MICS

FNSNF
Fonds national suisse
Schweizerischer Nationalfonds
Fondo nazionale svizzero
Swiss National Science Foundation

upstream

A Storage-centric Load Management System for Real-time Update Streams

Alexandru Moga and Nesime Tatbul, Systems Group, ETH Zurich, Switzerland

Systems@ETH zürich

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Streaming Applications with Update Semantics

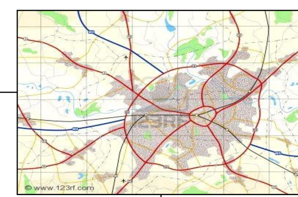
Both **data streams** and **continuous queries** can exhibit update semantics …

example 1: *Financial services*

```
Streams: Trades(time, symbolId, price, volume)
         Quotes(time, symbolId, bid, bidVol, ask, askVol)

Query: For each symbol, continuously update
       the total trade volume and average trading price
       of the last 10 minutes.
```

example 2: *Road traffic monitoring*

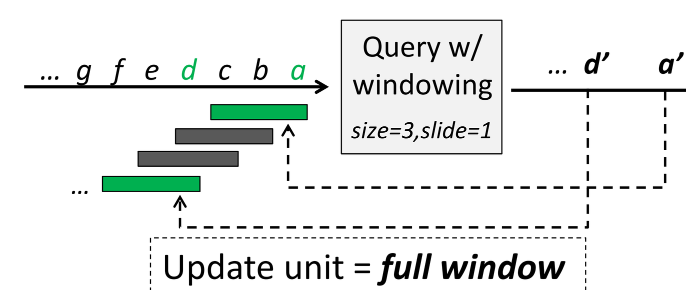


```
Stream: Positions(time, objectId, x, y, speed)

Query: For each object, continuously update
       the number of cars in range R of the object
       in the last minute.
```

Challenges: Controlled Load Shedding

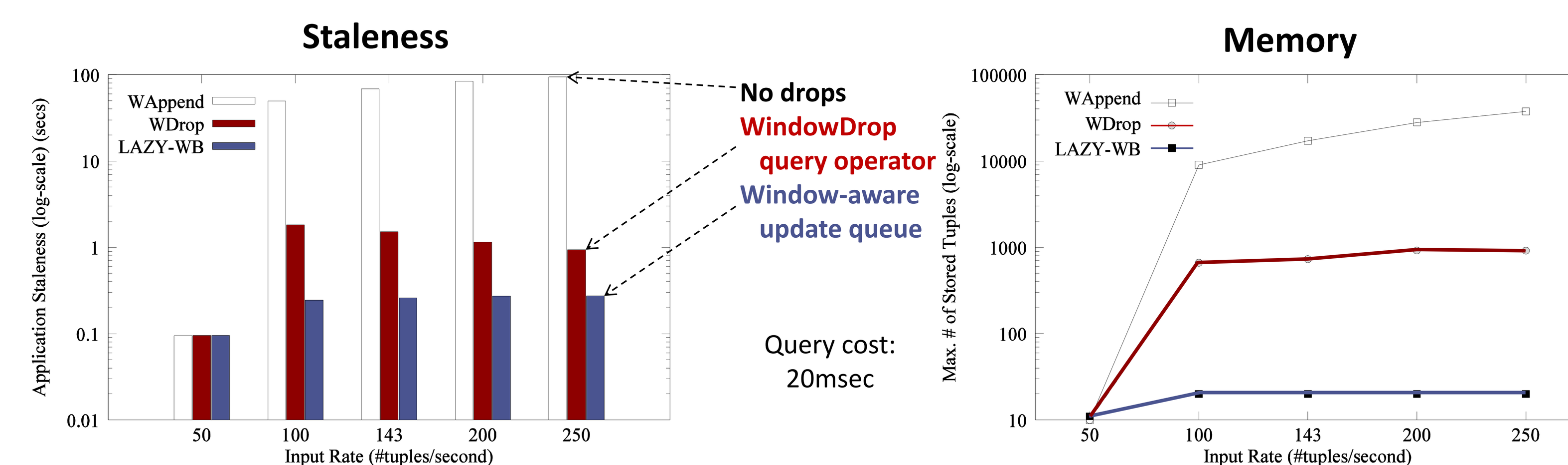
Update-aware load shedding via subset-based approximation



Window Buffers

- Keep the most recent window
- Shed older window tuples using
  - ✓ windowing information (size w, slide s)
  - ✓ tuple marking scheme

