Delft University of Technology

Masters Thesis

The Design and Implementation of a Stateful Streaming Rule Language for the Maritime Sector

Author:
Thomas Smith

Supervisor:
Dr. A. Katsifodimos

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

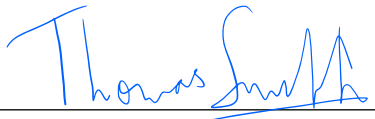
Web Information Systems Group
Software Technology

November 8, 2020

Declaration of Authorship

I, Thomas Smith, declare that this thesis titled, " The Design and Implementation of a Stateful Streaming Rule Language for the Maritime Sector " and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: 

Date: 08-11-2020

DELFT UNIVERSITY OF TECHNOLOGY

Abstract

Electrical Engineering, Mathematics and Computer Science
Software Technology

Master of Science

The Design and Implementation of a Stateful Streaming Rule Language for the Maritime Sector

by Thomas Smith

Stream processing has taken a prominent position in the software engineering industry. Many applications, from a small to large scale, embrace this paradigm to deal with the difficulties of responding to events that happen asynchronously: programmers define "operators" that perform small, independent tasks on individual events. These operators are then connected to form a (usually asyclic) network of tasks performed on one or multiple streams.

While very useful as can be witnessed by the vast amount of stream processing tools available to the industry, one problem remains difficult to tackle in this model: state. When comparing support for stateful operations in matured stream-processing systems, one will find that none of the systems agree on how state should be modeled. Some systems outright ignore state, while others provide ever different APIs to manage it.

In this thesis, we present a small but versatile model that combines traditional stream processing with stateful operations from relational algebra, providing one unified way of thinking about streams and state. To substantiate the models practicality, we present an end-to-end implementation of a real-world case study for Europe's largest shipping container port, along with a performance analysis. Moreover, we assert that this model is not just tied to one specific stream-processing solution, but can be implemented with ease on any actor-based platform.

Contents

Chapter 1

Introduction

For the past forty years, relational databases [23] have had an undeniably large impact on how data gets processed by businesses and organisations. As a paradigm, tables and relational algebra to manipulate them are ubiquitous and used in many systems at any scale.

As useful as relational databases are however, many organisations need to handle data in ways ill suited for the relational model. Specifically, processing data that arrives in real-time at high speeds and in large volumes is difficult to deal with since relational databases are mostly suited for querying and updating data at rest[32].

The need to process data in real-time without locking or mutating a global resource has spawned several systems that at their core do not follow the relational model, but rather follow a model of operating on timestamped bundles of data called events[12].

Streaming operators take events as inputs and produce events as outputs, sometimes merging or producing multiple separate streams. This paradigm draws much from early work on data flow programming and Kahn networks[26, 27, 31, 38].

While these systems excel at performing transformations on in-flight data, their lack of good query languages and incompatibility with the relational model were problematic. This gap has been filled by extensive research into methods for continuous querying of streams [13, 20]. Nowadays many have adopted a relational layer on top of the streaming model to solve this problem [19, 36, 39]. Research from the past two decades has also resulted resulted in many interesting designs of streaming systems, each with their own characteristics and lessons [1, 2, 9, 10].

In this thesis, we aim to give a better overview of how the unification of relational algebra on top of streams might be realised. To this end we have gathered a set of operations that capture the essential features of both streams and relations into a minimal language (or algebra). We substantiate the practicality and versatility of this minimal algebra by presenting a real-world use case for the maritime sector. After this we give an interpretation of our algebra into the Actor model, showing that our minimal algebra can be implemented on any actor system. Finally we present benchmarks of the entire system showing that our approach is works even for large-scale deployments on commodity hardware. The next section will introduce our case-study for the Port of Rotterdam and clarify the significance of our work for the maritime sector in general.

1.1 Case study: Port of Rotterdam

This works was undertaken in collaboration with the Port of Rotterdam. In 2018, the port serviced 29.476 seagoing vessels and 107.000 inland barges totalling 469 million tonnes of cargo. This makes it currently the largest European sea port. While the scope and scale of the operations performed on port ground is huge, the port itself employs the relatively small amount of 1200 people. The reason for this is that the

ports' role is merely to facilitate other companies in servicing vessels. Much of the land owned by the port is rented to a broad range of companies for them to perform their business activities on. Therefore, the port can obtain a competitive advantage by offering services that make it more attractive on two fronts:

- Services that make Rotterdam a more desirable destination for shipping lines will provide a direct boost in income,
- Making it more appealing for businesses to be on port terrain.

Improving on one of these two areas also benefits the other; having increased interest from large shipping lines makes it more beneficial for companies to be in Rotterdam, while having more interesting companies within the port makes it more desirable to consider Rotterdam as a destination for cargo.

1.2 Avoiding delays with early warnings

Improving the planning accuracy of vessels is one such service where the Port of Rotterdam can play an important role. Most vessels visiting Rotterdam make use of services of multiple companies during their stay in the port. Each of these parties will have their own planning for when to service which vessel, and those schedules have to line up: a vessel cannot be refuelled if it is not at its berth, and it cannot arrive at its berth in time without services from a pilot and/or tugboats. Inaccuracies in these schedules often cause cascading delays and waste of time and resources. Sadly, most of these inaccuracies are very hard or even impossible to prevent upfront, forcing businesses to react to impediments as they come up.

One very common cause for such impediments is related to weather and tide. By Dutch law, the port is mandated to only let vessels enter the port when the situation is safe to do so. To facilitate this, a protocol is in place where the port authority determines, on a case-by-case basis, whether a vessel is allowed to enter the port and manoeuvre to its destination berth. The rules that capture the different factors of this decision are called *restrictions*. This decision is made as soon as a vessel arrives near the anchor area roughly 50 kilometres away from the shore. At this moment, a ship captain usually requests for pilot services. Unless the captain has an exemption, a pilot must accompany the captain on the ship and give directions as to how the ship should sail to its destination berth. The port authority together with the pilot decide at this moment whether it is safe for the vessel to enter the port based on:

- weather data;
- tide data;
- tugboat availability;
- vessel size;
- type of cargo;
- other ships present in the vicinity.

Oftentimes, the interplay between these factors will not allow for safe entry of the vessel, causing the captain to have to wait in the anchorage for hours or even days until the situation is resolved.

For the catpain, the shipping company, the terminal that has to handle vessel cargo, and all other service-providers it would be beneficial to know in advance whether

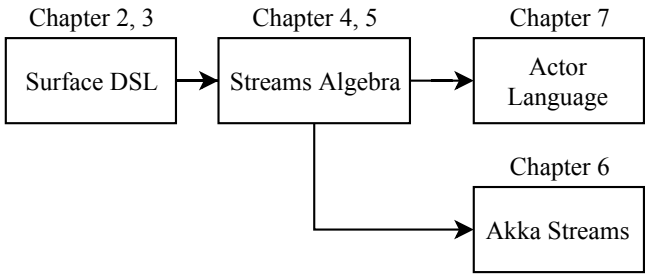


FIGURE 1.1: Architectural overview

conditions are likely to cause problems given the current schedule. Aside from potential savings due to reduced delays, minimising the waiting time of vessels also has significant environmental benefits. For Rotterdam alone, studies show that the port is responsible for 2% of all shipping emissions globally, and between 10% and 25% of emissions in the city itself [42, 43]. Not only do vessels waiting in the anchorage form a source of Nitrogen Oxide (NO_x) pollution, but these vessels also waste tonnes of fuel sailing to Rotterdam at full steam unnecessarily. In general one of the best ways for ports to reduce this pollution is to reduce vessel waiting times [42].

In this thesis we implement a system for recognising ahead of time when a vessel might not be able to enter the port due to a restriction violation. We create a Domain Specific Language (DSL) embedded in the Scala programming language to express these rules in [44]. The DSL enables software engineers to write these rules in such a way that they can be structured and maintained with ease while being intuitive to read and comprehend for domain experts without knowledge of programming. The variables of these restrictions (e.g. vessel size and wind speed) all arrive in the form of streaming data and are processed by our system in a streaming fashion. Furthermore, our restrictions DSL will be fully implemented in terms of a minimal streaming algebra.

1.3 Thesis structure

The structure of this thesis roughly reflects the architectural components of the system for our case-study. Figure 1.1 shows the abstract components of the system that will be discussed in this work.

- In chapter 2, we define the surface language used to express domain rules regarding vessel allowance. We give examples of how it might be used and touch upon the expressivity of this language, showing a correspondence to logic.
- Chapter 3 explores how one might go about implementing this DSL based on streams and shows the need for a model that combines streams with stateful relations (or tables).
- In chapter 4 we introduce a minimal algebra that combines streams and relational operators in one model.
- A translation from the surface DSL to this algebra is presented in chapter 5 along with a discussion on several key design choices and their impact on expressiveness.
- In chapter 6 we observe that our model for streams and tables relates to operators known from the reactive programming paradigm. We show that our

representation can be translated in a straightforward manner into stateful reactive programs and provide a concrete interpretation into Akka Streams, a popular library for reactive programming in Scala.

- In chapter 7, we conclude our exploration of the expressiveness of our streaming algebra by providing an interpretation into Act, a minimal actor language [5]. This translation shows that our system will be able to run on any actor system.
- Finally, the presentation of this end-to-end system would not be complete without an analysis of its performance. Chapter 8 is dedicated to this purpose. We present three optimisations commonly seen in other work, describe their implementation and study their compound effect. This analysis also includes a discussion on time-memory trade-offs and how eliminating or maintaining intermediate state affects run-times and memory usage.

Chapter 2

DSL for vessel restrictions

In this section we introduce the surface language for our case study by means of examples. As explained in section 1.2, the purpose of this language is to enable developers to express rules that characterise when it it safe for a vessel to enter the port. There are four requirements that this DSL has to fulfil:

1. It must be possible to write numerical and Boolean expressions on available data sources (e.g. wind speed and vessel size).
2. These expressions will be used in assertions to state conditions that should always hold.
3. It must be possible to write assertions that get checked conditionally, i.e. only when a specific condition is met.
4. One or more assertions can be tied to a location.

To facilitate these requirements, we have conceptually split the language into two parts: one which expresses statements and one for expressions. The part of the DSL that expresses statements deals with requirements $2, 3$ and 4, while numerical and Boolean expressions of requirement 1 are dealt with by expressions.

2.1 A note on embedded DSLs

It should be noted that any code presented in this chapter is Scala code [44]. Therefore, the DSL that we present is an *embedded* DSL. The act of embedding a DSL inside a host language has been known for quite some time and is often praised for requiring low upfront effort compared to full external DSLs [29, 35]. We therefore chose to embed our languages in Scala due to the fact that this grants us, as well as users of our system, the benefit of having the Scala compiler check our code for type and syntax errors. Since this is a language already in active use at the Port of Rotterdam, developers will be able to work with our DSL without having to get accustomed to foreign or custom tooling.

Our approach is similar to the one introduced by Elliott et al. in that we also embed an optimizing compiler [30] to produce code that runs on a well-known Scala streaming framework. To make processing and transformations easier we employ the Kiama language processing library [51]. This gives us a straightforward way of expressing necessary transformations of DSL terms based on rewrite rules similar to those seen in Stratego [54].

2.2 Statements

To get an impression of what actual rules in this DSL will look like, we will first discuss statements in depth, then move on to expressions in section 2.3.

The `require` statement

We start the discussion on statements by introducing the `require` statement. This statement accepts an expression from our expression language (explained in detail in the next section) and turns that expression in an assertion. As an example considering this rule: vessels can safely maneuver up to winds speeds of 45 knots. This can be achieved with a single line in our DSL:

Scala Listing 1 – Illustrating the `require` statement

```
1 require (wind.velocity <= 45.knots)
```

Similar to how assertions are the most basic building blocks in many unit testing frameworks, the `require` statement forms the core building block for port restrictions. Intuitively, we can think of the `require` statement as an instruction that emits a warning as soon as the condition is dissatisfied. We chose this formulation as opposed to the opposite (`warn(wind.velocity > 45.knots)`) because this lines up with existing documents on safety guidelines within the Port of Rotterdam.

Upon inspecting this example, some questions on the operational aspect of this rule may come to mind:

- When does this rule exactly emit a warning? After all, emitting the warning as soon as the wind is already above 45 knots means the warning issued too late for vessels already in the area.
- Who receives this warning? How does the system dispatch the warning to relevant parties?

To avoid emitting warnings too late, the system has to look at future predictions of the wind velocity and match those predictions with the expected time of arrival of a vessel. Since every vessel will arrive at a different time, this implies that each rule needs to be evaluated on a per-vessel basis. Thus, any generated warning is dispatched for a specific vessel. In the case of listing 1 this means that we will emit a warning for any vessel *planned* at a time when the wind is *predicted* to exceed 45 knots. Further discussion of these operational aspects is deferred to chapter 3 and onwards.

The anatomy and possible ways of writing the condition that triggers the warning is treated in section 2.3. We will first expand on the first example a bit to illustrate the different kind of statements.

The `when` statement

Next, we introduce the `when` statement that deals with the third system requirement. Consider the example we had on the previous section. The rule depicted by listing 1 is too simplistic for a real-world scenario. In reality, the size of a vessel matters greatly in how it is affected by wind speeds. We would like to update our previous rule to differentiate vessels that are 200 meters or longer and emit a warning for those vessels when the wind speed exceeds 35 knots:

Scala Listing 2 – Illustrating the `when` statement

```
require (wind.velocity <= 45.knots)
when (vessel.length >= 200.meters) {
  require (wind.velocity <= 35.knots)
}
```

The `when`-statement is analogous to `if`-statements in general programming languages. It takes one or more restrictions and only evaluates them if the given condition is satisfied. The rule in listing 2 thus emits a warning:

- for all vessels when wind speeds are above 45 knots, and
- when wind speeds are above 35 knots, but only for large vessels.

As of the moment of writing, the language has no `if-else` counterpart to keep the language as simple as possible. This feature can straightforwardly be achieved with two `when`-statements where one has the condition inverted.

The `location` statement

Now that we have accounted for larger vessels, there is one more consideration we might want to take care of. Currently, all of our expressible rules range over the entire port. However, in this domain, rules may change for different locations within the port. For example, vessels transporting dangerous chemicals are only allowed to berth in certain sections of the port. We would like to reflect in our language that rules may differ among locations. To this end we've added the `location` statement:

Scala Listing 3 – Illustrating the `location` statement

```
location (wilhelminakade) {
  require (not(
    vessel.vesselType === Chemical
  ))
}
```

This example states that for the location `wilhelminakade`, vessels that berth there are not allowed to carry chemicals. The `location` statement takes a location and one or more sub-restrictions as argument. The way in which locations are modelled is not in scope of this thesis as it is merely an implementation detail unessential to this discussion as a whole. We have now seen all three kinds of statements that a restriction can be constructed from. It is worth pointing out that these statements can be nested arbitrarily, meaning that the body of for example a `when`-statement can in turn contain multiple other other statements, including `when` iteself. This will become clear when we present the Scala types of these operations:

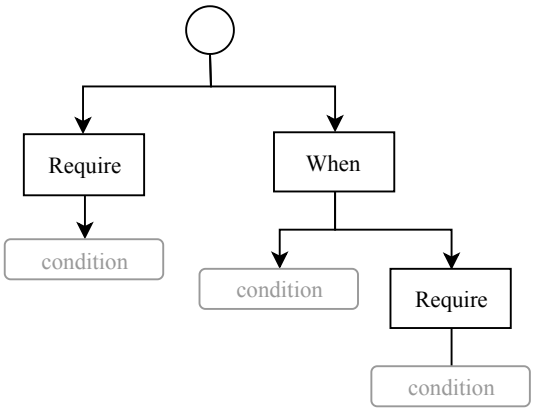


FIGURE 2.1: Visual representation of the object tree created by listing 2

Scala Listing 4 – Abstract syntax tree for statements

```scala
sealed trait RestrictStmt
case class Require(
  restriction: Observation[Boolean]) extends RestrictStmt
case class When(
  condition: Observation[Boolean],
  restrictions: List[RestrictStmt]) extends RestrictStmt
case class Location(
  location: LocationID,
  restrictions: List[RestrictStmt]) extends RestrictStmt
```

Each statement corresponds to a case class and statements that need a body can always take `List[RestrictStmt]` to allow for arbitrary nesting. Note that both `When` as well as `Require` take one `Observation` argument. Just as `RestrictStmt` is the type that captures all statements in our DSL, `Observation` is the type that contains the expression part of our language.

To support the exact syntax as shown in previous listings, we have implemented some additional constructs that allow, among other things, sequencing of multiple statements without having to explicitly wrap them in a `List`.

