline{r}, h)$	$\stackrel{def}{=}$	$\texttt{send}(r, h); \overline{\texttt{send}}(\overline{r}, h)$		

Semantic Entities

At run-time, a fwSkitter application is represented by its deployment data and its set of workers. The deployment data remains static, while the workers can change based on the data they process. Each worker contains a unique reference, a reference to its corresponding node, the expression it will evaluate when it receives a data record, a state, and a queue of values to be processed. The state and the queue of the worker may change when reduction rules are reduced.

Element		Set		Name		Form
				Configuration	::=	(Δ, W)
w	$\in$	W	$\subseteq$	**Worker**	::=	$\mathcal{W}\langle r, n, e, h, \overline{q}\rangle$
Δ			$\subseteq$	**Deployment**	::=	$n \mapsto h$
r	$\in$	R	$\subseteq$	**WorkerRef**		
h	$\in$	$\mathbb{H}$	$=$	**HybridDomain**	$=$	$\mathbb{D} \cup R \cup R^n$
$\overline{q}$					$\in$	$\mathbb{H}^n$

Initial Configuration

The initial configuration of a fwSkitter application is an empty set of workers and an empty deployment. Both are created by the application of (DEPLOY ALL).

$$(\varnothing, \varnothing)$$

Reduction Rules

fwSkitter defines two global reduction rules, (DEPLOY ALL) and (PROCESS). In turn, these rules invoke relations which may lead to the applications of other rules. In all of the reduction rules described below, N and L are available globally.

From an initial configuration of fwSkitter, only (DEPLOY ALL) may be applied. This rule repeatedly applies $\xrightarrow{deploy^*}$ until each node $n \in N$ is deployed.

(DEPLOY ALL)

$$\frac{\varnothing, \varnothing, N \xrightarrow{deploy^*} \Delta, W, \varnothing}{(\varnothing, \varnothing) \to (\Delta, W)}$$

(DEPLOY) reduces the `deploy` hook of the strategy of the node to a value, which then becomes the deployment data of that node.

(DEPLOY)

$$\frac{n = \mathcal{N}\langle \mathcal{S}\langle e, \_\rangle, \_\rangle \qquad e, n, \varnothing \xrightarrow{expr^*} h, n, W_n}{\Delta, W, N_d \uplus \{n\} \xrightarrow{deploy} \Delta\,[n \mapsto h]\,, W \cup W_n, N_d}$$

Once a worker has a message in its queue, (PROCESS) may be applied to process this value. This is done by invoking the `process` expression stored in the worker.

(PROCESS)

$$\frac{\begin{array}{c}e = [h_q/\texttt{msg}][h_s/\texttt{state}][\Delta\,(n)\,/\texttt{deployment}][r/\texttt{self}]e_p \\ e, \mathcal{W}\langle r, n, e_p, h_s, \overline{q}\rangle, (\Delta, W) \xrightarrow{process^*} h, \mathcal{W}\langle r, n, e_p, h_s, \overline{q}'\rangle, (\Delta, W')\end{array}}{(\Delta, W \uplus \{\mathcal{W}\langle r, n, e_p, h_s, \overline{q} \cdot h_q\rangle\}) \to (\Delta, W' \cup \{\mathcal{W}\langle r, n, e_p, h, \overline{q}'\rangle\})}$$

$\xrightarrow{expr}$ may be used while applying (PROCESS).

(EVAL EXPR)

$$\frac{w = \mathcal{W}\langle \_, n, \_, \_, \_\rangle \qquad e, n, W \xrightarrow{expr} e', n, W'}{e, w, (\Delta, W) \xrightarrow{process} e', w, (\Delta, W')}$$

During the application of (PROCESS), a worker may send itself a message.

(SEND SELF)

$$\begin{array}{c}\mathcal{E}[\texttt{send}(r, h_q)], \mathcal{W}\langle r, n, e, h_s, \overline{q}\rangle, (\Delta, W) \\ \xrightarrow{process} \mathcal{E}[\texttt{null}], \mathcal{W}\langle r, n, e, h_s, h_q \cdot \overline{q}\rangle, (\Delta, W)\end{array}$$

During the application of (PROCESS), a data record may be emitted. This will apply (DELIVER) for each node downstream of the current node.

(EMIT)

$$\frac{\begin{array}{c}w = \mathcal{W}\langle \_, n, \_, \_, \_\rangle \\ N_e = \{(n_t, i) \mid \mathcal{L}\langle n, n_t, i\rangle \in L\} \qquad d, N_e, (\Delta, W) \xrightarrow{deliver^*} d, \varnothing, (\Delta, W')\end{array}}{\mathcal{E}[\texttt{emit}(d)], w, (\Delta, W) \xrightarrow{process} \mathcal{E}[\texttt{null}], w, (\Delta, W')}$$

Applying (DELIVER) involves invoking the `deploy` hook of the downstream node.

(DELIVER)

$$\frac{\begin{array}{c}n = \mathcal{N}\langle \mathcal{S}\langle \_, e\rangle, \_\rangle \\ [d/\texttt{data}][i/\texttt{port}][\Delta\,(n)\,/\texttt{deployment}]e, n, W \xrightarrow{expr^*} \texttt{null}, n, W'\end{array}}{d, N_e \uplus \{(n, i)\}, (\Delta, W) \xrightarrow{deliver} d, N_e, (\Delta, W')}$$

We define several rules which can be used in any hook. (SPAWN) specifies how a worker is created for a node.

(SPAWN)

$$\frac{r \text{ fresh}}{\mathcal{E}[\texttt{spawn}(h, e)], n, W \xrightarrow{expr} \mathcal{E}[r], n, W \cup \{\mathcal{W}\langle r, n, e, h, \varnothing\rangle\}}$$

(SEND) specifies a message is sent to a worker by prepending the message to its queue.

(SEND)

$$\mathcal{E}[\mathtt{send}(r, h_q)], n, W \uplus \{\mathcal{W}\langle r, n, e, h_s, \overline{q}\rangle\} \xrightarrow{expr} \mathcal{E}[\mathtt{null}], n, W \cup \left\{\mathcal{W}\langle r, n, e, h_s, h_q \cdot \overline{q}\rangle\right\}$$

Callbacks can be called by looking up the callback in the operation of the current node and substituting the result into the remainder of the expression.

(CALL)

$$\frac{n = \mathcal{N}\langle \_, \Omega\rangle \qquad \rho = \Omega\,(cb) \qquad d'_s, d_r = \rho(d_s, d_a)}{\mathcal{E}[\mathtt{let}\ x_s, x_r = \mathtt{call}(cb, d_s, d_a)\ \mathtt{in}\ e], n, W \xrightarrow{expr} \mathcal{E}[[d'_s/x_s][d_r/x_r]e], n, W}$$

Values are bound by substituting them into an expression.

