CS 591 K1:
Data Stream Processing and Analytics

Spring 2020

2/04: Streaming languages and operator semantics

Vasiliki (Vasia) Kalavri
vkalavri@bu.edu

Languages for continuous data processing

Language Types

- **Transforming** languages define transformations specifying operations that process input streams and produce output streams.
    - **Declarative** languages specify the *expected results* of the computation rather than the execution flow.
    - **Imperative** languages are used to describe *plans of operators* the streams must flow through.
- **Pattern-based** languages specify conditions and actions to be taken when conditions are met.
    - **Conditions** are commonly described as patterns that can match input stream events on type, content, timing constraints.
    - **Actions** define how to produce results from the matches.

Declarative language: CQL

Three classes of operators:

- **relation-to-relation**: similar to standard SQL and define queries over tables.
- **stream-to-relation**: define tables by selecting portions of a stream.
- **relation-to-stream**: create streams through querying tables

CQL Example

```
Select IStream(*)
From S1 [Rows 5], S2 [Rows 10]
Where S1.A = S2.A
```

relation-to-stream

stream-to-relation

relation-to-relation

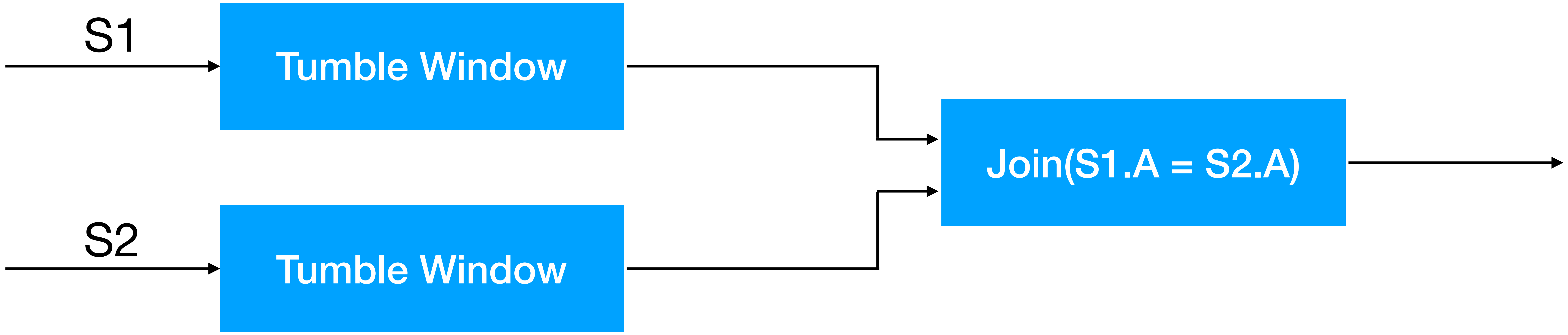
Last 5 elements of stream S1 and last 10 elements of S2

CQL relation-to-stream operators

- **Istream** (for “insert stream”) applied to relation R contains a stream element <s, τ> whenever tuple s is in R(τ) – R(τ – 1).
- **Dstream** (for “delete stream”) applied to relation R contains a stream element <s, τ> whenever tuple s is in R(τ – 1) – R(τ).
- **Rstream** (for “relation stream”) applied to relation R contains a stream element <s, τ> whenever tuple s is in R at time τ.

Imperative language: Aurora SQuAl

Queries are represented in graphical representation using *boxes* and *arrows*



Composite subscription pattern language

A, B, C are topics

X, Y, Z are inner fields

```
A(X>0) & (B(Y=10);[timespan:5] C(Z<5))[within:15]
```

The rule fires when
an item of type A having an attribute X > 0 *enters the system and also*
an item of type B with Y = 10 *is detected,*
*followed (**in a time interval of 5–15 s**) by*
***an item of type C with Z < 5**.*

Streaming Operators

Operator types (I)

- **Single-Item** Operators process stream elements one-by-one.
  - selection, filtering, projection, renaming.
- **Logic** Operators define rules for complex pattern detection without order constraints.
  - **conjunction** of items I_1, I_2, …, I_n is satisfied when all items have been detected.
  - **disjunction** of items I_1, I_2, …, I_n is satisfied when at least one item has been detected.
  - **repetition** of an item I of degree (m, n) is satisfied when I is detected at least m times but o more than n times.
  - **negation** of an item I is satisfied when I is not detected.