In essence, the code shown in listings 1-3 when run produces a `RestrictStmt` structure. This structure forms a syntax tree where the internal nodes are either `when`, `location` or `require` nodes, and the leaves terminating the tree are conditional expressions. This is illustrated visually in figure 2.1. How these condition nodes are created and represented is subject of the next section.

2.3 Expressions

Having discussed the part of the DSL that represents statements, we will now cover the expression language in this section. Expressions are the pieces of code passed as arguments to `require` and `when`. This part of the DSL has to enable developers to write numerical and Boolean expressions over observed values from data sources like weather, tide and the vessel for which a rule is evaluated. The goal of these expressions is twofold. First and foremost, expressions that evaluate to Boolean values are used when writing `require` and `when`-statements. Secondly, expressions serve to abstract

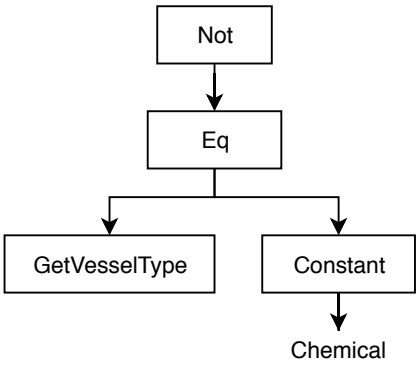


FIGURE 2.2: Visual representation of the object tree created by listing 7

common computations. One such computation for example is the *under keel clearance*, which is the term used to describe the amount of space between the bottom of a vessel and the floor of a berth. We will illustrate this point by example:

Scala Listing 5 – A rule that checks under keel clearance for a single berth

```
location (berthA) {
  require (berth.depth + tide.height - vessel.draught >=
  ↪  20.centimeters)
}
```

In this snippet, for `berthA` the distance between the vessel and the berth floor must always be greater than 20 centimeters. Rules like these are very common, different locations in the port require different mimimum values for this under keel clearance. By allowing expressions to be extracted into their own value, we can re-use the under keel clearance without causing duplicate code:

Scala Listing 6 – Re-using the previous expression for under keel clearance

```
val ukc = berth.depth + tide.height - vessel.draught
location (berthA) {
  require (ukc >= 20.centimeters)
}
location (berthZ) {
  require (ukc >= 0.5.meters)
}
```

To make extracting expressions like this possible without sacrificing type-safety, all expressions are typed with their result type. More on how these types work is explained in the next section. The underlying representation of expressions is similar to that of statements in that it is simply a series of case classes that can be nested to form specific terms. An example expression is given in the following listing:

Scala Listing 7 – Expression operating on a future value and a constant

```
not(vessel.vesselType === Chemical)
```

This computes whether a vessel does not carry chemical products. It has four interesting parts, also shown visually in Figure 2.2:

- `vessel.vesselType` represents a *future* value of the type of vessel for which a rule gets checked;
- `Chemical` represents a constant where the value is known at development-time;
- `===` is the operator for checking equality of two observed values;
- `not(...)` operates on Boolean observations, inverting their outcome.

Listing 8 shows an excerpt of the definition of the expression language with the terms used in the previous example.

Scala Listing 8 – Excerpted AST for expressions

```scala
sealed trait Observation[A]
case class Constant[A](value: A)    extends Observation[A]
case class GetVesselType()          extends Observation[VesselType]
case class Eq[A](l: Observation[A],
                 r: Observation[A]) extends Observation[Boolean]
...
```

An important property of these expressions is that they are not limited to Booleans. To promote re-usability, the DSL allows deverlopers to write expressions with many different kinds of units. Most reasonable operations on numeric and ordered values are supported, as well as some built-in expressions for representing data sources such as the current vessel, weather and tide measurements. Table 2.1 shows most of the operations and observations that are supported.

Types and reusability

While Table 2.1 does not explicitly mention types, expressions are well-typed by construction to remove the need for a hand-written type checker.

As an example, consider this expression that computes the clearance between the lowest point of a vessel and the floor of its berth:

Scala Listing 9 – Expression computing under keel clearance

```scala
val underKeelClearance: Observation[Length] =
  berth.depth + tide.height - vessel.draught
```

The type of `ukc` in this snippet can be fully inferred by the Scala compiler since the three data sources used in this expression all have the same type (`Observation[Length]`) and addition and subtraction do not change those types.

This is not always the case though. Consider the following expression for computing an estimate of the area of a vessel:

Scala Listing 10 – Computing vessel area

```scala
val area: Observation[Area] =
  vessel.length * vessel.beam
```

Here, `vessel.beam` and `vessel.length` are both of type `Observation[Length]`, but multiplying them produces an `Observation[Area]`.

Again we can let the Scala compiler infer these types for us. We rely on the type system and implicit resolution to achieve this. To this end, expressions returning

Built-in observations

Expression	Syntax	Explanation
`GetWindVelocity`	`wind.velocity`	Expression representing the relevant wind velocity
`GetWindDirection`	`wind.direction`	Expression representing wind direction in degrees
`GetTideHeight`	`tide.height`	The height of tide at the destination berth
`GetAvailableTugs`	`availableTugs`	Represents the amount of tugs available
`GetVesselName`	`vessel.name`	The human readable name of the vessel
`GetVesselMMSI`	`vessel.mmsi`	The vessels Maritime Mobile Service Identity number
`GetVesselDraught`	`vessel.draught`	The distance between the water line to the lowest point of the vessel under water
`GetVesselLength`	`vessel.length`	The distance between the bow and the stern of the vessel
`GetVesselBeam`	`vessel.beam`	The distance between the left and the right side of the vessel
`GetVesselType`	`vessel.vesselType`	Classification for the type of goods that a vessel carries

Operations

Expression	Syntax	Explanation
`Plus`	`x + y`	Add observed numeric values
`Minus`	`x - y`	Compute the difference between two observed numeric values
`Negate`	`-x`	Negates a numeric observed value
`Times`	`x * y`	Multiply two values, only allowed for units that can be multiplied to produce a valid unit as result (e.g.: Length and Time)
`Divide`	`x / y`	Divide two values, only allowed for units that can be divided to produce a valid unit as result (e.g.: Velocity and Time)
`Absolute`	`abs(x)`	Compute the absolute numeric value of an observation
`Eq`	`x === y`	Check equality between two observed values
`LessThan`	`x < y`	Check ordering between observed values
`LessOrEq`	`x <= y`	
`GreaterThan`	`x > y`	
`GreaterOrEq`	`x >= y`	
`And`	`x && y`	Perform Boolean AND on two observed Boolean values
`Or`	`x || y`	Perform Boolean OR on two observed Boolean values
`Not`	`not(x)`	Negate a Boolean observation

Table 2.1: Built-in `Observation`s and operations

numeric values keep track their relevant units, and operations on different units are checked to produce the correct unit as their result. This holds for simple cases where an operation only works on operands of exactly the same type (like `Eq` or `LessThan`), but also for operations that can work with different units. For example, when multiplying expressions with different types like `Observation[Length]` and `Observation[Time]`, the compiler ensures that the operands can be multiplied and that the result will be `Observation[Velocity]`. This gives users of the DSL freedom to write expressions with many different kinds of units while always being sure that the resulting program does not contain errors related to incorrect conversion or misrepresentation of numeric values.

2.4 An example scenario

Before we conclude our discussion of the restrictions DSL, let's take a look at one more example to conceptually see how it should execute at runtime.

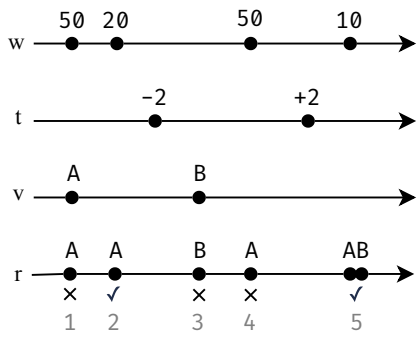


FIGURE 2.3: Timeline of events for the sources and outcome of Listing 11.

Scala Listing 11 – A restriction using multiple data sources

```
1 location (berthA) {
2   require (wind.velocity <= 45.knots)
3   when (vessel.draught >= 15.meters) {
4     require (
5       berth.depth + tide.height
6       - vessel.draught >= 0.5.meters)
7   }
8 }
```

The rule above uses wind, tide and vessel information to declare when a situation is safe. Let's consider what happens to the rule's outcome as these data sources change over time. Figure 2.3 depicts timelines of wind speed in knots (w), tide height in meters (above or below sealevel, t), vessels (v) and the outcome of the rule above (r). The timeline of vessels shows two planned arrivals. Vessel A is a small barge, not deeper than 6 meters. Vessel B however is a large cargo-carrying ship that exceeds 15 meters in draught. The outcome of our rule changes as follows:

1. As vessel A arrives, the wind is predicted to be too much, so our rule emits an event indicating that A can not enter the port at its planned time.

2. When wind speeds drop to 20 knots, our rule re-evaluates and emits that A is no longer restricted.

3. As soon as the event for vessel B arrives, we expect a warning to be emitted because the latest tide event would make line 5 of our rule fail. No warning is emitted since vessel A is small enough to not be affected by the under keel clearance requirement.

4. When the wind picks up again, our rule should re-evaluate that A is again restricted. No additional warning is emitted for B since it was already warned for in a previous event.

5. When wind calms and the tide height becomes acceptable again, both A and B are notified that their restrictions are lifted. This happens at the same time because even though tide height changes first, the excessive wind speeds still block safe entry for vessel B.

In summary, any time that a data source used in the rule changes, it should trigger partial re-evaluation of the rule. Expressions in this DSL thus describe a data flow that

Surface language	Corresponding propositional logic	
	Equivalent statement	**Type of statement**
`require(`a`)`	a	Propositional variable
`when(a) {` $r_0; r_1; ...; r_n$ `}`	$a \implies r_0 \wedge r_1 \wedge ... \wedge r_n$	Implication
`location(l) {` $r_0; r_1; ...; r_n$ `}`	$(vesselDestination = l) \implies r_0 \wedge r_1 \wedge ... \wedge r_n$	Implication

TABLE 2.2: Terms in the restriction algebra correspond to propositional logic

propagates changes from sources every time an update is observed. Intuitively we can view the timelines of Figure 2.3 as event streams. We can imagine that given the rule in Listing 11 there exists a transformation from our DSL to a combination of streaming operators that achieves the desired result stream. The streaming operations that this requires are subject of Chapter 4, where we will also give an intuitive explanation of how such a transformation could work.

2.5 Correspondence to propositional logic

The language of restrictions has the interesting property that it closely resembles formulas in propositional logic [18]. Looking at expressions, the only type of expressions usable in the language are the ones that eventually form a Boolean predicate over one or more inputs. Therefore, the `require`-statement can be seen as directly corresponding to a proposition in logic. The other two statements (`when` and `location`) can be seen as corresponding to implication where the body is a series of conjunctions. Table 2.2 informally depicts the correspondence. It is worth noting that the types of `when` and `location` rule out some forms of propositions, namely those where the antecedent itself contains an implication. This is impossible because the condition of a `when`-statement accepts only expressions (which never correspond to implication).

While mainly being an intellectually pleasing property, this observation could also yield a useful approach to optimising terms in our DSL. By converting rules into a representation of propositional logic, one could potentially transform the rules into conjunctive or disjunctive normal form and reduce the subsequent rule [47]. While reducing Boolean formulas is to their minimal equivalent is an NP-complete problem, many approximation exist that could be used in a practical context [24, 34, 53]. While such an optimization is not in the scope of this thesis, this is an avenue worth considering in future efforts.

Chapter 3

A first attempt at semantics

With the surface DSL fleshed out, we will now discuss our general approach to giving this DSL semantics in terms of streams. All sources of data that can be used in the surface DSL exist as streaming data. More specifically, all events are persisted to Apache Kafka, which is an append-only event log that can be consumed in a streaming fashion [39]. Therefore, all terminal expressions — like `GetWindVelocity` — are given semantics by interpreting them as streams. This in turn means that operators like `Eq` need to combine their operand streams and produce a stream containing the result of evaluating this operator, as would be the case, for example, in Listing 7.

In short, to achieve this we need to develop a translation that takes DSL terms and turns them into successive applications of streaming operations like `map`, `filter`. In this chapter we will first explore some intuitive ideas as to how the translation of expressions could work. We will identify several stumbling blocks that keep us from accomplishing satisfactory semantics. This will in turn motivate the creation of our intermediate representation for streams and relational operators on streaming tables.

3.1 Turning expressions into streams

Let's attempt to transform a simple expression into streaming operations:

Scala Listing 12 – A simple expression

```
1 not (vessel.length > 100.meters)
```

This expression mentions one stream: `vessel`. Events arrive in this stream each time a vessel updates some of its properties like position, planned arrival time, draught, etc. Since from this stream of ships we are only concerned about their length, performing a `map` would suffice:

```
map(vesselStream, _.length)
```

Here, `map` is the hypothetical stream operator that transforms each element of the first argument by applying the function given in the second argument. We will use this kind of notation throughout the thesis where we assume the operator we'd like to use exists as a Scala method, and applying it returns the stream produced by the operator.

Next, the stream of lengths has to be compared to the `100.meters` constant. Again, this can be done with the map operation. The same holds for the last `not` operation. In fact, as long as an expression only mentions one stream, the expression can always be translated to a streaming operation that contains chained applications of `map`:

Scala Listing 13 – Translating expression 12

```
map(map(map(vesselStream,
  _.length),
  _ > 100.meters),
  not(_))
```

But what if an expression mentions two or more streams? Combinations of vessel and tide data are expected to be commonplace in our DSL.

Scala Listing 14 – An expression with two data sources

```
(vessel.draught - tide.height) < 10.meters
```

Evaluating the subtraction presents a challenge, since it has to operate over two streams. It is not immediately clear how this should be realised. One might consider subtracting each element in one stream with every element in the other, essentially producing a cartesian product but this woulbe be needlessly memory-intensive. As only the last tide measurement matters at any point in time, there is no need to consider any measurements in the past. Another approach might therefore be to let the operator subtract only the *latest* values in both streams — as soon as a new element arrives in one of the streams, it is subtracted with the last seen element from the other stream. This is akin to `zip` on the `List` collection in Scala: taking two lists as input, it forms an output list containing tuples of the elements that are in matching position in the inputs. To this end we assume a `zip` operator on streams. Most reactive and streaming libraries support such an operation, usually under the name of `combineLatest` or something similar. Translating the above expression would then result in a combination of `zip` and `map`:

Scala Listing 15 – Translating expression 14 to streams

```
map(map(
  zip(map(vesselTable, _.draught),
      map(tideTable, _.height)),
  subtract),
  _ < 10.meters)
```

This way every time the tide stream receives an event, `zip` emits an update with this tide event and the last seen vessel event. Similarly, when a new vessel is scheduled and an update arrives on the vessel stream, this event is paired with the last tide measurement. Thus, each update to one of the streams results in an update in the result of the original expression.

3.2 The issue with multiple streams

Sadly, the approach outlined above does not give adequate semantics in all cases. Consider the event timelines depicted in Figure 3.1.

In this figure, tide is represented by timeline t, the vessel stream is line v, and the result of the whole expression `r`. Assuming that both vessel A and B have a draught of 9 meters, the result stream will see three elements emitted:

1. An 'allow' event for A because $9 - 2 < 10$.

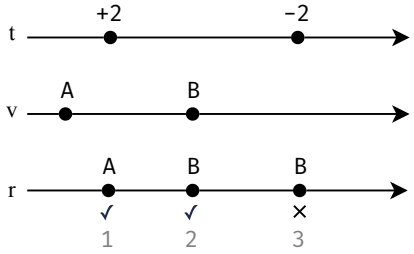


FIGURE 3.1: Combining multiple streams with `zip`

2. The same holds when the event for B arrives.

3. Tide changes again. The expression is re-computed with the *latest* event from the vessel stream. Since $9 - (-2) \geq 10$, a warning is emitted for B.

This poses a significant problem. By using `zip`, any vessel event that is in the past will stop being considered. As a result, rules are only re-evaluated for the last vessel that has received an update. Instead, we wish to also receive warning for vessel A on the second update from the tide stream. While a cartesian product or simply accumulating all events from the first argument of `zip` would accomplish this, we also do not wish to keep track of all vessels indefinitely. Keeping track of all vessels would not only consume an unbounded amount of memory as time goes on, but also pollute the result stream with events for vessels that have long left the port. Instead we wish to only consider vessels that arrive or leave somewhere in the near future. Making matters even more complicated, the vessel stream may emit multiple events for the same ship (for example when an estimated arrival time is updated). In such a scenario, we wish to accumulate the only latest event for each unique vessel.