(LET)

$$\mathcal{E}[\mathtt{let}\ x = h\ \mathtt{in}\ e], n, W \xrightarrow{expr} \mathcal{E}[[h/x]e], n, W$$

Translating FeatherweightStorm to FeatherweightSkitter

translate translates a fwStorm application into a fwSkitter application with equivalent behaviour. We forego the detailed explanation of the translation here and refer the reader to Section 5.1.

$$translate : \mathcal{P}(\mathbf{Component}) \times \mathcal{P}(\mathbf{Grouping}) \mapsto \mathcal{P}(\mathbf{Node}) \times \mathcal{P}(\mathbf{Link})$$
$$translate((C, G)) = translate'(C, addport(C, G))$$
$$translate' : \mathcal{P}(\mathbf{Component}) \times \mathcal{P}(\mathbf{Grouping} \times \mathbb{N}) \mapsto \mathcal{P}(\mathbf{Node}) \times \mathcal{P}(\mathbf{Link})$$
$$translate'(C, G_i) = \left(\{trans_{c\to n}(c, G_i) \mid c \in C\}, \left\{trans_{g\to l}(g, i, G_i) \mid (g, i) \in G_i\right\}\right)$$

$$addport : \mathcal{P}(\mathbf{Component}) \times \mathcal{P}(\mathbf{Grouping}) \mapsto \mathcal{P}(\mathbf{Grouping} \times \mathbb{N})$$
$$addport(C, G) = \{addport'(\{g \mid \mathcal{G}\langle \_, c, \_, \_\rangle \in G\}, 0) \mid c \in C\}$$
$$addport' : \mathcal{P}(\mathbf{Grouping}) \times \mathbb{N} \mapsto \mathcal{P}(\mathbf{Grouping} \times \mathbb{N})$$
$$addport'(\varnothing, \_) = \varnothing$$
$$addport'(G_c \uplus \{g\}, i) = \{(g, i)\} \cup addport'(G_c, i + 1)$$

$$trans_{g\to l} : \mathbf{Grouping} \times \mathbb{N} \times \mathcal{P}(\mathbf{Grouping} \times \mathbb{N}) \mapsto \mathbf{Link}$$
$$trans_{g\to l}(\mathcal{G}\langle c_f, c_t, \_, \_\rangle, i, G_i) = \mathcal{L}\langle trans_{c\to n}(c_f, G_i), trans_{c\to n}(c_t, G_i), i\rangle$$

$$trans_{c\to n} : \textbf{Component} \times \mathcal{P}(\textbf{Grouping} \times \mathbb{N}) \mapsto \textbf{Node}$$
$$trans_{c\to n}\big(s_p = \mathcal{S}_p\langle p, d_0, \sigma_s\rangle, G_i\big) = \mathcal{N}\langle \mathcal{S}\langle e_{dp}, \texttt{null}\rangle, [next \mapsto (d_s, \varnothing \mapsto \sigma_s(d_s))]\rangle$$
where
$$e_{pr} = \texttt{let } (x_s, x_\Gamma) = \texttt{state in}$$
$$\texttt{let } x'_s, x_r = \texttt{call}(next, x_s, \varnothing) \texttt{ in}$$
$$\texttt{let } x'_\Gamma, x_{\bar{i}} = emit_\Gamma(x_\Gamma, x_r) \texttt{ in}$$
$$\texttt{send}(\texttt{self}, \texttt{null}); \texttt{emit}((x_r, x_{\bar{i}})); (x'_s, x'_\Gamma)$$
$$e_{dp} = \overline{\texttt{send}\big(\texttt{spawn}\big(\big(d_0, trans_{c\to\Gamma}\big(s_p, G_i\big)\big), e_{pr}\big), \texttt{null}\big)}^{\,p}$$
$$trans_{c\to n}(b_o = \mathcal{B}_o\langle p, d_0, \sigma_b\rangle, G_i) = \mathcal{N}\langle \mathcal{S}\langle e_{dp}, e_{dl}\rangle, [react \mapsto (d_s, d_a \mapsto \sigma_b(d_s, d_a))]\rangle$$
where
$$e_{pr} = \texttt{let } (x_s, x_\Gamma) = \texttt{state in}$$
$$\texttt{let } x'_s, x_r = \texttt{call}(react, x_s, \texttt{msg}) \texttt{ in}$$
$$\texttt{let } x'_\Gamma, x_{\bar{i}} = emit_\Gamma(x_\Gamma, x_r) \texttt{ in}$$
$$\texttt{emit}((x_r, x_{\bar{i}})); (x'_s, x'_\Gamma)$$
$$e_{dp} = \overline{\texttt{spawn}\big(\big(d_0, trans_{c\to\Gamma}(b_o, G_i)\big), e_{pr}\big)}^{\,p}$$
$$e_{dl} = \texttt{let } (x_d, x_{\bar{i}}) = \texttt{data in}$$
$$\overline{\texttt{send}}\big(\texttt{deployment}[x_{\bar{i}}\big(trans_{b_o\to\overline{g}}(b_o, G_i)[\texttt{port}]\big)], x_d\big)$$

$$trans_{c\to\Gamma} : \textbf{Component} \times \mathcal{P}(\textbf{Grouping} \times \mathbb{N}) \mapsto \mathcal{P}(\textbf{Grouping} \times \mathbb{D})$$
$$trans_{c\to\Gamma}(c, G_i) = \{(g, d) \mid (g = \mathcal{G}\langle c, \_, d, \_\rangle, \_) \in G_i\}$$

$$emit_\Gamma : \mathcal{P}(\textbf{Grouping} \times \mathbb{D}) \times \mathbb{D} \mapsto \mathcal{P}(\textbf{Grouping} \times \mathbb{D}) \times \mathcal{P}(\textbf{Grouping} \times \mathbb{N}^*)$$
$$emit_\Gamma(\Gamma, d) = \big(\{(g, d_s) \mid (g, (d_s, \_)) \in X\}, \{\big(g, \overline{i}\big) \mid \big(g, \big(\_, \overline{i}\big)\big) \in X\}\big)$$
where
$$X = \big\{\big(g, \sigma_g(d_s, d, p)\big) \mid \big(g = \mathcal{G}\langle \_, \mathcal{B}_o\langle p, \_, \_\rangle, \_, \sigma_g\rangle, d_s\big) \in \Gamma\big\}$$

$$trans_{b_o\to\overline{g}} : \textbf{Component} \times \mathcal{P}(\textbf{Grouping} \times \mathbb{N}) \mapsto \textbf{Grouping}^*$$
$$trans_{b_o\to\overline{g}}(b_o, G_i) = \overline{g}$$
where
$$\forall \{(g, i) \mid (g = \mathcal{G}\langle \_, b_o, \_, \_\rangle, i) \in G_i\} : \overline{g}[i] = g$$

B. An Elixir Primer

This appendix provides a brief primer on the Elixir language. Its goal is not to provide readers with a complete overview of the language, but to provide the information required to understand the code present in this dissertation. Readers interested in learning more about Elixir are referred to its official website.[1]

Elixir is a dynamically typed, functional programming language. It combines a Ruby-inspired syntax with the capabilities of BEAM, Erlang's virtual machine. Elixir inherits many of the benefits of Erlang such as scalability, fault-tolerance and distributed execution. On top of these benefits, it aims to provide a friendly syntax, unified tooling and extensibility through the use of macros.