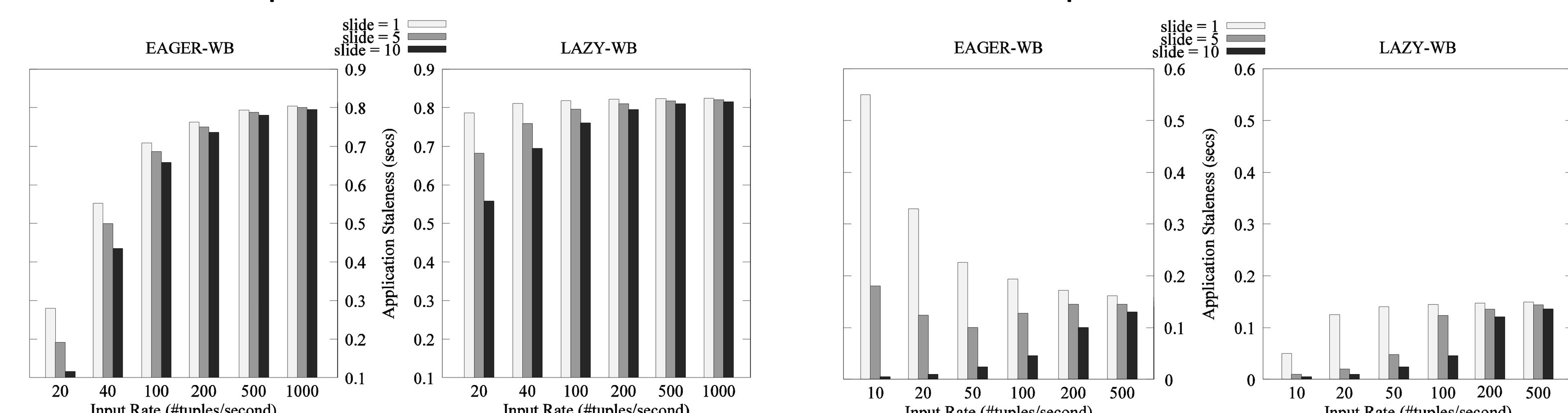
Window updates vs. Window random drops



EAGER vs. LAZY window updates

EAGER: update on **window start**

LAZY: update on **window end**



Incremental window processing (cost per tuple)

Non-incremental window processing (cost per window)

The Problem

Update semantics: **the latest result is all that really matters**

Staleness: **how out-of-date results are compared to latest data**

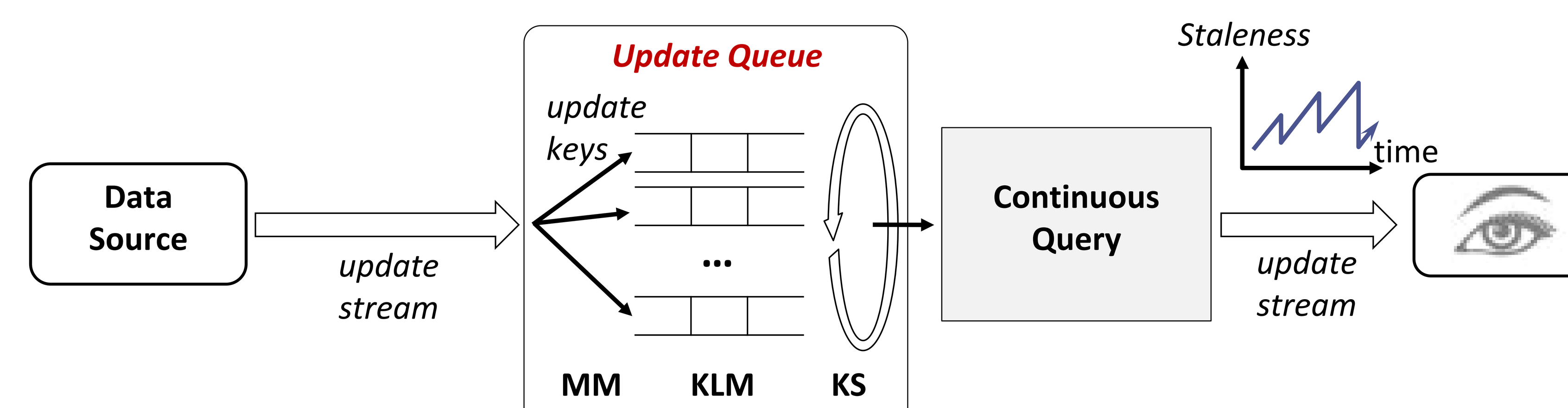
"How to minimize
staleness of results for
streaming applications with **update semantics**
under conditions of **high load**?"

Overload: **High-volume streams can exceed processing capacity leading to stale results**

The UpStream Framework

Key insight: Exploit update semantics!

I. Shed load to keep up with new updates
II. Push update semantics upstream
   Why? More effective load management → *Preventive*
   Where? Closer to source of overload → *Memory efficient*
   How? **Storage-centric framework** based on **update queues**



Memory Management (MM)
- On-demand paging
- Early vs. Late garbage collection

Key Scheduling (KS)
- Group stream tuples by **update key**
- Order update keys for processing

Key Load Management (KLM) →Update-aware load shedding
- *Update semantics*: keep the most recent update (**in-place updates**)
- *Query semantics*: represent the update unit (e.g. tuple, window of tuples)