Devising a way to fulfil all these requirements in terms of operations on streams would quickly become a convoluted task. Indeed, the domain of (mostly stateless) streaming operations does not seem to be a good fit to solve this problem.

3.3 Dealing with state using tables

A much more natural fit would be to model our data sources as stateful tables instead of just plain streams. Capturing vessel events in a table keyed by the ships unique identifier would allow us to keep track of all relevant vessels. Removing vessels as soon as they have left the port is then as simple as a filter on this table. Moving from streams to tables may seem like a radical shift in our model, but as we'll see in the next chapter, streams and tables are actually easily interchangable.

In this section we will envision an small API for tables. This API requires a type for tables that contain entries where `K` is the key and `V` is the value:

```
trait Table[K,V]
```

We will assume that within these tables, entries are uniquely identified by their key. This allows us to model a table of vessels by their unique MMSI (Maritime Mobile Service Identifier):

```
val vesselTable: Table[MMSI, Vessel]
```

Let's see what happens when translating the expression of Listing 14 if instead of streams we imagine data sources to be tables. Just as before, selecting the draught from `vessels` requires a mapping operation over tables. In terms of relational algebra,

`map` over a table is akin to the projection operation (Π). For the subtraction, instead of `zip` we need to have some way of combining two tables and then mapping the subtraction function over the result. Borrowing inspiration from relational algebra, we know that combining two tables means performing a join:

Scala Listing 16 – Translating expression 14

```
map(map(
  join(map(vesselTable, _.draught),
       map(tideTable, _.height)),
  subtract),
  _ < 10.meters)
```

The expression is the same as in Listing 15, except that `zip` is now replaced by `join`. This brings up another question: how exactly are these tables joined? In core relational algebra, cartesian join suffices to model all other variants on joins, but in our discussion of `zip` we have already discarded this as a solution since it would be prohibitively expensive to produce such joins. Instead, our `join` will take two tables that have matching key types, and produce a table of all entries where the keys were equivalent:

```
def join(t: Table[K,A], u: Table[K,B]):
  Table[K, (A,B)]
```

Implied in this choice is that we can only join `vesselTable` and `tideTable` if their keys are the same type. Since `vesselTable` must have `MMSI` as its key, so should `tideTable`. At first this may seem like a significant impediment. After all, it requires us to construct table that maps each vessel MMSI to a tide measurement. From a domain perspective however, this turns out to be useful. Previously, we have modeled tide and wind measurements as singular — e.g.: there could only ever be one up-to-date measure of the tide height for the entire port. This is too simplistic though, as tide heights will differ substantially throughout different locations in the harbor. The same holds for other weather measurements as well such as wind speeds and tidal stream rates. Having a table with tide entries for each vessel would solve this problem. A detailed discussion of how these tables can be obtained is deferred until Chapter 5.

The model that we have arrived at now has proven to be sufficiently powerful. The DSL of restrictions can be fully translated to this model and shows the desired semantics. In the next chapter, we define an algebra of streaming and relational operations and link the two models.

Chapter 4

Algebra for streams and relations

In the previous chapter we have observed, on a high level, how our DSL can be compiled to an API on tables. This chapter will fully shape that API by defining it as an algebra (or intermediate representation) that encompasses the operations needed for many tasks with streams and tables. This algebra focuses on providing a small set of operations to manipulate streaming data in a stateless as well as stateful manner. Stateless operations are concentrated in a subset of the algebra dedicated to streams, whereas stateful operations are concentrated in a subset of the algebra that models streams as tables. With this algebra we hope to provide a small intellectual framework for thinking about streaming applications that need to maintain state and demonstrate that this framework is robust enough to build a real-world case study on. Our work most closely relates to that of Arasu et al. on CQL [9], with the major difference that relations in our work are always keyed to provide for an easier implementation of well-performing joins.

Before CQL, there were numerous early efforts on languages for streaming queries starting with the work from Terry et al [52]. Most of these systems aim to provide SQL-like languages that execute on a periodic basis [16, 22, 41]. Later systems shift focus to providing windowing constructs to deal with the unbounded nature of streams [21, 25, 46]. While most prevous work focuses mainly on SQL-like dialects for querying streams, our work aims to explore a minimal set of fundamental operators needed to combine streams and relations. A noteworthy difference to most previous work is that our representation of relations and streams has no built-in notion of windows, which makes our language considerably smaller. The algebra we propose does however support windowing in terms of the `CombineUpdates` operator.

As with the surface DSL, the streaming algebra is implemented completely in terms of Scala case classes along with a compiler. This means that for example performing a join in this representation merely constructs an abstract syntax tree instead of directly evaluating the join. In some sense, the implementation of this algebra is therefore also an embedded DSL. The algebra is given semantics in later chapters when we discuss two possible execution targets. We will start by answering a natural question: how are streams and tables related? Following on this, we will examine the two parts of the algebra: first the streaming operations and then tables and relational operators. After describing our representation of streams and tables we will discuss how the original surface language of restrictions maps to operations in the streaming domain.

Figure 4.1 displays an overview of the complete set of operators available in the algebra and their relations. Operations on streams are separated from operations on tables by their type. For example, a table of vessels will have the type `TableExp[MMSI, Vessel]`, whereas a stream of wind velocities will have the type `StreamExp[Velocity]`. It might therefore be useful to view this work as two algebras that are connected mainly by the `ToTable` and `Changelog` operators. `Reduce` also forms a connection from tables to streams, which will be explained later in this chapter.

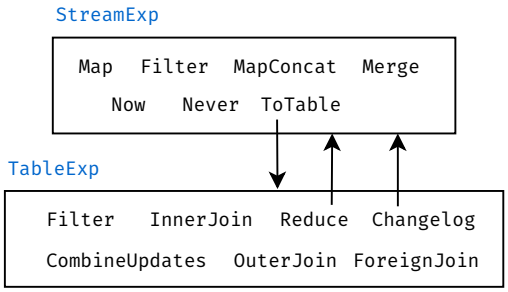


FIGURE 4.1: Overview of the algebra on streams and relations.

4.1 The table-stream equivalence

To connect our table algebra to the streaming algebra we make use of the observation that streams and relational tables have an equivalent representation: any table can be converted into a stream by simply emitting all deletes and updates that occur in the table. The other way around also holds, any stream of updates or deletes can be accumulated in memory to form the current state of the table represented by the stream. This approach is also taken in previous work on CQL and Kafka [9, 50]. Listing 17 illustrates the data types that were added to support this equivalence. The constructors introduced there allow us to treat any table as a stream of updates, and vice versa.

Scala Listing 17 – Conversion between streams and relations

```
sealed trait RowUpdate[K,+A]
case class Update[K,A](key: K, value: A) extends RowUpdate[K,A]
case class Delete[K,A](key: K) extends RowUpdate[K,A]

case class Changelog[K,A](relation: TableExp[K,A]) extends
  ↪ StreamExp[RowUpdate[K,A]]
case class ToTable[K,A](stream: StreamExp[RowUpdate[K,A]]) extends
  ↪ TableExp[K,A]
```

For these to truly form an equivalence, we also have to ensure that the conversion between streams and tables can in no way alter the stream of events. That is, when evaluating a `StreamExp` xs or a `TableExp` t, the following properties must hold:

$$\forall \texttt{xs} : \texttt{Changelog(ToTable(xs))} = \texttt{xs}$$

and

$$\forall \texttt{t} : \texttt{ToTable(Changelog(t))} = \texttt{t}$$

When compiling `StreamExp` and `TableExp` terms, we achieve these properties by treating both `Changelog` and `ToTable` as no-ops. This has the added benefit that none of these conversions carry a runtime cost, a property which we will use later when discussing `map` on relations.

4.2 Operations on streams

Table 4.1 outlines the algebra of streams and stream operators with their types. The table presents operator arguments and types in Scala syntax. Here, the colon in `x:`

Expression	Arguments	Result type	Explanation
`Never`	*None*	`StreamExp[A]`	An empty stream of `A`s.
`Now`	`x: A`	`StreamExp[A]`	A stream emitting `a` once.
`Map`	`xs: StreamExp[A]` `f: A => B`	`StreamExp[B]`	Alter the value of each stream element by applying `f`.
`MapConcat`	`xs: StreamExp[A]` `f: A => List[B]`	`StreamExp[B]`	Similar to `map`, but allows emitting multiple output events per input.
`Filter`	`xs: StreamExp[A]` `p: A => Boolean`	`StreamExp[A]`	Omit elements from `xs` for which applying `p` returns `false`.
`Merge`	`xs: StreamExp[A]` `ys: StreamExp[A]`	`StreamExp[A]`	Combine all elements from `xs` and `ys` into one stream.
`Changelog`	`t: TableExp[K, A]]`	`StreamExp[RowUpdate[K, A]]`	Given a table, represent it as its stream of updates

TABLE 4.1: Abstract syntax representation for streams and operations on them

`A` states that the argument named `x` has type `A`. Functions from `A` to `B` are represented by the arrow-type: `A => B`.

There are two ways to create a stream from other non-stream values. A stream can be created from a single constant value (`Now`) or without any value at all (`Never`) to represent streams that never emit. The other rows represent the operations possible on streams.

In addition, our system also allows for creating `StreamExp` terms from streams that already exist externally. Our specific implementation of this algebra thus has an additional operation for lifting streams from the Akka framework[8] into the algebra domain:

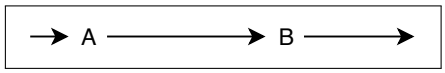
```
FromAkka(xs: Source[A, NotUsed]): StreamExp[A]
```

This additional operation is not fundamental to the algebra itself and merely allows us to provide for a way of incorporating existing streams in our model. Instead of Akka Streams, one could also choose a different framework and a dedicated lifting operation for it.

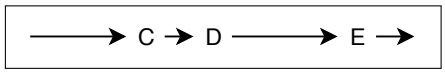
In the remaining part of this section we will first discuss the types and intended behaviour of these constructs at an intuitive level, after which we will substantiate or choice of these specific operators.

`MapConcat`

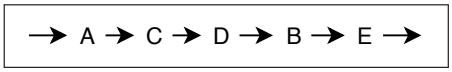
This operation gives the user the possibility to expand the amount of items in a stream. The function supplied to the operator is evaluated for each item emitted by the operand. The values that this function returns are then emitted downstream in the same order as they appear in the returned list. Since the function may be called at any point in time and not necessarily in the exact order in which events arrive from upstream, it is important to ensure that any side-effects that this function produces are local to each invocation. In other words: the supplied operation should not keep state between calls.



(A) Stream `xs`



(B) Stream `ys`



(C) `Merge(xs, ys)`

FIGURE 4.2: Example of merging two streams

`Merge`

Merge allows users to combine multiple separate streams where the emitted items are of the same type. Elements from either upstream sources are immediately emitted downstream, preserving the timing of each item. This makes it possible to implement `merge` in a guaranteed memory-safe way since items never have to be buffered. Figure 4.2 shows progression of time from left to right and shows how timing of events is preserved when merging streams. The `merge` operator is fundamental in our representation: its behaviour can not be achieved with any combination of other operators since none of them allow both combining two streams as well as preserving timing of elements.

`Map` and `Filter`

While not strictly necessary for expressiveness, we have also included the very common `map` and `filter` operations in our representation. Both operators can also be implemented as specialisations of `mapConcat`, but we have chosen to treat them separately for performance reasons. `Map` and `filter` are such common operations that we have decided the performance benefit of not having to allocate unnecessary intermediate `Lists` — as would happen with `mapConcat` — is worth the price of having more constructs in our representation.

A note on the selected operators

One could argue that the presented set of operators for streams is a very small one. After all, many operators can be found in other streaming or reactive programming frameworks that are not present here, like `take`, `drop`, `groupBy` and `reduce`. The reason for this is twofold:

- First of all, we want streaming operations to be mostly stateless. Since tables are stateful by nature, we would like to handle as many stateful concerns in that part of the algebra and avoid the need to mix state into parts of the algebra where they are not necessarily needed.
- Second, most reactive programming operators can be expressed in terms of our minimal set of constructs. Later in this chapter we will see how many of the common functionality from reactive programming libraries can be recovered by implementing them in terms of relational operators.

Expression	Arguments	Result type	Explanation
`FilterTable`	`t: TableExp[K,A]` `p: A => Boolean`	`TableExp[K,A]`	Omit rows from the table where the values do not satisfy predicate `p`.
`InnerJoin`	`t: TableExp[K,A]` `u: TableExp[K,B]`	`TableExp[K,(A,B)]`	Join tables `t` and `u` by their key, creating pairs for each join result.
`OuterJoin`	`t: TableExp[K,A]` `u: TableExp[K,B]`	`TableExp[K,Ior[A,B]]`	Join tables `t` and `u` by their key. The resulting table contains each row from `t` and `u`, pairing up elements when their keys match.
`ForeignJoin`	`t: TableExp[K,A]` `u: TableExp[A,B]`	`TableExp[K,B]`	Join tables `t` and `u`, where the row value of `t` matches the keys of `u`.
`Reduce`	`t: TableExp[K,A]` `g: Group[A]`	`StreamExp[A]`	Incrementally compute a function over all rows in a table.

`CombineUpdates`	`t: TableExp[K,A]`
	`initial: B`
	`combine: (B,A) => B`
		`TableExp[K,B]`	Compute a function over individual rows when one gets updated.

`ToTable`	`t: StreamExp[RowUpdate[K, A]]`	`TableExp[K, A]]`	Given a stream of table updates, represent it as a table.

TABLE 4.2: Abstract syntax representation for operations on streaming relations.

4.3 Streaming Relational Algebra

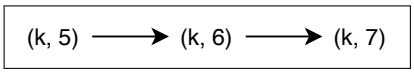
Of course we would be nowhere if we would not have some means of stateful operations on streams. Our model approaches this by modeling state as tables. We will now discuss the operations added to our streaming model to support a form of relational algebra on streams. This extension of our model is similar to relational algebra [23]. We have taken care to support the four classical relational operators (selection, projection, renaming and joining) as well as two additional types of aggregations. However, as opposed to the relational model as originally invented by E. Codd, where a relation is a set of n-tuples, we model tables as a set of 2-tuples where the first part *must* be unique, and the second part may be any shape of data (including arbitrary length tuples). We can view the unique part as the key of a relation. Therefore all relations in our model have a key. From the perspective of Scala code, the type of an expression representing a table is therefore:

```
sealed trait TableExp[K, +A]
```

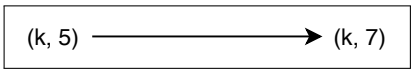
where `K` is the unique key and `A` the table value. Table 4.2 summarizes the operators in our relational algebra.

Absence of `Map`

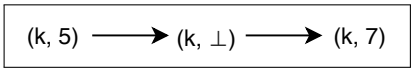
Observant readers might notice that there is no `Map` operator in Table 4.2, while in the previous section a `Map` operator was present for streams. Does this mean our algebra does not support mapping over tables like we can over streams? Thankfully not! Mapping a function over a table can perfectly be achieved without keeping track of any state. Therefore a `Map` primitive is simply not necessary in the relational part of our algebra. Instead, we define map in terms of only streaming constructs:



(A) `Changelog(t)`



(B) `Filter(Changelog(t), p)`



(C) `Changelog(FilterTable(t, p))`

FIGURE 4.3: The effect of filtering over streams versus tables

Scala Listing 18 – `Map` over tables in terms of streams

```
def map[K, A, B](t: TableExp[K, A], f: A => B): TableExp[K, B] =
  ToTable(Map(Changelog(t), {
    case Update(k, v) } => Update(k, f(v))
    case Delete(k) => Delete(k)
  }))
```

The keys in the relation remain untouched, meaning that `map`ing over a table will never change the amount of rows present in the table when materialised. The definition of `map` as above can be used for projection (Π) as well as renaming (ρ) in the context of relational algebra.

`Filter`

Scala Listing 19 – Definition of the `filter` operation on relations

```
case class FilterTable[K,A](
  t: TableExp[K,A],
  p: A => Boolean)
  extends TableExp[K,A]
```

While `Map` can be defined completely in terms of streams because it requires no persistent state between events, it turns out that this does not hold for `FilterTable`. There is a subtle semantic difference between `filter` on streams and tables that makes this a stateful operation on tables. This is best explained by illustrating the difference between

`Changelog(FilterTable(t, p))`

and

`Filter(Changelog(t), p)`

where `t` and `p` can be any arbitrary relation and predicate. Note that the first expression denotes a filter over a table, whereas the second expression denotes one over a stream. Figure 4.3 depicts what happens to a stream of table updates under these two ways of filtering. Figure 4.3a shows the changes of table `t` over time. It emits three updates for the same key, implying that the table has one entry that increases its value as time passes. Now assume `p` is a predicate that selects only odd numbers:

$p(i) = isOdd(i)$. If we apply this filter predicate on the changelog of the table — that is, evaluate the expression `Filter(Changelog(t), p)` — the resulting stream will emit only the odd elements as seen in Figure 4.3b.