Some of the example listings in this examples contain lines starting with `iex>`. These show interactions with `iex`, Elixir's interactive shell, (i.e. its REPL).

Basic Types

The following of Elixir's basic data types are relevant for this dissertation:

Integers such as `8121991` or `-1305` represent integers of an arbitrary size.

Atoms are constants whose value is their own name. They correspond to what is known as a "symbol" in some other languages. Atoms are prefixed with a colon, e.g. `:atom`.

Strings are "binaries", i.e. sequences of bytes, which contain text encoded as UTF-8. As in many other languages, strings are denoted using quotes: `"I am a string."`.

Lists are used to store a variable amount of (heterogeneous) data. Lists are single-linked lists, made up from a sequence of *pairs*. They are denoted

[1] `https://elixir-lang.org/` (visited on 11/03/2026)

using square brackets: the empty list is denoted as `[]`, while a list with the numbers from 1 to 3 is denoted as `[1, 2, 3]`.

Tuples are constant-sized, heterogeneous containers. They are created through the use of `{}`, e.g. `{:a, "tuple"}`. Tuples can be accessed by index in constant time.

Variables and Pattern Matching

Variables can be bound through the use of `=`, the *match* operator.

```
iex> x = 5
5
iex> x
5
```

As its name implies, the match operator can be used to match values, potentially destructuring them:

```
iex> {x, _} = {"hello", :world}
{"hello", :world}
iex> x
"hello"
```

Modules & Functions

Elixir applications are defined by several modules, which contain functions. Modules are defined using *defmodule*, functions using *def*.

```
defmodule Math do
  def fac(n) do
    if n == 0 do
      1
    else
      n * fac(n - 1)
    end
  end
end
```

Functions definitions can pattern match on their arguments:

```
defmodule Math do
  def fac(0), do: 1
  def fac(n), do: n * fac(n - 1)
end
```

Pattern matching on arguments is syntactic sugar for using the `case` expression inside the body of the function:

```
defmodule Math do
  def fac(arg) do
    case arg do
      0 -> 1
      n -> n * fac(n - 1)
    end
  end
end
```

Calling a function outside of the module where it is defined requires it to be prefixed with the name of its defining module:

```
iex> Math.fac(5)
120
```

This is not needed when calling primitives, functions defined in the same module, or functions whose modules have been imported.

```
iex> import Math
Math
iex> fac(5)
120
iex> length([1, 2]) # primitive
2
```

Elixir functions are first-class citizens. However, functions live in a separate namespace from values. This results in some oddities.

First, functions can be *captured* to turn them into a value. This is done by passing the name, arity, and module of the function to the capture operator, `&`.

```
iex> fac = &Math.fac/1
&Math.fac/1
```

Second, to avoid ambiguity, calling a value function uses a different syntax from calling a defined function:

```
iex> fac.(5)
120
```

Anonymous functions can be created by using the `fn` notation:

```
iex> doubled_fac = fn x -> 2 * Math.fac(x) end
#Function<42.113135111/1 in :erl_eval.expr/6>
iex> doubled_fac.(5)
240
```

Anonymous functions can capture values from their environment:

```
iex> fac = &Math.fac/1
&Math.fac/1
iex> x = 5
5
iex> x_fac = fn n -> x * fac.(n) end
#Function<42.113135111/1 in :erl_eval.expr/6>
iex> x_fac.(5)
600
```

Keyword Lists, Maps & Structs

Elixir has several ways to represent collections of key-value pairs. The most basic of these is the *keyword list*, which is a list of `{key, value}` tuples, where `key` is always an atom. Elixir offers syntactic sugar for the creation of these keyword lists: writing `[a: 1, b: 2]` is equivalent to writing `[{:a, 1}, {:b, 2}]`. A value can be fetched from a keyword list in linear time by indexing it:

```
iex> [a: 1, b: 2][:a]
1
```

Maps only enable one value to be stored for each key and can be accessed in logarithmic time. They are created with the `%{}` syntax, e.g. `%{:key => "value"}`. Values for keys can be accessed through the use of `[]`, like keyword lists. When atoms are used as keys, `%{key: "value"}` can be used as a shorthand for the creation of a map; additionally, the dot notation can be used to access the map:

```
iex> %{a: 1, b: 2}[:a]
1
iex> %{a: 1, b: 2}.a
1
```

Structs are a special type of maps wich are used to represent structured data. The keys of each struct are known upfront, as they are defined by the programmer.

```
defmodule User do
  defstruct [:name, streets: 0]
end
```

Structs are created similar to maps:

```
iex> %User{name: "Mathijs"}
%User{name: "Mathijs", streets: 0}
```

Values in structs can only be accessed through the use of the dot notation:

```
iex> me = %User{name: "Mathijs"}.name
"Mathijs"
```

Elixir provides a special notation for updating an existing key in a map or struct.

```
iex> me = %User{name: "Mathijs"}
%User{name: "Mathijs", streets: 0}
iex> %{me | streets: 7870}
%User{name: "Mathijs", streets: 7870}
```

Syntactic Sugar

Keyword Arguments

Functions in Elixir (and Erlang) have a fixed arity and do not accept optional arguments, nor do they accept keyword arguments. To work around this, Elixir functions often accept a keyword list. When this keyword list is the final argument of a function, the surrounding brackets may be omitted.

```
defmodule Options do
  def fun(_mandatory, keywords) do
    keywords
  end
end
```

```
iex> Options.fun("ignored", key1: 1, key2: "another value")
[key1: 1, key2: "another value"]
```

Pipe Operator

Elixir provides the *pipe* operator, `|>`, which uses the result of the expression before the operator as the first argument of the expression after the operator. This is commonly used to write data transformation pipelines:

```
iex> [1, [2], 3] |> List.flatten() |> Enum.map(fn x -> x * 2 end)
[2, 4, 6]
```

This is equivalent to:

```
iex> Enum.map(List.flatten([1, [2], 3]), fn x -> x * 2 end)
[2, 4, 6]
```

Sigils

Elixir provides *sigils* as a way to extend the language with custom textual representations. Sigils have a name which consists of a single alphabetic character; when used, this character is preceded by a tilde (~), and succeeded by the argument to the sigil, wrapped in a pair of delimiters.

As an example, writing `~r"foo"` is syntactic sugar for calling `sigil_r("foo")`, which creates a regular expression which matches the word "foo". In Section 6.2.4, we discuss the `~f` sigil which is used to access named fields of the callback state.

