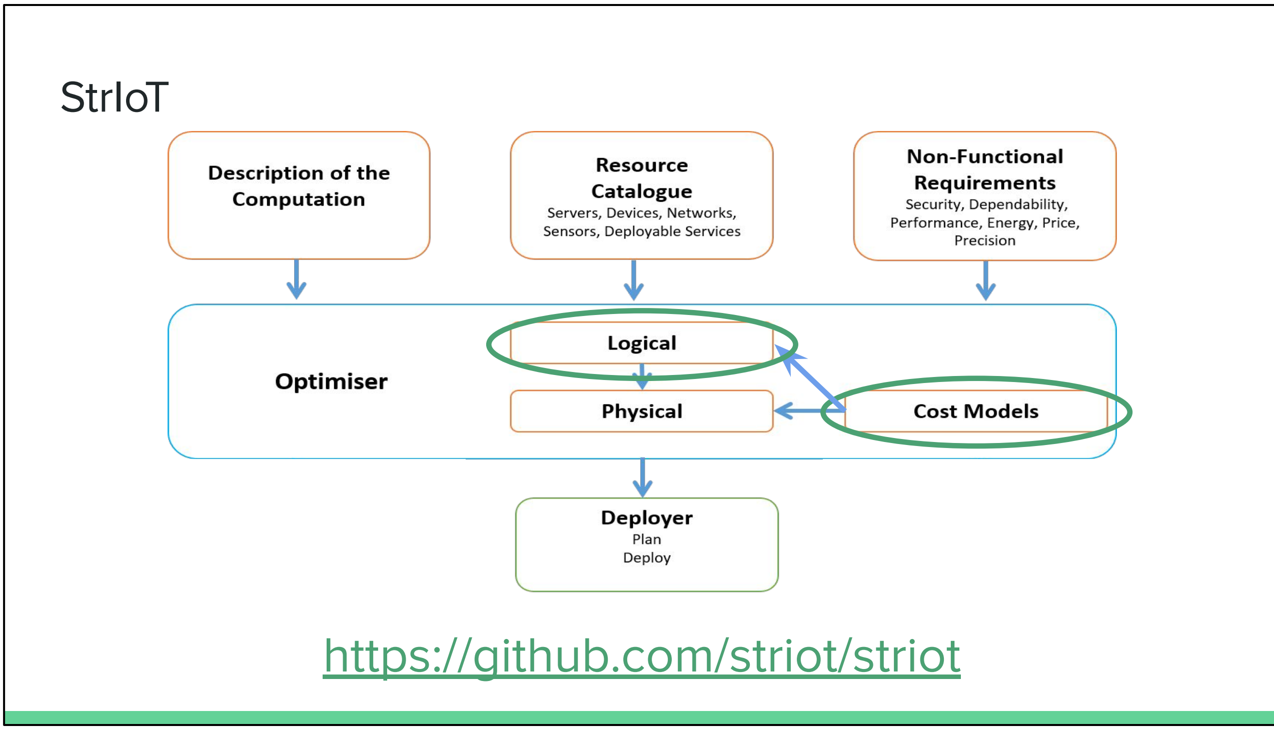
Picking a winner
cost models for evaluating stream-processing programs

Jonathan Dowland <jon.dowland@ncl.ac.uk>
UK Systems ‘21

Jon Dowland
p/t PhD student year now in 5th year



Purely-functional stream-processing system
Implemented in Haskell, open source
User composes program from a fixed set of functional operators
Without considering deployment issues (a single contiguous program)
Optimisers re-write the program to perform better according to NFRs

Logical optimiser and cost models my focus

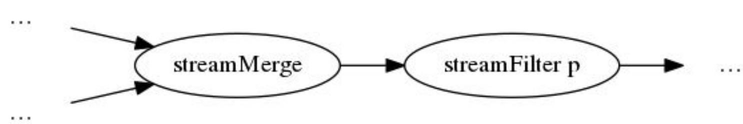
StrIoT operators

Filter	streamFilter	$\alpha \rightarrow \alpha$
	streamFilterAcc	$\alpha \rightarrow \alpha$
Map	streamMap	$\alpha \rightarrow \beta$
	streamScan	$\alpha \rightarrow \beta$
Window	streamWindow	$\alpha \rightarrow [\alpha]$
	streamExpand	$[\alpha] \rightarrow \alpha$
Combine	streamMerge	$[\alpha] \rightarrow \alpha$
	streamJoin	$\alpha \rightarrow \beta \rightarrow (\alpha,\beta)$