However, filtering on the table instead of the changelog of that table will have a different effect. Namely, if we filter all even elements from the table, the changelog stream should also emit *removal* of the filtered rows as soon as the predicate stops holding. During the entire timespan between the second and third event, our table will be empty. This is indicated in Figure 4.3c by $\perp$. To emit this delete-event, the filter operator on tables must know which rows are in the table at any given point in time. Therefore, implementing filter on relations is not possible without materialising intermediate state, making it distinct from a streaming filter.

Joins

Any approximation of a relational algebra demands some form of joins. The purpose of joins is to combine tables in one of two ways: constructing the intersection or the union. Our algebra contains three such joining operations to combine tables, two for intersections and one for unions: inner join, foreign key join and outer join.

The first of these — inner join — computes the intersection between two tables. Each row in the resulting table will be a combination of rows from the operand tables where their keys match. Similar to inner join, foreign join also computes the set intersection of two tables, but here the key of one is the row value of the other.

Scala Listing 20 – Definition of the `innerJoin` on relations

```
1 case class InnerJoin[K,A,B](
2   t: TableExp[K,A],
3   u: TableExp[K,B])
4   extends TableExp[K,(A,B)]
```

Scala Listing 21 – Definition of the `foreignJoin` on relations

```
1 case class ForeignJoin[K,A,B](
2   left: TableExp[K,A],
3   right: TableExp[A,B])
4   extends TableExp[K,B]
```

While `innerJoin` and `foreignJoin` are used to compute table intersections, `outerJoin` can be used to construct the union of two tables. Note that in the return type of `outerJoin` we use `Ior[A,B]`. In Scala the `Ior` data type models a disjunction of two types where either `A` or `B` or both may be present. The resulting table of an `outerJoin` therefore contains all keys from both its left and right operand.

Scala Listing 22 – Definition of the `outerJoin` on relations

```
1 case class OuterJoin[K,A,B](
2   t: TableExp[K,A],
3   u: TableExp[K,B])
4   extends TableExp[K,Ior[A,B]]
```

As we will see in section 4.4, `zip` (or `combineLatest`) can be expressed as a join on two single-entry tables.

Reduce

Most real-world implementations of relational algebra support some form of aggregations like counting or summing elements in a relation. Furthermore, reactive and streaming frameworks like RxJava, Kafka and Akka Streams usually also support aggregating multiple values in a stream (`groupBy` for example). For this reason, we have decided to adopt two more primitives: `combineUpdates` and `reduce`.

The combineUpdates operation allows to aggregate changes to on a per-row basis, whereas `reduce` allows for computing aggregations over all rows present in a table. We will start by introducing `reduce` in this section, by means of an example.

Imagine we are given a table containing the wind speed at several locations in the port of Rotterdam:

```
val windSpeeds: TableExp[ID, Velocity]
```

A simple scenario for `reduce` would be to compute the sum of all wind speeds in this table. This sum could in turn be used to get the average wind speed, but for simplicity we will only consider summing the velocities here. Ideally, we would like to compute this sum in an incremental manner — each time the table receives an update, we would like to update and emit the new sum. For this to work we need to consider what happens to the running total when the `windSpeeds` table receives an update:

- Initially, when the table is still empty, the sum should be 0.
- When an update arrives for a sensor:
  - If this is the first value for that sensor, we emit: $(sum + newVelocity)$
  - If this is not the first value for that sensor, we emit: $(sum - oldVelocity + newVelocity)$

This way, at each point in time, the last value emitted accurately describes the sum of all entries in the table. A significant advantage of this approach is that recomputing the sum takes constant time; we only need the running total, old, and new value for the changed row and avoid having to scan the entire table each time an entry is updated.

To enable these semantics in our `reduce` operator, we need to provide `reduce` with a way to both add new values as well as subtract old values when they are updated. This requirement is captured exactly by the mathematical abstraction of groups. A group is an abstraction over sets of values that have a means to combine values (e.g. addition) and invert them (subtration/negation):

Scala Listing 23 – Definition of `Group` in Scala

```
1 trait Group[A] {
2   def empty: A
3   def combine(x: A, y: A): A
4   def inverse(x: A): A
5 }
```

Furthermore, to ensure that `combine` and `inverse` work together as expected, a group also needs an `empty` element such that the following always holds:

```
combine(x, inverse(x)) = empty
```

As seen in the previous example, integers form a group where `combine` adds integers together, `inverse` negates an integer and `empty` is the integer 0. Any time a table update arrives, the new running total can be computed as:

```
combine(
  combine(total, inverse(oldValue)),
  newValue)
```

This pattern of "subtracting" by combining with an inverse is so common that we have supplemented `Group[A]` with an additional method:

```
def subtract(x: A, y: A): A = combine(x, inverse(y))
```

By using `Group` as the central abstraction for reducing tables, we can now give the signature of reduce as follows:

Scala Listing 24 – Definition of the `reduce` operator

```
case class Reduce[K,A](t: TableExp[K,A], g: Group[A])
  extends StreamExp[A]
```

Any table where the row values form a group can be reduced with this scheme and produce a stream of the results. This brings us to another advantage of relying on `Group` as an abstraction. Summing integers is not the only possibility, computing an average for example is also possible with this mechanism:

Scala Listing 25 – Implementing an averaging aggregation with `Group`

```
case class Avg(sum: Int, count: Int)
implicit val avgGrp = new Group[Avg] {
  def empty = Avg(0, 0)
  def combine(x, y) =
    Avg(x.sum + y.sum,
        x.count + y.count)
  def inverse(x) =
    Avg(-x.sum, -x.count)
}
```

In summary, by using `Group` we do not limit the usage of `reduce` to only count, sum and avg, but give the user of our algebra the ability to use any reduction mechanism they see fit, as long as it can be implemented using a group. In addition, the group abstraction allows `reduce` to perform well even for tables with many rows, as each update only takes constant time to compute.

`CombineUpdates`

Scala Listing 26 – Definition of the `combineUpdates` operator

```
case class CombineUpdates[K,A,B](
  t: TableExp[K,A],
  initial: B, combine: (B,A) => B)
  extends TableExp[K, B]
```

`CombineUpdates` was defined in the same spirit as `reduce`, but instead of reducing all rows in a table, it aggregates the updates to each row over time. This enables users

to determine what happens when an update to a table row arrives. Instead of simply replacing the old row value with the new one, the operator will call `combine(oldVal, newVal)` and use the result as the new row. In the case where there is no `oldVal`, it takes the `initial` element in its place. With this mechanism, row values can be aggregated per key, over time. For example, by using `List`, where `combine` appends two lists together, we can keep track of all updates that ever occur for a certain table key. This construct enables, among other things, the same behaviour as `groupBy` from most common streaming frameworks and SQL.

A note on constants

While our algebra of streams has ways to create streams from constants, our algebra of tables has no constructs to do this. This is due to the fact that we can already achieve these things by first lifting the constant values into a `StreamExp`, and then converting the result to a `TableExp` with `ToTable`. As we will later see, these conversions do not carry any runtime overhead and can thus be performed entirely for free.

4.4 Expressivity

In this section, we will demonstrate how the entire algebra can be used to recover several well-known features from streaming frameworks and libraries like RxJava and Akka Streams[8, 49]. We will also discuss some limitations of the algebra and how those can potentially be remedied.

Implementing ReactiveX operators

Reactive programming[14, 37, 48] libraries offer a rich set of operators to manipulate asynchronous streams of events. While many of these can trivially be implemented in terms of our stateless streaming algebra, some important ones achieve their functionality by keeping internal state. For those operators it is not always obvious how they could be realised with our algebra. With this we show that our streams/table algebra is not only sufficient for our specific use case, but is also expressive enough to recreate many common ReactiveX operators.

Singleton tables

Many operators rely on keeping track of one stateful value in their implementation. We will first give three definitions that help us deal with such state in a clean manner. Singular values can be approached in our model as a table that has at most one entry. Since storing a single value in a table requires associating it with a key, we use the following as dummy key:

Scala Listing 27 – Definitions for creating single-valued table from any stream.

```
1 case object Singleton
2
3 def tableOfLatest[A](xs: StreamExp[A]): TableExp[Singleton, A] =
4   ToTable(Map(xs, x => Update(Singleton, x)))
5
6 def updateStream[A](t: TableExp[Singleton, A]): StreamExp[A] =
7   Map(Chanelog(t), { case Update(Singleton, x) => x }
```

The `Singleton` type has only one possible value, namely itself. A table with `Singleton` as key can therefore only ever hold one entry. Using this type, we derive two new operators: one that turns a stream into a table always containing the last seen event in the stream, and another that recovers the original stream from a singleton table. These operators are inverses of eachother. That is, the following equations will always hold for any approriately typed `xs` and `t`.

```
updateStream(tableOfLatest(xs)) = xs
tableOfLatest(updateStream(t)) = t
```

The next sections will focus on five commonly used reactive programming operatios and detail how they can be obtained in our algebra.

Aggregating streams with `scan`

The first operator chosen is `scan`. Informally, this operator applies a function to each event from its input stream along with the last event emitted by itself. As an example, `scan` allows computing a running sum of a stream of integers:

```
val sums: StreamExp[A] = scan(xs, 0, (total, x) => total + x)
```

Starting with a total of 0, every time `xs` emits, the value is added to the running total and emitted downstream. Scan uses an initial value so it can produce at least one result even if the input stream never emits any events. Using our algebra, this operator can be accomplished with an aggregation over a singleton-table:

Scala Listing 28 – Scan in terms of our algebra.

```
1 def scan[A,B](xs: StreamExp[A], initial: B, f: (A,B) => B):
  ↪  StreamExp[B] =
2   updateStream(CombineUpdates(tableOfLatest(xs), initial, f))
```

Note that in the definition above, types `A` and `B` can be the same, but are not required to. Some variants of `scan` in other libraries do not require the `initial` argument and will simply emit the first event from xs as its first result. This can also be achieved in our algebra with slight modifications to listing 28.

Combining multiple streams with `zip` (or `combineLatest`)

As demonstrated in section 3.1, `zip`ing allows two streams to be combined in a way that enables downstream operators to process elements from both streams at the same invocation. This can for example be used to subtract the latest element occurring in two distinct streams:

```
def subtract(xs: StreamExp[Int], ys: StreamExp[Int]): StreamExp[Int] =
  map(zip(xs, ys), (x, y) => x - y)
```

The desired behaviour is as follows: whenever a new element arrives at one of the two input streams, `zip` will emit this element together with the last seen element from the other stream (if any). This is achieved fairly easily by joining the singleton-tables of both inputs:

Scala Listing 29 – Zip in terms of our algebra.

```
def zip(xs: StreamExp[A], ys: StreamExp[B]): StreamExp[(A,B)] =
  updateStream(InnerJoin(tableOfLatest(xs), tableOfLatest(ys)))
```

Since two `Singleton` values are always equal, joining will always result in a table containing the latest elements of both inputs. Additionally, the behaviour is consistent with `zip` in other libraries in the case where one of the two input streams has not yet emitted any events: the join ensures that a result is only emitted if an event is seen from both inputs.

Filtering elements until a predicate holds with `skipWhile`

The purpose of this operator is to filter elements from a stream until one event is seen that dissatisfies a predicate. After this, all events from the input stream should be emitted regardless of the predicate. Intuitively this is like `filter`, except that the operator has to keep track of whether it has seen any event that fails the predicate. Even though `filter` over streams is stateless, this operator is not.

To keep track of whether an event has occurred that dissatisfies the predicate, we make use of the `Option[A]` type as state. This state starts out as `Option.empty[A]` and switches to `Option.of(x)` whenever an event `x` does not pass the predicate. As soon as the state is non-empty, the predicate will no longer be checked and all received events will be forwarded. Finally, to ensure that only values of type `A` are emitted downstream, we filter the empty `Option`s and unpack the non-empty ones.

Scala Listing 30 – Defining skipWhile in our algebra.

```
def skipWhile[A](xs StreamExp[A], p: A => Boolean): StreamExp[A] =
  def needToEmit(previouslyEmitted: Option[A], x: A): Option[A] =
    if (previouslyEmitted.isEmpty && p(x))
      Option.empty[A]
    else
      Option(x)

  val skipTable: TableExp[Singleton, Option[A]] =
   CombineUpdates(tableOfLatest(xs), Option.empty(), needToEmit)

  Map(Filter(updateStream(skipTable), _.isDefined), _.get)
```

Rolling `window` operator

Windowing is an important feature in any stateful streaming system. As remarked earlier in this chapter, most SQL-like streamin query languages explicitly incorporate windowing constructs. Our algebra supports windowing even though they are not explicitly accounted for. The following code shows an implementation of a rolling window with a maximum size.

Scala Listing 31 – Defining a rolling window in our algebra.

```
def window[A](xs StreamExp[A], maxSize Int): StreamExp[List[A]] =
  def updateWindow(window: List[A], x: A) =
    if (window.size < maxSize)
      window.append(x)
    else
      window.append(x).dropRight(1)

  updateStream(CombineUpdates(tableOfLatest(xs), List(),
  ↪  updateWindow))
```

Grouping streams by key with `groupBy`

Being able to group events is an important feature in many streaming libraries. There are multiple possible approaches to the `groupBy` operator. One may be tempted to accumulate events in a table, producing an API like the following:

```
def groupBy[K,A](stream: StreamExp[A], select A => K): TableExp[K,List[A]]
```

Here, all updates in all groups are accumulated. The problem with this approach is that lists and entries will only ever be added, consuming a potentially unbounded amount of memory. For this specific reason, many streaming / reactive libraries force their users to perform an aggregation (like `sum` or `count`) on the groups, and specify the amount of possible groups manually. Our algebra does not impose any of these restrictions. `groupBy` can be defined with the signature above (and potentially consume unbounded memory), or in such a way that it requires an aggregation over the groups:

Scala Listing 32 – Defining a naive `groupBy`.

```
def groupBy[K,A](stream: StreamExp[A],
                 select A => K): TableExp[K,List[A]] =
  val groups = ToTable(Map(xs, x => Update(select(x), x)))
  CombineUpdates(groups, List(), (state, a) => state.append(a))
```

Scala Listing 33 – Defining `groupBy` with forced aggregation

```
def groupBy[K,A,B](stream: StreamExp[A],
                   select A => K,
                   initial: B,
                   aggregate: (B, A) => B): TableExp[K,B] =
  val groups = ToTable(Map(xs, x => Update(select(x), x)))
  CombineUpdates(groups, initial, aggregate)
```

In both cases, the input stream is first converted to a table with the correct keys. The choice in how groups should be aggregated can always be made afterwards by using `combineUpdates`. It may be more useful to view `groupBy` as a combination of two operators: `keyBy` followed `combineUpdates` to determine the desired aggregation:

Scala Listing 34 – Defining `keyBy` in our algebra.

```
def keyBy[K,A](xs: StreamExp[A], select: A => K): TableExp[K,A] =
  ToTable(Map(xs, x => Update(select(x), x)))
```

Chapter 5

Compiling restrictions to our streaming algebra

This chapter will tackle the translation from DSL terms to the algebra more in-depth. We will not give a formal semantics, but focus on explaining the inner workings of the actual compiler because we believe that this will give a better understanding of the actual system implementation, while also not requiring readers to have a background in programming languages and formal semantics.

The compilers' architecture is relatively straigtforward. For it to compile a restriction rule to streaming algebra, it has two functions. One serves to compile expressions to an algebra expression:

```
def compileExp[A](
  exp: Observation[A],
  vessels: TableExp[MMSI, Vessel]):
  TableExp[MMSI, A]
```

The other function exists to compile statements:

```
def compileStmt(
  rules: RestrictStmt,
  vessels: TableExp[MMSI, Vessel]):
  TableExp[MMSI, Result]
```

Expressions and statements are always compiled for a given table of vessels. This enables us to filter the set of vessels for which rules get checked appropriately when compiling sub-statements inside `when` and `location` rules. The need for this will become more clear when discussing how `restrict` and `when`-statements are compiled. Furthermore, when compiling rules, the caller of the compiler is responsible for supplying the tables that represent external sources:

Scala Listing 35 – Several tables are pre-defined

```
1 val vesselTable: TableExp[MMSI, Vessel]
2 val berthTable: TableExp[BerthID, Berth]
3 val windTable: TableExp[SensorID, Wind]
4 val tideTable: TableExp[SensorID, Tide]
5 ...
```

We will later demonstrate how some of these tables are constructed. As with previous secions, we will split this discussion into two parts. First the translation of statements will be covered, then we will build onto this by covering how expressions are compiled.

