Verified Time-Aware Stream Processing

Rafael Castro G. Silva

rasi@di.ku.dk

Department of Computer Science
University of Copenhagen

04/06/2024

Introduction

Time-Aware Stream Processing

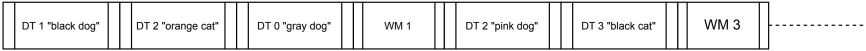
- What is Time-Aware Stream Processing?

Time-Aware Stream Processing

- What is Time-Aware Stream Processing?
  - Stream Processing: programs that compute unbounded sequences of data (streams)

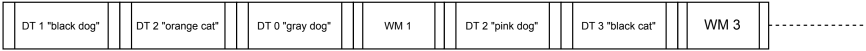
Time-Aware Stream Processing

- What is Time-Aware Stream Processing?
  - Stream Processing: programs that compute unbounded sequences of data (streams)
  - Time-Aware: Data has timestamp metadata; timestamps are bounded by watermarks
    - Timestamp: A partially-ordered value associated with the data (e.g., unix-time, int, pairs of ints, etc...)
    - Watermark: A value of the same type of the timestamp. Represents data-completeness.



Time-Aware Stream Processing

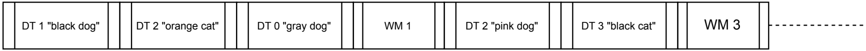
- What is Time-Aware Stream Processing?
  - Stream Processing: programs that compute unbounded sequences of data (streams)
  - Time-Aware: Data has timestamp metadata; timestamps are bounded by watermarks
    - Timestamp: A partially-ordered value associated with the data (e.g., unix-time, int, pairs of ints, etc...)
    - Watermark: A value of the same type of the timestamp. Represents data-completeness.



- Asynchronous Dataflow Programming: Directed graph of interconnected operators that perform event-wise transformations
- E.g.: Apache Flink, Apache Samza, Apache Spark, Google Cloud Dataflow, and Timely Dataflow

Time-Aware Stream Processing

- What is Time-Aware Stream Processing?
  - Stream Processing: programs that compute unbounded sequences of data (streams)
  - Time-Aware: Data has timestamp metadata; timestamps are bounded by watermarks
    - Timestamp: A partially-ordered value associated with the data (e.g., unix-time, int, pairs of ints, etc...)
    - Watermark: A value of the same type of the timestamp. Represents data-completeness.



- Asynchronous Dataflow Programming: Directed graph of interconnected operators that perform event-wise transformations
- E.g.: Apache Flink, Apache Samza, Apache Spark, Google Cloud Dataflow, and Timely Dataflow

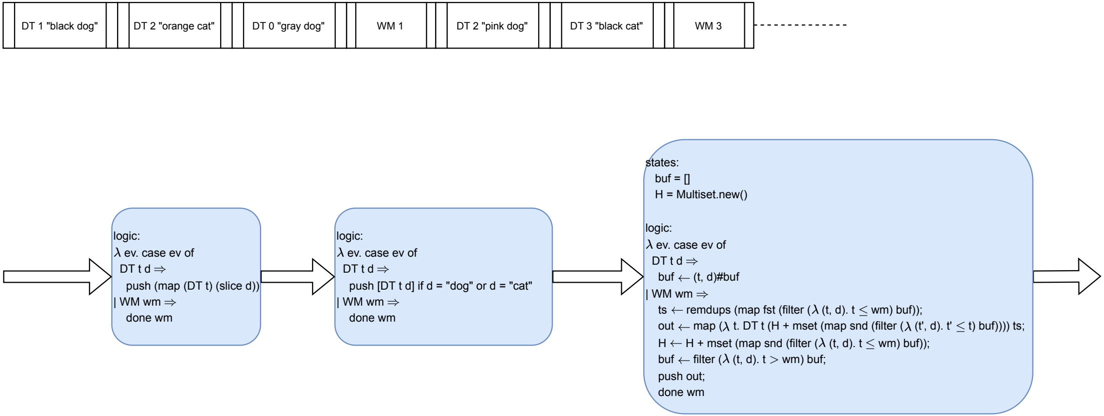
Cloud DataFlow Flink

- Why?
  - Highly Parallel
  - Low latency (output as soon as possible)
  - Incremental computing

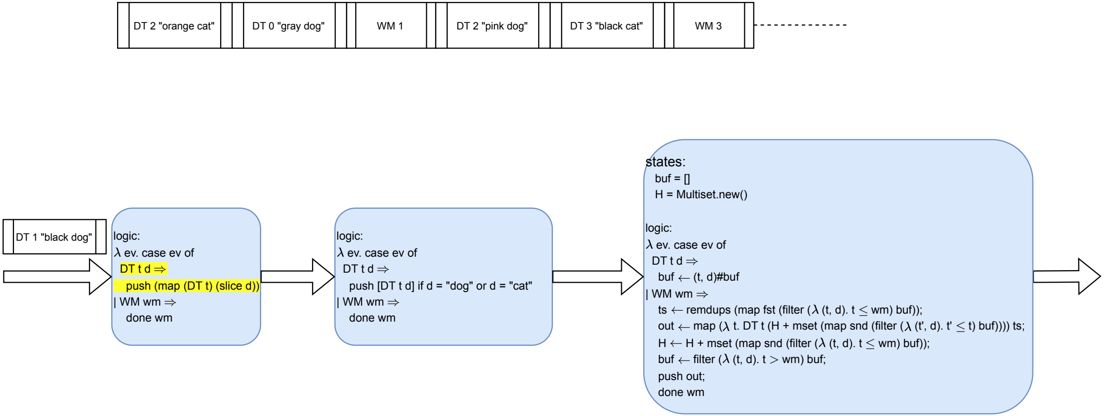
Time-Aware Stream Processing Example

Example:
Incrementally count the occurrences of the words “dog” and “cat”

