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

N DFA

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 N DFA 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 N DFA 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 N DFA, 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
\[ \sum \in \left[ x, y \right]^n \]

Diagram:

\[ \begin{array}{c}
\text{X} & \text{X} & \text{X} & \text{X} & \text{X} \\
\text{X} & \text{X} & \text{X} & \text{X} & \text{X} \\
\text{X} & \text{X} & \text{X} & \text{X} & \text{X} \\
\text{X} & \text{X} & \text{X} & \text{X} & \text{X} \\
\text{X} & \text{X} & \text{X} & \text{X} & \text{X} \\
\text{X} & \text{X} & \text{X} & \text{X} & \text{X} \\
\end{array} \]

\[ e_{13} e_{13} e_{1,23} e_{1,23} e_{1,23} e_{1,33} \]
SELECT x, y, z FROM [stream]

FILTER (\( p \) \(\in\) \(\text{Stream}\))

\( p \) \(\in\) \(\text{RA select im 0 \& p}\)

\( p \) \(\in\) \(\text{Stream}\) \(\Rightarrow\) NEXT (\( p \) \(\in\) \(\text{Stream}\))

\( p \) \(\in\) \(\text{Stream}\) \(\Rightarrow\) JOIN (lite)

\( p \) \(\in\) \(\text{Stream}\) \(\Rightarrow\) for each A find next B matching \( p \)
Stream → Filter → SQL/RA

SELECT

Aggregate (Group-By)

PROJECT

JOIN?
Aggregate

0 | 0 | 0 | 0 | 0 | 0 | 0 | -

Cumulative

慢慢 per tuple

慢慢 per 'breakpoint'

慢慢 externally triggered

GROUP BY

WINDOW
JOIN

\[ \underline{R} \]

\[ \underline{\text{Window}} \]

- MERGE JOIN (sorted DMM)

- 1-1 Join

But need to make sure matches show up fast

\[ \underline{L} = \text{1-Many Join + Aggregate} \]
Stocks (Ticker, Price)

\[ \emptyset \]

Stocks NEXT if \( $1.\text{ticker} = $2.\text{ticker} \) AND \( $1.\text{price} > $2.\text{price} \)

\(<700, \ast>300\)

<IBM $22

<MSFT $23
A

[Stream] FOLD &
Group
done,
aggregate
3 [Stream]
B

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