5.1 Statement compilation

As discussed in Chapter 2, there are three possible ways to construct a statement. We will lay out the translation for each of the three constructs: `require`, `when` and `location`. In terms of code, compiling a statement means pattern matching on it to bring the fields in scope:

Scala Listing 36 – Compiling each statement

```
def compileStmt(rule: RestrictStmt,
  vessels: TableExp[MMSI, Vessel]) =
  rule match {
    case Require(conditionExp) => ???
    case When(conditionExp, stmts) => ???
    case Location(berthID, stmts) => ???
  }
```

`Require`-statements

Let's first look at the simplest case, compiling a `require`-statement. In the AST, require has just one field: the expression that needs to be checked. If this expression evaluates to `false` for one of the vessels, a warning must be emitted in the result stream. This requires compiling the condition expression, but when doing so we are faced with a choice:

```
val resultBools: TableExp[Vessel, Bool] =
  compileExp(conditionExp,
             ??? /* vessles / vesselTable */)
```

The three question marks have to be filled in by an algebra expression representing a table of vessels. One could be inclined to provide `vesselTable` as argument, as this would result in the condition being checked against *all* vessels. However, this will not produce correct results if the `require`-statment is nested inside a `when`. In that case, the `require` should only range over vessels that have passed the condition given in the `when`. For this reason, we always compile statements and expressions with the vessel table that was provided as function argument:

```
val resultBools: TableExp[Vessel, Bool] =
  compileExp(conditionExp, vessels)
```

Finally we use the table of Booleans to turn them into `Results`. The `Result` type adds more bits of information to the plain true/false result such as which expression was checked.

`When`-statements

We will now look at `when`-statements. Its AST node has two fields: the condition, and a list the statements representing the body of the `when`. These are statements that need to be evaluated only if the condition becomes true. To determine for which vessels the body needs to be evaluated, we first compile the condition:

```
val cResults: TableExp[MMSI, Bool] =
  compile(conditionExp, vessels)
```

Next, the body (`stmts` in Listing 36) needs to be compiled. This is be done by recursively calling `compileStmt` on each of the body's statements. As we have seen in the translation of `Require`, we need to ensure at this step that the body gets compiled with a filtered version of the original table of vessels. We therefore filter all vessels using the result of the condition to obtain a vessel table with only those for which the body should be evaluated:

```
val affectedVessels =
  map(
    filter(
      join(vessels, cResult),
      _.snd),
    _.fst)
```

Now the body of the `when` can be compiled. Since this involves a list of statements, we need to compile each individual statement:

```
val bodyTables =
  stmts.map(compile(_, affectedVessels))
```

The above results in a `List[TableExp[MMSI, Result]]`; i.e. multiple algebra expressions. The signature of `compileStmt` requires us to return a single algebra expression though. So lastly, to obtain a single algebra expression, we need to combine all these individual table expressions into a single one. Conceptually, this goes as follows. Imagine the following snippet:

```
when (vessel.length > 100.meters) {
  when (vessel.beam < 15.meters) {
    require (vessel.draught < 12.meters)
  }
  require (wind.velocity < 35.knots)
}
```

Compiling the body here results in a list with two `TableExp` terms — one which would yield in a table with results of the innermost `when`-statement, the other a table of the `require`. To combine these two tables into a single table, a join is needed. However, the way of joining matters: whenever one of the two tables contains a result for a vessel, we would like to keep that result on the final output. Moreover, in the scenario where both tables have a result for the same ship, we need the final output to keep any of the negative results. The suitable operator to obtain this behaviour is `outerJoin` combined with a `map`:

```
bodyTables.reduceLeft((resultsA, resultsB) =>
  map(outerJoin(resultsA, resultsB), {
    case Ior.Left(a) => a
    case Ior.Right(b) => b
    case Ior.Both(a b) =>
      if (a.isNegative) a else b
  })
)
```

5.2 Expression compilation

This section will outline how expressions are compiled. To gain an intuitive understanding of the expressions that we consider here, we encourage the reader to look at

section 2.3 and onwards in Chapter 2. Table 2.1 displays all constructs that an expression can be made of. There are many constructors in the language, but most of them can be treated very similarly. As with statements, to compile an expression, the `compileExp` method simply pattern matches on each of the possible constructors, building a program with our relational algebra along the way. We will discuss expressions in four parts, starting with the simple ones: constants and the `Map` expression. After this we continue with the constructs that perform functions over other expressions like `LessOrEq` and `Not`. Lastly we explain how expressions for data sources like `vessel` and `tide` are compiled. Note that compiling DSL expressions to our relational algebra forms a natural transformation between the `Observation[_]` and `TableExp[MMSI, _]` types. That is, compiling any DSL expression `Observation[A]` will always result in a `TableExp[MMSI,A]`. The `A` in these types will always be preserved by the compiler.

Compiling constants

To get an intuition of what compiling to `TableExp[MMSI, _]` entails, let's first consider constants. Any Scala value can be treated as a constant in the restrictions DSL. For example, one could write the following:

```
val number: Observation[Int] = 100
```

The integer 100 is converted via implicit conversion to the following AST:

```
val number = Constant[Int](100)
```

If constants are used in the DSL, the underlying syntax tree will therefore contain `Constant`-nodes. What should the result of compiling a constant be? Looking at the type signature for `compileExp` indicates that even for single constants the compiler needs to translate it to a table where the keys are `MMSI`:

```
compileExp(
  number,
  vessels): TableExp[MMSI, Int]
```

This raises another question: how many rows should this table have? We could produce a table with zero rows, but this would mean the constant will essentially be erased. We could also produce a table with one row, but then we would be forced to choose one specific MMSI for its key; an arbitrary choice which is not likely to produce the desired result. Instead, we make sure the result has exactly as many rows as in the provided `vessels` table, thus duplicating the constant over multiple rows:

```
case Constant(x) =>
  map(vessels, v => (v, x))
```

This principle also holds when compiling other expressions. The table-expression produced by the compiler will always have one result for each vessel.

Compiling Map

The `map` operation is prevalent in many expressions written in the surface language. For example, any expression with a selector like `vessel.length`, `wind.velocity` or `berth.depth` is internally represented by `Map`:

```
val vesselLenght =
  Map(GetVessel(), _.length)
```

Compiling `Map` is trivial since our target domain of relational operators also supports `map`:

```
case Map(subExp, f) =>
  map(compileExp(subExp, vessels), f)
```

Compiling operators

The DSL has many operators that work on one or two arguments. Single-argument operators like `Not` and `Abs` are trivially compiled by using `map`:

```
case Not(exp) =>
  map(compileExp(exp, vessels), x => !x)
case Abs(exp) =>
  map(compileExp(exp, vessels), Math.abs(_))
```

Operators with two arguments additionally require a join. Let's take `LessOrEq` as an example:

```
case LessOrEq(leftExp, rightExp) =>
  val leftTable =
    compileExp(leftExp, vessels)
  val rightTable =
    compileExp(rightExp, vessels)
  map(
    join(leftTable, rightTable),
    { case (l, r) => l <= r }
  )
```

After compiling the left- and right hand side individually, the less-then-or-eq operator can only be applied if both tables are joined. By using `join` we ensure that entries in both tables get compared where they have a matching MMSI. This strategy applies to any operator with two arguments: compile and join the arguments, then map the corresponding operator function over it.

Compiling data source expressions

At the core of each expression are the data sources used. These, together with `Constant` form the leaves of all expressions. Aside from `GetVessel`, all other leaf-expressions are compiled in mostly the same manner. `GetVessel` is the simplest one. Because `compileExp` is already called with the current table of vessels as argument, compiling `GetVessel` is as simple as returning the table of vessels:

```
def compileExp(exp, vessels) = exp match {
  ...
  case GetVessel() => vessels
```

Translating the others is a bit more involved as it requires selecting and joining the external tables from Listing 35. We will discuss translation of `GetTide` here, as the others differ only on non-essential aspects. To produce the table of tide measurements for each vessel, we need to link each vessel to a relevant tide sensor:

```
compileExp(GetTide(), vessels):
  TableExp[MMSI, Tide]
```

This correlation can be made by taking the tide sensor that is closest to the destination of the vessel. Assuming the tables form Listing 35 are accessible, the following yields the desired table with one tide measurement for each vessel:

```
case GetTide() =>
  val destinations: TableExp[MMSI, BerthID] =
    map(vessels, _.destination)
  val berthLoc: TableExp[BerthID, LatLong] =
    map(berthTable, _.locationCoordinates)
  val vesselLoc: TableExp[MMSI, LatLong] =
    joinForeign(destinations, berthLoc)
  val vesselSensors: TableExp[MMSI, SensorID]=
    map(vesselLoc, closestTideSensorID(_))
  joinForeign(vesselSensors, tide)
```

This works similarly for wind, tidal streams, and any other kind of data that originates from multiple sensors.

Chapter 6

Interpreting our algebra on Akka Streams

This chapter will take the next step in our end-to-end DSL implementation. Having discussed how restriction rules are compiled into our streaming/relational algebra, we will now focus on semantics for this algebra. Two interpretations are discussed, one in this chapter and another in chapter 7. In the first we will translate our streaming/relational algebra to Akka Streams library API calls[8].

6.1 The Akka Streams API

Akka Streams is a library for building stream processing pipelines in Scala or Java. It was developed to facilitate intuitive and safe formulation of stream processing systems. Users of the API can construct stream processing pipelines using a high-level API with operators much like those from the Scala collections API. Moreover, users can define their own operators by using a lower-level API based on callbacks. This lets developers determine how their operator should respond when upstream events arrive and when downstream operators are ready to process new events.

In the high-level API, there are two ways to produce a stream of events. The most simple method is to provide one event upfront:

```
val stream: Source[Int, NotUsed] =
  Source.single(42)
```

This stream will immediately emit a single integer as soon as it is materialised. In Akka, materialization is the process of allocating all resources needed to start a stream processing pipeline. Any `Source[_, _]` can be materialised by connecting it to a `Sink`. This is usually done by calling one of the `run` methods on a `Source`. To materialize our single-integer stream, we could do the following:

```
val stream: Source[Int, NotUsed] =
  Source.single(42)
val sink: Sink[Int, Future[Done]] =
  Sink.foreach(println)
val done: Future[Done] =
  stream.runWith(sink)
```

Running this code will print the number 42. The returned future will resolve to `Done` once the source stream signals its completion. Note that the type of this stream mentions two type arguments: `Int` and `NotUsed`. The second argument represents the type of the materialised value. This is a value that is returned after a stream is materialised. In our initial stream this value is ignored, but materializing the `Sink` results in the future that signals completion of the pipeline.

Between source and sink, multiple operators can be chained together to do processing on the actual events:

```
stream.map(x => x*x)
  .filter(_ >= 50)
  .drop(10)
  .runWith(sink)
```

Many operators are supported, some serve to transform a single stream, others act on multiple streams. As we will see in a moment, most of the operators from of the streaming part of our algebra can directly be translated into Akka Streams library calls. For the relational operators that maintain state, we define our own Akka Streams operators with their low-level API.

An important property of Akka Streams is that `Source`s can also be created from events that happen externally, like calls to a web API endpoint. This is done by pre-materializing a Source that has a queue as materialised value:

```
val source: Source[Int, SourceQueue[Int]] =
  Source.queue[Int](100, backpressure)
```

Materializing this stream will thus give us a queue to which we can push elements. These elements will then be propagated through the data flow graph:

```
val (queue, source2) =
  source.map(x => x*x).preMaterialize()
source2.runForeach(println)
queue.offer(8) // Prints 64
```

This allows us to describe a stream processing pipeline with functions like `map` and `filter` and afterwards run events through the pipeline by `offer`ing them to the queue.

We will now discuss how our streaming/relational algebra is compiled to Akka Streams operations. The compiler that translates algebra expressions is again a simple recursive function that pattern matches on algebra terms, recurses on their components, and combines the results using functions from the Akka Streams library.

6.2 Compiling the algebra

As our streaming/relational algebra is comprised of two types of expressions (`StreamExp[A]` and `TableExp[K,A]`), our compiler implements two functions to translate both types to Akka Streams:

```
def compile[A](stream: StreamExp[A]):
  Source[A, NotUsed]
def compile[K,A](stream: TableExp[K,A]):
  Source[RowUpdate[K,A], NotUsed]
```

Note the difference in the returned Akka stream. Compiling purely streaming operations will result in a stream of values whereas compiling a table-expression results in a stream of keyed updates. These updates can be accumulated to materialize the full table in memory. While we have considered using Akka's materialised value mechanism for carry the table state, this turned out to be impractical and, as we will see later, would make `ToTable` and `Changelog` conversions carry an unwanted runtime cost.

The `compile` method for streaming expressions is trivial to implement and shown below:

Scala Listing 37 – The complete compiler for StreamExp.

```
def compile[A](stream: StreamExp[A]): Source[A, NotUsed] =
  stream match {
    case Changelog(t)     => compile(xs)
    case Map(xs, f)       => compile(xs).map(f)
    case Filter(xs, p)    => compile(xs).filter(p)
    case MapConcat(xs, f) => compile(xs).mapConcat(f)
    case Merge(xs, ys)    => compile(xs).merge(compile(ys))
    case Now(x)           => Source.single(x)
    case Never()          => Souce.empty
  }
```

Each operation in our algebra conveniently has a corresponding operation in Akka Streams. Also note that compiling `Changelog` terms does not result in the introduction of any additional operation in Akka Streams: `Changelog` is a no-op. The same is true for `Changelog`s counterpart `ToTable` as we will see in the next section, where we will outline the implementation of all stateful relational operators.

Compiling TableExp

The stateful operators in our algebra require a more involved compilation step as those have no direct counterparts in Akka Streams. The remaining part of this chapter is therefore dedicated to explaining how these operators were realised. Note that as before, compiling `TableExp` terms does involve defining a `compile` method as we did before: matching on the `TableExp` term and producing the right Akka Streams program for that term:

Scala Listing 38 – Excerpt of the compiler for `TableExp`

```
def compile(t: TableExp[K,A]) = t match {
  case ToTable(xs)       => xs
  case FilterTable(u, p) => compile(u).via(new FilterOperator(p))
  case InnerJoin(l, r) => ...
  // etc..
}
```

For all but `ToTable`, we introduce our own operators using a low-level callback API provided by the library, and then use those new operators in our compiler. Those are then passed to `via` to connect them recursively to Akka Sources created from compiling sub-expressions.

When defining a custom operators for Akka Streams, one creates a class with a method (`onPush`) that gets called when the operator receives a new element. In turn, there are three methods that can be used in response to a new event:

- `push(outlet, event)`: propagates a new event downstream.
- `pull(inlet)`: signals to an input that the operator is ready for new events.
- `grab(inlet)`: aquire the last event that was emitted by an input.

Furthermore, stateful operators create an internal table to hold the state necessary for computing output events.

`FilterTable`

Pseudo-code implementation of `FilterOperator` is shown in scala listing 39. Note that many details have been omitted in the following snippets to beter convey the essential behavior of the operator. These details mostly relate to how inputs and outputs of a custom operator are declared as well as some logic to determine whether outputs are ready to receive new events.

