Noria:
Partially-Stateful Data-flow

Ashvin Goel

Electrical and Computer Engineering
University of Toronto

ECE1724

Authors: Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, and Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek and Robert Morris
Motivation

Modern web apps add an in-memory cache in front of traditional databases for high performance.

- Classic database reads are expensive
- Database + in-memory cache reads are fast
Motivation

In-memory cache work well, but programmer needs to handle invalidations correctly

Incorrect design may lead to stale data in caches forever

Database + in-memory cache reads are fast
Motivation

What about using a dataflow/streaming database?

Write path is expensive
Primarily support window operations

Generate data that may never be read
Require significant memory for caching
Key Idea in Noria

Use a partial-state dataflow model

- Cache state on reads: Like traditional cache
- Update cached state on writes: Like dataflow databases
- Evict cache state if needed: Limits memory requirements
- No cost for updating evicted state: Reduces cost of writes
- Enables dynamically updating dataflow model: Simplifies adding and removing queries that need caching
Partially-stateful data-flow

Data-flow state is *partial*: entries for some keys are absent (⊥).
Partially-stateful data-flow

Data-flow state is *partial*: entries for some keys are absent (⊥).
Partially-stateful data-flow

Data-flow state is *partial*: entries for some keys are absent (⊥).
Partially-stateful data-flow: upqueries
Partially-stateful data-flow: upqueries

Need to fill absent entry
Partially-stateful data-flow: upqueries

Solution: *upquery* through data-flow.
- Compute missing entry from upstream state

Need to fill absent entry
Partially-stateful data-flow: upqueries

Solution: upquery through data-flow.
- Compute missing entry from upstream state
- Response fills missing entry
Partially-stateful data-flow: upqueries

Solution: upquery through data-flow.
• Compute missing entry from upstream state
• Response fills missing entry
Partial state enables live data-flow changes

Start new views and operator state **empty**, fill via **upqueries**.
Partial state enables live data-flow changes

Start new views and operator state **empty**, **fill via upqueries**.
Partial state enables live data-flow changes

Start new views and operator state **empty**, fill via **upqueries**.
Partial state enables live data-flow changes

Start new views and operator state **empty**, fill via **upqueries**.
Partial state enables live data-flow changes

Start new views and operator state empty, fill via upqueries.
High performance requires concurrency

Process operators concurrently. Read from views concurrently. Process shards concurrently.

Without global coordination!
Challenges implementing partially-stateful data-flow

1. Concurrent upqueries and update processing — races!

Must maintain correctness under concurrency!
Correctness under concurrency

**Goal:** upquery restores state as if present all along.
Correctness under concurrency

**Goal:** upquery restores state as if present all along.
Correctness under concurrency

**Goal:** upquery restores state as if present all along.
Correctness under concurrency

**Goal:** upquery restores state as if present all along.
Correctness under concurrency

**Goal:** upquery restores state as if present all along.
Correctness under concurrency

Goal: upquery restores state as if present all along.

Upquery response is a snapshot of state
Correctness under concurrency

**Goal:** upquery restores state as if present all along.

Upquery response is a **snapshot** of state

includes 2 1

does not include 3
Correctness under concurrency

**Goal:** upquery restores state as if present all along.

Upquery response is a **snapshot** of state

includes **2 1**
does **not** include **3**

**Solution:** Maintain **order** of upquery response and surrounding updates, despite lack of global coordination.
Challenges implementing partially-stateful data-flow

1. Concurrent upqueries and forward processing — races!

   Must maintain **correctness** under concurrency!

2. Update processing may require absent state
Challenges implementing partially-stateful data-flow

1. Concurrent upqueries and forward processing — races!

   Must maintain **correctness** under concurrency!

2. Update processing may require absent state
Challenges implementing partially-stateful data-flow

1. Concurrent upqueries and forward processing — races!
   Must maintain **correctness** under concurrency!

2. Update processing may require absent state

   Drop updates that touch absent state, future upquery repeats them.
Noria implementation

MySQL adapter

- 45k lines of Rust + 15k libraries
- RocksDB for base table storage
- ZooKeeper for leader election

```sql
1 /* base tables */
2 CREATE TABLE stories
3   (id int, author int, title text, url text);
4 CREATE TABLE votes (user int, story_id int);
5 CREATE TABLE users (id int, username text);
6 /* internal view: vote count per story */
7 CREATE INTERNAL VIEW VoteCount AS
8   SELECT story_id, COUNT(*) AS vcount
9   FROM votes GROUP BY story_id;
10 /* external view: story details */
11 CREATE VIEW StoriesWithVC AS
12   SELECT id, author, title, url, vcount
13     FROM stories
14     JOIN VoteCount ON VoteCount.story_id = stories.id
15     WHERE stories.id = ?;
```
Evaluation

1. Can Noria improve a real web application’s performance?
Case study: Lobsters (http://lobste.rs)

- Ruby-on-Rails application with MySQL backend
- Hand-optimized by developers to pre-compute aggregations
- Noria data-flow with 235 operators, 35 views
- Emulate production load
Can Noria improve Lobsters’ performance?

Noria with natural queries supports 5x MySQL’s throughput.
Noria — Summary

• New partially-stateful data-flow model.
• Noria: new web application backend based on data-flow.
• Partial state saves space and allows live change.
• Supports high throughput on one or more machines.
• Open source, try it out!
Q1

Noria ensures that clients read eventually consistent data. What problem are they trying to solve? Why is eventual consistency okay?
How does Noria ensure eventual consistency?

Consider the following example: say there are two updates u1, u2 that update two views, and a upquery uq issued by the user also updates these views.

base table
view1 (upstream) op1:
view2 (downstream) op2:
Q3

What complicates Join processing in Noria?

Consider the following example:

<table>
<thead>
<tr>
<th>Table A:</th>
<th>Table B:</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[K A1]</td>
<td>[K B1]</td>
<td>[K A2]</td>
<td>[K B2]</td>
<td></td>
</tr>
<tr>
<td>[K A1 B1]</td>
<td>[K A2 B1]</td>
<td>[K A2 B2]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[K A1 B2]</td>
<td>[K A2 B2]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Join output: