Overview

Stream Processing

- **Applications**
  - Stock Markets
  - Internet of Things
  - Intrusion Detection
- **Central Idea**
  - **Classical Queries**: Queries Change, Data Fixed
  - **View Maintenance**: Data Changes, Queries Fixed, Slow Response
  - **Here**: Data Changes, Queries Fixed, Fast Response
- **Language Models**
  - Classical SQL w/ Windows
  - Stream-specific query langs
- **Challenges & Advantages**
  - Limited Compute Time: Want to deal with large numbers of records as they come in quickly.
  - All compute requirements (structurally, at least) are given upfront.
  - Typically specialized for bounded data sizes

Cayuga

Stream Definition Operators

- **SELECT x, y, z FROM [stream]**
  - Classical Projection. Optionally defines a new stream
  - Optional PUBLISH clause names the stream
- **FILTER { condition } [stream]**
  - Classical Selection. Pass only tuples that pass a condition
- **[stream] NEXT { condition } [stream]**
  - “JOIN”-like operation
  - For each tuple on the LHS
    - Find (and emit) the next tuple from the RHS that matches the condition
- **[stream] FOLD { group_condition, done_condition, aggregate } [stream]**
  - “JOIN+AGGREGATE”-like operation
  - For each tuple on the LHS
    - Start a group
    - Attach each tuple from the RHS that matches group_condition
    - Update the group with the aggregate expression
    - If the RHS tuple matches done_condition, close out the group and emit the aggregate

Discussion

- **Why not use regular joins**
  - Regular Joins are Non-Streaming
    - Unclear when a tuple stops being relevant
      - Unbounded memory use
      - Steadily growing compute
    - Language chosen to ensure finite state per tuple being joined
      - NEXT: State = unmatched tuples from LHS
        - One-One join
      - FOLD: State = unfinished groups: Constant per LHS tuple
        - One-Many join
    - **What about many/many?**
  - Hard to express temporal relationships w/ joins
    - WHERE t2 > t1 and/or some sort of nested subquery trickery to get LIMIT

Autometa

DFA

Data Model

- Nodes represent states
- Edges represent transitions
- One node designated as the “start” state
- One or more nodes designated as “terminal” or “output” states

Language

- Start with an alphabet [Sigma]
- Edges labeled with letters in the alphabet
- Every node has an out-edge for every letter in the alphabet
  - Implicit ‘error’ state if no edge for a letter given explicitly

Evaluation

- Given a string in [Sigma]
- For each letter in the string travel the edge with the same label.
- "Success" if you end in one of the terminal states.

NDFA

Data Model

- Same as DFA, but allowed to have >1 edges with the same label.

Evaluation

- At any given point in time, you can be “present” at multiple nodes/states
- If at a state with multiple out-edges labeled with the same letter as the next letter in the string, travel to all of them in parallel

Reduction to DFA

- Given an NDFA with N states (e.g., {A, B, C}), create a new graph with 2^N states, call them hyperstates ({ {}, {A}, {B}, {C}, {AB}, {AC}, {BC}, {ABC})
- Each state represents the state of the NDFA where you are in some subset of the N states (there are 2^N such states)
- For each hyperstate (e.g., {AB})...
  - For each letter in the alphabet
    - For each state in the hyperstate (e.g., A and B)
      - Compute the set of states that the state would transition to for that letter
    - Compute the union of these states
    - This is the hyperstate that you transition to