Logic Operators Example

Explicit conjunction and disjunction

```
(A & B) || (C & D)
```

Implicit conjunction in CQL

```
Select IStream(S1.A, S2.B)
From S1 [Rows 50], S2 [Rows 50]
```

Consider events from stream S1 *and* stream S2

Operator types (II)

- **Sequence** Operators capture the arrival of an *ordered set* of events.
    - common in pattern languages
    - events must have associated timestamps
- **Iteration** Operators define sequences of events or processing that satisfies a *loop condition*.
    - not commonly supported
    - a termination condition must be defined, e.g. time limit

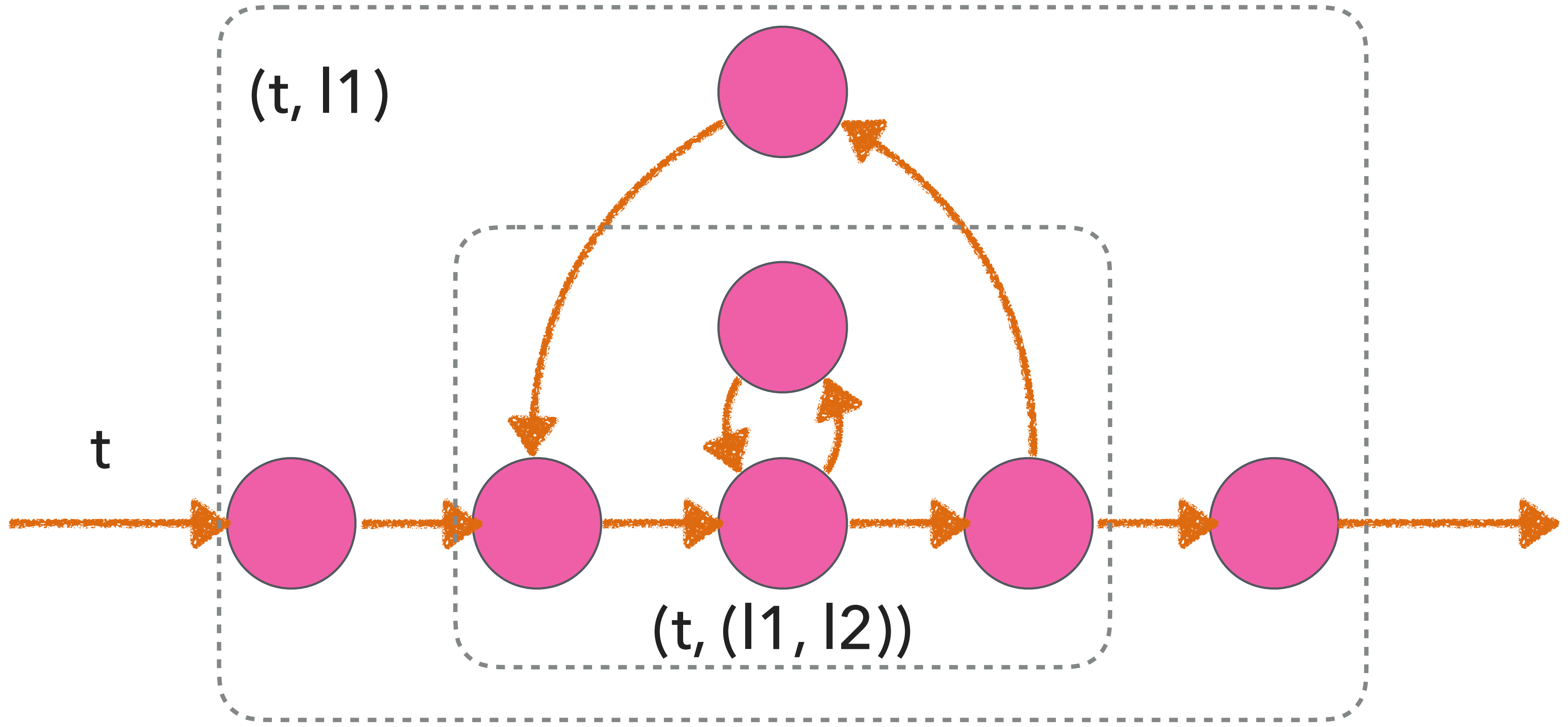
Streaming Iteration Example

Terminate after 100 iterations

```
timely::example(|scope| {

    let (handle, stream) = scope.loop_variable(100, 1);
    (0..10).to_stream(scope)
           .concat(&stream)
           .inspect(|x| println!("seen: {:?}", x))
           .connect_loop(handle);
});
```

Create the feedback loop



Blocking vs. Non-Blocking operators

- A **Blocking query operator** can only return answers when it detects the end of its input.
  - NOT IN, set difference and division, traditional SQL aggregates
- A **Non-blocking query operator** can produce answers incrementally as new input records arrive.
  - projection, selection, union

Window Operators

- Probably the most important operators in stream processing systems
- Almost universally supported across streaming systems and languages albeit with *various names and semantics*
- Allow un-blocking the processing of blocking operators by defining bounded portions of the stream on which computations can be performed

Window types (I)

- **Time-based** (logical) windows define their contents as a function of time.
  - average price of items bought within the last 5 minutes
- **Count-based** (physical) windows define their contents according to the number of events.
  - average price of last ten items bought

Window types (II)

- **Fixed** windows have bound which don't move
    - events received between 1/1/2019 and 12/1/2019