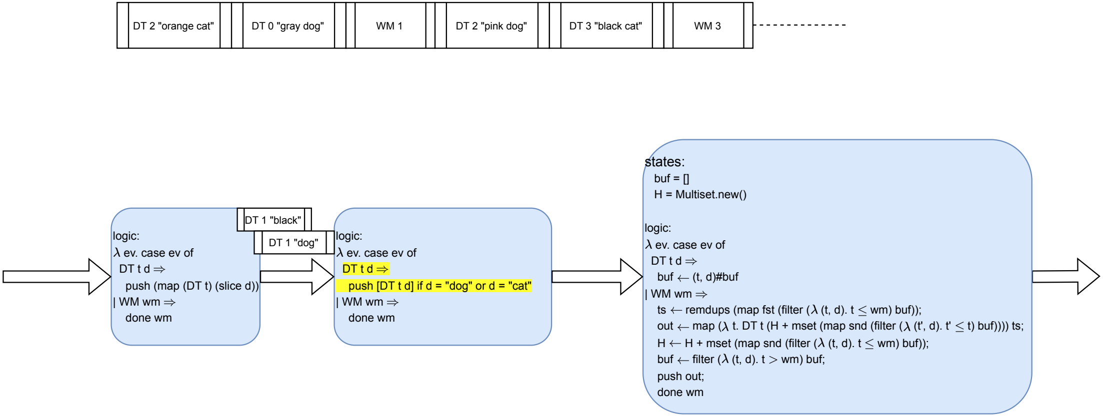
Time-Aware Stream Processing Example



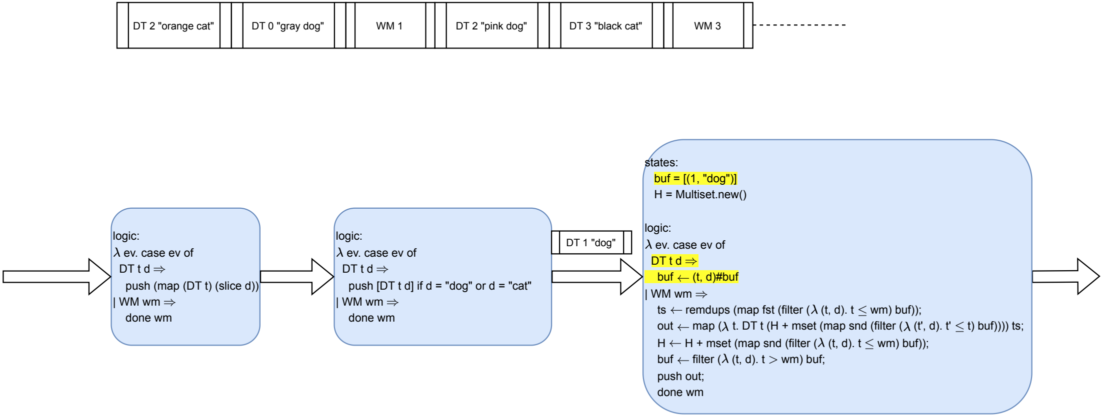
Time-Aware Stream Processing Example



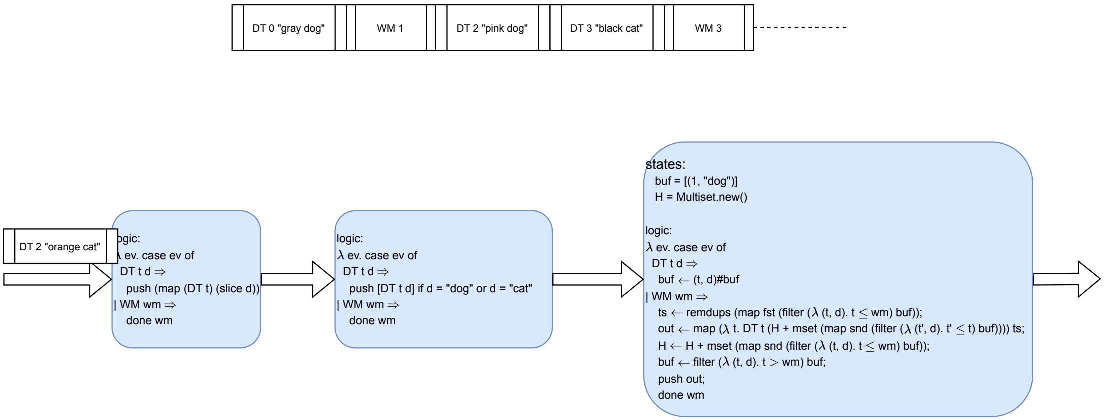
Time-Aware Stream Processing Example



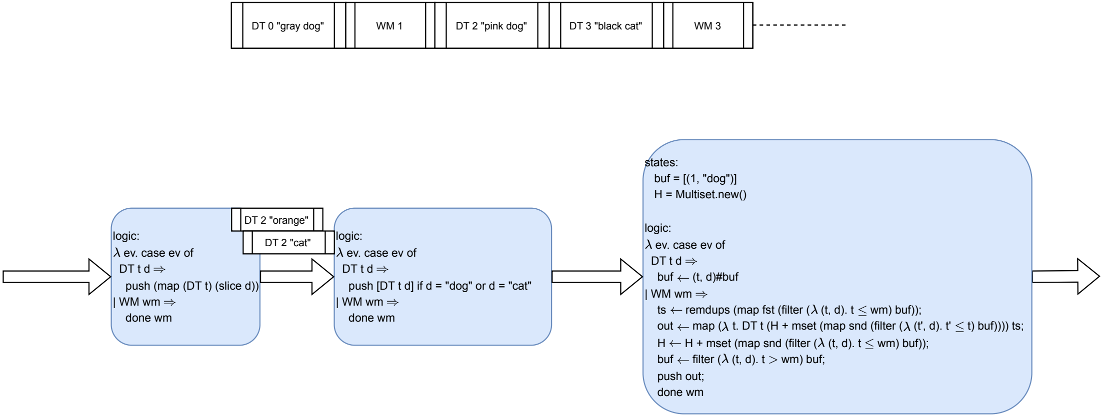
Time-Aware Stream Processing Example



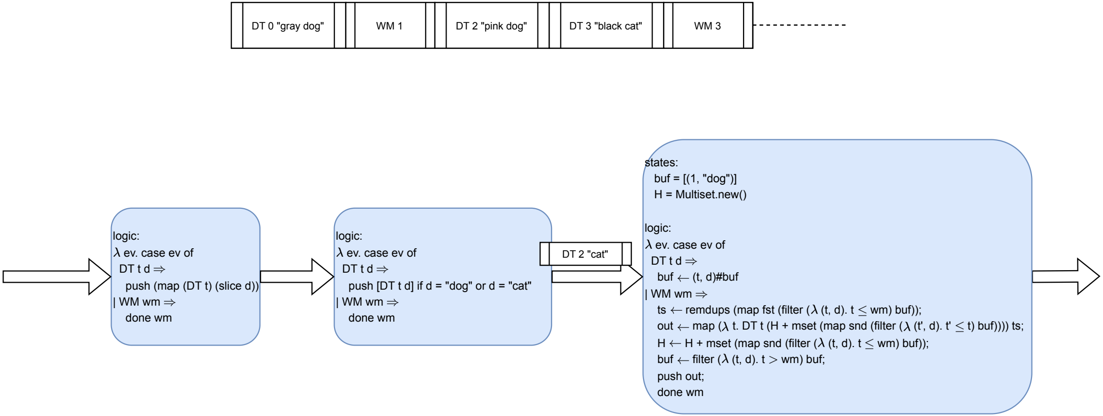
Time-Aware Stream Processing Example



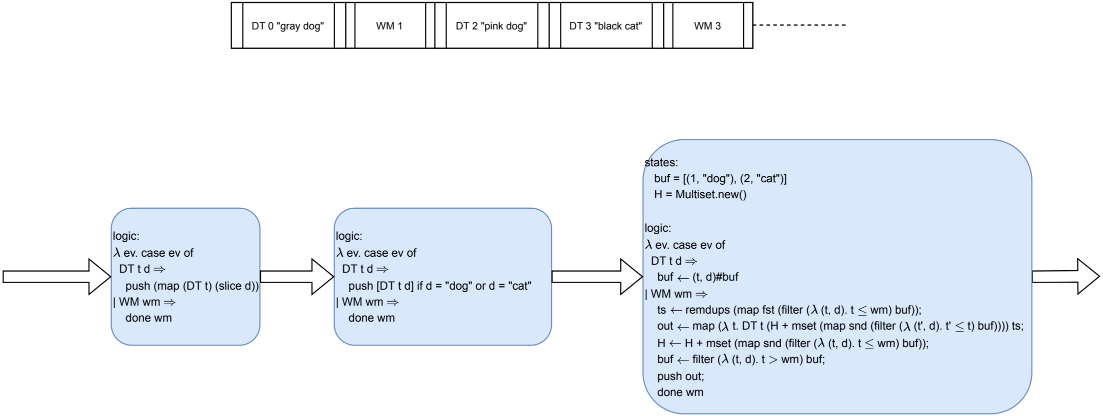
Time-Aware Stream Processing Example



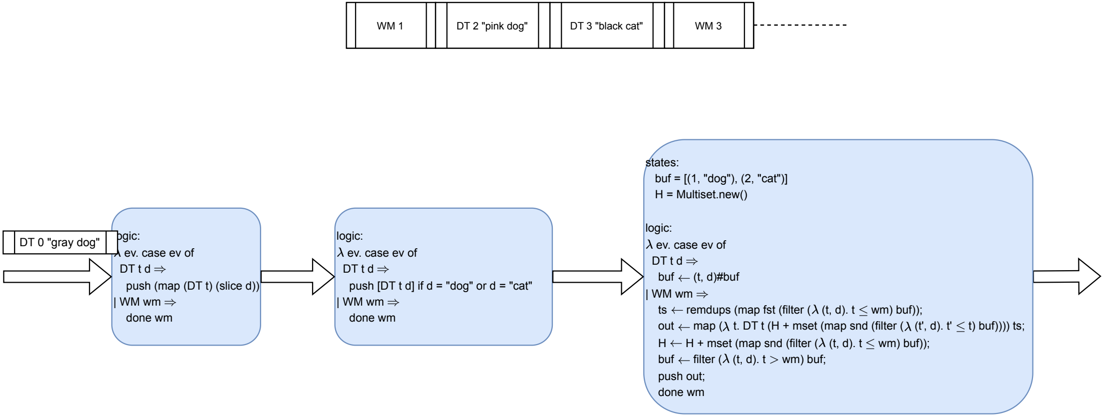
Time-Aware Stream Processing Example



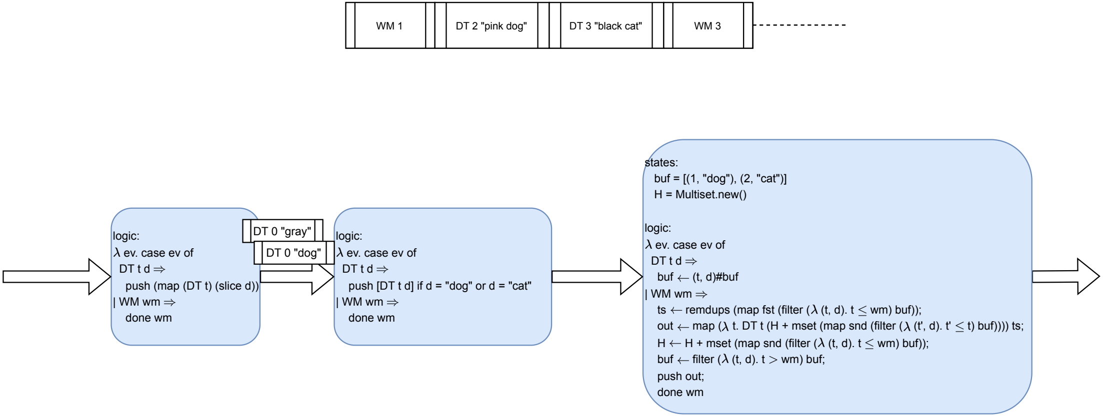
Time-Aware Stream Processing Example



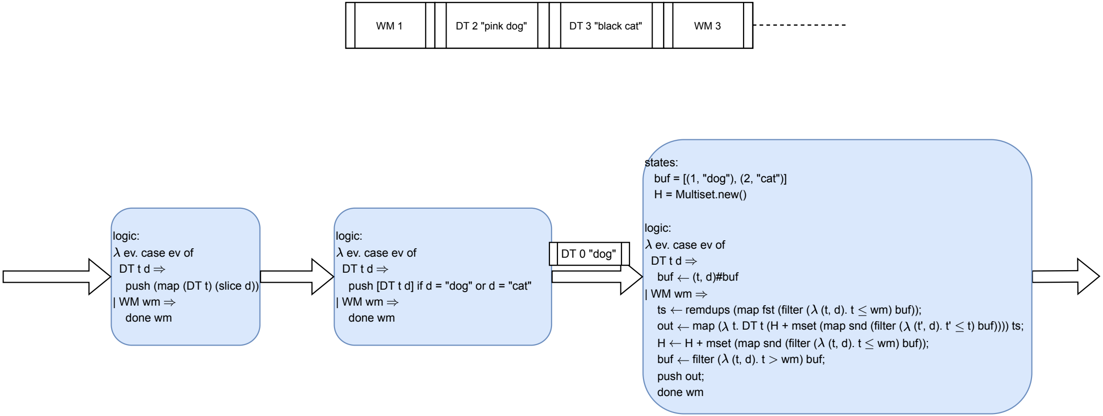
Time-Aware Stream Processing Example