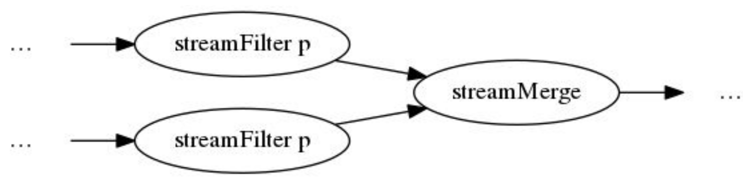
4 classes of operators; 8 operators total; some inverses of others eg window/expand
Simplified types: just in and outs here

(highlight: merge, filter)

Logical Optimiser: term-rewriting




```
streamFilter p (streamMerge [s1,s2…])
  = streamMerge [ streamFilter p s1,
                  streamFilter p s2, … ]
```



Since the input program is a pure-functional program we can use equational reasoning and term rewriting

A set of **21** semantically-preserving rewrite rules (and a further **6** semantically-altering)

Derived by pair-wise comparison of the operators

Example rule: filter hoisting

Rewrite rule implementation

```
-- streamFilter f >>> streamFilter g = streamFilter (\x -> f x && g x)

filterFuse :: RewriteRule
filterFuse (Connect (Vertex a@(StreamVertex _ Filter (p:_) _ _))
                    (Vertex b@(StreamVertex _ Filter (q:_) _ _))) =

let c = a { parameters = [[| (\p q x -> p x && q x) $(p) $(q) |]] }

in Just (removeEdge c c . mergeVertices (`elem` [a,b]) c)
```

A happy accident: it was possible to implement rewrite rules as plain functions
The left-hand side as a pattern-match, due to the choice of graph library we use
Andrey Mokhov

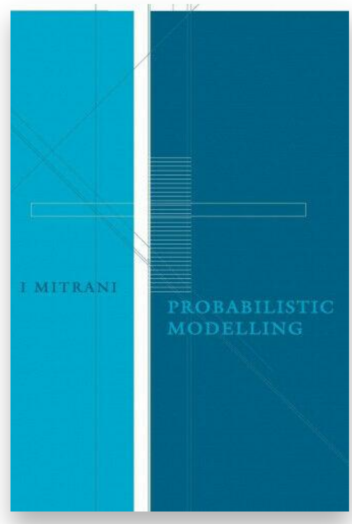
Cost models for evaluation

We can generate program variants with rewrite rules
We need a way of determining which variant is best

Queuing system model

Mitrani, I. (1997). *Probabilistic Modelling*. Cambridge: Cambridge University Press. doi:10.1017/CBO9781139173087.001

