Storage-centric Load Management for Data Streams with Update Semantics

Alexandru Moga, Systems Group @ ETH Zurich
Motivation

Real-time applications and update semantics

- **Data Stream Processing**
  - Continuous queries over append-only streams
  - Process everything with low latency

- **Real-time applications need up-to-date information**
  - Update semantics: Staleness is more important!

- **Overload happens ...**
  - Typical solution: random load shedding
    - Latency under control
    - Focus on accuracy loss
  - Problems:
    - Overhead in resource utilization
    - Random shedding may introduce unnecessary staleness
The UpStream Framework

**Problem**
- Minimizing staleness for streaming applications with update semantics

**Approach**
- Push update semantics down the processing pipeline
  - Why? Smarter and more effective load management
  - Where? As close to the source of overload as possible
  - How? Storage-centric framework based on update queues

**Goals**
- Optimize for staleness and resource utilization
- Control and stability

**Challenges**
- Correct stream processing
  - Windowing operations, etc.
- Optimization opportunities
  - Exploit stream & application characteristics
Outline

- Models
- Handling windows
- Exploiting stream characteristics
- Future work & Wrap-up
How to push-down update semantics?

Staleness:
To what extent does the output reflect the input? VS.

Latency:
How fast is the output produced?

Data Source

input stream

Continuous Query

update stream

output stream

update stream

Push down update semantics
How to push-down update semantics?

Key Load Management
- “in-place” updates

Key Scheduling
- order of processing keys

Data Source

Update Queue

Continuous Query

Push down query semantics

Update stream

Push down update semantics
Handling windows

- **Window-aware update queue**
  - Update unit = full window!
    - Push down windowing information (size, slide)
  - Keep track of the most recent window (MRW)

- **Challenges**
  - Correctness $\rightarrow$ Tuple marking
  - Define MRW + Memory management $\rightarrow$ Window buffers
Window buffers

- Data structures
  - Newly forming windows (enqueue buffer)
  - Committed windows (dequeue buffer)

- What constitutes the most recent window?
  - **LAZY** approach
    - (When the current window closes)
  - **EAGER** approach
    - (When a new window opens)

Max. memory requirements: \(2w + s\)

Max. memory requirements: \(w + s\)
EAGER vs. LAZY

Incremental processing
- query applies processing per window tuple

Non-incremental processing
- query applies processing per window result
Key Scheduling

- Key-based queries

- Stream segregation into key sub-streams using hashing
  - Update queue is a queue of keys (not tuples!)

- IN-PLACE key scheduling
  - Minimize the maximum waiting time
  - ~ Fair queuing
IN-PLACE vs. Random Drops

Staleness per key gets bounded w/ LF

Staleness scales linearly w/ # of Keys

Given all keys update at the same frequency, IN-PLACE is the best
Non-uniform update frequencies

- **Problem:** IN-PLACE treats all keys “too” fairly

- **LINECUTTING**
  - **Approach:** process slow updating keys as soon as possible
  - **How?** Minimize the maximum \((S+W)\)
    - **S** ("slowness"): enables the promotion of slow keys
    - **W**: prevents starvation for fast keys

- **Symmetrical distribution (20 keys)**
  - Higher skew favors LINECUTTING
  - Higher load limits the benefits
Future work

- **Real-time scheduling**
  - Generalized LINECUTTING
  - Optimality
  - Access frequencies
  - Multi-query scheduling

- **Adaptive load management**
  - Hybrid semantics
  - Decentralized processing model
Wrap-up

✓ Efficient load management for applications with update semantics

✓ Means to push-down semantics in a stream processing system

✓ Storage-centric approach: update queues
  ▪ Key-level management (query semantics push-down: e.g. windowing)
  ▪ Key scheduling (exploiting stream characteristics)
  ▪ Prototype implementation in Borealis stream processing engine

✓ UpStream
  ➢ Control and adaptability to load
  ➢ Minimum and stable staleness levels
  ➢ Bounded memory usage