Patterns for Stream Processing in a Functional Programming Language

Draft 0.2

# Introduction

The map-reduce functional programming pattern has dominated large-scale data analytics for the past decade. This combination of two common functional programming operations has proved useful across a wide range of problem domains. There are several reasons for this:

* It’s a simple pattern that is easily understood by users
* It has wide applicability
* Programmers just have to write the map and reduce functions. All the complexity of building a high performance, scalable, dependable to execute those functions on vast amounts of data is done by underlying infrastructure: the user does not need to concern themselves with how this infrastructure is implemented, nor even understand how it works.

As streaming data becomes increasingly important, it is worth asking whether there is a simple functional pattern or patterns that are appropriate to event based data processing. This paper introduces a possible set of patterns to achieve this. Our design goals were:

* Simplicity for programmers:
  + they should focus on high-level operations on the streaming data, not on the infrastructure that supports those operations
  + they should program in the “pure” core of functional programming
* Distribution: the functionality in IoT applications is inherently distributed (e.g. across sensors, field gateways and clouds). It must be possible for the user to write IoT applications that can exploit this.
* Non-functional requirements should be provided by the underlying infrastructure with no, or as little as possible, explicit programming by the user (as Hadoop does for map-reduce). This includes:
  + exploiting parallelism to scale performance
  + security
  + dependability

In the rest of this paper, we describe the overall approach, and the patterns that fit within it: we use the Haskell functional language is illustrate our approach. The approach is illustrated with a set of examples. We then discuss in high-level terms how the non-functional requirements can be addressed.

# Modelling Streaming Data

We model a stream as an infinite list of events. Each event may contain a timestamp:

data Event alpha = E {time::Timestamp, value::alpha} |

V { value::alpha} |

T {time::Timestamp}

deriving (Eq, Ord)

type Timestamp = UTCTime

type Stream alpha = [Event alpha]

The “T” form of Event is used for time windowing as will be described later).

Note that as Event is defined as a polymorphic type, a stream can consist of data of any type (e.g. integers, strings, tuples, lists, trees, graphs and even functions). An important principle is that separation between the view of the system seen by the user and the infrastructure: the details of the processing infrastructure are hidden allowing for multiple different implementations, and the addition of functionality to transparently support non-functional requirements. The user only sees a simple view of a stream as a list of type alpha:

[alpha]

In the rest of the paper we first describe the patterns as seen by the user for stream processing. We then describe the infrastructure used to execute them.

# Processing Patterns

By analysis of the Complex Event Processing literature [?], there are 6 main tasks any system must perform:

* Filter (select only those events that match a criterion)
* Map (transform all events into another type of event)
* Aggregate (break a stream of events into windows; combine all events in each window)
* Merge (combine streams of the same type)
* Join (combine streams of different types)

We now define patterns (implemented as Haskell functions) that users can use to perform these operations.

## Filter

Users use the streamFilter function which takes a stream as input, and generates a stream containing only those events that meet a user-provided criteria. The type signature is:

streamFilter:: EventFilter alpha -> -- The function applied to each

-- input event to determine if it

-- meets the criteria

Stream alpha -> -- The input stream

Stream alpha -- The output stream

type EventFilter alpha = alpha -> Bool -- type of user-provided function

This illustrates the fact that the user does not need to know or understand the exact format of the events as seen by the infrastructure: they only need provide a function of type EventFilter to filter out all events whose value does not meet a particular criterion. This user-provided function is applied by the infrastructure to each the value in each element of the stream, and only those that return “True” are selected.

To show how filterStream can be used, assume a temperature sensor generates a stream of events of type Int.

tempSensor:: Stream Int

If we are interested only in temperatures of over 100 then the user can define a function:

over100:: EventFilter Int

over100 temp = temp>100

and stream can be filtered:

filterStream over100 tempSensor

Alternatively, the user may choose to use lambda notation to provide an unnamed function, and so the filter can also be written as:

filterStream (\temp->temp>100) tempSensor

## Filtering with an Accumulating Parameter

Sometimes, it is necessary to filter events based partly on past results. This can be done using the function streamFilterAcc whose signature is as follows:

streamFilterAcc:: beta -> -- initial accumulator value

(alpha -> beta -> beta) -> -- the accumulator fn:

-- takes head of stream &

-- accumulator and computes

-- the new accumulator

(alpha -> beta -> Bool) -> -- the filter fn: takes head

-- of stream & accumulator

Stream alpha -> -- input stream

Stream alpha -- output stream

An example of its use is where we only want to propagate an event if its value is different to the previous event (though the timestamp may differ). This can be written:

changes:: Eq alpha=> Stream alpha -> Stream alpha

changes s = streamFilterAcc (value $ head s)

(\h acc-> if (h==acc) then acc else h)

(\h acc->(h/=acc))

(tail s)

## Mapping Streams

The function mapStream is used to transform the values in a stream. The user supplies a function of type EventMap which is applied to the value within each event in the input stream to generate the output stream. The function streamMap has the signature:

streamMap:: EventMap alpha beta -> -- the user-supplied map function

Stream alpha -> -- input stream

Stream beta -- output stream

type EventMap alpha beta = alpha -> beta -- type of the

-- user-supplied map function

To illustrate this with the running example, let us say that after filtering all temperatures over 100, the user wants to use 100 as the baseline temperature, and represent all temperatures as their value over 100. To do this we can define a function:

amountOver100:: EventMap Int Int

amountOver100 temp = temp-100

and then include this in the application:

mapStream amountOver100 $ filterStream over100 tempSensor

again, this can also be written using lambda functions:

mapStream (\temp->temp-100) $ filterStream (\temp->temp>100) tempSensor

Note that the user does not have to understand the format of events, nor how map or filter are enacted. Instead they can focus solely on how the data in the events is to be processed.

## One to Many Mapping

Sometimes, mapping may generate multiple events from each input event. A function provided for this is:

type EventMap1toN alpha beta = alpha -> [beta]

streamMap1toN:: EventMap1toN alpha beta -> -- the user-provided 1 to N

-- mapping function

Stream alpha -> -- input stream

Stream beta -- output stream

An example is a function that takes in a stream of Twitter tweets, and generates a stream of the hashtags found in those tweets. Assume a function getHashtags with signature:

getHashtags:: String -> [String]

This function takes as input a tweet and outputs a list of all the hashtags contained within it.

This could then be used to generate the stream of hashtags:

streamMap1toN getHashtags s

## Map with an Accumulating Parameter

There is sometimes the need for a map function which can take into account previous events.

streamMapAcc:: beta -> -- initial accumulator value

(alpha -> beta -> beta) -> -- the accumulator fn:

-- takes head of stream &

-- accumulator and computes

-- the new accumulator

(alpha -> beta -> beta) -> -- the map fn:

-- takes head of stream &

-- accumulator and computes

-- the output event value

Stream alpha -> -- input stream

Stream beta -- output stream

An example is emitting an event giving a count of the number of events received so far in the stream:

counter:: Stream alpha -> Stream Int

counter s = streamMapAcc 0 (\h count-> count+1) (\h acc -> acc) s

## Aggregation

Windowing gives a way to aggregate a sub-stream of events. A stream of events is broken into windows; and a function then combines all events in each window. Experience shows that it is useful in many stream applications, and as a result, most CEP systems support a range of types of windowing.

The signature of the function provided for windowed stream processing is:

streamWindowAggregate:: WindowMaker alpha -> -- turns stream

-- into stream of

-- windows

WindowAggregator alpha gamma -> -- aggregates

-- window into

-- single value

Stream alpha -> -- input stream

Stream gamma -- output stream

type WindowMaker alpha = Stream alpha -> [Stream alpha]

type WindowAggregator alpha beta = [alpha] -> beta

WindowMaker is a function that generates windows to be aggregated. To save the user from having to define this function we provide a pre-defined set of functions covering the common cases. However, users are free to define their own windowing functions if they require an alternative, specialist window creator. Examples of pre-defined functions are:

-– create sliding windows that are a fixed number of events in length

sliding:: Int -> WindowMaker alpha

-– create sliding windows that are a fixed time length

slidingTime:: NominalDiffTime -> WindowMaker alpha

-– create non-overlapping windows that are a fixed # of events in length

chop:: Int -> WindowMaker alpha –- creates non-overlapping windows

-– create non-overlapping windows that are a fixed time length

chopTime:: NominalDiffTime -> WindowMaker alpha

The user-defined function WindowAggregator takes a list of the values contained in a window of events, and aggregates them into a single value.