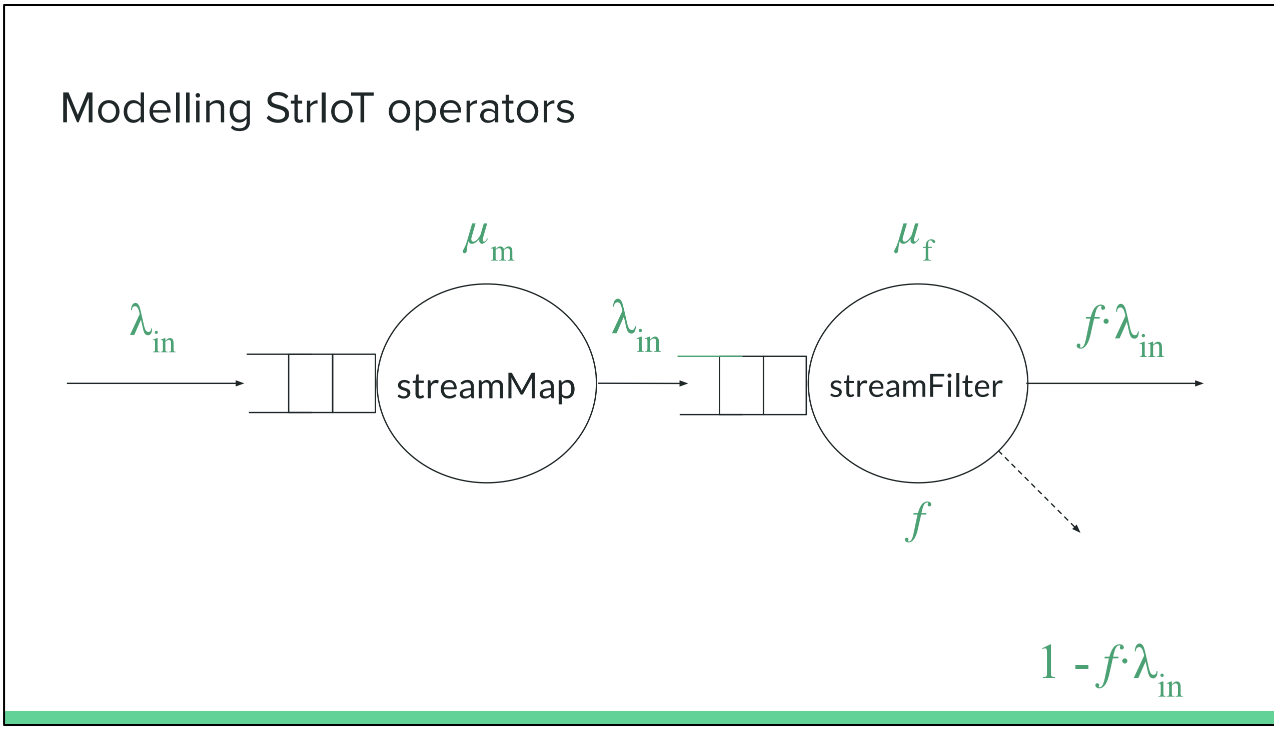
Utilisation (ρ) = arrival rate (λ) / service rate (μ)



I'm exploring representing a StrIoT program as a queuing system
Working with Dr Paul Ezhilchelvan and Emeritus Prof. Isi Mitrani
Outside my comfort zone
The holy grail for me has been Isi Mitrani's book

For what I'm going to show today key formula is utilisation

Steady state



StrIot operators map to “servers” in queuing theory parlance. We define some additional metadata to represent parameters for the model:

- For each operator instance we define a (mean avg.) service rate: how fast that operator can process events
- We model (mean avg.) arrival rates into the program. Note that arrival rates are not influenced by service rates, so the rate out of that map matches the rate in
- To model the filter rejecting events, we define a selectivity and route the rejected events out of the stream

Encoding queueing theory properties

```
data StreamVertex = StreamVertex
{ vertexId :: Int
, operator :: StreamOperator
, parameters :: [ExpQ]
, intype :: String
, outtype :: String
                       }

data StreamOperator = Map | Filter
| Expand | Window | Merge | Join | Scan
| FilterAcc
| Source
| Sink deriving (Show,Ord,Eq)
```

Straightforward to extend data types with Queuing theory parameters
Before
after

Re-write rules and queueing theory

```
-- streamFilter f >>> streamFilter g = streamFilter (\x -> f x && g x)

filterFuse :: RewriteRule
filterFuse (Connect (Vertex a@(StreamVertex _ (Filter sel1) (p:_) _ _ s1))
                    (Vertex b@(StreamVertex _ (Filter sel2) (q:_) _ _ s2))) =

let c = a { operator = Filter (sel1 * sel2)
          , parameters = [[| (\p q x -> p x && q x) $(p) $(q) |]]
          , serviceTime = s1 + (sel1 * s2) }

in Just (removeEdge c c . mergeVertices (`elem` [a,b]) c)
```

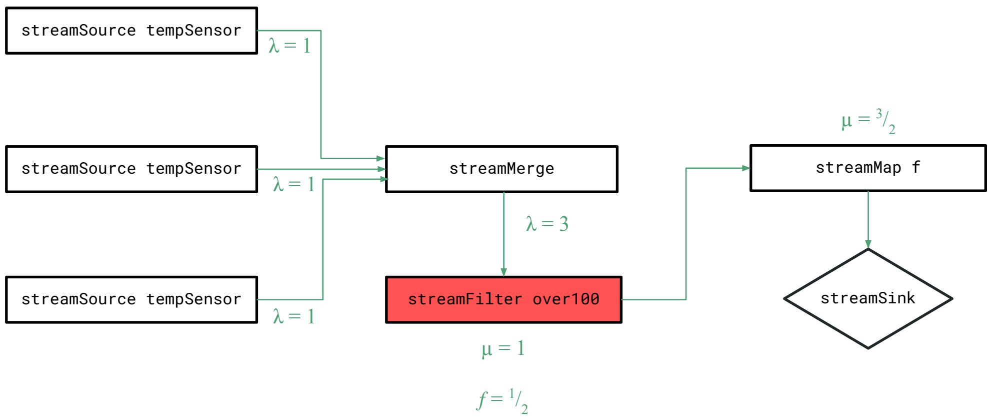
And extending re-write rules similarly straightforward
Highlighted section new