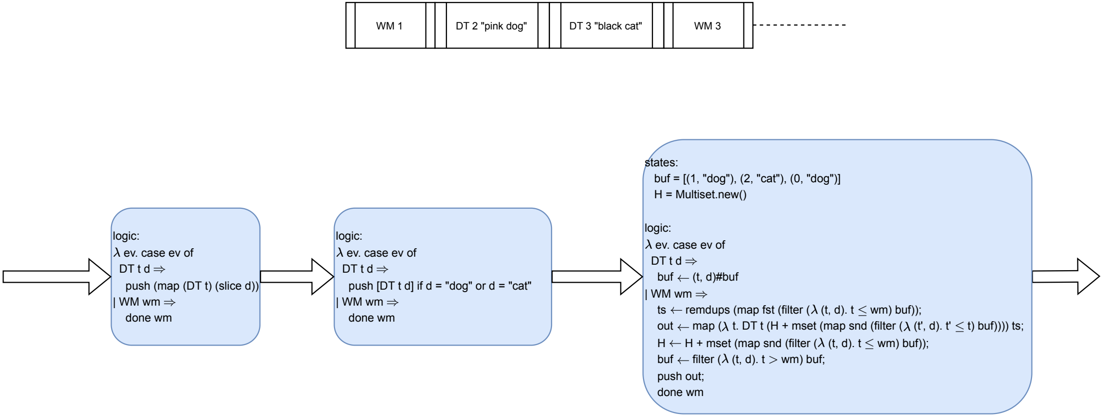
Time-Aware Stream Processing Example



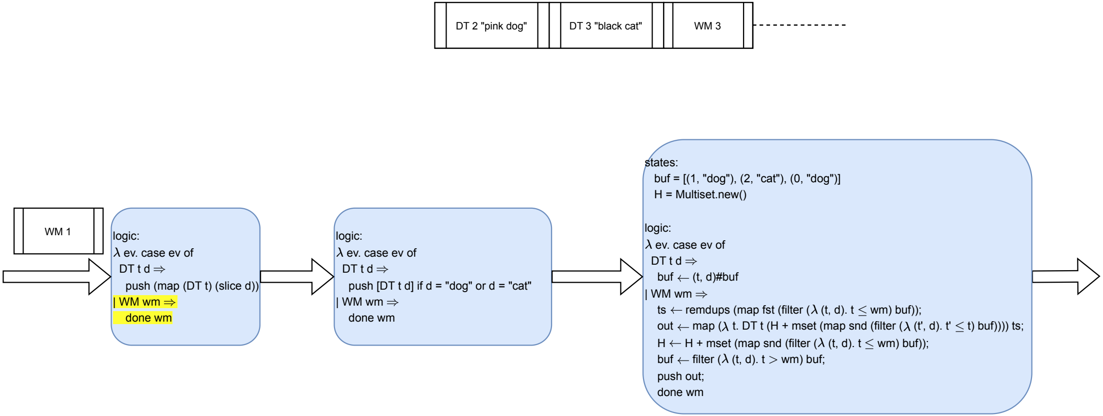
Time-Aware Stream Processing Example



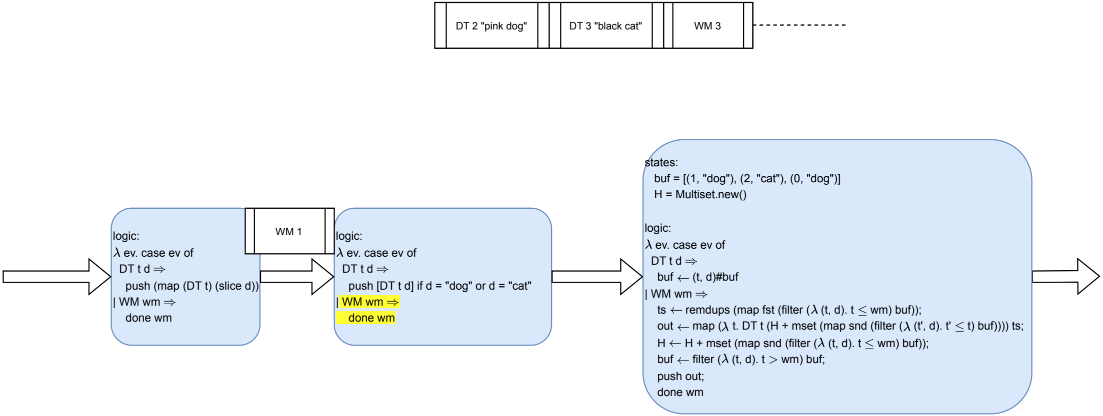
Time-Aware Stream Processing Example



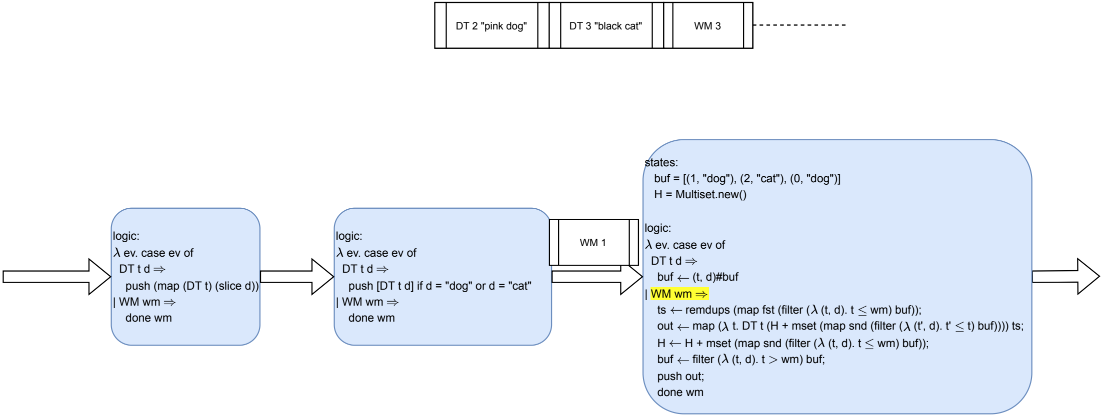
Time-Aware Stream Processing Example



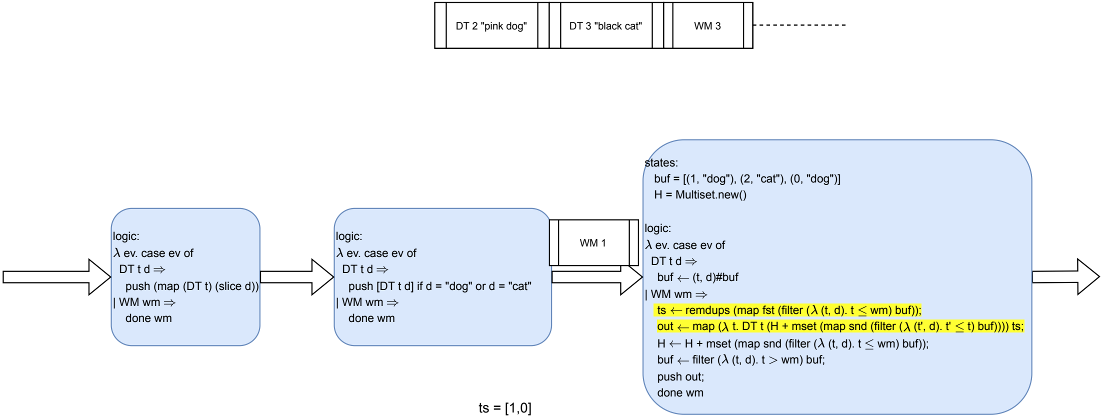
