Towards Execution Guarantees for Stream Queries

Rafael J. Fernández-Moctezuma, David Maier, and Kristin A. Tufte

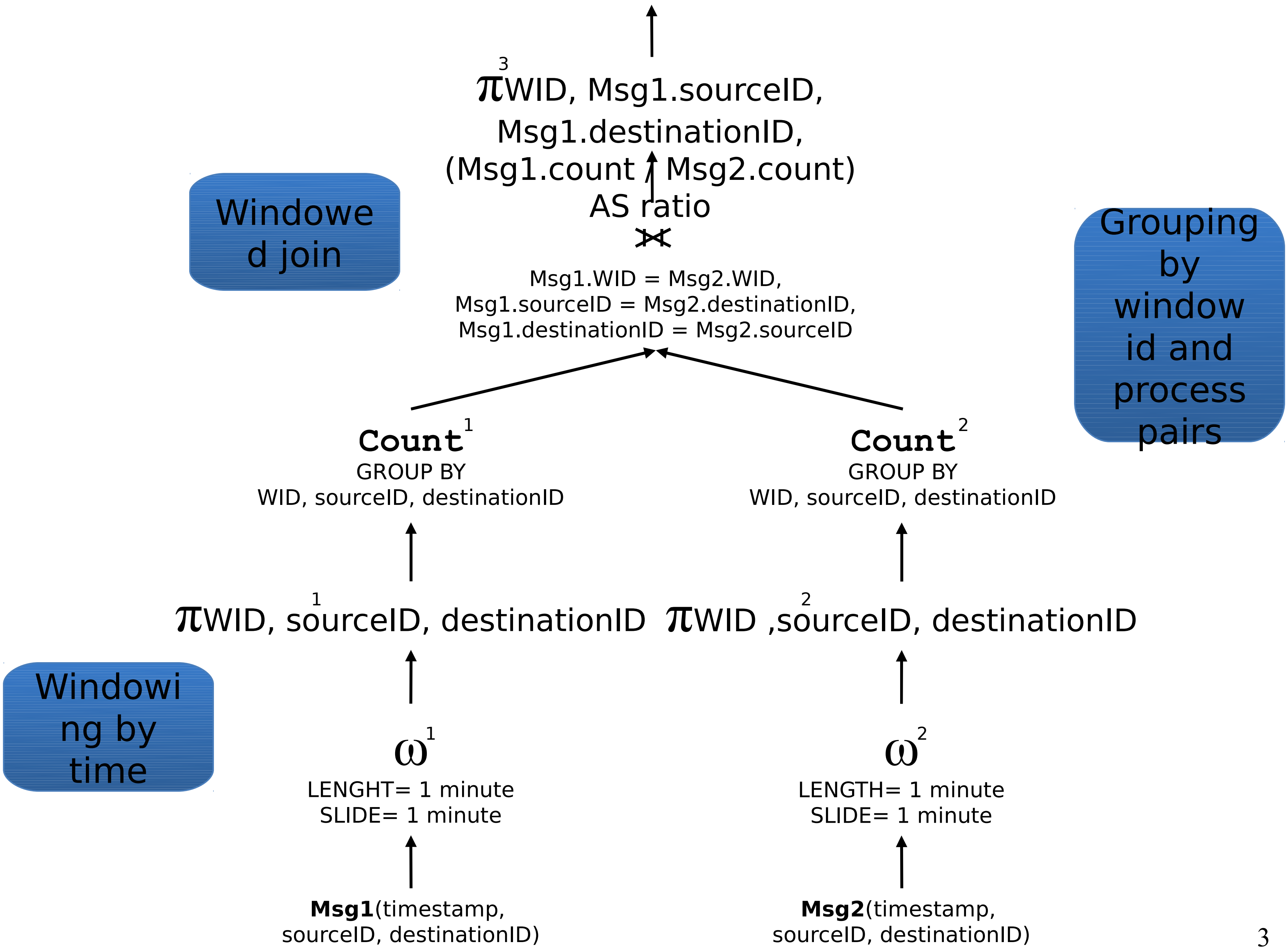
Acknowledgements: Lois Delcambre, Len Shapiro, Tim Chevalier, Jeremy Steinhauer, CONACyT México (178258), NSF (IIS-0917349)

SSPS 2010

Continuous Queries

Data Stream Management Systems allow evaluation of *continuous queries* over *data streams* – data flows *through* the query plan

Contrast with Database Management Systems, where data *sets* are static and queries are issued against them to produce result *sets*