Example outcome #1 of 3

Reject over-utilised operators

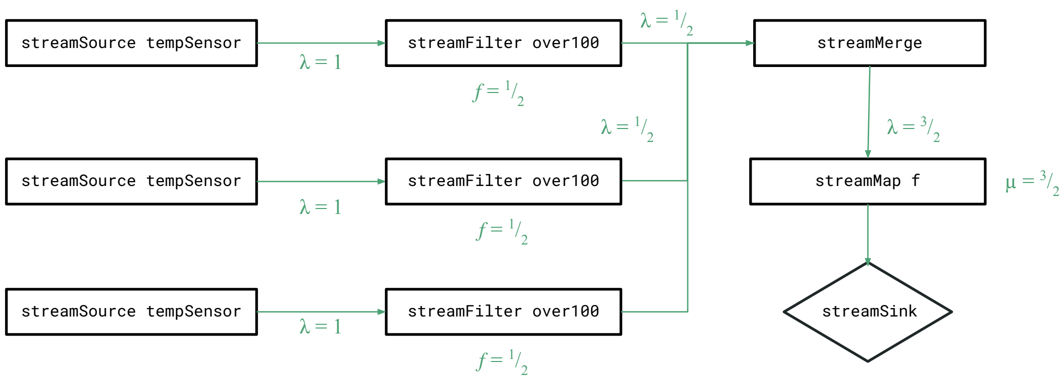
The first example of what we can do is at the operator level

Input program: over-utilised operator



ANIMATION
Here the filter operation is determined to be overutilised
This would be ruled out by the cost model

Re-written program: no over-utilised operators



Re-write rules applied,
Several program variants derived (how many?)
One of the program variants generated by the logical optimiser results in no overutilisation

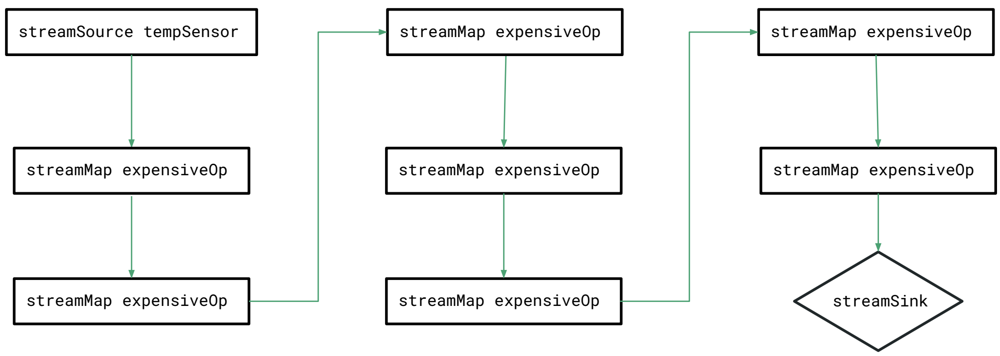
Example outcome #2 of 3

Discard plans with Nodes above a utilisation threshold

The next two examples are at the Node level in a deployment

Reduce the number of nodes needed for deployment by hoisting a map upstream to the Edge, increasing the utilisation of Edge nodes