Time-Aware Stream Processing Example



Time-Aware Stream Processing Example



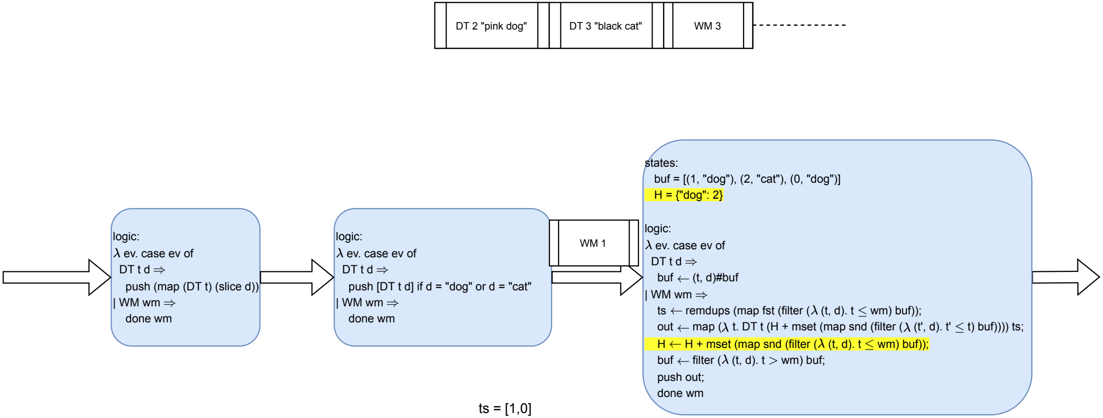
Time-Aware Stream Processing Example



ts = [1,0]

out = [DT 1 {"dog": 2}, DT 0 {"dog": 1}]

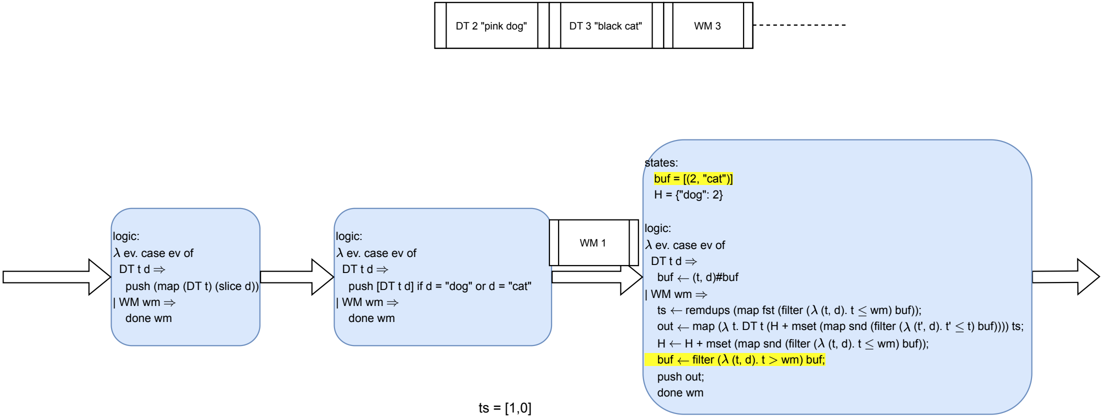
Time-Aware Stream Processing Example



ts = [1,0]

out = [DT 1 {"dog": 2}, DT 0 {"dog": 1}]