- **Landmark** windows have a fixed lower bound and the upper bound advances for every new event
    - all events since 1/1/2019
- **Sliding** windows have fixed size but both their bounds advance for new events
    - last 10 events or event in the last minute
- **Tumble** windows are non-overlapping fixed-size
    - events every hour
- **Custom** windows have neither fixed bounds nor fixed size
    - events in a period during which a user was active

Flow Management Operators (I)

- **Join** operators merge two streams by matching elements satisfying a condition
  - commonly applied on windows
- **Union** operators combine two or more streams without ordering guarantees
  - elements have to be of the same type
- **Difference** operators take two streams and output elements present in the first but not in the second
  - it is blocking and must be defined over a window

Flow Management Operators (II)

- **Duplicate/Copy** Operator replicates a stream, commonly to be used as input to multiple downstream operators.
- **Group by / Partition** Operators split a stream into sub-streams according to a function or the event contents.
    - one stream per customer Id
    - round-robin assignment

CQL GroupBy Example

```
Select IStream(Count(*))
From S1 [Rows 1000]
Group By S1.B
```

Count the number or events in the last 1000 rows for each value of B

What kind of queries can we express and support on data streams?

Non-blocking (monotonic) queries are the only continuous queries that can be supported on data streams.

Proposition:

Only monotonic queries can be expressed by non-blocking operators.

Then:

Can all monotonic queries be expressed using only non-blocking operators?

Model and formalization (I)

A stream is a sequence of unbounded length, where tuples are **ordered** by their arrival time.

Sequence: Let $t_1, \dots, t_n$ be tuples from a relation R. The list $S = [t_1, \dots, t_n]$ is called a sequence, of length n, of tuples from R.

The empty sequence [ ] has length 0.

We use $t \in S$ to denote that, for some $1 \leq i \leq n$, $t_i = t$.