π^3 WID, Msg1.sourceID, Msg1.destinationID, (Msg1.count / Msg2.count) AS ratio
Msg1.WID = Msg2.WID, Msg1.sourceID = Msg2.destinationID, Msg1.destinationID = Msg2.sourceID
Windowed join
Grouping by window id and process pairs
Count 1
GROUP BY WID, sourceID, destinationID
Count 2
GROUP BY WID, sourceID, destinationID
π^1 WID, sourceID, destinationID
π^2 WID ,sourceID, destinationID
Windowing by time
ω^1
LENGHT= 1 minute
SLIDE= 1 minute
ω^2
LENGTH= 1 minute
SLIDE= 1 minute
Msg1(timestamp, sourceID, destinationID)
Msg2(timestamp, sourceID, destinationID)

Assessing Data Stream Progress

Data Streams are unbounded – don’t know for sure when they end

“Average speed on US 26”. Let me know when you’ve seen all cars. I may not be willing to wait.

“Average speed on US 26 for yesterday”. Let me know when you’ve seen all data points for yesterday.

Punctuated Data Streams

How do we know for sure when you've seen all data points in a stream?

May be out of order

Latency in result production, correctness of the result, and efficient use of resources are important.

Punctuations (Tucker *et al.*) are delimiters in the stream that help track progress

Punctuated Data Streams

<ts: 10:00:00 p.m., sensorID: 1, speed: 20>

<ts: 10:00:00 p.m., sensorID: 2, speed: 30>

<ts: 10:00:30 p.m., sensorID: 1, speed: 25>

<ts: 10:00:30 p.m., sensorID: 2, speed: 25>

[ts:≤10:00:30 p.m., sensorID: *, speed: *]

<ts: 10:00:00 p.m., sensorID: 1, speed: 20>

<ts: 10:00:00 p.m., sensorID: 2, speed: 30>

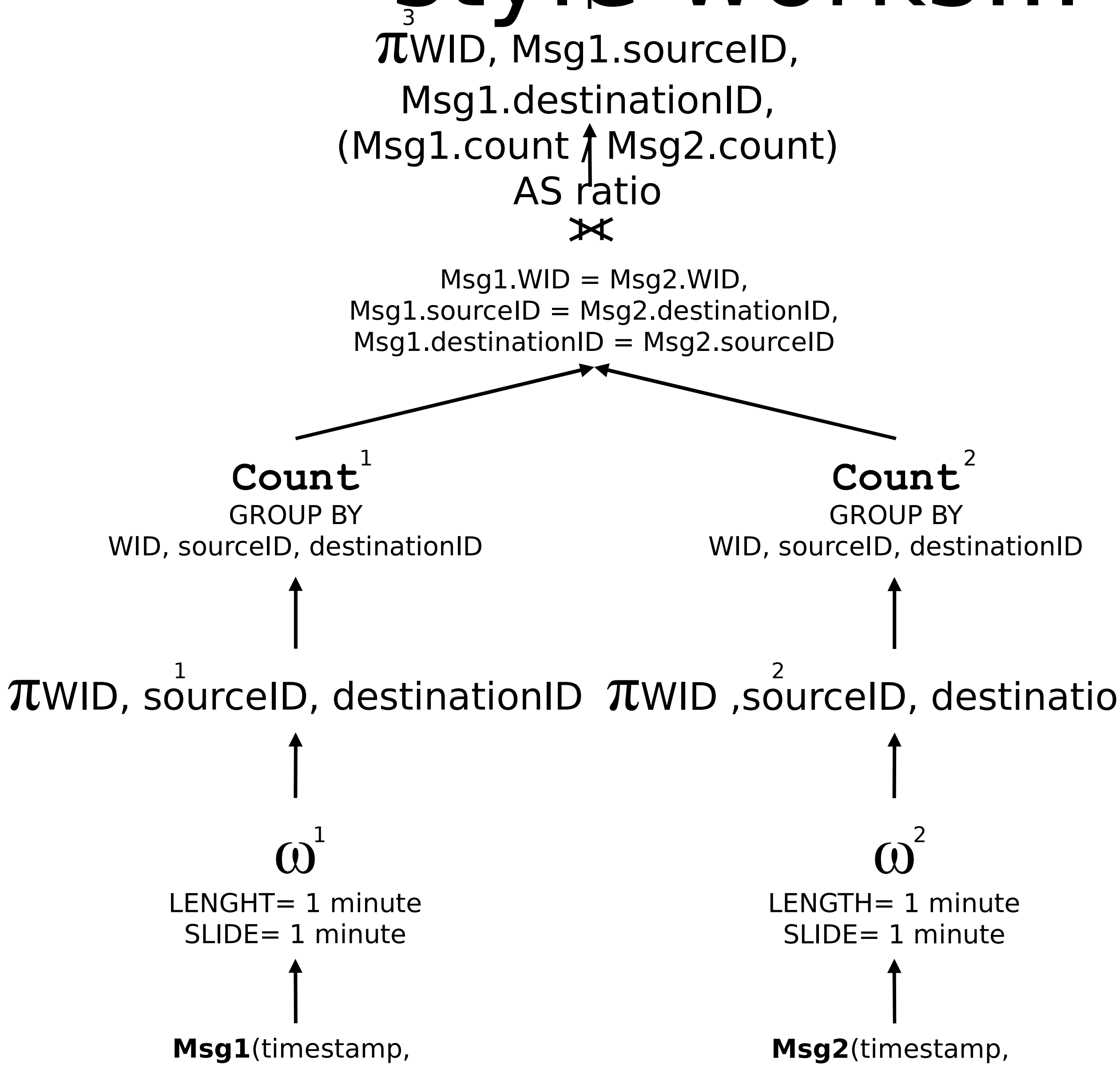
<ts: 10:00:30 p.m., sensorID: 1, speed: 25>