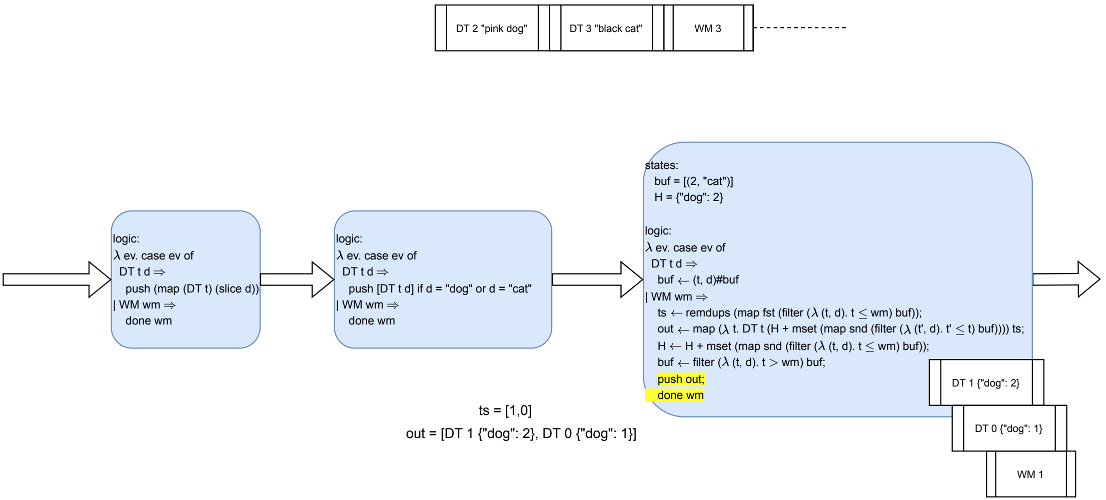
Time-Aware Stream Processing Example



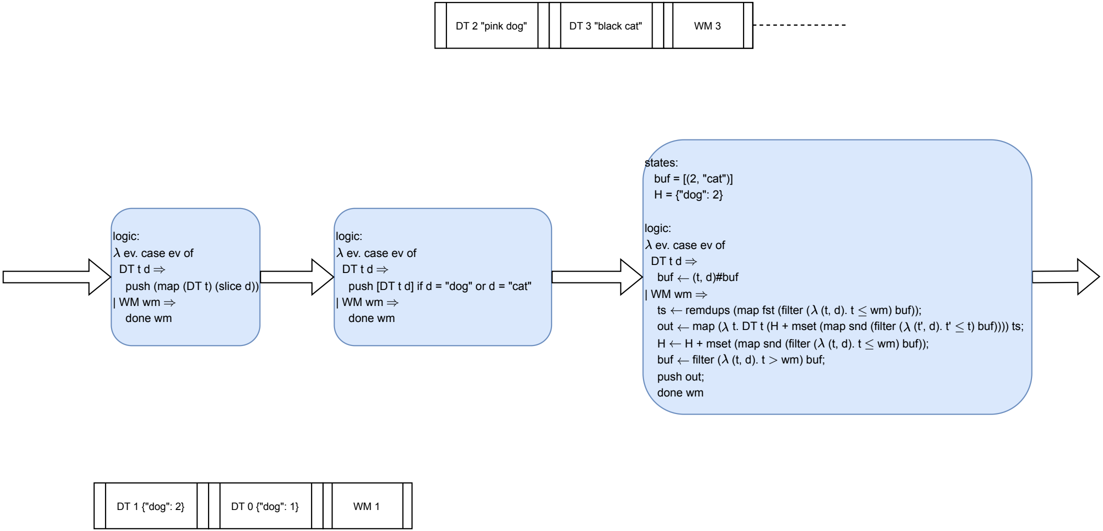
ts = [1,0]

out = [DT 1 {"dog": 2}, DT 0 {"dog": 1}]

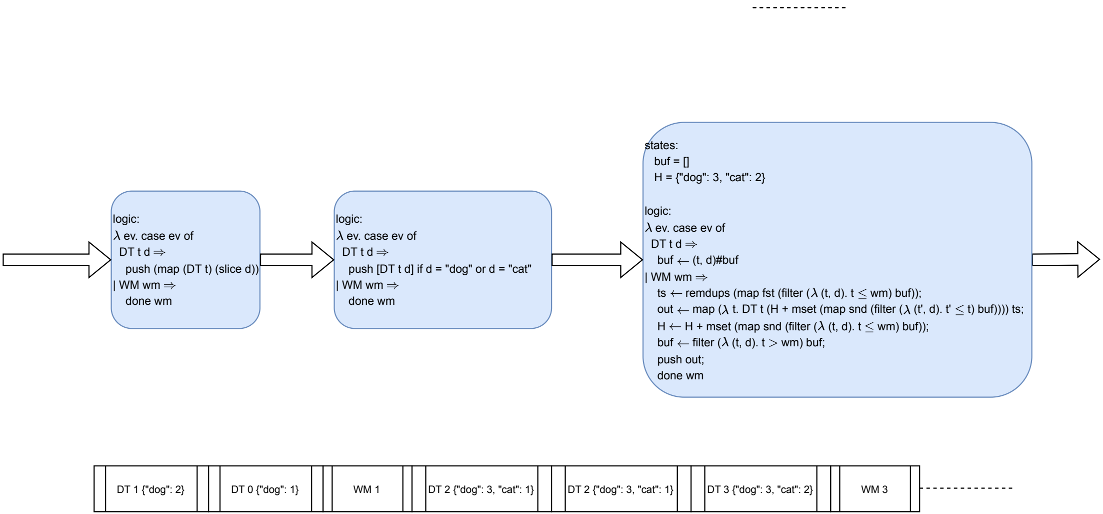
Time-Aware Stream Processing Example



Time-Aware Stream Processing Example



Time-Aware Stream Processing Example



Properties

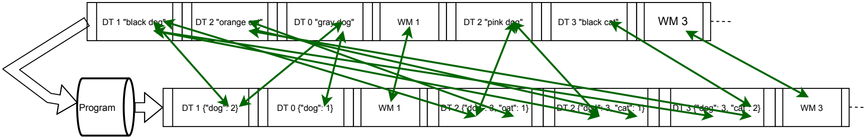
- How do we know if our Dataflow program is what we want?