Meta-programming

Elixir offers strong support for meta-programming by enabling developers to write macros, which operate on abstract syntax trees (ASTs). When a macro is called on a set of arguments, these arguments are not evaluated immediately; instead, the macro receives the AST of each argument. In turn, the macro must return a new AST which is evaluated by Elixir later.

Through the use of these macros, developers can extend Elixir with any construct that can be expressed in its syntax. Skitter.ex uses these macros heavily to implement its various DSLs.

Function Reference

The table below provides an overview of the (Elixir) functions used inside the listings in this dissertation. Note that this overview does not include the primitives introduced by Skitter.ex.

Function	Description
`rem(dividend, divisor)`	modulo
`tuple_size(tuple)`	obtain the size of `tuple`
`elem(tuple, index)`	get the element at `index` in `tuple`
`put_elem(tuple, index, value)`	store `value` at `index` in `tuple`
`Tuple.to_list(tuple)`	convert `tuple` to a list

Continued on next page

(Continued)

`Enum.map(list, fun)`	map `fun` over `list`, returning a new list with the results
`Enum.each(list, fun)`	apply `fun` to each element in `list`, ignore the results
`Enum.reduce(list, acc, fun)`	accumulate over `list` with initial accumulator `acc`
`List.to_tuple(list)`	convert `list` to a tuple
`List.duplicate(elem, n)`	make a list containing `n` copies of `elem`
`Map.get(map, key, default)`	fetch value of `key`, return `default` if `key` is not present in `map`
`Map.get_lazy(map, key, fun)`	like `Map.get`, but lazily calculate default
`Map.put(map, key, value)`	store `value` in `map` for `key`
`Map.new()`	create an empty map
`Map.new(lst)`	create a map from a list of key-value pairs
`Map.values(map)`	get all the values stored in `map`
`String.split(string)`	split `string` where whitespace occurs
`Keyword.put(list, key, value)`	like `Map.put`, but for keyword lists
`Murmur.hash_x86_32(data)`	hash `data`
`Murmur.hash_x86_32(data, seed)`	hash `data`, randomised with `seed`

C. Additional Code Listings

```
defstrategy Split do
  defhook deploy(_) do
    buckets =
      Remote.on_all_workers(fn -> local_worker(Map.new(), :bucket) end)
      |> Enum.map(fn {_, worker} -> worker end)
      |> List.to_tuple()

    forwarder_state = strategy().forwarder_state(context(), buckets)

    forwarders =
      Remote.on_all_workers(fn ->
        local_worker(forwarder_state, :forwarder)
      end)
      |> Map.new()

    mergers =
      Remote.on_all_workers(fn -> local_worker(Map.new(), :merger) end)
      |> Enum.map(fn {_, worker} -> worker end)
      |> List.to_tuple()

    {forwarders, buckets, mergers}
  end

  defhook deliver(data) do
    {forwarders, _, _} = deployment()
    send(forwarders[Remote.self()], data)
  end

  defhook process(data, state, :forwarder) do
    {_, buckets, _} = deployment()
    key = call(:key, args: [data]).result
    strategy().forward(context(), data, buckets, key, state)
  end

  defhook process(data, state_map, :bucket) do
    key = call(:key, args: [data]).result
    state = Map.get(state_map, key, initial_state())
    res = call(:react, state: state, args: [data])
```

```
39
40      {_, _, mergers} = deployment()
41      idx = rem(Murmur.hash_x86_128(key), tuple_size(mergers))
42      send(elem(mergers, idx), {key, self(), res.state})
43
44      Map.put(state_map, key, res.state)
45    end
46
47    defhook process({key, bucket, state}, state_map, :merger) do
48      state_map = Map.put(state_map, {key, bucket}, state)
49
50      merged_state =
51        state_map[key]
52        |> Map.values()
53        |> Enum.reduce(fn state, merged ->
54          call(:merge, state: merged, args: [key, state]).state
55        end)
56
57      emit(call(:merge, state: merged_state, args: [key, initial_state()]).emit)
58      state_map
59    end
60  end
```

Listing C.1: Unabbreviated version of Listings 6.7 and 7.3.

```
1   defoperation Rate, in: [sales, clicks], out: rate, strategy: KeyedState do
2     state_struct clicks: 0, sales: 0
3
4     defcb key(data) do
5       data.ad_id
6     end
7
8     defcb react(data) do
9       case port_of(data) do
10        :sales -> sales <~ ~f{sales} + 1
11        :clicks -> clicks <~ ~f{clicks} + 1
12      end
13
14      {data.ad_id, ~f{sales} / ~f{clicks}} ~> rate
15    end
16
17    defcb merge(key, other) do
18      sales <~ sales + other.sales
19      clicks <~ clicks + other.clicks
20
21      {key, ~f{sales} / ~f{clicks}} ~> rate
22    end
23  end
```

Listing C.2: `Rate` operation with support for merging split keys.

```
defoperation KeyBy, in: _, out: _ do
  defcb conf([key_fn]), do: key_fn

  defcb key(value) do
    key_fn = config()
    key_fn.(value)
  end
end
```

Listing C.3: Definition of the `KeyBy` operation described in Section 7.4.

```
defoperation Reduce, in: _, out: _ do
  defcb conf(args), do: args

  defcb initial_state do
    [_, initial] = config()
    initial
  end

  defcb react(val) do
    [red_fn, _] = config()
    {new_state, emit} = red_fn.(val, state())
    state <~ new_state
    emit ~> _
  end
end
```

Listing C.4: Definition of the `Reduce` operation described in Section 7.4.

```
defstrategy StructuredStreams.Reduce do
  defhook deploy(args) do
    config = call_if_exists(:conf, args: [args]).result

    mappers =
      Remote.on_all_workers(fn -> local_worker(nil, :map) end))
      |> Map.new()

    reducers =
      Remote.on_all_workers(fn -> local_worker({Map.new(), 0}, :reduce) end)
      |> Enum.map(fn {_node, worker} -> worker end)
      |> List.to_tuple()

    {mappers, reducers, config}
  end

  defhook deliver(keyed_batch) do
    {mappers, _reducers, _config} = deployment()
    send(mappers[Remote.self()], {:process_keyed_batch, keyed_batch})
  end

  defhook process({:process_batch, batch}, _, :map) do
```

```
23      {_mappers, reducers, config} = deployment()
24
25      partial_results = Enum.reduce(batch, Map.new(), fn {key, el}, acc ->
26        current = Map.get_lazy(acc, key, fn -> initial_state(config) end)
27        res = call(:react, state: current, config: config, args: [el])
28        Map.put(acc, key, res.state)
29      end)
30
31      Enum.each(partial_results, fn {key, aggregated} ->
32        idx = rem(Murmur.hash_x86_128(key), tuple_size(reducers))
33        reducer_pid = elem(reducers, idx)
34        send(reducer_pid, {:partial, key, aggregated})
35      end)
36
37      reducers |> Tuple.to_list() |> Enum.each(&send(&1, :mapper_done)
38
39      nil
40    end
41
42    defhook process(:mapper_done, {state, mappers_done}, :reduce) do
43      {mappers, _reducers, _config} = deployment()
44
45      if mappers_done + 1 == map_size(mappers) do
46        emit(to_port(0, state))
47        {Map.new(), 0}
48      else
49        {state, mappers_done + 1}
50      end
51    end
52
53    defhook process({:partial, key, partial}, {state, done_count}, :reduce) do
54      {_mappers, _reducers, config} = deployment()
55      current = Map.get_lazy(state, key, fn -> initial_state(config) end)
56      res = call(:react, state: current, config: config, args: [part_val])
57      new_state = Map.put(state, key, res.state)
58      {new_state, done_count}
59    end
60  end
```

Listing C.5: Full definition of Structured Streaming's `Reduce` Strategy.