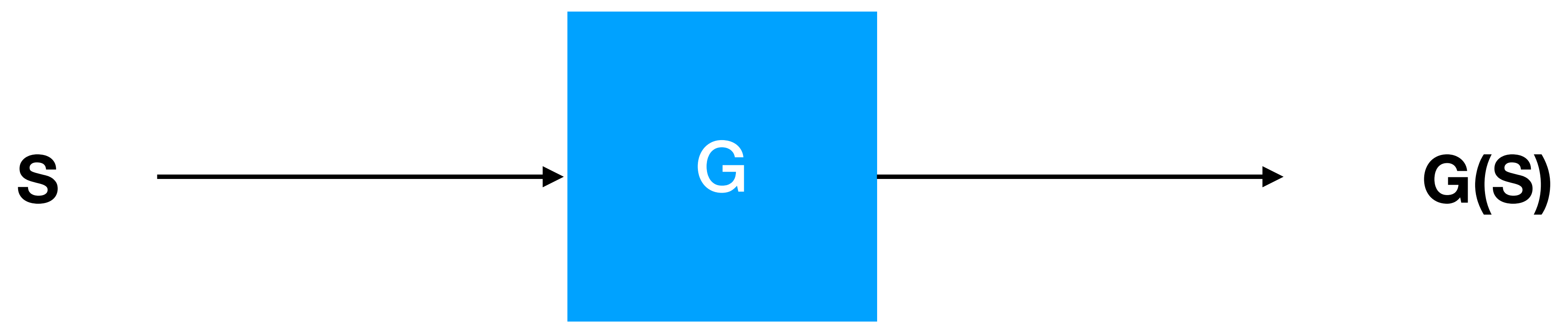
Model and formalization (II)

Pre-sequence (prefix): Let $S = [t_1, \ldots, t_n]$ be a sequence and $0 < k \leq n$. Then, $t_1, \ldots, t_k$ is the pre-sequence of S of length k, denoted by S^k .

[ ] is the zero-length pre-sequence of S.

Partial Order: Let S and L be two sequences. Then, if for some k, $L^k = S$ we say that S is a pre-sequence of L and write $S \subseteq L$.

If $k < n$, we say that S is a *proper* pre-sequence of L and write $S \subset L$.



Given a relation R, ⊆ is a partial order on sequences of tuples from R.

Streaming operators take sequences (streams) as input and return sequences (streams) as output:

For each new input tuple in S, G adds zero, one, or several tuples to the output.

Let $G^j(S)$ be the **cumulative** output produced by G up to step j.

What if n = 5 and S = [3, 3, 4, 4, 5]?

Consider a sequence of length n, i.e., $S = S^n$.

If G is a *traditional (blocking)* sum:

- what is $G^j(S)$ for $j < n$?
- for $j = n$?

What if $n = 5$ and $S = [3, 3, 4, 4, 5]$?

Consider a sequence of length n, i.e., $S = S^n$.

If G is a *continuous* sum, so that it returns the sum of all tuples seen so far:

- what is $G^j(S)$ for $j < n$?
- for $j = n$?

$G^j(S) \subseteq G^k(S)$, for $j \leq k$ — i.e., the output produced till step j is a pre-sequence of that produced till step k.

A **null** operator N is one where N(S) = [ ] for every S.

A non-null operator G is

- **blocking**, when for every sequence S of length n, $G^j(S)$ = [ ] for every $j < n$, and $G^n(S) = G(S)$
- **non-blocking**, when for every sequence S of length n, $G^j(S) = G(S^j)$, for every $j \leq n$.
- **partially blocking**, when it does not satisfy either definition, i.e., those where, for some S and j: $[\,] \subset G^j(S) \subset G(S^j)$

What functions on streams can be expressed using non-blocking operators?

Proposition: A function F(S) on a sequence S can be computed using a non-blocking operator, iff F is monotonic with respect to the partial ordering $\subseteq$.

A query Q on a stream S can be implemented by a non-blocking query operator iff Q(S) is monotonic with respect to $\subseteq$.

The traditional aggregate operators (max, avg, etc.) always return a sequence of length one and they are all non-monotonic, and therefore blocking.