Properties

- How do we know if our Dataflow program is what we want?
- We need a correctness specification

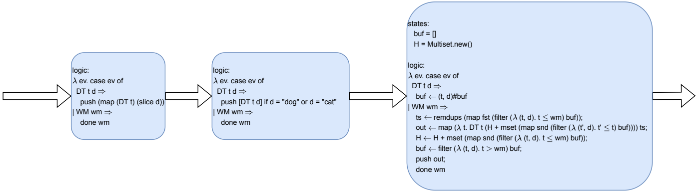
Properties

- How do we know if our Dataflow program is what we want?
- We need a correctness specification
- **Intuition of the specification**:
  - Soundness: for every output $DT\ t\ H$, the "dog" count in H is the count of events with timestamp $(\leq)t$ which contains the string "dog"; similarly for "cat". The count for any other word is always 0.
  - Completeness: The other way around.

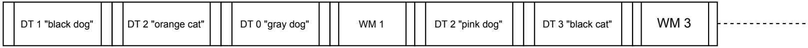


How to prove it

- **lets break down the problem!**:
  - The correctness of the entire Dataflow emerges from the correctness of each part (operator)
    - Operator 1: Slicer
    - Operator 2: Filter
    - Operator 3: Incremental histogram
  - Assumptions about the incoming stream:
  1. Monotone: after WM wm no $DT\ t\ d$ such that $t \leq wm$.
  2. Productive: after $DT\ t\ d$ eventually WM wm such that $t \leq wm$



- The original incoming stream must respect monotonicity and productivity



- Each operator must preserve monotonicity and productivity!

Writing it down in a proof assistant!

Isabelle/HOL

- Classical higher-order logic (HOL): Simple Typed Lambda Calculus + (Hilbert) axiom of choice + axiom of infinity + rank-1 polymorphism

Isabelle/HOL

- Classical higher-order logic (HOL): Simple Typed Lambda Calculus + (Hilbert) axiom of choice + axiom of infinity + rank-1 polymorphism
- Isabelle: A generic proof assistant

- Isabelle/HOL: Isabelle's flavor of HOL

Isabelle/HOL: (Co)datatypes

- Datatypes and Codatatypes

  codatatype (lset: *'a*) *llist* = lnull: LNil | LCons (lhd: *'a*) (ltl: *'a llist*)
  for map: lmap **where** ltl LNil = LNil

- Examples:
  - LNil
  - LCons *1* (LCons *2* (LCons *3* LNil))
  - LCons *0* (LCons *0* (LCons *0* (...)))

Isabelle/HOL: (Co)datatypes

- Datatypes and Codatatypes

  codatatype (lset: $'a$) *llist* = lnull: LNil | LCons (lhd: $'a$) (ltl: $'a$ *llist*)
  for map: lmap **where** ltl LNil = LNil

- Examples:
  - LNil
  - LCons *1* (LCons *2* (LCons *3* LNil))
  - LCons *0* (LCons *0* (LCons *0* (...)))
- Coinductive principle for lazy list equality:

Isabelle/HOL: Recursion and While Combinator

- Recursion

$$\textbf{fun}\ \mathsf{lshift} :: {'a}\ list \Rightarrow {'a}\ llist \Rightarrow {'a}\ llist\ (infixr\ @@\ 65)\ \textbf{where}$$
$$\mathsf{lshift}\ []\ lxs = lxs$$
$$|\ \mathsf{lshift}\ (x \# xs)\ lxs = \mathsf{LCons}\ x\ (\mathsf{lshift}\ xs\ lxs)$$

- While Combinator

$$\textbf{definition}\ \mathsf{while\_option} :: ({'a} \Rightarrow bool) \Rightarrow ({'a} \Rightarrow {'a}) \Rightarrow {'a} \Rightarrow {'a}\ option\ \textbf{where}$$
$$\mathsf{while\_option}\ b\ c\ s = \ldots$$

- While rule for invariant reasoning (Hoare-style):
  - There is something that holds before a step; that thing still holds after the step

Isabelle/HOL: Corecursion

- Corecursion is like recursion, but instead of always eventually reducing an argument it always eventually produces something

Isabelle/HOL: Corecursion

- Corecursion is like recursion, but instead of always eventually reducing an argument it always eventually produces something
- Corec:

$\textbf{corec}\ \mathsf{lapp} :: {}'a\ \mathit{llist} \Rightarrow {}'a\ \mathit{llist} \Rightarrow {}'a\ \mathit{llist}\ \textbf{where}$
$\mathsf{lapp}\ \mathit{lxs}\ \mathit{lys} = \mathtt{case}\ \mathit{lxs}\ \mathtt{of}\ \mathsf{LNil} \Rightarrow \mathit{lys} \mid \mathsf{LCons}\ x\ \mathit{lxs'} \Rightarrow \mathsf{LCons}\ x\ (\mathsf{lapp}\ \mathit{lxs'}\ \mathit{lys})$

Isabelle/HOL: (Co)inductive Predicates

- Inductive predicate
  - Finite number of introduction rule applications

$\textbf{inductive}$ in_llist $:: {}'a \Rightarrow {}'a\ llist \Rightarrow bool$ $\textbf{where}$
  In_llist: in_llist x (LCons x lxs)
| Next_llist: in_llist x lxs $\Rightarrow$ in_llist x (LCons y lxs)

in_llist 2 (LCons 1 (LCons (2 (...))))

Isabelle/HOL: (Co)inductive Predicates

- Inductive predicate
  - Finite number of introduction rule applications

inductive in_llist $:: 'a \Rightarrow 'a\ llist \Rightarrow bool$ **where**
  In_llist: in_llist x (LCons x lxs)
| Next_llist: in_llist x lxs $\Rightarrow$ in_llist x (LCons y lxs)

in_llist 2 (LCons 1 (LCons (2 (. . .))))

- Coinductive predicate
  - Infinite number of introduction rule applications

coinductive lprefix $:: 'a\ llist \Rightarrow 'a\ llist \Rightarrow bool$ **where**
  LNil_lprefix: lprefix LNil lys
| LCons_lprefix: lprefix lxs lys $\Rightarrow$ lprefix (LCons x lxs) (LCons x lys)

lprefix (LCons 1 (LCons (2 (. . .)))) (LCons 1 (LCons (2 (. . .))))