References

Adamic, Lada A. and Bernardo A. Huberman (2002). "Zipf's law and the Internet". In: *Glottometrics* 3, pp. 143–150. URL: https://www.ram-verlag.eu/wp-content/uploads/2018/08/g3zeit.pdf (visited on 21/10/2025).

Agha, Gul A. (1985). *Actors: a Model of Concurrent Computation in Distributed Systems*. MIT Press. ISBN: 9780262010924. URL: https://apps.dtic.mil/sti/tr/pdf/ADA157917.pdf (visited on 13/03/2026).

Akidau, Tyler, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt and Sam Whittle (2015). "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing". In: *Proceedings of the Very Large Data Bases Endowment* 8.12, pp. 1792–1803. DOI: 10.14778/2824032.2824076.

Ameryckx, Tim (2025). "Failure Handling in Open Distributed Stream Processing Frameworks". Master's thesis. Brussels, Belgium: Vrije Universiteit Brussel.

Arapakis, Ioannis, Souneil Park and Martin Pielot (2021). "Impact of Response Latency on User Behaviour in Mobile Web Search". In: *Proceedings of the ACM SIGIR Conference on Human Information Interaction and Retrieval* (14th–19th Mar. 2021). Canberra ACT, Australia: ACM, pp. 279–283. DOI: 10.1145/3406522.3446038.

Armbrust, Michael, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica and Matei Zaharia (2018). "Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data* (10th–15th June 2018). Houston, TX, USA: ACM, pp. 601–613. DOI: 10.1145/3183713.3190664.

Armbrust, Michael, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi and Matei Zaharia (2015). "Spark SQL: Relational Data Processing in Spark". In: *Proceedings of the ACM SIGMOD International Conference on Manage-*

ment of Data (31st May–4th June 2015). Melbourne, Victoria, Australia: ACM, pp. 1383–1394. DOI: 10.1145/2723372.2742797.

Aslam, Adeel, Hanhua Chen and Hai Jin (2021). "Pre-filtering based summarization for data partitioning in distributed stream processing". In: *Concurrency and Computation: Practice and Experience* 33.20. DOI: 10.1002/CPE.6338.

Assunção, Marcos Dias de, Alexandre Da Silva Veith and Rajkumar Buyya (2018). "Distributed data stream processing and edge computing: A survey on resource elasticity and future directions". In: *Journal of Network and Computer Applications* 103, pp. 1–17. DOI: 10.1016/J.JNCA.2017.12.001.

Begoli, Edmon, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior and Daniel Lemire (2018). "Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data* (10th–15th June 2018). Houston, TX, USA: ACM, pp. 221–230. DOI: 10.1145/3183713.3190662.

Berinde, Radu, Piotr Indyk, Graham Cormode and Martin J. Strauss (2010). "Space-optimal heavy hitters with strong error bounds". In: *ACM Transactions on Database Systems* 35.4, 26:1–26:28. DOI: 10.1145/1862919.1862923.

Berlinska, Joanna and Maciej Drozdowski (2018). "Comparing load-balancing algorithms for MapReduce under Zipfian data skews". In: *Parallel Computing* 72, pp. 14–28. DOI: 10.1016/J.PARCO.2017.12.003.

Bracha, Gilad and David M. Ungar (2004). "Mirrors: design principles for meta-level facilities of object-oriented programming languages". In: *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (24th–28th Oct. 2004). Vancouver, BC, Canada: ACM, pp. 331–344. DOI: 10.1145/1028976.1029004.

Carbone, Paris, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter and Kostas Tzoumas (2017). "State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing". In: *Proceedings of the Very Large Data Bases Endowment* 10.12, pp. 1718–1729. DOI: 10.14778/3137765.3137777.

Carbone, Paris, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi and Kostas Tzoumas (2015). "Apache Flink™: Stream and Batch Processing in a Single Engine". In: *IEEE Data Engineering Bulletin* 38.4, pp. 28–38.

URL: http://sites.computer.org/debull/A15dec/p28.pdf (visited on 13/06/2025).

Chandy, K. Mani and Leslie Lamport (1985). "Distributed Snapshots: Determining Global States of Distributed Systems". In: *ACM Transactions on Computer Systems* 3.1, pp. 63–75. DOI: 10.1145/214451.214456.

Chaudhary, Salaar Ahmed (2025). "Capturing the Essential Complexity of Distribution Strategies in Stream Processing Frameworks. A Language-Based Approach". Master's thesis. Brussels, Belgium: Vrije Universiteit Brussel.

Chen, Hanhua, Fan Zhang and Hai Jin (2021). "PStream: A Popularity-Aware Differentiated Distributed Stream Processing System". In: *IEEE Transactions on Computers* 70.10, pp. 1582–1597. DOI: 10.1109/TC.2020.3019689.

Codd, Edgar F. (1972). "Relational Completeness of Data Base Sublanguages". In: *Database Systems*.

De Koster, Joeri, Tom Van Cutsem and Wolfgang De Meuter (2016). "43 years of actors: a taxonomy of actor models and their key properties". In: *Proceedings of the International Workshop on Programming Based on Actors, Agents, and Decentralized Control* (30th Oct. 2016). Amsterdam, the Netherlands: ACM, pp. 31–40. DOI: 10.1145/3001886.3001890.

Elnozahy, E. N., Lorenzo Alvisi, Yi-Min Wang and David B. Johnson (2002). "A survey of rollback-recovery protocols in message-passing systems". In: *ACM Computing Surveys* 34.3, pp. 375–408. DOI: 10.1145/568522.568525.

Elseidy, Mohammed, Abdallah Elguindy, Aleksandar Vitorovic and Christoph Koch (2014). "Scalable and Adaptive Online Joins". In: *Proceedings of the Very Large Data Bases Endowment* 7.6, pp. 441–452. DOI: 10.14778/2732279.2732281.

Fang, Junhua, Rong Zhang, Tom Z. J. Fu, Zhenjie Zhang, Aoying Zhou and Junhua Zhu (2017). "Parallel Stream Processing Against Workload Skewness and Variance". In: *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing* (26th–30th June 2017). Washington, DC, USA: ACM, pp. 15–26. DOI: 10.1145/3078597.3078613.