As an example, consider an extension to the running example in which we in each want to know how may times in each hour the sensor has measured a temperature over 100. This can be done using the function:

streamWindowAggregate (chopTime 3600) length

$ filterStream over100 tempSensor

Here, length is the standard Haskell function that returns the length of a list.

## Merge

Sometimes there is a need to merge multiple streams of the same type into one. This is done with the mergeStreams function.

mergeStreams:: [Stream alpha]-> Stream alpha

For example, if there are three temperature sensors then we can combine them as follows:

mergeStreams [tempSensor1,tempSensor2,tempSensor3]

In the running example, the merged stream can then be used as input to one of the previously described stream processing functions, e.g.:

streamWindowAggregate (chopTime 3600) length

$ filterStream over100

$ mergeStreams [tempSensor1,tempSensor2,tempSensor3]

The infrastructure will automatically order the input streams by timestamp so that temporal order is preserved (see the discussion of Time handling in Section ???)

## Join

The join pattern is used to combine two streams of different types. Two different types of Join are provided. The first - joinStreamsE – has the signature:

type JoinFilter alpha beta = alpha -> beta -> Bool

type JoinMap alpha beta gamma = alpha -> beta -> gamma

joinStreamsE:: Stream alpha -> -- 1st input stream

WindowMaker alpha -> -- create windows from stream 1

Stream beta -> -- 2nd input stream

WindowMaker beta -> -- create windows from stream 2

JoinFilter alpha beta -> -- determines if this pair of

-- values meets join criteria

JoinMap alpha beta gamma -> -- combines the pair of values

-- into the output event

Stream gamma -- the output stream

The user provides a WindowMaker function for each of the two input streams. The Cartesian product of the values in the events in each window is then formed. The resulting pairs of values are then filtered using the user-defined JoinFilter function to determine if the pair of values meets the join criteria. A user-defined map function – JoinMap – is then applied to each conforming pair and the result appears on the output stream.

For example, if our running example temperature sensors are upgraded to ones which give a wider range of data, including location and carbon dioxide:

data EnvSensor = EnvSensor {loc::Location,temp::Int,co2::Int}

And are joined by another set of sensors with traffic data:

data Traffic = Traffic {loc::Location,load::Int}

where load is a measure of the volume of traffic per second. Then, we may want to understand the correlation between load and CO2 levels every minute. As a first step we can combine data at all the locations where there are both types of sensor.

We can do this using the following function (where envSensors is the merge of the data from all the environmental sensors, and trafficSensors is the merge of the data from all the traffic sensors):

joinSreamsE envSensors (chopTime 60)

trafficSensors (chopTime 60)

(close 50)

(\(loc1,temp1,co2level)(loc2,ld) -> (loc1,temp1,ld,c02level))

Here close is a function that is true if the locations given by its second and third parameters are within a number of meters specified by its first parameter. Its signature is:

close:: Int -> Location -> Location -> Bool

We also provide another join function – joinStreamsW - that does not perform a cartesian product of the events in the two windows that are being joined. Instead, it operates on whole windows, applying a user defined function to the two windows of data taken from the two input streams.

It is defined as:

joinStreamsW:: Stream alpha -> -- 1st input stream

WindowMaker alpha -> -- create windows from stream 1

Stream beta -> -- 2nd input stream

WindowMaker beta -> -- create windows from stream 2

([alpha]->[beta]->gamma) -> -- combines the two windows

-- into the output event

Stream gamma -- the output stream

## Summary

|  |  |
| --- | --- |
| **Stream Processing** | |
| **Category** | **Function** |
| Filter | streamFilter |
| streamFilterAcc |
| Map | streamMap |
| streamMapAcc |
| streamMap1toN |
| Aggregate sub-stream | streamWindowAggregate |
| Combine Streams | mergeStreams |
| joinStreamsE |
| joinStreamsW |