<ts: 10:00:30 p.m., sensorID: 2, speed: 25>

[ts: *, sensorID: 1, speed: *]

…

More than one punctuation style works...



Can track progress
by time, or by process termination.

How do we know, before execution, if a query executes successfully given a specific punctuation style?

Punctuation on process ID vs. Punctuation on time

Execution Guarantees

A query will execute successfully if:

Every *correct* output will be eventually delivered by the query

No piece of state remains indefinitely in any query operator

Framework

Tucker *et al.* Characterized how an operator processes *one* punctuation

Frees up internal state

Emit output

Emit punctuation

We want to consider all the punctuations in a stream

Punctuation Templates

Three styles : some tell you “up to value *x*”, others about a specific item *y, others tell you about “anything”*

A *template* captures these styles:

“+” for the “up to” pattern

“#” for the “point” pattern

“-” for the “anything” pattern

Punctuation Templates

[[a:+, b:#, c:-]]

describes punctuations such as:

[a:<‘11:30 p.m.’, b:26, c:*]

but not:

[a:*, b:26, c:*]

[a:<‘11:30 p.m.’, b:<26, c:*]

[a:<‘11:30 p.m.’, b:26, c:3]

Punctuation Scheme

Operators may be able to process more than one template. A *punctuation scheme* is a set of one or more punctuation templates:

```
PS1 = {[[a:+,b:#,c:-]]}

PS2 = {[[a:+,b:-,c:-]], [[a:-,b:#,c:-]]}
```

A stream *S* obeys a scheme *PS* if:

- Any punctuation *p* in *S* conforms to *at least one* punctuation template *T* in *PS*
- For any tuple *t* in *S*, and each template *T* in *PS*, there is a *p* in *S* s.t. *p* conforms to *T* and *t* matches *p*.

PS1 = {[[a:+,b:#,c:-]]}

[a:<'10:00 p.m.',b:1,c:*]

[a:<'10:00 p.m.',b:2,c:*]

[a:<'10:05 p.m.',b:1,c:*]

[a:<'10:05 p.m.',b:2,c:*]

…

- obeys PS1

[a:<'10:00 p.m.',b:*,c:*]

[a:*,b:2,c:*]

- does not obey PS1

A stream *S* obeys a scheme *PS* if:

- ·Any punctuation *p* in *S* conforms to *at least one* punctuation template *T* in *PS*
- ·For any tuple *t* in *S*, and each template *T* in *PS*, there is a *p* in *S* s.t. *p* conforms to *T* and *t* matches *p*.

```
PS2 = {[[a:+,b:-,c:-]],
```

```
[a:<'10:00 p.m.',b:*,c:*]
[a:<'10:05 p.m.',b:*,c:*]
[a:*,b:1,c:*]
[a:<'10:10 p.m.',b:*,c:*]
[a:*,b:2,c:*]
[a:<'10:15 p.m.',b:*,c:*]
...
```

- obeys PS2

- does not obey PS2

Punctuation Contracts

Records of punctuation schemes corresponding to each input and output of an operator.