Cayuga-Autometa

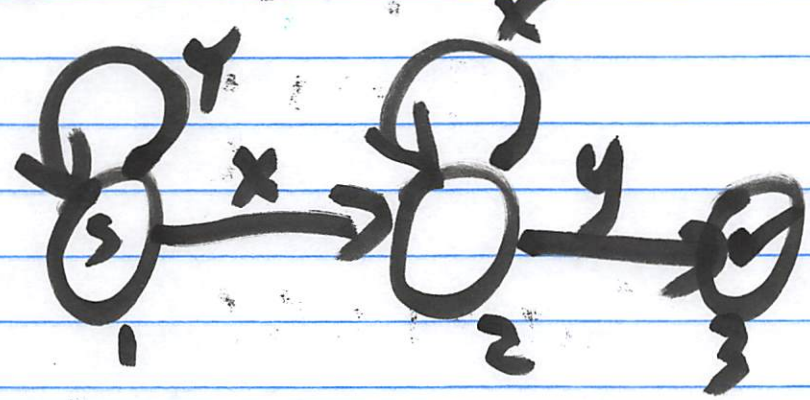
Data Model

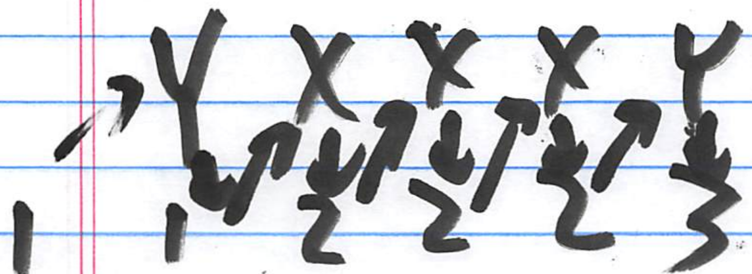
- Same as NDFA, but extended in one additional dimension: Every state has a set of associated instances
  - Like a generalization from Zeroth- to First-order logic
  - AliceIsAStudent -> AliceIsInClass vs IsStudent(x) -> IsInClass(x)
  - Strictly more powerful (infinite number of states)
- In short, every state behaves like a relation
- Edges represent opportunities for tuples to travel from one relation to another.
- Edges are labeled with
  - Condition (for the tuple to travel)
  - Projection rule (for generating the new tuple)

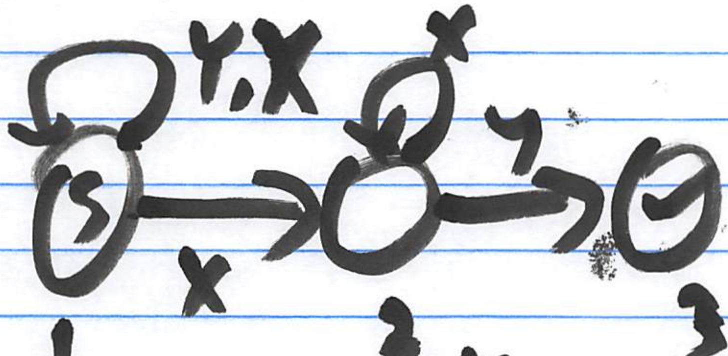
Reducing CEL to Cayuga

- SELECT
  - (True, Projection Targets) -> Next State
- NEXT
  - (~condition, ID) -> Same State
  - (condition, ID) -> Next State
- FOLD
  - (group_condition, aggregate) -> Same State
  - (~group_condition, ID) -> Same State
  - (done_condition, ID) -> Next State

alpha $\Sigma = [x, Y]^N$







Y x x x Y

{1} {1} {1,2} {1,2} {1,2} {1,3}

stream :=

SELECT x,y,z FROM $\boxed{\text{stream}}$

↳ RA proj

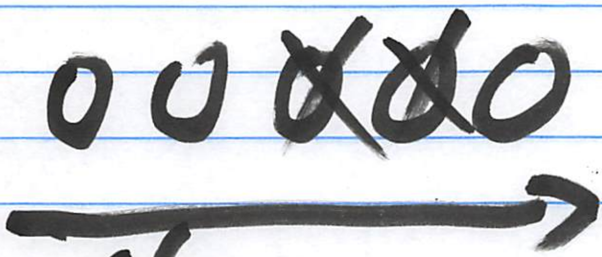
FILTER { φ } [$\boxed{\text{stream}}$]

↳ RA selection on φ

[$\overset{A}{\boxed{\text{stream}}}$] NEXT { φ } [$\overset{B}{\boxed{\text{stream}}}$]

↳ JOIN lite

↳ for each A
find <u>next</u> B matching φ



stream

SQL/RA

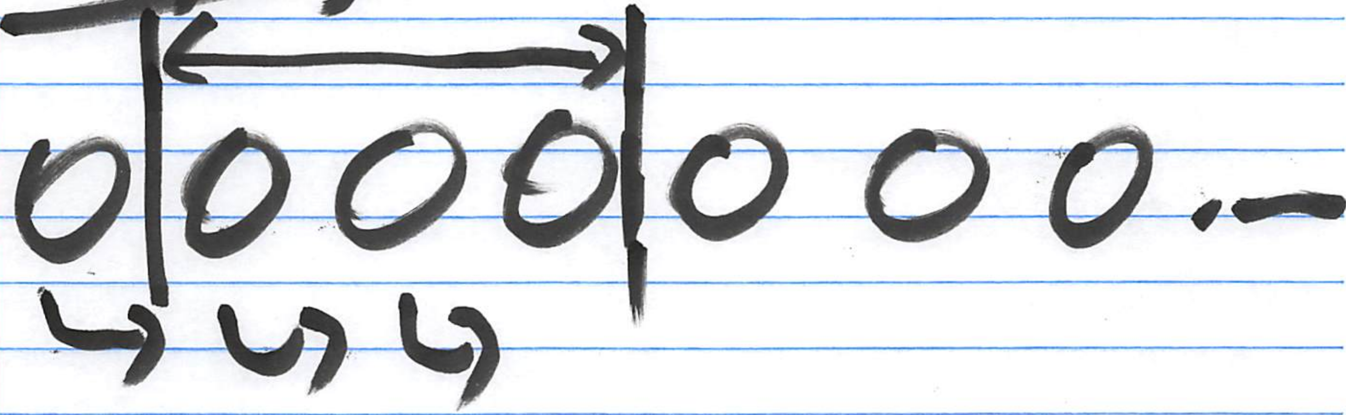
Filter == SELECT

Aggregate
(Group-By)