Fang, Junhua, Rong Zhang, Yan Zhao, Kai Zheng, Xiaofang Zhou and Aoying Zhou (2021). "A-DSP: An Adaptive Join Algorithm for Dynamic Data Stream

on Cloud System". In: *IEEE Transactions on Knowledge and Data Engineering* 33.5, pp. 1861–1876. DOI: 10.1109/TKDE.2019.2947055.

Felleisen, Matthias and Robert Hieb (1992). "The Revised Report on the Syntactic Theories of Sequential Control and State". In: *Theoretical Computer Science* 103.2, pp. 235–271. DOI: 10.1016/0304-3975(92)90014-7.

Fu, Zhongming, Zhuo Tang, Li Yang, Kenli Li and Keqin Li (2020). "ImRP: A Predictive Partition Method for Data Skew Alleviation in Spark Streaming Environment". In: *Parallel Computing* 100, p. 102699. DOI: 10.1016/j.parco.2020.102699.

Gedik, Bugra (2014). "Partitioning functions for stateful data parallelism in stream processing". In: *International Journal on Very Large Data Bases* 23.4, pp. 517–539. DOI: 10.1007/S00778-013-0335-9.

Hirzel, Martin, Robert Soulé, Scott Schneider, Buğra Gedik and Robert Grimm (2014). "A catalog of stream processing optimizations". In: *ACM Computing Surveys* 46.4, 46:1–46:34. DOI: 10.1145/2528412.

Hwang, Jeong-Hyon, Magdalena Balazinska, Alex Rasin, Ugur Çetintemel, Michael Stonebraker and Stanley B. Zdonik (2005). "High-Availability Algorithms for Distributed Stream Processing". In: *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan* (5th–8th Apr. 2005). Tokyo, Japan: IEEE Computer Society, pp. 779–790. DOI: 10.1109/ICDE.2005.72.

Jafarpour, Hojjat and Rohan Desai (2019). "KSQL: Streaming SQL Engine for Apache Kafka". In: *Proceedings of the International Conference on Extending Database Technology* (26th–29th Mar. 2019). Lisbon, Portugal: OpenProceedings.org, pp. 524–533. DOI: 10.5441/002/EDBT.2019.48.

Ji, Hangxu, Su Jiang, Yuhai Zhao, Gang Wu, Guoren Wang and George Y. Yuan (2023). "BS-Join: A novel and efficient mixed batch-stream join method for spatiotemporal data management in Flink". In: *Future Generation Computer Systems* 141, pp. 67–80. DOI: 10.1016/J.FUTURE.2022.11.016.

Karimov, Jeyhun, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen and Volker Markl (2018). "Benchmarking Distributed Stream Data Processing Systems". In: *IEEE International Conference on Data Engineering* (16th–19th Apr. 2018). Paris, France: IEEE Computer Society, pp. 1507–1518. DOI: 10.1109/ICDE.2018.00169.

Katsipoulakis, Nikos R., Alexandros Labrinidis and Panos K. Chrysanthis (2017). "A holistic view of stream partitioning costs". In: *Proceedings of the Very Large Data Bases Endowment* 10.11, pp. 1286–1297. DOI: 10.14778/3137628.3137639.

Kreps, Jay, Neha Narkhede, Jun Rao et al. (2011). "Kafka: A distributed messaging system for log processing". In: *Proceedings of the International Workshop on Networking Meets Databases* (12th–14th June 2011). Athens, Greece. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2017/09/Kafka.pdf (visited on 21/10/2025).

Li, Wenxin, Duowen Liu, Kai Chen, Keqiu Li and Heng Qi (2021). "Hone: Mitigating Stragglers in Distributed Stream Processing With Tuple Scheduling". In: *IEEE Transactions on Parallel and Distributed Systems* 32.8, pp. 2021–2034. DOI: 10.1109/TPDS.2021.3051059.

Lin, Qian, Beng Chin Ooi, Zhengkui Wang and Cui Yu (2015). "Scalable Distributed Stream Join Processing". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data* (31st May–4th June 2015). Melbourne, Victoria, Australia: ACM, pp. 811–825. DOI: 10.1145/2723372.2746485.

Liu, Gang, Zeting Wang, Amelie Chi Zhou and Rui Mao (2024). "Adaptive key partitioning in distributed stream processing". In: *CCF Transactions on High Performance Computing* 6.2, pp. 164–178. DOI: 10.1007/S42514-023-00179-3.

Liu, Guipeng, Xiaomin Zhu, Ji Wang, Deke Guo, Weidong Bao and Hui Guo (2018). "SP-Partitioner: A novel partition method to handle intermediate data skew in spark streaming". In: *Future Generation Computer Systems* 86, pp. 1054–1063. DOI: 10.1016/J.FUTURE.2017.07.014.

Liu, Xunyun and Rajkumar Buyya (2021). "Resource Management and Scheduling in Distributed Stream Processing Systems: A Taxonomy, Review, and Future Directions". In: *ACM Computing Surveys* 53.3, 50:1–50:41. DOI: 10.1145/3355399.

Liu, Xunyun, Aaron Harwood, Shanika Karunasekera, Benjamin I. P. Rubinstein and Rajkumar Buyya (2017). "E-Storm: Replication-Based State Management in Distributed Stream Processing Systems". In: *International Conference on Parallel Processing, Bristol, United Kingdom, August 14-17, 2017* (14th–

17th Aug. 2017). Bristol, United Kingdom: IEEE Computer Society, pp. 571–580. DOI: 10.1109/ICPP.2017.66.

Manku, Gurmeet Singh and Rajeev Motwani (2002). "Approximate Frequency Counts over Data Streams". In: *Proceedings of the International Conference on Very Large Data Bases* (20th–23rd Aug. 2002). Hong Kong: Morgan Kaufmann, pp. 346–357. DOI: 10.1016/B978-155860869-6/50038-X.

Marconi, Francesco, Marcello M. Bersani, Madalina Erascu and Matteo Rossi (2016). "Towards the Formal Verification of Data-Intensive Applications Through Metric Temporal Logic". In: *Proceedings of the International Conference on Formal Engineering Methods* (14th–18th Nov. 2016). Vol. 10009. Tokyo, Japan, pp. 193–209. DOI: 10.1007/978-3-319-47846-3_13.

Marconi, Francesco, Marcello M. Bersani and Matteo Rossi (2017). "Formal verification of storm topologies through D-VerT". In: *Proceedings of the Symposium on Applied Computing* (3rd–7th Apr. 2017). Marrakech, Morocco: ACM, pp. 1168–1174. DOI: 10.1145/3019612.3019769.

Metwally, Ahmed, Divyakant Agrawal and Amr El Abbadi (2005). "Efficient Computation of Frequent and Top-k Elements in Data Streams". In: *Proceedings of the International Conference on Database Theory* (5th–7th Jan. 2005). Edinburgh, UK: Springer, pp. 398–412. DOI: 10.1007/978-3-540-30570-5_27.