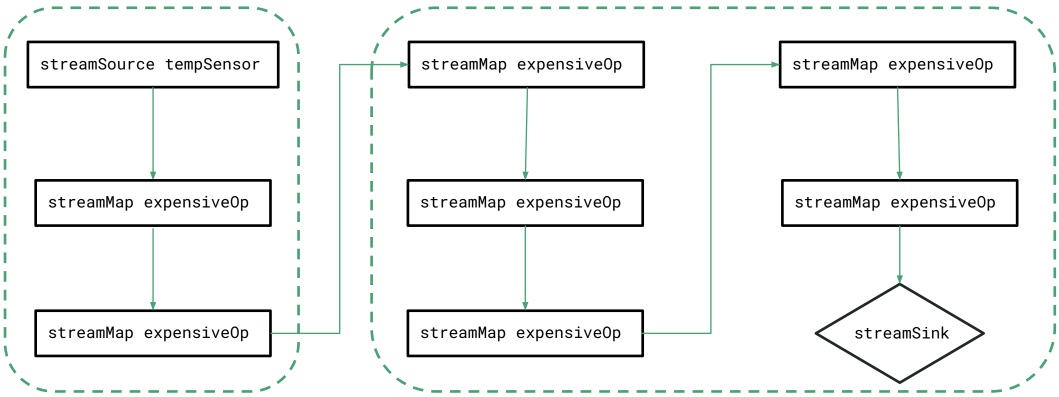
Input program



7 expensive operations (each $\rho = 1$)

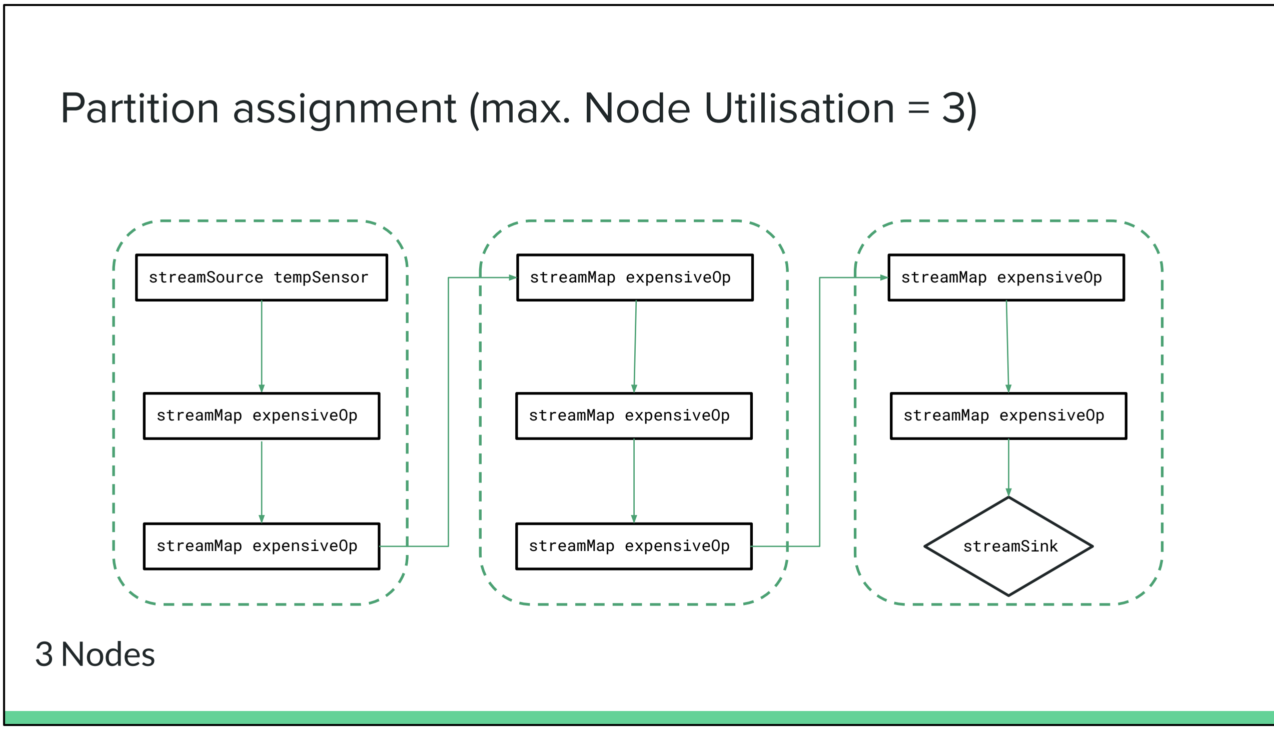
A series of expensive operations each rho = 1

Partition assignment (no max. Node Utilisation threshold)