Continuous count and sum are monotonic and non-blocking, and thus suitable for continuous queries.

Non-blocking SQL

Let *NB*-SQL be the non-blocking subset of SQL that excludes non-monotonic constructs:

- EXCEPT, NOT EXIST, NOT IN and ALL
- all standard blocking aggregates

Can we express all streaming (monotonic queries) with NB-SQL?

Non-blocking SQL

Some queries expressed using aggregates are *monotonic*:

```
SELECT DeptNo
FROM empl
GROUP BY DeptNo
HAVING SUM(empl.Sal) > 10000
```

The introduction of a new empl can only *expand* the set of departments that satisfy this query

However this sum query cannot be expressed without the use of aggregates!

SQL extensions and SQL-like languages

SQL extensions for streams

Why SQL-based approaches?

- Ideally, we would like to use the same language for querying both streaming and static data.

Requirements (or why SQL is not enough)

- Push-based model as opposed to the pull-based model of SQL, i.e. an application or client asks for the query results when they need them.
- The stream might never end in which case how to define blocking operators, e.g. groupBy?
- The data might be too large to store for future use.

ESL: Expressive Stream Language

- **Ad-hoc** SQL queries
- **Updates** on database tables
- **Continuous** queries on data streams
- New streams (derived) are defined as **virtual views** in SQL
  - Semantics are equivalent to having an append-only table to which new tuples are continuously added.

Example: CREATE STREAM

```
CREATE STREAM OpenAuction(
itemID INT, sellerID CHAR(10),
start_price REAL, start_time TIMESTAMP)
ORDER BY start_time SOURCE …
```

It needs to define external source and timestamp field.

```
CREATE STREAM expensiveItems AS(
SELECT itemID, start_price, start_time
FROM OpenAuction WHERE start_price > 1000
```

Derived stream as an append-only table.

User-Defined Aggregates (UDAs)

Constructs that allow the definition of custom aggregations using three statement groups:

- `INITIALIZE`: initialized local state.
- `ITERATE`: update state based on new element and current state.
- `TERMINATE`: produce the result.

Note that it is allowed to define and maintain local tables as state.

Example: AVG UDA

```
AGGREGATE myavg(Next Int): Real
{     TABLE state(tsum Int, cnt Int);
       INITIALIZE: {
            INSERT INTO state VALUES(Next, 1);
       }
       ITERATE: {
            UPDATE state
                 SET tsum=tsum+Next, cnt=cnt+1;
       }
       TERMINATE: {
            INSERT INTO RETURN
                 SELECT tsum/cnt FROM state;
       }
}
```

Allocated just before INITIALIAZE and deallocated just after TERMINATE.

Can you define a MIN UDA?

```
AGGREGATE mymin(Next Int): Int
{     TABLE state(…);
       INITIALIZE: {

       }
       ITERATE: {

      }
       TERMINATE: {


       }
}
```

Example: AVG UDA

```
AGGREGATE myavg(Next Int): Real
{     TABLE state(tsum Int, cnt Int);
      INITIALIZE: {
            INSERT INTO state VALUES(Next, 1);
      }
      ITERATE: {
            UPDATE state
                  SET tsum=tsum+Next, cnt=cnt+1;
      }
      TERMINATE: {
            INSERT INTO RETURN
                  SELECT tsum/cnt FROM state;
      }
}
```

This is a blocking UDA: TERMINATE is executed once the stream is over.

Example: Non-blocking AVG UDA

```
AGGREGATE myavg(Next Int): Real
{    TABLE state(tsum Int, cnt Int);
     INITIALIZE: {
          INSERT INTO state VALUES(Next, 1);
     }
     ITERATE: {
          UPDATE state
               SET tsum=tsum+Next, cnt=cnt+1;
     }
     TERMINATE: {}
}
```

While iterating the stream elements maybe?

Can we return results earlier than TERMINATE?