Two contracts for `SELECT`:

```
CT1 = <In={[[a:+,b:-,c:-]]},
        Out={[[a:+,b:-,c:-]]}>
CT2 = <In={[[a:+,b:-,c:-]],[[a:-,b:#,c:-]]},
```

Execution Guarantees

For operator *R* with an input stream that obeys the input punctuation scheme in R's contract *CT*, the following *guarantees* hold:

1. *R*'s output stream obeys the output punctuation scheme in *CT*
2. No piece of state remains is held by *R* forever
3. *R* produces the maximal possible

JOIN characterization

I1, I2 = input schemas of JOIN.

J = set of joining attributes (J in I1, J in I2).

L and R = sets of attributes exclusive to inputs 1 and 2, respectively

(L = I1 - J, R = I2 - J).

General contract forms:

Full-query analysis

An *accordance* is a pairing of selections of contracts from operator contracts:

Stream1 □ Op1

Stream1 Offering = `{C1}`

`Op1 Offering = {C2, C3}`

`Accordances (C1, C2), (C1, C3)`

Full-query analysis

C1 = <Out={[[a:+, b:-]]}>

C2 = <In={[[a:#, b:-]]}, Out={[[a:#,b:-]]}>

C3 = <In={[[a:+, b:-]]}, Out={[[a:+,b:-]]}>

Stream1   Op1

Stream1 Offering = {C1}

Op1 Offering = {C2, C3}

Accordances: (C1, C2), **(C1, C3)**

One **consistent accordance** is found.

Finding an accordance as a join problem

Contract offerings for each operator are relations, each contract is a row

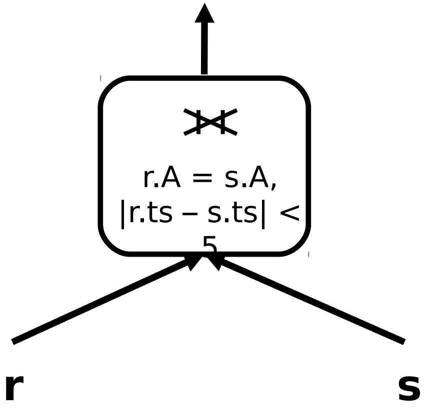
Offering for operator A	**In**	**Out**
	{[[a:+, b:-]]}	{[[a:+, b:-]]}
	{[[a:#, b:-]]}	{[[a:#, b:-]]}

Offering for operator B	**In**	**Out**
	{[[a:+, b:-]]}	{[[a:+, b:-]]}

If the query is a DAG, can be cast as a Full Reducers problem, which admits an efficient solution.

Further Considerations 1

No permanent lodging of state, but doesn't bound state at any instance



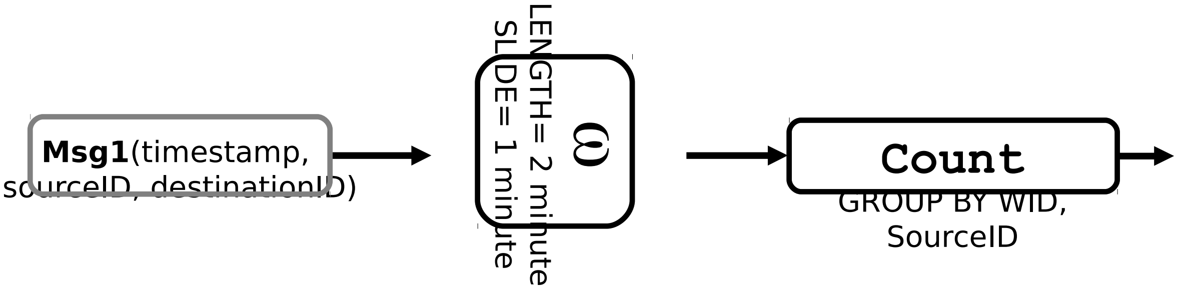
Band join at right: Needs to buffer 5 minutes of tuples