Nasir, Muhammad Anis Uddin, Gianmarco De Francisci Morales, David García-Soriano, Nicolas Kourtellis and Marco Serafini (2015). "The Power of Both Choices: Practical Load Balancing for Distributed Stream Processing Engines". In: *IEEE International Conference on Data Engineering* (13th–17th Apr. 2015). Seoul, South Korea: IEEE Computer Society, pp. 137–148. DOI: 10.1109/ICDE.2015.7113279.

Nasir, Muhammad Anis Uddin, Gianmarco De Francisci Morales, Nicolas Kourtellis and Marco Serafini (2016). "When two choices are not enough: Balancing at scale in Distributed Stream Processing". In: *IEEE International Conference on Data Engineering* (16th–20th May 2016). Helsinki, Finland: IEEE Computer Society, pp. 589–600. DOI: 10.1109/ICDE.2016.7498273.

Noghabi, Shadi A., Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta and Roy H. Campbell (2017). "Samza: stateful scalable stream processing at LinkedIn". In: *Proceedings of the Very Large Data Bases Endowment* 10.12, pp. 1634–1645. DOI: 10.14778/3137765.3137770.

Olston, Christopher, Benjamin C. Reed, Utkarsh Srivastava, Ravi Kumar and Andrew Tomkins (2008). "Pig latin: a not-so-foreign language for data processing". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data* (10th–12th June 2008). Vancouver, BC, Canada: IEEE Computer Society, pp. 1099–1110. DOI: 10.1145/1376616.1376726.

Pacaci, Anil and M. Tamer Özsu (2018). "Distribution-Aware Stream Partitioning for Distributed Stream Processing Systems". In: *Proceedings of the ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond* (15th June 2018). Houston, TX, USA: ACM, 6:1–6:10. DOI: 10.1145/3206333.3206338.

Peng, Boyang, Mohammad Hosseini, Zhihao Hong, Reza Farivar and Roy H. Campbell (2015). "R-Storm: Resource-Aware Scheduling in Storm". In: *Proceedings of the 16th Annual Middleware Conference* (7th–11th Dec. 2015). Vancouver, BC, Canada: ACM, pp. 149–161. DOI: 10.1145/2814576.2814808.

Qiu, Yuan, Serafeim Papadias and Ke Yi (2019). "Streaming HyperCube: A Massively Parallel Stream Join Algorithm". In: *International Conference on Extending Database Technology* (26th–29th Mar. 2019). Lisbon, Portugal: OpenProceedings.org, pp. 642–645. DOI: 10.5441/002/EDBT.2019.76.

Ragan-Kelley, Jonathan, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman P. Amarasinghe and Frédo Durand (2018). "Halide: decoupling algorithms from schedules for high-performance image processing". In: *Communications of the ACM* 61.1, pp. 106–115. DOI: 10.1145/3150211.

Requeno, José Ignacio, José Merseguer and Simona Bernardi (2017). "Performance Analysis of Apache Storm Applications Using Stochastic Petri Nets". In: *Proceedings of the International Conference on Information Reuse and Integration* (4th–6th Aug. 2017). San Diego, CA, USA: IEEE Computer Society, pp. 411–418. DOI: 10.1109/IRI.2017.64.

Rivetti, Nicoló, Leonardo Querzoni, Emmanuelle Anceaume, Yann Busnel and Bruno Sericola (2015). "Efficient key grouping for near-optimal load balancing in stream processing systems". In: *Proceedings of the ACM International Conference on Distributed Event-Based Systems* (29th June–3rd July 2015). Oslo, Norway: ACM, pp. 80–91. DOI: 10.1145/2675743.2771827.

Saey, Mathijs (2025). *Accepted Artifact for "Skitter: A Distributed Stream Processing Framework with Pluggable Distribution Strategies"*. Version 1.0. DOI: 10.5281/zenodo.14714125.

Schroeder, Bianca and Garth A. Gibson (2010). "A Large-Scale Study of Failures in High-Performance Computing Systems". In: *IEEE Transactions on Dependable and Secure Computing* 7.4, pp. 337–351. DOI: 10.1109/TDSC.2009.4.

Shah, Mehul A., Joseph M. Hellerstein, Sirish Chandrasekaran and Michael J. Franklin (2003). "Flux: An Adaptive Partitioning Operator for Continuous Query Systems". In: *IEEE International Conference on Data Engineering* (5th–8th Mar. 2003). Bangalore, India: IEEE, pp. 25–36. DOI: 10.1109/ICDE.2003.1260779.

Shukla, Anshu, Shilpa Chaturvedi and Yogesh Simmhan (2017). "RIoTBench: An IoT benchmark for distributed stream processing systems". In: *Concurrency and Computation: Practice and Experience* 29.21. DOI: 10.1002/CPE.4257.

Sun, Dawei, Minghui Wu, Zhihong Yang, Atul Sajjanhar and Rajkumar Buyya (2023). "A two-tier coordinated load balancing strategy over skewed data streams". In: *Journal of Supercomputing* 79.18, pp. 21028–21056. DOI: 10.1007/S11227-023-05473-Z.

Toshniwal, Ankit, Siddarth Taneja, Amit Shukla, Karthikeyan Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal and Dmitriy V. Ryaboy (2014). "Storm@twitter". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data* (22nd–27th June 2014). Snowbird, UT, USA: ACM, pp. 147–156. DOI: 10.1145/2588555.2595641.

Tran, Geoffrey Phi C., John Paul Walters and Stephen P. Crago (2017). "Dynamically Improving Resiliency to Timing Errors for Stream Processing Workloads". In: *International Conference on Parallel and Distributed Computing, Applications and Technologies* (18th–20th Dec. 2017). Taipei, Taiwan: IEEE Computer Society, pp. 469–476. DOI: 10.1109/PDCAT.2017.00080.

Tran, Geoffrey Phi C., John Paul Walters and Stephen P. Crago (2018). "Reducing Tail Latencies While Improving Resiliency to Timing Errors for Stream Processing Workloads". In: *IEEE International Conference on Services Computing* (2nd–7th July 2018). San Francisco, CA, USA: IEEE, pp. 278–281. DOI: 10.1109/SCC.2018.00048.

van Dongen, Giselle and Dirk Van den Poel (2020). "Evaluation of Stream Processing Frameworks". In: *IEEE Transactions on Parallel and Distributed Systems* 31.8, pp. 1845–1858. DOI: 10.1109/TPDS.2020.2978480.

van Dongen, Giselle and Dirk Van den Poel (2021). “A Performance Analysis of Fault Recovery in Stream Processing Frameworks”. In: *IEEE Access* 9, pp. 93745–93763. DOI: 10.1109/ACCESS.2021.3093208.