Filter is declared to have one input and one output as witnessed by line 4 and 5. Since this is an operator on tables, both the input and output handle events of the `RowUpdate` type. Whenever a new event arrives, we pattern match on it to determine whether it is an update or a delete. The operator can then update the table, remove an existing row, or keep the table as is. The only scenario in which a table entry is updated or inserted is when the received event satisfies the predicate. If the event is a delete or and update for an existing entry that dissatisfies the predicate, then that entry should be removed. All other cases require no changes to the table. Note that each time the in-memory table is updated, an event reflecting this change is emitted downstream for other operators to process. The `pull(in)` on line 14 sends a signal upstream indicating that the operator is ready to process new events. Similarly, if the downstream operator signals readiness, our operator propagates this signal upstream. To use the custom operator, we make use of the `via` operation in Akka Streams:

Scala Listing 39 – Pseudocode for implementing a new operator in Akka Streams.

```
1  class FilterOperator[K,A](predicate: A => Boolean) {
2    val table: Table[K,A] = Table.empty[K,A]
3
4    val in: Inlet[RowUpdate[K,A]] = ... // defines an input to the
   ↪  operator
5    val out: Outlet[RowUpdate[K,A]] = ... // defines an output of the
   ↪  operator
6
7    override def onPush(): Unit = grab(in) match {
8      case Update(key, value) if predicate(value) =>
9        table.update(key, value)
10       push(out, Update(key, value))
11     case rowUpdate if table.contains(rowUpdate.key) =>
12       table.remove(rowUpdate.key)
13       push(out, Delete(rowUpdate.key))
14     case _ => pull(in)
15   }
16
17   override def onPull(): Unit = pull(in)
18 }
```

`InnerJoin`, `OuterJoin` and `ForeignJoin`

We will now discuss the implementation of `InnerJoin`, `OuterJoin` and `ForeignJoin`. All join operators take two inputs and produce one output where the type differs per join. For `InnerJoin`, the output is an intersection: the result table contains one entry for each key that is present in both input tables.

```
case class InnerJoin[K,A,B](
  t: TableExp[K,A],
  u: TableExp[K,B])
  extends TableExp[K,(A,B)]
```

Since the join has two inputs, our operator defines two handlers: `onPushLeft` and `onPushRight` for both respective inputs. When an event arrives for one of the inputs, the operator needs to perform a lookup in the other table to see if the received key is already present in that table and emit an update if this is the case. Performing this lookup demands that both input tables are materialised in the operator. Therefore, `InnerJoinOperator` initializes two tables: `tableLeft` and `tableRight`. Figure 40 shows how events from the left input are handled. The handler for the second input is symmetric to the first in that it is the same but all references to `tableLeft` are swapped with `tableRight`.

Scala Listing 40 – Implementing inner-join in Akka Streams.

```
1  def onPushLeft(): Unit = grab(in1) match {
2    case Update(k, v) =>
3      tableLeft.update(k, v)
4      if (tableRight.contains(k)) {
5        push(out, Update(k, (v, tableRight(k))))
6      }
7    case Delete(k) =>
8      tableLeft.remove(k)
9      if (tableRight.contains(k)) {
10       push(out, Delete(k))
11     }
12 }
```

Outer join is similar in implementation to `InnerJoin`, but instead of producing an intersection, it produces a union. To see how, it is useful to reiterate the type of outer join:

```
case class OuterJoin[K,A,B](
  t: TableExp[K,A],
  u: TableExp[K,B])
  extends TableExp[K,Ior[A,B]]
```

Noteworthy here is that the input tables can hold values of different types. The output produced by the union is an inclusive disjunction of the left and the right type. That is, values of the `Ior[A,B]` type can either be an `A`, a `B` or a tuple of both. This means that the union result should indeed have a row for each key in either input table. Figure 41 again shows how events arriving from the left input are handled. Just as with `InnerJoin`, computing the union requires both input tables to be materialised. From the code in this listing it can be deduced that this indeed performs a union. A row in the output is updated whenever any of the two input rows is updated. Furthermore, rows in the output are only deleted when the key is not present in both inputs.

Scala Listing 41 – Implementing outer-join in Akka Streams.

```
def onPushLeft(): Unit = grab(in0) match {
  case Update(k, v) =>
    tableLeft.update(k, v)
    if (tableRight.contains(k)) {
      push(out, Update(k, Ior.both(v, tableRight(k))))
    } else {
      push(out, Update(k, Ior.left(v)))
    }
  case Delete(k) =>
    tableLeft.remove(k)
    if (tableRight.contains(k)) {
      push(out, Update(k, Ior.right(tableRight(k))))
    } else {
      push(out, Delete(k))
    }
}
```

The last join in our algebra is `ForeignJoin`. With this, we can join two tables where the row values of one match with the keys of another:

```
case class ForeignJoin[K,A,B](
  left: TableExp[K,A],
  right: TableExp[A,B])
  extends TableExp[K,B]
```

In this operator we are again computing an intersection, but this time there is no symmetry between the two input tables — an update in the left table needs to be treated differently from an update to the right table. This idea is presented in Figure 42. The performance characteristics of this operator may be questioned. While an update to the left table only requires one lookup in the other, an update to the right table is much more expensive. Since tables are indexed by key, joining a key to a value means the whole table has to be scanned to obtain any matches. A more clever implementation could probably solve this issue by also indexing the foreign keys in the left table, but we have refrained from this to aid simplicity.

Scala Listing 42 – Implementing foreign key join in Akka Streams.

```
def onPushLeft(): Unit = grab(inLeft) match {
  case Update(key, foreignKey) =>
    tableLeft.update(key, foreignKey)
    if (tableRight.contains(foreignKey)) {
      push(out, Update(key, tableRight.get(foreignKey)))
    }
  case Delete(key) =>
    val foreignKey = tableLeft.get(key)
    tableLeft.remove(key)
    if (tableRight.contains(foreignKey.get)) {
      push(out, Delete(key))
    }
}

def onPushRight(): Unit = grab(inRight) match {
  case Update(fk, value) =>
    tableRight.update(fk, value)
    tableLeft.foreach {
      case (key, fk2) =>
        if (fk == fk2) {
          push(out, Update(key, value))
        }
    }
  case Delete(fk) =>
    tableRight.remove(fk)
    tableLeft.foreach {
      case (key, fk2) =>
        if (fk == fk2) {
          push(out, Delete(key))
        }
    }
}
```

Reduce

The reduce operator takes a single table and reduces the rows of the table by combining their values. FOn every update to this table, the reduction is recomputed and the new anwser is emitted downstream. For this reason, `reduce` outputs a stream. The AST node carries two values: the table to reduce, and the operations necessary (re)compute the output:

```
case class Reduce[K,A](t: TableExp[K,A], g: Group[A])
  extends StreamExp[A]
```

As discussed in section 4.3, this `Group` argument allows us to recompute the reduce without performing a scan over the entire table. Whenever an update arrives, the old value is removed from the current reduce outcome before adding the new value. The essential pieces of the reduce operator are outlined in scala listing 43. When a row is removed, its value no longer contributes to the output of the reduce. Therefore, the

output is recomputed by `subtract`ing the removed value from the last known output on line 13. The same happens when a row is updated: the old value before from the update is removed to undo its effect on the outcome, while at the same time adding the newly received value.

Line 6 uses the `empty` value to handle the scenario where a new row is inserted. In this case, there will be no old value in the table to subtract from the previous computed result. By the laws of groups, the `empty` value ensures that adding or subtracting it has no effect, making it a suitable stand-in for values that are not always present. The same thing holds for the initial outcome when the reduce operator starts: when no events have been received yet, the output of the reduce is set to `empty`.

Scala Listing 43 – Implementing the `Reduce` operator in Akka Streams.

```
1  import group.{empty, combine, subtract}
2  var currentOutput = empty
3
4  override def onPush() = grab(in) match {
5    case Update(key, newValue) =>
6      val oldValue = table.getOrElse(key, empty)
7      currentOutput = combine(subtract(currentOutput, oldValue),
   ↪  newValue)
8      push(out, currentOutput)
9      table.update(key, newValue)
10   case Delete(key) =>
11     if (table.contains(key)) {
12       val oldValue = table.get(key)
13       currentOutput = subtract(currentOutput, oldValue)
14       table.remove(key)
15       push(out, currentOutput)
16     }
17 }
```

`CombineUpdates`

Similar to `Reduce`, this operator consumes events from a single table:

```
case class CombineUpdates[K,A,B](
  t: TableExp[K,A],
  initial: B, combine: (B,A) => B)
  extends TableExp[K, B]
```

As explained in depth in section 4.3, `CombineUpdates` allows users to determine how individual rows in a table should be updated. The code in scala listing 44 shows how this operator processes updates to its input table. Deletions are simply propagated as-is while also removing the deleted row from the materialised table. On an update however, the user-provided `combine` function is used to determine the new row value based on the current and incoming one. The `initial` value is used only when a new, previous unknown, row is inserted into the upstream table.

Scala Listing 44 – Implementing the `CombineUpdates` operator in Akka Streams.

```
// 'initial' and 'combine' are in scope as parameters of
↪ CombineUpdates
override def onPush() = grab(in) match {
  case Update(key, updatedValue) =>
    val oldValue = table.getOrElse(key, initial)
    val newValue = combine(oldValue, updatedValue)
    table.update(key, newValue)
    push(out, Update(key, newValue))
  case Delete(key) =>
    if (table.contains(key)) {
      table.remove(key)
      push(out, Delete(k))
    }
}
```

Chapter 7

Actor semantics for our algebra

In this chapter we go over the second interpretation of our streaming algebra. With this translation we aim to convince the reader that the operations in the algebra do not impose impracticable requirements in order to be implemented. Namely, aside from being implementable in terms of Akka Streams, a very specific stream processing tool, the algebra is also implementable in terms of a very minimal actor language.

The actor model is a model for asynchronous, distributed computation connceived in the late 1970s[4, 33]. It views systems as a set of stand-alone programs (actors) that process input independently. Each actor processes incoming messages from a queue one by one. How a particular message is handled is decided by an actors *behaviour*: a function that takes a single message and performs a sequence of actions. An action can be one the following three:

- create new actors,
- send messages to actors of which it knows the address,
- specify the replacement behaviour for processing subsequent messages.

Except for the three actions mentioned above, it is not possible for an actor to modify internal state by means of variable assignment or perform other side-effects. Keeping track of changing state can only be achieved by constructing a replacement behaviour that contains the newly modified data. Furthermore, actors can only influence each other via the mechanism of message passing — there is no shared memory or other communication possible between actors. These properties leave the door open for highly parallel and pipelined execution of programs.

Viewed from the outside, an actor system has a set of *receptionists*: actors inside the system that are externally visible and can receive input from the outside world. Similarly, *external* actors are known within the system to send messages to the outside world. Receptionists and external actors constitute the interface of the system. This is notably similar to the source/sink model known from stream processing.

The actor model has been studied in fields from artificial intelligence to embedded systams and has also received significant attention from industry practitioners[3, 11, 15, 40]. Several programming languages have been created around actors and many mainstream programming languages now also support actor-based development through libraries and frameworks[6, 7, 11, 17, 28].

Because we would like our system to possibly also run in a distributed setting, we dedicate this chapter to compiling our algebra to a rudimentary actor language. The language that we use is an extension of a small functional language with actor primitives. This language shares its constructs and semantics with the one presented by Agha et al [5]; we merely use differentiate in syntax.

The existance of this translation furthermore implies that our algebra is suitable for any actor system, regardless of language or ecosystem.

7.1 The actor language

Before we describe the actual compilation, we briefly introduce the language that serves as our compilation target. For a full discussion we refer the reader to the original work of Agha et al[5]. The language assumes a small functional core featuring lambda abstraction, let-bindings, conditional expressions with an if-else construct, a `seq` operator for sequential composition of actions and a fixed-point operator to enable recursion. In the following we will regularly mention meta-variables like e_1, k, and v. Meta-variables k and v range over values. Those of the form e_n range over any expression in the actor language and are therefore not limited to merely values but can also be identifiers, if-expressions, etc.

Actor actions

This small functional core is extended with three actor-specific statements:

- `create(`e_1`)`: instantiates a new actor from the expression e_1 and returns the actors address. This expression must evaluate to a behaviour-function that takes a message as argument.
- `send(`e_1`, `e_2`)` sends a message to an actor. The first argument e_1 must evaluate to an actor address while the result of e_2 is used as the message. Sending a message is an action without a return value.
- `become(`e_1`)` instantiates an anonymous actor that serves to replace the current actors' behaviour. For this, e_1 must evaluate to a behaviour-function just like with `create`. Just like `send`, this operation returns no usable value as result.
- `skip` is a no-op action that has no effect.

Values

In the language presented by Agha et al, values are either lambda terms or integers. Our adaptation has no need for integer arithmetic but does rely on distinguishing between updates and deletes in a table. Therefore we assume a value representing updates to table rows with the following operations:

- `mk-update(`e_k`, `e_v`)` constructs a row update where e_k is the row key and e_v the value.
- `mk-delete(`e_k`)` constructs a value signaling deletion of a table-row with key e_k.
- `is-update(`e`)` checks whether a row-update is an update or a delete. Returns *true* if it is an update.
- `key-of(`e`)` returns the key that was used when constructing the row-update or delete.
- `value-of(`e`)` returns the value if e evaluates to a row update. This operation is only defined on values constructed with `mk-update`.

These row-update values are passed between actors to singal changes in their state and allow downstream operators to act on changes in upstream tables.

Of course, to maintain state inside actors we also need a means of storing and searching key-value pairs. For this we also assume a simple table API is available to us where tables are values and three operations exist:

- `empty`: creates an empty table,
- `update(`e_t`, `k`, `v`)`: updates table e_t by associating key k with value v. Inserts a new entry when needed. If there is already a value associated with the key, it replaces the existing value with the provided one. This operation returns a new table with the modified contents.
- `delete(`e_t`, `k`)`: removes an entry from the table with key k and returns the modified table. This has no effect if the key is not present.
- `contains(`e_t`, `e_k`)` checks if a value is associated with key e_k in the table.

To make sure our target language does not facilitate side-effects beside the three actor-specific actions, we restrict this table API to be a pure one [45]. That is, the table data structure is persistent — any call to update or remove an entry returns an altered copy of the table while keeping the original unchanged.

Lastly, some operators require the target language to have some way of representing lists of values and tuples. Simple cons-lists with `head`, `tail` and `is-cons` operations are sufficient for this. We also make use of a `pair(l, r)` operator to construct a tuple.

Examples

Below are two simple examples of programs written in this actor language. One of the most trivial actors imaginable is the actor that simply ignores its input:

Actor Listing 1 – An actor that ignores messages.

```
1 let { sink = create(fix(λs.λm. become(s))) }
2 send(sink, 1)
```

The behaviour of this actor only performs one action: `become(s)`. With this action, the actor registers itself as successor for processing subsequent messages. The self-reference is made available by using the fixed-point operator. This actor will only ever perform the same behaviour and therefore always ignore the message m. Instantiating it and sending the integer 1 thus has no effect.

Our second example illustrates the mechanism for keeping track of state:

Actor Listing 2 – An actor behaviour that keeps track of state.

```
1 let {
2   countBehaviour = fix(λs.λi.λm.
3     seq(become(s(i + 1)), send(m, i))
4   )
5 } ...
```

This behaviour increments a counter each time it receives a message and replies the current counter value to the sending actor, assuming the received message contains the address of the sending actor. The behaviour defined here takes three arguments: the message to handle m, a self-reference provided by `fix`, and the current state of the actor i. Whenever a message arrives, this behaviour registers $s(i + 1)$ as the replacement behaviour. Via this mechanism the actor registers itself for processing new messages but replaces the state i with an updated value (since the s refers to the function of i and m). The `become` action is therefore essential in supporting mutable state in an actor system.

7.2 Translating streaming operators

In this section we will demonstrate how streaming operators are converted to the actor language introduced above. Each operator is represented as a one or more actors, the behaviour of which will process messages from upstream sources. Before diving into the translation rules, let's discuss the behaviours of each operator as expressed in the actor language. For every operator, we define a behaviour-function from which an actor can be instantiated. These behaviours are lambda terms in our target language. To create the behaviours, our translation makes use of auxilary functions that construct the correct behaviour. For example, the auxilary function B_{map} constructs the behaviour of the map-operator:

Actor Listing 3 – Constructing the `Map` behaviour

```
1 B_map(f, out) = fix(λs.λm.seq(become(s), send(out, f(m))))
```

When provided with the actor reference to publish results to as well as the mapping function, B_{map} constructs the behaviour for `Map` in the actor language. From the usage of `fix` and `become(`s`)` we can see that an actor instantiated from this behaviour will always perform the same task without any state changes.

Similar constructors exist for all other operators:

Actor Listing 4 – Constructing the `Filter` behaviour

```
1 B_filter(p, out) = fix(λs.λm.
2   seq(become(s), if p(m) then send(out, m) else skip)
```

Actor Listing 5 – Constructing the `Merge` behaviour

```
1 B_merge(out) = fix(λs.λm. seq(become(s), send(m, out)))
```

The behaviour of `MapConcat` differs slightly form the previous two because it has to emit multiple messages downstream. After calling $f(m)$ which returns a list of messages to send downstream, we use a recursive function `sendAll(`xs`)` to emit each message in the list:

Actor Listing 6 – Constructing the `MapConcat` behaviour

```
1 B_mapConc(f, out) = fix(λs.λm.
2   let { sendAll = fix(λsendAll.λxs.
3     if is-cons(xs)
4     then seq(send(out, head(xs)), sendAll(tail(xs)))
5     else skip)
6   }
7   seq(become(s), sendAll(f(m)))
```

We do not define behaviour constructors for `Now` and `Never` since these are sources that simply emit values and do not receive messages themselves. Translation of these constructs is made explicit later.