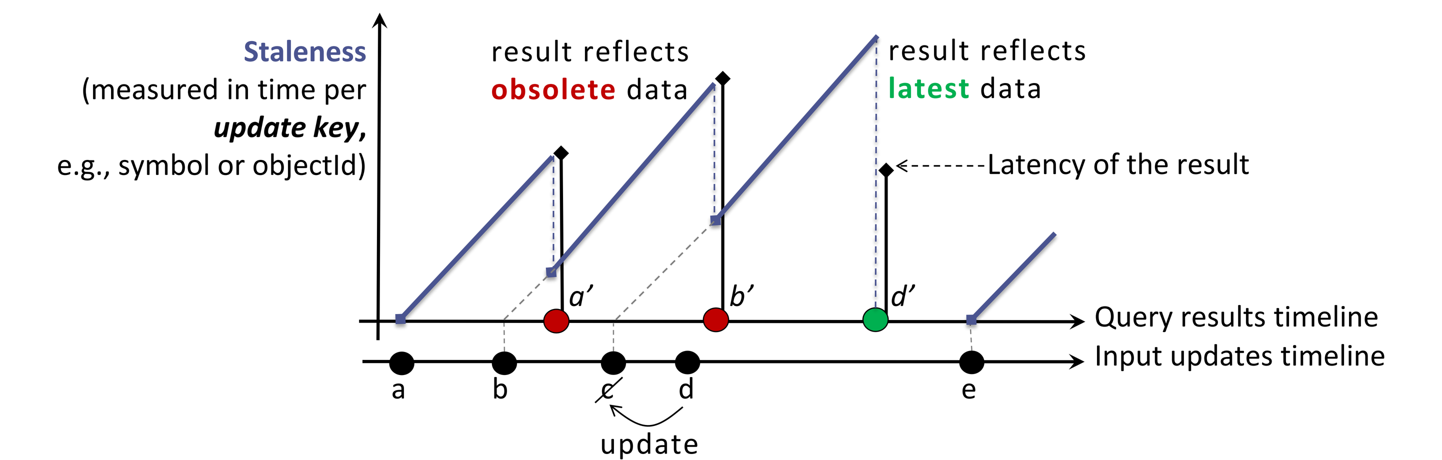
Further details

- The UpStream system is a prototype implementation on top of the **Borealis** SPE.
- **Website**: http://www.systems.ethz.ch/research/projects/upstream/
- See also: "*UpStream: Storage-centric Load Management for Streaming Applications with Update Semantics*", **VLDB Journal**, 2011.

Staleness

We study the notion of staleness in the streaming context

- Push-based processing: updates and queries are part of the same pipeline
- Data is transient, queries are persistent
- Other studies: web and soft real-time databases, stream warehouses



Staleness is **intrinsic** to applications with **update semantics**

Opportunities: Frequency-Aware Key Scheduling

Uniform update rates: **IN-PLACE Key Scheduling**

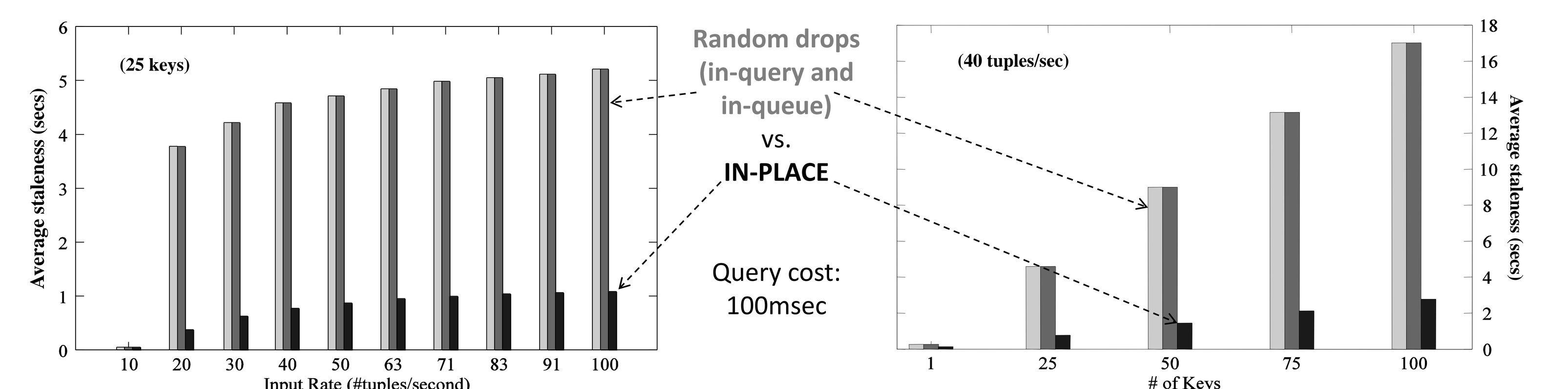
- Order keys by earliest enqueue time
- ~ Fair Queuing with in-place updates

… < W(i) < … < W(2) < W(1)

… | k(i) | … | k(2) | k(1) → CQ

IN-PLACE minimizes the maximum waiting time *W*

- Bounded average staleness per key
- Optimal performance for uniform update rates



Non-uniform update rates: **LINECUTTING Key Scheduling**

- Update frequencies are rarely uniform (e.g., stock price updates)
- IN-PLACE treats all keys "too fairly"
- Idea: Adaptively promote *slow updating keys*

LINECUTTING minimizes the maximum
S + W

W prevents starvation of *fast* keys

S ("slowness") enables promotion of *slow* keys