Example: Non-blocking AVG UDA

```
AGGREGATE online_avg(Next Int): Real
{     TABLE state(tsum Int, cnt Int);
      INITIALIZE: {
          INSERT INTO state VALUES(Next, 1);
      }
      ITERATE: {
          UPDATE state
              SET tsum=tsum+Next, cnt=cnt+1;

  INSERT INTO RETURN

    SELECT tsum/cnt FROM state

    WHERE cnt % 200 = 0;
      }
      TERMINATE: {}
}
```

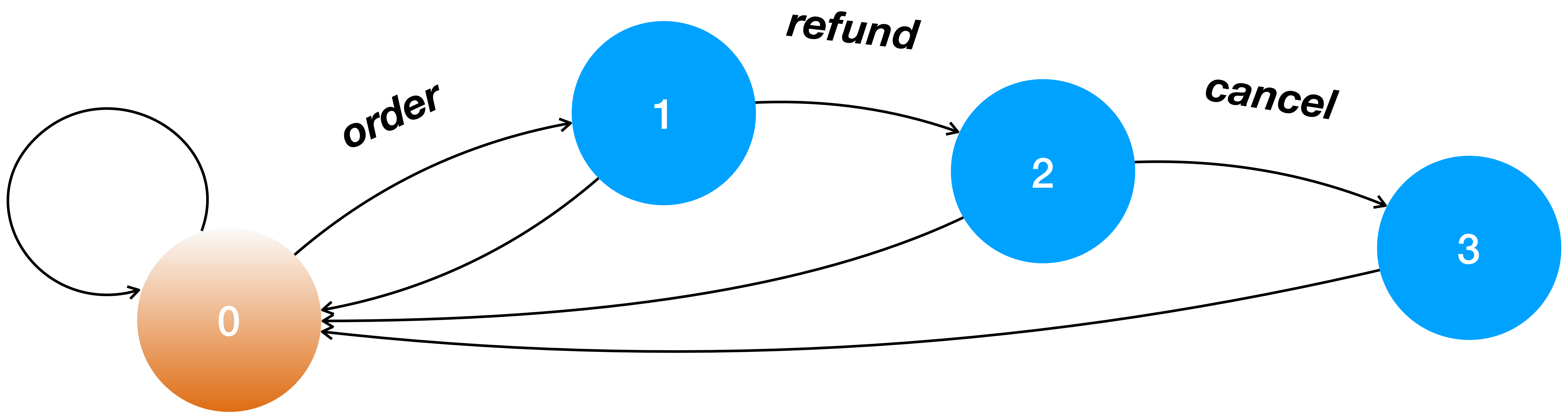
Continuously return a result every 200 tuples.

If TERMINATE is empty, the aggregate is non-blocking.

Pattern Queries with UDAs

- UDAs process streams tuple-per-tuple
- How can we write a UDA that detects a sequence of actions?
  - e.g. detect users who place an order, ask for a refund immediately, and then cancel the order

webevents(CustomerID, ItemID, Event, Amount, Time)



Pattern-Matching UDA

```
AGGREGATE pattern(CustomerID Char, Next Char):(Char, Char)
{     TABLE state(sno Int);
        INITIALIZE : {
          INSERT INTO state VALUES(0);
          UPDATE state SET sno = 1 WHERE Next = 'order';}
        ITERATE: {
          UPDATE state SET sno = 0
            WHERE NOT (sno = 1 AND Next = 'refund')
              AND NOT (sno = 2 AND Next = 'cancel')
              AND Next <> 'order'
          UPDATE state SET sno = 1 WHERE Next = 'order';
          UPDATE state SET sno = sno+1
            WHERE (sno = 1 AND Next = 'refund')
              OR (sno = 2 AND Next = 'cancel')
          INSERT INTO RETURN
            SELECT CustomerID,'pattern123' FROM state
              WHERE sno = 3;
} }
```