Data Density

Further Considerations 2

Distribution of data values can also affect operator memory needs

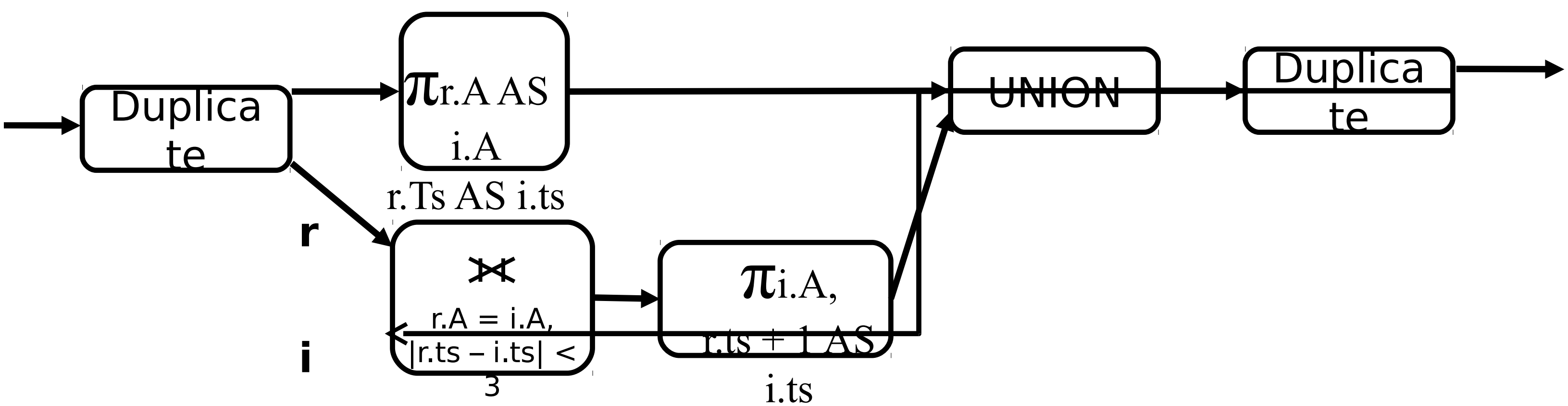
In windowed aggregate below, number of distinct SourceIDs in 2 minutes determines entries in `Count`



Further Considerations 3

Even if an event is cleared from state, its progeny may live on

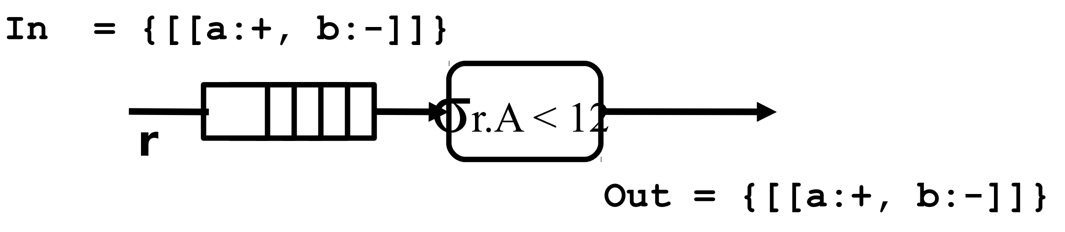
Autocorrelation query below permits chains of derived tuples



Further Considerations 4

Need to consider data outside of operator state

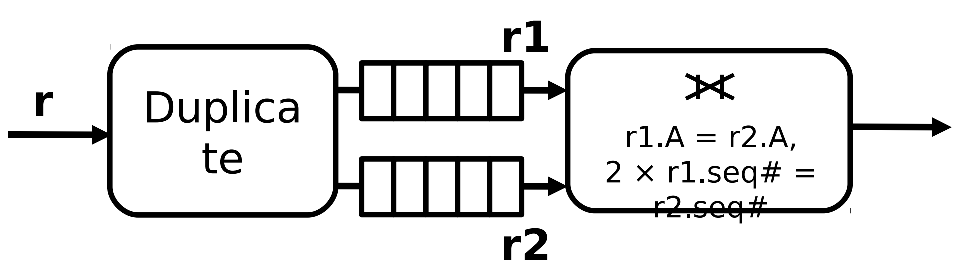
"Reticent" select operator below stops reading input once it produces its final output



Further Considerations 5

Even reasonable operator implementations can result in unbounded buffer growth

Evil query below has unbounded growth on r1 because of different consumption rates



The Four D's

Key properties in determining memory and CPU use

Density: Items per logical time unit

Disorder: Specifically, how late can an item be

Distribution: Number and density of groups

Divergence: Offset in time stamps between streams

Future Work

Extension to query processing styles in which contextual information flows contrary to the stream direction

Need to adjust punctuation density to match data density (*e.g.*, “you won’t see more than 500 events without a punctuation”)

Revisiting adaptivity in the light of the four D’s. If you don’t address those, you might not get much benefit.