Vogel, Adriano, Sören Henning, Esteban Pérez-Wohlfeil, Otmar Ertl and Rick Rabiser (2024). “A Comprehensive Benchmarking Analysis of Fault Recovery in Stream Processing Frameworks”. In: *Proceedings of the ACM International Conference on Distributed and Event-Based Systems* (25th–28th July 2024). DEBS ’24. Villeurbanne, France: ACM, pp. 171–182. DOI: 10.1145/3629104.3666040.

Wang, Qihang, Decheng Zuo, Zhan Zhang, Siyuan Chen and Tianming Liu (2023). “An adaptive non-migrating load-balanced distributed stream window join system”. In: *The Journal of Supercomputing* 79.8, pp. 8236–8264. DOI: 10.1007/S11227-022-04991-6.

Wang, Qihang, Decheng Zuo, Zhan Zhang, Yanjun Shu, Xin Liu and Mingxuan He (2024). “Low-Latency Adaptive Distributed Stream Join System Based on a Flexible Join Model”. In: *Proceedings of the ACM on Management of Data* 2.3, p. 150. DOI: 10.1145/3654953.

Wu, Minghui, Dawei Sun, Shang Gao, Keqin Li and Rajkumar Buyya (2025). “Ls-Stream: Lightening Stragglers in Join Operators for Skewed Data Stream Processing”. In: *IEEE Transactions on Computers* 74.8, pp. 2841–2855. DOI: 10.1109/TC.2025.3575917.

Wu, Song, Mi Liu, Shadi Ibrahim, Hai Jin, Lin Gu, Fei Chen and Zhiyi Liu (2018). “TurboStream: Towards Low-Latency Data Stream Processing”. In: *IEEE International Conference on Distributed Computing Systems* (2nd–6th July 2018). Vienna, Austria: IEEE, pp. 983–993. DOI: 10.1109/ICDCS.2018.00099.

Xu, Luna, Ali Raza Butt, Seung-Hwan Lim and Ramakrishnan Kannan (2018). “A Heterogeneity-Aware Task Scheduler for Spark”. In: *IEEE International Conference on Cluster Computing* (10th–13th Sept. 2018). Belfast, UK: IEEE Computer Society, pp. 245–256. DOI: 10.1109/CLUSTER.2018.00042.

Yu, Shuiying, Hanhua Chen and Hai Jin (2023). “Nereus: A Distributed Stream Band Join System With Adaptive Range Partitioning”. In: *IEEE Transactions on Consumer Electronics* 69.4, pp. 949–961. DOI: 10.1109/TCE.2023.3249292.

Zaharia, Matei, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker and Ion Stoica (2012).

"Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". In: *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation* (25th–27th Apr. 2012). San Jose, CA, USA: USENIX Association, pp. 15–28. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia (visited on 25/10/2025).

Zaharia, Matei, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker and Ion Stoica (2013). "Discretized Streams: Fault-Tolerant Streaming Computation at Scale". In: *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles* (3rd–6th Nov. 2013). Farmington, PA, USA: ACM, pp. 423–438. DOI: 10.1145/2517349.2522737.

Zhang, Fan, Hanhua Chen and Hai Jin (2019). "Simois: A Scalable Distributed Stream Join System with Skewed Workloads". In: *IEEE International Conference on Distributed Computing Systems* (7th–10th July 2019). Dallas, TX, USA: IEEE, pp. 176–185. DOI: 10.1109/ICDCS.2019.00026.

Zhao, Hongyan, Huibiao Zhu, Yucheng Fang and Lili Xiao (2019). "Modeling and Verifying Storm Using CSP". In: *IEEE International Symposium on High Assurance Systems Engineering* (3rd–5th Jan. 2019). Hangzhou, China: IEEE, pp. 192–199. DOI: 10.1109/HASE.2019.00037.

Zhou, Shunjie, Fan Zhang, Hanhua Chen, Hai Jin and Bing Bing Zhou (2019). "FastJoin: A Skewness-Aware Distributed Stream Join System". In: *IEEE International Parallel and Distributed Processing Symposium* (20th–24th May 2019). Rio de Janeiro, Brazil: IEEE, pp. 1042–1052. DOI: 10.1109/IPDPS.2019.00111.

Glossary & Abbreviations

ABS, Asynchronous Barrier Snapshotting A checkpointing technique used to make DSPAs resilient to partial failure. Discussed in Section 6.4.

Bolt Storm's abstraction to express a source. See Section 3.2.1.

Callback Skitter's abstraction for implementing data processing logic inside an operation. Discussed in Sections 4.1.2 and 6.1.2.

Cluster A set of computers connected through a fast network, treated and programmed as a single system.

Cluster Node A single computer that is part of a cluster.

DAG directed acyclic graph.

Deployment Data Skitter's abstraction for accessing the return value of the `deploy` hook. Discussed in Sections 4.1.3 and 6.1.3.

Distribution Strategy Logic that determines how an operation is distributed over a cluster.

DSL domain-specific language.

DSPA, Distributed Stream Processing Application An application which processes high-volume data streams with low latency by distributing the processing of this incoming data over a cluster.

DSPF, Distributed Stream Processing Framework A type of framework which enables the expression of DSPAs.

Grouping Storm's abstraction to represent a data dependency. See Section 3.2.1.

Hook Skitter's abstraction for implementing distribution logic inside a strategy. These hooks are called in response to predefined events. Discussed in Sections 4.1.3 and 6.1.3.

Link Skitter's abstraction for representing a data dependency between two nodes in a workflow. Discussed in Sections 4.1 and 6.1.1.

Operation A data processing step which may consume incoming data records and potentially produce new data records.

Operator A well-known high-level operation (such as `map` or `reduce`) which is parametrized with a function.

PKG, Partial Key Grouping A distribution strategy which splits the state of a keyed operation over two workers to improve parallelism. Discussed in Sections 2.2.1 and 7.1.1.

Port Skitter's abstraction to refer to a specific input or output of a data processing step. Discussed in Sections 4.1.1 and 6.1.2.

Role Skitter.ex's abstraction to distinguish between groups of workers which belong to the same distribution strategy but perform a different job. Discussed in Section 6.1.3.

Sink An operation which consumes input but does not produce any output.

Skew An uneven distribution of data over keys where most of the data is associated with a small set of keys typically leading to an unbalanced distribution of work over the cluster. Discussed in Section 2.2.1.

Source An operation that does not accept any input but produces output.

Spout Storm's abstraction to express any non-source node. See Section 3.2.1.

Task A computational entity in Storm (and fwStorm) which processes data records for a bolt, or which generates data records for a spout. This is the (featherweight) Storm terminology to denote a worker. See Section 3.2.2.

Token A fwSkitter data record wrapped with its associated meta information. See Section 6.2.3.

Topology Storm's abstraction to represent a data processing graph. See Section 3.2.1

Worker A computational entity which processes data records for an operation. For the general definition, see Section 2.2. For its use in Skitter, see Sections 4.2 and 6.1.3.

Workflow Skitter's abstraction to represent a data processing DAG. Discussed in Sections 4.1.1 and 6.1.1.