Check next event

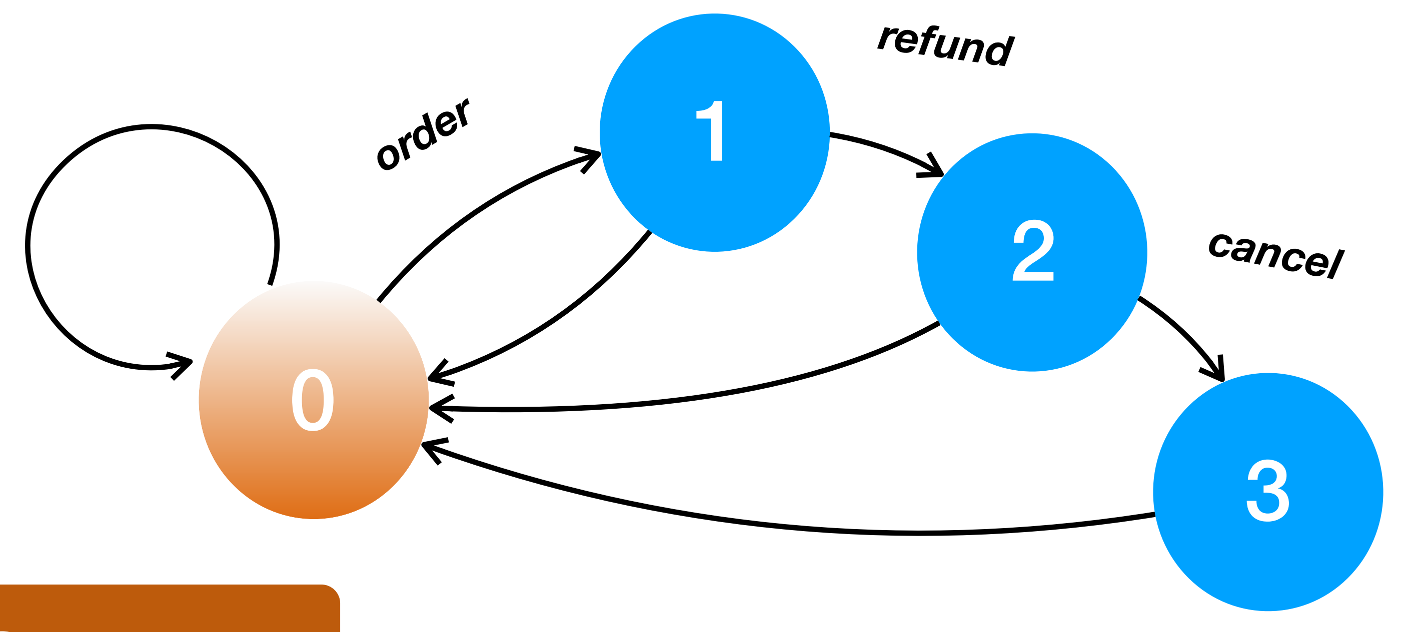
Initialize state to 0

Pattern failed

Order matched

Refund and cancel matched

Output success!



Pattern-Matching: a simpler approach

```
SELECT 'modified-pattern123', X.CustomerId
FROM webevents
    PARTITION BY CustomerId
    AS PATTERN (X Y Z)
WHERE
    X.Event = 'order' AND
    Y.Event = 'rebate' AND Y.ItemID = X.ItemID AND
    Z.Event = 'cancel' AND Z.ItemID = Y.ItemID
```

Partitions the stream into substreams according to a key

A sequence of events that immediately follow one another

- **Match zero or more successive events:**

```
AS PATTERN (X V* Y W* Z)
```

- **Match within a time limit:**

```
Z.Time - Y.Time < 60
```

NB-Completeness

Proposition: Every computable monotonic function on timestamped data streams can be expressed using *NB*-UDAs and union

where

NB-UDAs are those where TERMINATE is empty.