PROJECT

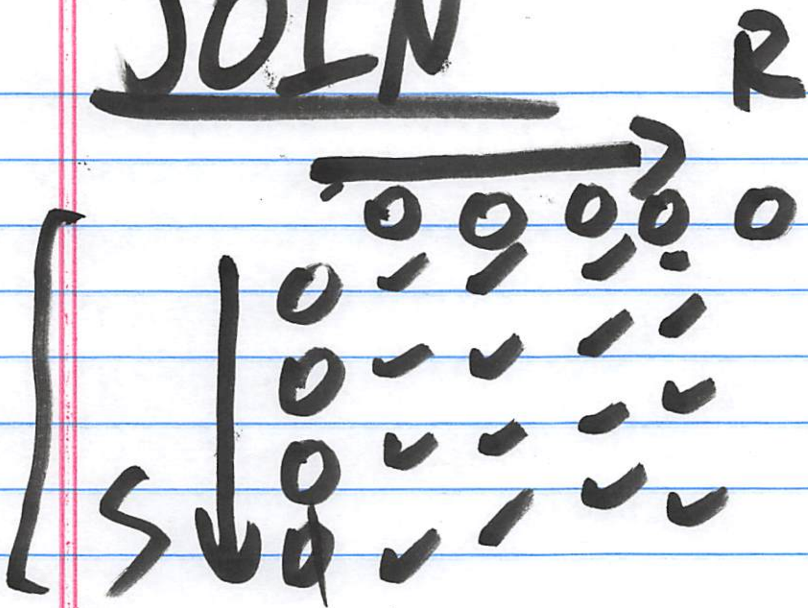
JOIN?

Aggregate



- Cumulative
  - per tuple
  - per 'breakpoint'
  - externally triggered
- GROUP BY
- WINDOW

JOIN



- WINDOW

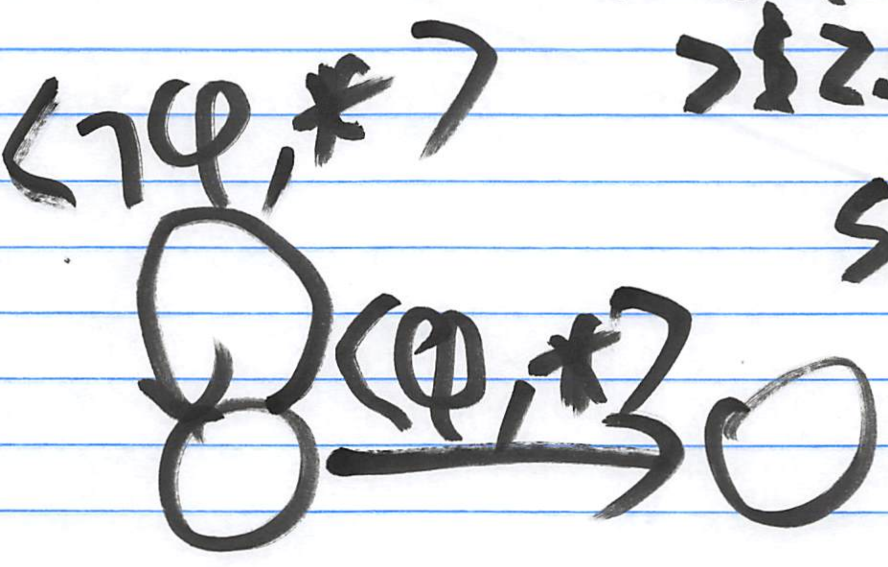
NEXT
- MERGE JOIN (sorted Data)
- 1-1 Join
  - ↳ But need to make sure matches show up fast

OLD
- 1-Many Join + Aggregate

Stocks (Ticker, Price)

$1 ϕ

Stocks NEXT{$1.ticker = $2.ticker AND $1.price > $2.price}$2 Stocks$2



IBM $22

MSFT $23

A

[stream] FOLD {
Group,
done,
aggregate
} [stream]

B

↳ 1-many join + agg
↳ Every tuple in A starts a group
↳ Agg over tuples in B that join on group
↳ Emit when done