Recursive translation of whole algebra expressions

In the previous we have outlined how streaming operator work individually in an actor language. We will now use these behaviours to translate entire programs from out

algebra into the target actor domain. This is done via a recursive function with the following notation:

$$[\![alg, out]\!] = e$$

Here alg can be any expression in our streaming/table algebra, out is a term in the target language that represents a reference to an actor, and e is the result expression of the translation. Informally stated this says that algebra term alg translates to actor program e which publishes output messages to actor out. All translation rules are listed together in Figure 7.1.

$$[\![\texttt{Never()}, out]\!] = \texttt{skip} \qquad [\![\texttt{Now}(a), out]\!] = \texttt{send}(out,\ a)$$

$$[\![\texttt{Map}(f,\ xs), out]\!] = [\![xs, \texttt{create}(B_{map}(f, out))]\!]$$

$$[\![\texttt{Filter}(p,\ xs), out]\!] = [\![xs, \texttt{create}(B_{filter}(p, out))]\!]$$

$$[\![\texttt{MapConcat}(f,\ xs), out]\!] = [\![xs, \texttt{create}(B_{mapConc}(f, out))]\!]$$

$$\begin{aligned}[\![\texttt{Merge}(xs,\ ys), out]\!] = {} & \texttt{let } \{\ A_{merge} = \texttt{create}(B_{merge}(out))\ \} \\ & \texttt{seq}([\![xs, A_{merge}]\!],\ [\![ys, A_{merge}]\!])\end{aligned}$$

$$[\![\texttt{FilterTable}(p,\ xs), out]\!] = [\![xs, \texttt{create}(B_{Rfilter}(p, out))]\!]$$

$$\begin{aligned}[\![\texttt{InnerJoin}(xs,\ ys), out]\!] = {} & \texttt{let } \{\ A_{ijoin} = \texttt{create}(B_{ijoin}(out,\texttt{nil},\texttt{nil}))\ \} \\ & \texttt{let } \{\ A_L = \texttt{create}(B_{map}(\lambda m.\texttt{mk-left}(m), A_{ijoin}))\ \} \\ & \texttt{let } \{\ A_R = \texttt{create}(B_{map}(\lambda m.\texttt{mk-right}(m), A_{ijoin}))\ \} \\ & \texttt{seq}([\![xs, A_L]\!],\ [\![ys, A_R]\!])\end{aligned}$$

$$\begin{aligned}[\![\texttt{OuterJoin}(xs,\ ys), out]\!] = {} & \texttt{let } \{\ A_{outerjoin} = \texttt{create}(B_{outerjoin}(out,\texttt{nil},\texttt{nil}))\ \} \\ & \texttt{let } \{\ A_L = \texttt{create}(B_{map}(\lambda m.\texttt{mk-left}(m), A_{outerjoin}))\ \} \\ & \texttt{let } \{\ A_R = \texttt{create}(B_{map}(\lambda m.\texttt{mk-right}(m), A_{outerjoin}))\ \} \\ & \texttt{seq}([\![xs, A_L]\!],\ [\![ys, A_R]\!])\end{aligned}$$

$$\begin{aligned}[\![\texttt{ForeignJoin}(xs,\ ys), out]\!] = {} & \texttt{let } \{\ A_{foreignjoin} = \texttt{create}(B_{foreignjoin}(out,\texttt{nil},\texttt{nil}))\ \} \\ & \texttt{let } \{\ A_L = \texttt{create}(B_{map}(\lambda m.\texttt{mk-left}(m), A_{foreignjoin}))\ \} \\ & \texttt{let } \{\ A_R = \texttt{create}(B_{map}(\lambda m.\texttt{mk-right}(m), A_{foreignjoin}))\ \} \\ & \texttt{seq}([\![xs, A_L]\!],\ [\![ys, A_R]\!])\end{aligned}$$

$$[\![\texttt{Reduce}(xs,\ g), out]\!] = [\![xs, \texttt{create}(B_{reduce}(\texttt{g.empty},\texttt{g.combine},\texttt{g.subtract}, out))]\!]$$

$$[\![\texttt{CombineUpdates}(xs,\ i,\ c), out]\!] = [\![xs, \texttt{create}(B_{combineUpdates}(i, c, out))]\!]$$

$$[\![\texttt{ToTable}(xs), out]\!] = [\![xs, out]\!] \qquad [\![\texttt{Changelog}(t), out]\!] = [\![t, out]\!]$$

FIGURE 7.1: Translation rules that give our algebra semantics in terms of a minimal actor language.

The two simplest rules are those for `Never` and `Now`:

$$[\![\texttt{Never()}, out]\!] = \texttt{skip} \qquad [\![\texttt{Now}(a), out]\!] = \texttt{send}(out, a)$$

Informally, these rules state that translating `Never` results in an actor program that sends nothing to the output actor, while `Now`(a) translates to a program that sends a single value to the output. The rules above are non-recursive because the algebra terms that they translate are terminal; they do not themselves contain any further terms to translate. Since all other constructs in the algebra are non-terminal, their translation rules are recursive.

For instance, a `Map` algebra expression is translated by instantiating an actor with the B_{map} behaviour and using that actor as output when translating xs recursively:

$$[\![\texttt{Map}(f, xs), out]\!] = [\![xs, \texttt{create}(B_{map}(f, out))]\!]$$

As a step-wise illustration, translating `Map`$(\lambda i.i + 1,$ `Now`$(1))$ would result in the following:

$$\begin{aligned} &[\![\texttt{Map}(\lambda i.i + 1, \texttt{Now}(1)), out]\!] = \\ &[\![\texttt{Now}(1), \texttt{create}(B_{map}(\lambda i.i + 1, out))]\!] = && (\texttt{Map}\text{-rule}) \\ &\texttt{send}(\texttt{create}(B_{map}(\lambda i.i + 1, out)), 1) && (\texttt{Now}\text{-rule}) \end{aligned}$$

In the case of `Merge`, the translation has to take care of combining the two imput streams. We achieve this by simply recursively translating both input stream expressions and set their output to be the merge-actor:

$$\begin{aligned} [\![\texttt{Merge}(xs, ys), out]\!] = &\texttt{let } \{ A_{merge} = \texttt{create}(B_{merge}(out)) \} \\ &\texttt{seq}([\![xs, A_{merge}]\!], [\![ys, A_{merge}]\!]) \end{aligned}$$

We can use B_{merge} to output messages from both xs and ys because both of the merged streams emit events of exactly the same type.

7.3 Translating relational operators

Having discussed the implementation and translation of all streaming operators we will now continue with relational operators. Since the translation rules for these operators are structurally very similar as those explained in the previous section, we will only focus on the behaviours that carry out the actual operations.

The behaviours for relational operators share a lot of their conceptual logic with the Akka Streams implementations that we have seen in chapter 6. As before, we will omit parts of the code for join operators when their behaviour is symmetric on both inputs.

`FilterTable`

Starting with the relational version of `Filter`, the behaviour is given in actor listing 7. As can be seen from this code, the operator only outputs an update if the triggering message is an update and the predicate still holds for the new value. Otherwise, a

delete is emitted if an entry is present in the table but it is invalidated due to an upstream delete-message or the predicate not holding anymore.

Actor Listing 7 – Constructing the `Filter` behaviour for tables.

```
1  B_Rfilter(p, out) = fix(λs.λt.λm.
2    if is-update(m)
3      then let { key = key-of(m) }
4           let { value = value-of(m) }
5           if p(value)
6             then seq(become(s(update(t, k, v))), send(out, m))
7             else
8               if contains(t, key)
9                 then seq(become(s(delete(t, k))),
10                         send(out, mk-delete(k)))
11                else skip
12     else let { key = key-of(m) }
13          if contains(t, key)
14            then seq(become(s(delete(t, key))),send(out, m))
15            else skip
```

The translation rule for this operator is conceptually the same as `Filter` on streams: it instantiates the filtering behaviour and translates the subexpression with the created actor as target.

Joins

The actor implementations of the three join operators are presented below in listings 8, 9 and 10. These behaviours are slightly more involved. To start, each behaviour takes two additional state-arguments: t_L and t_R. These are the materialised tables for both input streams. Note that propagation and mutation of state is achieved by updating the two input tables and threading them through `become`-calls, as explained in section 7.1. Any time a join behaviour receives a message, it needs to determine which of the two materialised tables to check against: if the message originates from the left input stream, the right table needs to be checked to see if a join result can be produced. To allow join-behaviours to distinguish which input a message comes from, we have to tag messages from both inputs before they are sent to the behaviour. This can be seen in the translation rule for `InnerJoin`:

$$
\begin{aligned}
[\![\texttt{InnerJoin}(xs,\ ys), out]\!] = &\texttt{let } \{\ A_{innerjoin} = \texttt{create}(B_{innerjoin}(out,\texttt{nil},\texttt{nil}))\ \} \\
&\texttt{let } \{\ A_L = \texttt{create}(B_{map}(\lambda m.\texttt{mk-left}(m), A_{innerjoin}))\ \} \\
&\texttt{let } \{\ A_R = \texttt{create}(B_{map}(\lambda m.\texttt{mk-right}(m), A_{innerjoin}))\ \} \\
&\texttt{seq}([\![xs, A_L]\!],\ [\![ys, A_R]\!])
\end{aligned}
$$

The translation creates an actor program which performs several tasks. Firtly, it instantiates an actor with the innerjoin behaviour which has both state variables set to `nil`. Next, two actors are created that tag messages as originating from the "left" or "right" upstream source. Both these actors use $A_{innerjoin}$ as their output and will therefore send their results to the innerjoin operator. Finally, the input streams are translated with the tagging-actors as their targets, ensuring that all input events are correctly tagged and eventually forwarded to $A_{innerjoin}$.

All translation rules for joins follow a pattern similar the one just described.

Actor Listing 8 – Constructing the `InnerJoin` behaviour.

```
1  B_innerjoin(out) = fix(λs.λt_L.λt_R.λm.
2    let { key = key-of(m) }
3    if is-left(m) then
4      if is-update(m)
5        then let { value_L = value-of(m) }
6             seq(become(s(update(t_L, key, value_L), t_R)),
7                 if contains(t_R, key)
8                   then
9                     let { value_r = get(t_R, key) }
10                    send(out, mk-update(key, pair(value_L, value_R)))
11                  else skip)
12       else seq(become(s(delete(t_L, key))),
13                if contains(t_R, key)
14                  then send(out, mk-delete(key))
15                  else skip)
16   else
17     ...
```

Actor Listing 9 – Constructing the `OuterJoin` behaviour.

```
1  B_outerjoin(out) = fix(λs.λt_L.λt_R.λm.
2    let { key = key-of(m) }
3    if is-left(m) then
4      if is-update(m)
5        then let { value_L = value-of(m) }
6             seq(become(s(update(t_L, key, value_L), t_R)),
7                 if contains(t_R, key)
8                   then let { value_r = get(t_R, key) }
9                        send(out, mk-update(key, pair(value_L value_R)))
10                  else send(out, mk-update(key, value_L)))
11       else seq(become(s(delete(t_L, key))),
12                if contains(t_R, key)
13                  then send(out, mk-delete(key))
14                  else send(out, mk-update(key, value_R)))
15   else
16     ...
```

Actor Listing 10 – Constructing the `ForeignJoin` behaviour.

```
1  B_foreignjoin(out) = fix(λs.λt_L.λt_R.λm.
2    let { key = key-of(m) }
3    if is-left(m) then
4      if is-update(m)
5        then let { value_L = value-of(m) }
6             seq(become(s(update(t_L, value_L))),
7                 if contains(t_R, key)
8                   then send(out, mk-update(value_L, get(key, t_R)))
9                   else skip)
10       else seq(become(s(delete(t_L, key))),
11                if contains(t_R, key)
12                  then send(out, mk-delete(key))
13                  else skip)
14   else
15     if is-update(m)
16       then let { value_R = value-of(m) }
17            let { for-each = fix(λs.λxs.λf.
18                    if is-cons(xs)
19                      then seq(f(head(xs)), s(tail(xs), f))
20                      else skip) }
21            let { emitEntry = λentry.
22                    if value-of(entry) == key
23                      then send(out, mk-update(key-of(entry), value_R))
24                      else skip }
25            seq(become(s(update(t_R, value_R))),
26                for-each(get-entries(t_L), emitEntry))
27       else let { emitDelete = λentry.
28                    if value-of(entry) == key
29                      then send(out, mk-delete(key-of(entry)))
30                      else skip }
31            seq(become(s(delete(t_R, key))),
32                for-each(get-entries(t_L), emitDelete))
33   )
```

`Reduce` and `CombineUpdates`

Actor Listing 11 – Constructing the `Reduce` behaviour.

```
1  B_reduce(e, c, s, out) = fix(λs.λt.λr.λm.
2    let { key = key-of(m) }
3    if is-update(m)
4      then
5        let { old = if contains(t, k) then get(t, k) else e }
6        let { v = value-of(m) }
7        let { rNew = c(s(r, old), v) }
8        seq(become(s(update(t, k, v), rNew)), send(out, rNew))
9      else
10       if contains(t, k)
11         then let { old = get(t, k) }
12              let { rNew = s(r, old) }
13              seq(become(s(delete(t, k), rNew)), send(out, rNew))
14         else skip
15 )
```

Actor Listing 12 – Constructing the `CombineUpdates` behaviour.

```
1  B_combineUpdates(e, c, s, out) = fix(λs.λt.λr.λm.
2    let { key = key-of(m) }
3    if is-update(m)
4      then
5      else
6 )
```

Stream and table conversions

Two constructs in our algebra remain: `ToTable` and `Changelog`. As outlined before, these operations only serve as static book-keeping constructs that witness the isomorphism between relations and streams. At runtime the conversion between stream and table should not be observable since a table is represented as a stream that emits row updates, as can be seen from the rules in Figure 7.1. This translation is valid for two reasons:

- First of all, both operators do not need to materialise any state. Any state potentially kept by `ToTable` or `Changelog` is only observable for downstream operators, all of which already materialise state themselves if they need it.
- Second, the expected output messages do not change under both operators: `ToTable` expects a to receive row updates and deletes (as witnessed by the type of xs), and promises to send row updates to the output actor. The same holds for `Changelog`.

This allows the translation to simply ignore these operators entirely.

Chapter 8

Benchmarks

This chapter outlines the results of several end-to-end benchmarks of our system. The aim of these is to give an indication of the systems' performance in the setting of our case-study.

8.1 Benchmarked programs

We have selected a set of five programs written with the restrictions-DSL. Each of these programs shows a structural characteristic commonly seen in restriction-rules.

Baseline (program A)

We start with a very simple baseline program to get a sense of the maximum throughput possible for trivial rules. This program only performs a single test on the dimensions of all vessels:

Scala Listing 45 – Baseline program for benchmarking.

```
1 require (vessel.length <= 100.meters)
```

This program trivially involves only a single stream (the vessel stream). Since this rule has no need for any joins, we expect the throughput to be unaffected by the amount of unique vessels.

Two streams (program B)

Next, we extend the trivial program by involving two streams instead of one. Evaluating a rule with multiple streams means joins will happen, which will affect the throughput as materialized tables grow larger.

Scala Listing 46 – Program with two streams that need to be joined.

```
1 require (vessel.draught - tide.height <= 10.meters)
```

Nested branching (program C)

A very common pattern in real-world restrictions is exemplified by the following program. This program was re-created from a textual description of an actual rule enforced by the port authority. It contains nested contitional checks to determine which requirement should hold.

Scala Listing 47 – Program with nested conditional checks.

```
when (vessel.direction === Inbound) {
  when (vessel.draught >= 14.3.meters) {
    when (tidalStream.direction >= 180.degrees) {
      require (tidalStream.rate <= 1.knots)
    }
  }
  when (vessel.draught >= 19.meters) {
    require (tidalStream.direction < 180.degrees)
  }
}
```

Many requires (program D)

The fourth program features many independent requirements over all possible streams. Such rules are common for large sections of the port the contain many berths to specify requirements that must hold for all these berths.