UDAs on a single Stream

Every monotonic function F on an input data stream can be computed by a UDA that uses three local tables, IN, TAPE, and OUT, and performs the following operations for each new arriving tuple:

1. Append the encoded new tuple to IN,
2. Copy IN to TAPE, and compute F(IN) – OUT
3. Return the result obtained in 2 and append it to OUT.

Non-blocking

Timestamped streams

Pre-sequence: Let S and R be two sequences ordered by their timestamp and R^{τ} be the set of tuples of R with timestamp less than or equal to $\tau > 0$.

If $S = R^{\tau}$ for some τ , then S is pre-sequence of R, denoted $S \subseteq^{\tau} R$.

In general, if $S_1, ..., S_n$ and $R_1, ..., R_n$ be timestamped sequences, then

$(S_1, ..., S_n) \subseteq^{\tau} (R_1, ..., R_n)$ when $(S_1, ..., S_n) = (R_1, ..., R_n)$ for some τ .

A *unary* operator G is **monotonic** if $L_1 \subseteq^\tau S_1$ implies $G(L_1) \subseteq^\tau G(S_1)$.

A *binary* operator H is **monotonic** when $(L_1, L_2) \subseteq^\tau (S_1, S_2)$ implies $H(L_1, L_2) \subseteq^\tau H(S_1, S_2)$.

For $\tau = 0$, $S\ \tau = \varnothing$ is an empty sequence.

A query operator is **null** when it returns the empty sequence for every possible value of its argument(s).

A non-null *unary* operator G is **non-blocking**, when $G^\tau (S) = G(S^\tau)$, for every τ.

A non-null *binary* operator G is **non-blocking**, when, $G^\tau (L, S) = G(L^\tau , S^\tau)$, for every τ.

Union. Let $\cup^{\tau}$ denote the stream operator implementing union, i.e. $\cup^{\tau}$ returns, at any given time τ, the union of the τ-pre-sequences of its inputs:

$$L \cup^{\tau} S = L^{\tau} \cup S^{\tau}$$

Languages supporting union operators and non-blocking UDAs on data streams are *complete*, in the sense that they can express every monotonic function on their input.

Union & UDA example

Consider two streams of phone-call records: **`StartCall(callID, time)`** and **`EndCall(callID, time)`**

```
SELECT callID, length(time, tag) AS CallLength,
FROM
     (SELECT callID, time, 'start'
     FROM StartCall
     UNION ALL
     SELECT callID, time, 'end'
     FROM EndCall) AS
  CallRecord (callID, time, tag)
GROUP BY callID;
```

Computes the length of each call

```
AGGREGATE length(time, tag) : (CallLength)
{     TABLE state(ttime);
      INITIALIZE:
      ITERATE :{
        INSERT INTO state VALUES(time);
        INSERT INTO RETURN
          SELECT time-ttime FROM state
          WHERE tag='end';
        INSERT INTO RETURN
          SELECT ttime-time FROM state
          WHERE tag='start';
      }
}
```

Why do we need all INSERT blocks?

Summary

Today you learned:

- there are various types of languages for data streams
    - patterns, transformations, declarative
- traditional blocking operators don't work on streams
    - non-blocking versions or windows
- how to define non-blocking aggregates
- *NB*-SQL can be extended with union and UDAs to express all non-blocking, streaming queries

Lecture references

- Gianpaolo Cugola and Alessandro Margara. **Processing flows of information: From data stream to complex event processing**. ACM Comput. Surv. 44, 3, Article 15 (June 2012).

- Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. **Data Stream Management: Processing High-Speed Data Streams**. Springer-Verlag, Berlin, Heidelberg.

- David Maier, Jin Li, Peter Tucker, Kristin Tufte, and Vassilis Papadimos. **Semantics of data streams and operators**. In Proceedings of the 10th international conference on Database Theory (ICDT'05).

- Yan-Nei Law, Haixun Wang, and Carlo Zaniolo. **Query languages and data models for database sequences and data streams**. In Proceedings of the Thirtieth international conference on Very large data bases - Volume 30 (VLDB '04).