Isabelle/HOL: (Co)inductive Predicates

- Inductive predicate
  - Finite number of introduction rule applications

inductive in_llist :: $'a \Rightarrow 'a$ *llist* $\Rightarrow$ *bool* **where**
  In_llist: in_llist x (LCons x lxs)
| Next_llist: in_llist x lxs $\Rightarrow$ in_llist x (LCons y lxs)

in_llist 2 (LCons 1 (LCons (2 $(\ldots)$)))

- Coinductive predicate
  - Infinite number of introduction rule applications

coinductive lprefix :: $'a$ *llist* $\Rightarrow 'a$ *llist* $\Rightarrow$ *bool* **where**
  LNil_lprefix: lprefix LNil lys
| LCons_lprefix: lprefix lxs lys $\Rightarrow$ lprefix (LCons x lxs) (LCons x lys)

lprefix (LCons 1 (LCons (2 $(\ldots)$))) (LCons 1 (LCons (2 $(\ldots)$)))

- Coinduction principle

Lazy Lists Processors

Operator formalization

- Operator as a codatatype
  - Taking $'i$ as the input type, and $'o$ as the output type:
    $\textbf{codatatype}\ ('o, 'i)\ \mathit{op} = \text{Logic}\ (\text{apply}: ('i \Rightarrow ('o, 'i)\ \mathit{op} \times 'o\ \mathit{list}))$

Operator formalization

- Operator as a codatatype
  - Taking $'i$ as the input type, and $'o$ as the output type:
    $\textbf{codatatype}\ ('o, 'i)\ \mathit{op} = \mathrm{Logic}\ (\mathrm{apply}\!: ('i \Rightarrow ('o, 'i)\ \mathit{op} \times 'o\ \mathit{list}))$
  - Infinite trees: applying the selector apply "walks" a branch of the tree

Execution formalization

- Produce function: applies the logic (co)recursively throughout a lazy list

definition produce$_1$ *op lxs* = while_option ...

corec produce **where**
produce *op lxs* = (case produce$_1$ *op lxs* of
  None ⇒ LNil
| Some (*op′*, *x*, *xs*, *lxs′*) ⇒ LCons *x* (*xs* @@ produce *op′ lxs′*))

Execution formalization

- Produce function: applies the logic (co)recursively throughout a lazy list

  definition $\mathsf{produce}_1\ \mathit{op}\ \mathit{lxs} = \mathsf{while\_option}\ \ldots$

  corec produce **where**
  $\mathsf{produce}\ \mathit{op}\ \mathit{lxs} = (\mathtt{case}\ \mathsf{produce}_1\ \mathit{op}\ \mathit{lxs}\ \mathtt{of}$
  $\quad \mathsf{None} \Rightarrow \mathsf{LNil}$
  $\mid \mathsf{Some}\ (\mathit{op}', x, \mathit{xs}, \mathit{lxs}') \Rightarrow \mathsf{LCons}\ x\ (\mathit{xs}\ @@\ \mathsf{produce}\ \mathit{op}'\ \mathit{lxs}'))$

- $\mathsf{produce}_1$ has an induction principle based on the while invariant rule

Operators: Count

- Example:

$$\textbf{corec}\ \mathsf{count\_op}\ \textbf{where}\ \mathsf{count\_op}\ P\ n =$$
$$\quad \mathsf{Logic}\ (\lambda e.\ \texttt{if}\ P\ e\ \texttt{then}\ (\mathsf{count\_op}\ P\ (n + 1),\ [n{+}1])\ \texttt{else}\ (\mathsf{count\_op}\ P\ n,\ []))$$

$$\mathit{stream}_1 = \boxed{0}\,\boxed{3}\,\boxed{3}\,\boxed{6}\,\boxed{24}$$

$$\mathsf{produce}\ (\mathsf{count\_op}\ \mathit{is\_even}\ 0)\ \ \mathit{stream}_1 = \boxed{1}\,\boxed{2}\,\boxed{3}$$

Sequential Composition operator

- Sequential composition: take the output of the first operator and give it as input to the second operator.
- Correctness:

  $\text{produce } (\text{comp\_op } op_1 \; op_2) \; lxs = \text{produce } op_2 \; (\text{produce } op_1 \; lxs)$

- Proof: coinduction principle for lazy list equality and produce_1 induction principle

Time-Aware Operators

Time-Aware Streams

- Time-Aware lazy lists

  $\textbf{datatype}\ ('t{::}order, 'd)\ \mathit{event} = \mathrm{DT}\ (\mathrm{tmp}{:}\ 't)\ (\mathrm{data}{:}\ 'd) \mid \mathrm{WM}\ (\mathrm{wmk}{:}\ 't)$

Time-Aware Streams

- Time-Aware lazy lists

$$\textbf{datatype}\ ('t{::}\mathit{order}, 'd)\ \mathit{event} = \mathrm{DT}\ (\mathrm{tmp}{:}\ 't)\ (\mathrm{data}{:}\ 'd) \mid \mathrm{WM}\ (\mathrm{wmk}{:}\ 't)$$

- Generalization to partial orders
  - Cycles
  - Operators with multiple inputs
- Productive and monotone streams: Coinductive predicates over lazy lists of events.

Proving histogram correct: building Blocks

- Histogram operator: batching and incremental computing
- Building Blocks: reusable operators
  - Batching: batch_op
  - Incremental computing: incr_op
  - Soundness, completeness, preservation of monotonicity and productivity

Compositional Reasoning

- batch_op and incr_op can be composed

  definition incr_batch_op *buf1 buf2* = comp_op (batch_op *buf1*) (incr_op *buf2*)

- Soundness, completeness, and monotone and productive preservation

Histogram Operator

corec map_op **where** map_op f = Logic (λ ev. case ev of
WM $wm \Rightarrow$ (map_op f, [WM wm]) | DT t $d \Rightarrow$ (map_op f, [DT t (f t d)]))

definition incr_hist_op $buf1$ $buf2$ =
comp_op (incr_batch_op $buf1$ $buf2$) (map_op incr_coll)

- Soundness, completeness, and monotone and productive preservation

Other shapes

Join Operator

- Relation Join
- Use the `sum` type in the timestamps to represent two stream as one
- Partial order for the `sum`: lefts and rights are incomparable
- Defined using `incr_batch_op`
- Soundness, completeness, and monotone and productive preservation

Next Steps

Next Steps

- Feedback loop

Questions, comments and suggestions