2 Nodes

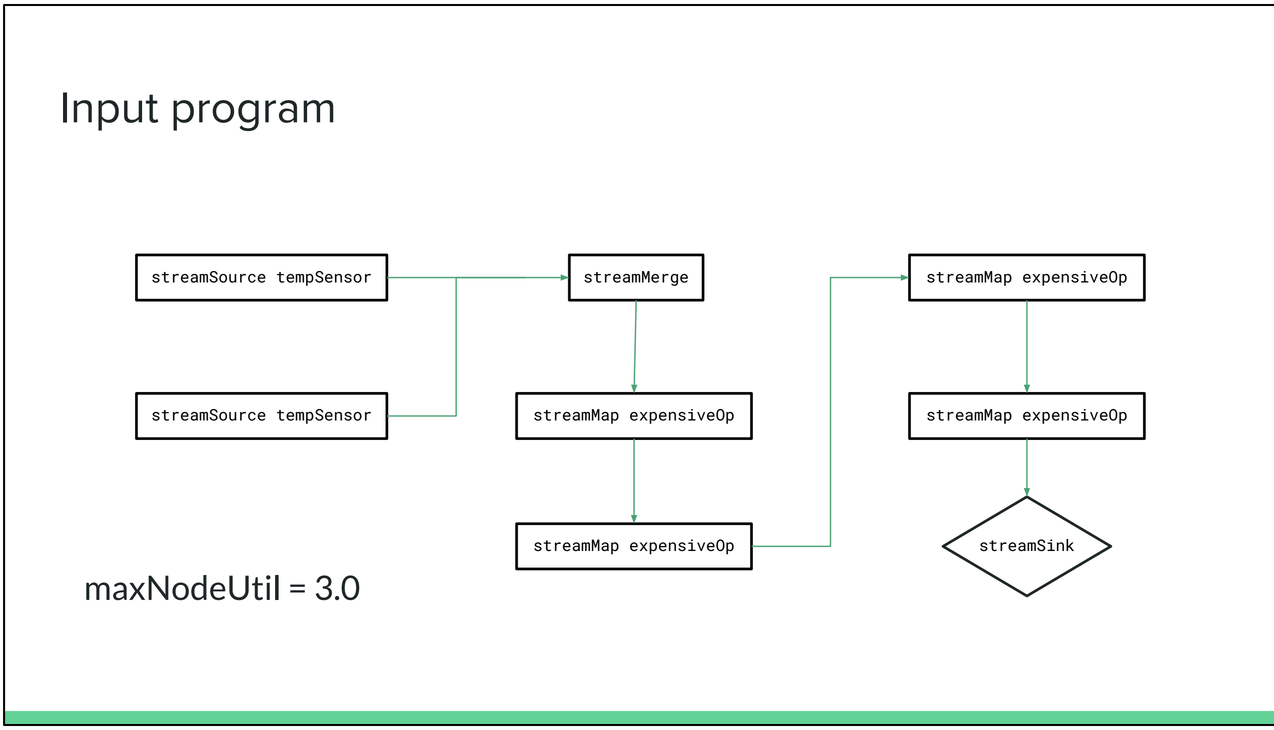
Without no max node utilisation specified, the cost model would choose a partitioning scheme that allocated 2 nodes



Considering max node utilisation specified as 3, so no more than 3 of the expensive operations can be allocated to a single node, the smallest viable partitioning scheme becomes 3 nodes (and is picked by the cost model)