Scala Listing 48 – Program with multiple independent requirements.

```
require (tidalStream.rate <= 1.knots)
require (tidalStream.direction < 0.degrees)
require (vessel.length <= 10.centimeters)
require (vessel.beam < 10.centimeters)
require (wind.direction < 180.degrees)
require (wind.velocity < 30.knots)
require (tide.height < 3.meters)
require (tide.low_water > 0.meters)
```

8.2 Benchmark methodology

The performance of most programs depends on the amount of entries in each of the materialized tables. We have therefore measured the throughput of each of the above programs for varying table sizes. To facilitate this, we pre-generate the contents of all possible input tables. The data is generated in such a way that any possible join between any pair of tables returns the largest possible result, thus accounting for a worst-case scenario. For example, we pre-fill the vessel table by sending n vessel events through the system. Each such event has a unique vessel id as well as a unique destination berth id. Then, we pre-fill the berth table by sending n events through the system each with a unique berth id that matches exactly one of the previously mentioned vessels. We repeat this for all tables such that for all rows in any given table, there is a unique corresponding row in all other tables that matches. This also ensures that each table has the same amount of rows before the benchmark starts. Besides randomly generated table data, we also generate beforehand all events that will be fed to the system during the benchmark. This ensures that the benchmark will only measure the time it takes for events to propagate through the system. After the pre-fill phase we perform an additional warmup phase to account for cold-start of the JVM.

During the benchmark itself, we feed pre-generated events into the system for 30 seconds, and measure at one-second intervals how many events were processed. When emitting an event, the event data as well as the stream to emit to is picked randomly with uniform probability. The events fed as input are ensured to always match a random row in the tables that were pre-filled to make sure each passes through the entire pipeline of operators. Note that the above setup results in the heaviest possible workload as each event fed to the system always matches a row in every table and all tables fully join with each other. Real-world workloads will most likely always be a lot more forgiving.

All measurements were performed on a laptop featuring an Intel Core i7-7700HQ with 4 cores running at 2.8 GHz. The heap space of the JVM was pre-allocated and restricted to 8 GB.

8.3 Results

Figure 8.1 shows the measured throughput for each of the benchmarked programs. The error-bars in this chart show the maximum and minimum observed throughput. For every configuration, 30 measurements were taken. Benchmarks were performed with tables containing 10 entries up to 100000 entries.

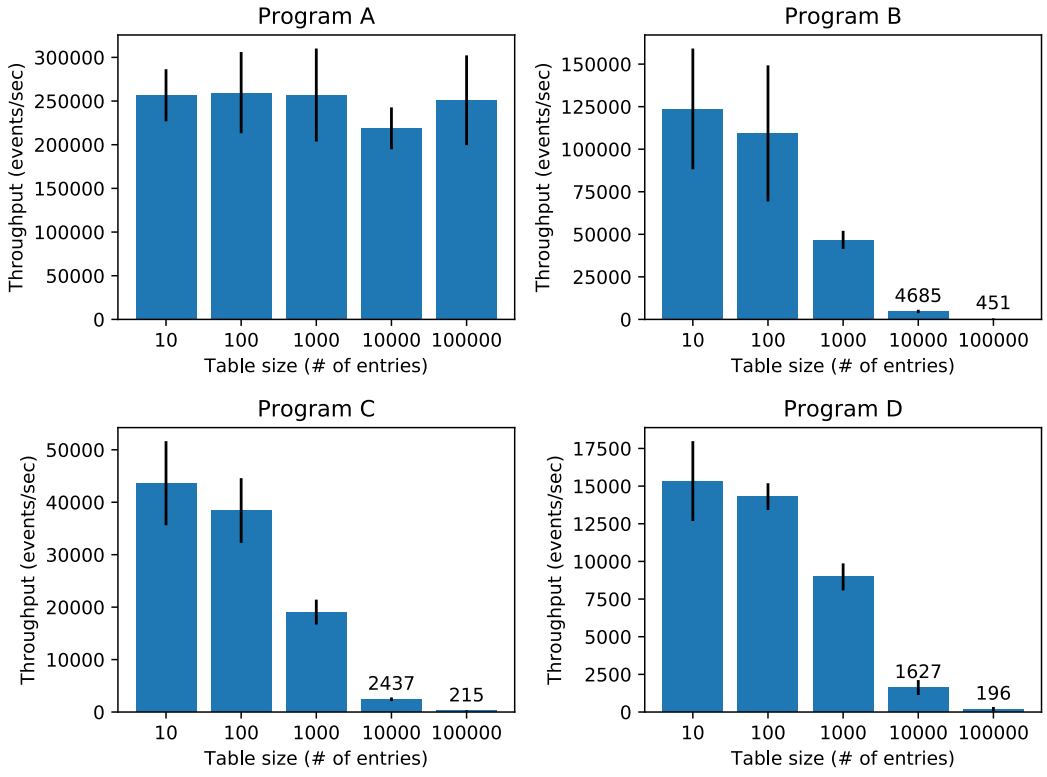


FIGURE 8.1: Throughput for four different restriction-dsl programs with different table sizes.

As expected, program A ran at a constant throughput. At about 250.000 events per second, this is in line with upper-limits that we've seen in exploratory testing with Akka Streams: a simple source-to-sink setup in Akka Streams showed about the same throughput. This implies that for programs that do not involve joins, one can expect

the same performance as an equivalent program written directly in the underlying technology.

For programs that do involve joins, we clearly see a different result. As table sizes go up, the amount of work necessary to perform streaming joins increases seemingly linearly. At first sight this may be surprising. After all, a standard `InnerJoin` between two streams should exhibit $O(log\ n)$ time complexity for updates. However, both the `wind` and `tide` DSL primitives actually perform a `ForeignJoin` to get their answers (see section 5.2). In turn, our interpretation of ForeignJoin on Akka Streams is still quite naive: lookups on forein keys require a linear traversal of one of the materialized tables. This could potentially be improved a lot by implementing an indexing mechanism as is done in any relational database system.

At the same time, we did not see an immediate need for performance improvements in this case-study. Recounting the use-case for the Port of Rotterdam, materialized tables with more than 1000 vessels would rarely occur. The port accepts about 80 seagoing vessels a day. Assuming an average stay of four days there would be around 320 vessels materialized at any point. When all vessels transmit an update every minute, this would result in about 30 incoming events per second (also including events from weather/tide sensors) For the benchmarked programs, that means the system can sustain a much larger load than required for the Port of Rotterdam. With these results it would be feasible to run a deployment for multiple large harbors with many more rules and vessels without any optimisations necessary.

Chapter 9

Future work

Several aspects of this work could be extended or improved upon. Starting with the restrictions DSL, our approach to designing the DSL has been to provide an intuitive interface to programmers. While we would consider the rule language intuitive to use, and all developers that have interacted with it shared this opinion, the rules in the end exist as Scala-code. While this works well with developers, domain experts without programming experience generally had a harder time. The necessity of an IDE and using the Scala compiler means that constructing and maintaining rules is currently only accessible to software engineers. Future efforts may therefore focus building an alternative front-end to these rules to allow more personel to contribute to these rules. For example, one could imagine a web-based interface for interactively editing rules in a point-and-click manner.

On the other end of the system, more performance related improvements may be worth considering. Although throughput of the initial system is decent, a growing set of rules will slow the system over time. Strides in performance can be made in three possible ways. First of all, the surface DSL may be optimised. Since this DSL corresponds to a restricted form of propositional logic, any set of rules can be treated as a formula in conjunctive or disjunctive normal form and reduced from there. While CNF- or DNF-minimisation is an NP-complete problem, approximations and heuristics exist that stand a good change of reducing large rule-sets into a minimal equivalent.

Optimisations on the actor-level also provide an avenue for improvement. Execution of multiple operators may for example be performed in a single actor instead of multiple to avoid communication overhead. Additionally, indices may be used to speed up the `ForeignJoin` operator and different table structures may be used depending on the real-time velocity of streams.

Chapter 10

Conclusion

In this thesis we have explored a small algebra that combines streaming operations (e.g.: map, filter, merge) with relational operators (e.g.: inner join, outer join and aggregations). The algebra aims to be a minimal set of operators needed to build stateful streaming programs based on tables.

In a case-study, we have used this algebra to implement an end-to-end rule-based DSL for the maritime sector. The system is a proof-of-concept built for the Port of Rotterdam that can automatic determine whether conditions are safe for vessels to maneuvre through the ports' waterways. Rules written in our DSL compile via our relational streaming algebra, into stream processor. Any time vessel, berth, weather or tide data is updated, the stream processor will automatically re-evaluate the relevant rules.

Furthermore, we show viability of this algebra in two ways.

- Firstly, we review several operators commonly found in streaming and reactive-programming libraries and demonstrate how those would be implemented via our algebra. This shows that the algebra representation is both expressive enough for our particular use-case but is can also support most well-known streaming/reactive functionality.
- Secondly, we provide a translation of our algebra to a small, theoretical actor language (*Act*), showing that our algebra can be implemented on top of both a specific streaming solution (Akka streams) as well as any actor system.

Lastly, a performance evaluation shows that the system has a high enough throughput to facilitate a potential global-scale deployment on commodity hardware.

Bibliography

[1] Daniel Abadi et al. "Aurora: a data stream management system". In: *SIGMOD Conference*. Citeseer. 2003, p. 666.

[2] Daniel J Abadi et al. "The design of the borealis stream processing engine." In: *Cidr*. Vol. 5. 2005. 2005, pp. 277–289.

[3] Gul Agha et al. "Towards a theory of actor computation". In: *CONCUR '92*. Ed. by W.R. Cleaveland. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 565–579. ISBN: 978-3-540-47293-3.

[4] Gul A Agha. *Actors: A model of concurrent computation in distributed systems*. Tech. rep. MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, 1985.

[5] GUL A. AGHA et al. "A foundation for actor computation". In: *Journal of Functional Programming* 7.1 (1997), 1–72. DOI: 10.1017/S095679689700261X.

[6] *Akka Actors*. https://doc.akka.io/docs/akka/current/index.html. [Online; accessed 4-Dec-2019].

[7] *Akka .NET*. https://getakka.net/. [Online; accessed 4-Dec-2019].

[8] *Akka Streams*. https://doc.akka.io/docs/akka/current/stream/index.html. [Online; accessed 4-Dec-2019].

[9] Arvind Arasu, Shivnath Babu, and Jennifer Widom. "The CQL continuous query language: semantic foundations and query execution". In: *The VLDB Journal* 15.2 (2006), pp. 121–142.

[10] Arvind Arasu et al. "Stream: The stanford stream data manager". In: *IEEE Data Eng. Bull.* 26.1 (2003), pp. 19–26.

[11] Joe Armstrong et al. *Concurrent Programming in ERLANG*. 1993.

[12] Brian Babcock et al. "Models and issues in data stream systems". In: *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM. 2002, pp. 1–16.

[13] Shivnath Babu and Jennifer Widom. "Continuous queries over data streams". In: *ACM Sigmod Record* 30.3 (2001), pp. 109–120.

[14] Engineer Bainomugisha et al. "A survey on reactive programming". In: *ACM Computing Surveys (CSUR)* 45.4 (2013), p. 52.

[15] Henry G Baker Jr. *Actor Systems for Real-Time Computation*. Tech. rep. MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE, 1978.

[16] Daniel Barbará. "The characterization of continuous queries". In: *International Journal of Cooperative Information Systems* 8.04 (1999), pp. 295–323.

[17] Philip A Bernstein et al. "Orleans: Distributed virtual actors for programmability and scalability". In: *MSR-TR-2014–41* (2014).

[18] Hans Kleine Büning and Theodor Lettmann. *Propositional logic: deduction and algorithms*. Vol. 48. Cambridge University Press, 1999.

[19] Paris Carbone et al. "Apache flink: Stream and batch processing in a single engine". In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015).

[20] Sirish Chandrasekaran and Michael J Franklin. "Streaming queries over streaming data". In: *Proceedings of the 28th international conference on Very Large Data Bases*. VLDB Endowment. 2002, pp. 203–214.

[21] Sirish Chandrasekaran et al. "Telegraphcq: Continuous dataflow processing for an Uncertain world." In: *Cidr*. Vol. 2. 2003, p. 4.

[22] Jianjun Chen et al. "NiagaraCQ: A scalable continuous query system for internet databases". In: *ACM SIGMOD Record*. Vol. 29. 2. ACM. 2000, pp. 379–390.

[23] Edgar F Codd. "A relational model of data for large shared data banks". In: *Communications of the ACM* 13.6 (1970), pp. 377–387.

[24] Olivier Coudert. "Two-level logic minimization: an overview". In: *Integration* 17.2 (1994), pp. 97–140.

[25] Chuck Cranor et al. "Gigascope: a stream database for network applications". In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM. 2003, pp. 647–651.

[26] Alan L Davis and Robert M Keller. "Data flow program graphs". In: (1982).

[27] Jack B Dennis. "First version of a data flow procedure language". In: *Programming Symposium*. Springer. 1974, pp. 362–376.

[28] *Elixir language*. https://elixir-lang.org/. [Online; accessed 4-Dec-2019].

[29] Conal Elliott. "An embedded modeling language approach to interactive 3D and multimedia animation". In: *IEEE transactions on software engineering* 25.3 (1999), pp. 291–308.

[30] Conal Elliott, Sigbjørn Finne, and Oege De Moor. "Compiling embedded languages". In: *Journal of functional programming* 13.3 (2003), pp. 455–481.

[31] KAHN Gilles. "The semantics of a simple language for parallel programming". In: *Information processing* 74 (1974), pp. 471–475.

[32] Lukasz Golab and M Tamer Özsu. "Issues in data stream management". In: *ACM Sigmod Record* 32.2 (2003), pp. 5–14.

[33] Carl Hewitt. "Viewing control structures as patterns of passing messages". In: *Artificial intelligence* 8.3 (1977), pp. 323–364.

[34] Se June Hong, Robert G. Cain, and Daniel L. Ostapko. "MINI: A heuristic approach for logic minimization". In: *IBM journal of Research and Development* 18.5 (1974), pp. 443–458.

[35] Paul Hudak. "Modular domain specific languages and tools". In: *Proceedings. Fifth International Conference on Software Reuse (Cat. No. 98TB100203)*. IEEE. 1998, pp. 134–142.

[36] Hojjat Jafarpour, Rohan Desai, and Damian Guy. "KSQL: Streaming SQL Engine for Apache Kafka." In: *EDBT*. 2019, pp. 524–533.

[37] Roland Kuhn Jonas Bonér Dave Farley and Martin Thompson. *The Reactive Manifesto*. https://www.reactivemanifesto.org/. [Online; accessed 4-Dec-2019].

[38] Gilles Kahn and David MacQueen. "Coroutines and networks of parallel processes". In: (1976).

[39] Jay Kreps, Neha Narkhede, Jun Rao, et al. "Kafka: A distributed messaging system for log processing". In: *Proceedings of the NetDB*. 2011, pp. 1–7.

[40] Edward A Lee, Stephen Neuendorffer, and Michael J Wirthlin. "Actor-oriented design of embedded hardware and software systems". In: *Journal of circuits, systems, and computers* 12.03 (2003), pp. 231–260.

[41] Ling Liu, Calton Pu, and Wei Tang. "Continual queries for internet scale event-driven information delivery". In: *IEEE Transactions on Knowledge and Data Engineering* 11.4 (1999), pp. 610–628.

[42] Olaf Merk. "Shipping Emissions in Ports". In: (2014). DOI: https://doi.org/https://doi.org/10.1787/5jrw1ktc83r1-en. URL: https://www.oecd-ilibrary.org/content/paper/5jrw1ktc83r1-en.

[43] Olaf Merk. "The competitiveness of global port-cities: synthesis report". In: (2013).

[44] Martin Odersky et al. *An overview of the Scala programming language*. Tech. rep. 2004.

[45] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[46] Kostas Patroumpas and Timos Sellis. "Window specification over data streams". In: *International Conference on Extending Database Technology*. Springer. 2006, pp. 445–464.

[47] Willard V Quine. "The problem of simplifying truth functions". In: *The American mathematical monthly* 59.8 (1952), pp. 521–531.

[48] *ReactiveX - An API for asynchronous programming with observable streams*. http://reactivex.io. [Online; accessed 4-Dec-2019]. 2014.

[49] *RxJava GitHub repository*. https://github.com/ReactiveX/RxJava. [Online; accessed 4-Dec-2019].

[50] Matthias J Sax et al. "Streams and tables: Two sides of the same coin". In: *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics*. ACM. 2018, p. 1.

[51] Anthony M Sloane. "Lightweight language processing in Kiama". In: *International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer. 2009, pp. 408–425.

[52] Douglas Terry et al. *Continuous queries over append-only databases*. Vol. 21. 2. ACM, 1992.

[53] Christopher Umans. "The minimum equivalent DNF problem and shortest implicants". In: *Journal of Computer and System Sciences* 63.4 (2001), pp. 597–611.

[54] Eelco Visser. "Program transformation with Stratego/XT". In: *Domain-specific program generation*. Springer, 2004, pp. 216–238.