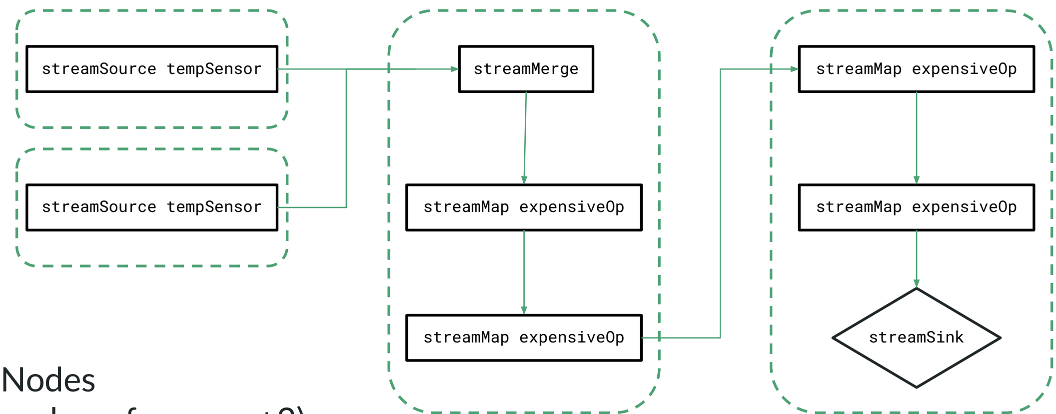
Example outcome #3 of 3

Reduce required Cloud nodes by increasing Edge utilisation



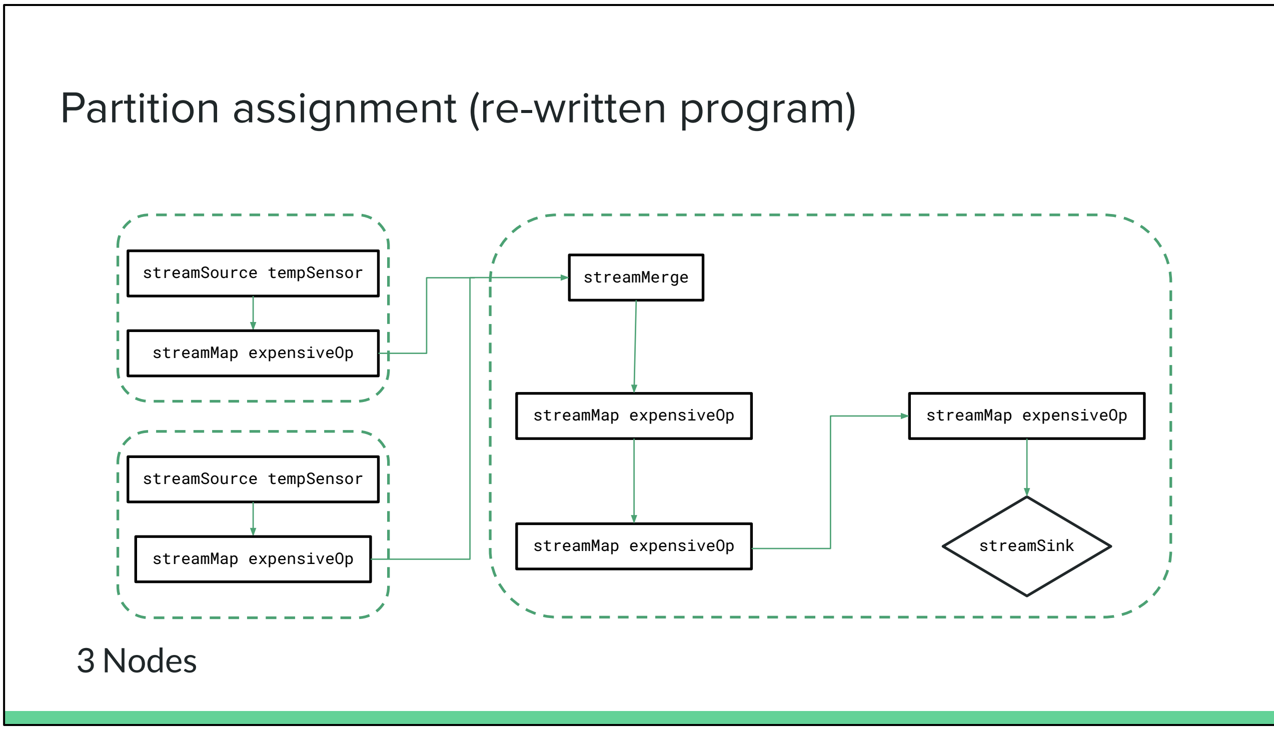
Here consider a program with a string of expensive operations which would again force a partitioning plan with more “cloud” nodes than desired

Partition assignment (input program)



4 Nodes
(number of sources+2)

Considering that limitation, the smallest number of nodes in a plan is four: the sources have to be separate; then a maximum of three of the Maps per cloud node



However the logical optimiser had produced a derivative program which hoisted one of the map operations upstream to the “edge” nodes. This increased their utilisation (within the limit) but reduced the number of nodes requires in a partitioning plan (corresponding to fewer “cloud” nodes needed)

Future work

- Heterogeneous nodes
  - (capabilities, limitations, costs...)
- Non-functional requirements
  - Bandwidth
- Further modelling work
- Operator semantics (streamWindow)
- quickSpec - machine-assisted law discovery

Thank you!

Q&A

Jonathan Dowland <jon.dowland@ncl.ac.uk>
UK Systems ‘21