Dynamic, partially-stateful data-flow for high-performance Web applications

Jon Gjengset
Jonathan Behrens
Eddie Kohler

Malte Schwarzkopf
Lara Timbó Araújo
M. Frans Kaashoek

Martin Ek
Robert Morris
Backend

Frontend
Backend

<table>
<thead>
<tr>
<th>Stories</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Votes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Frontend
Slow reads, repeated work!

90% reads
10% writes
Precomputed results

Vote

Stories

Votes

JOIN

COUNT

FILTER

Query

READ

Frontend
Store in base table?
— manual, slow.
Store in base table?
- manual, slow.
memcached?
- complex
[Facebook NSDI’13].
Store in base table?  
— manual, slow.

memcached?  
— complex  
[Facebook NSDI’13].

Streaming data-flow?
Streaming data-flow?
Streaming data-flow?
Streaming data-flow?

- **Stories**
  - Orange
  - Green
  - Blue

- **Votes**
  - Green
  - Blue
  - Orange

- **JOIN**
- **COUNT**
- **FILTER**

**Materialized view**

**Frontend**
Streaming data-flow?

Challenges

Frontend
Challenges

State-of-the-art data-flow systems:

- Change queries? Restart!
Challenges

State-of-the-art data-flow systems:

- Change queries? **Restart!**
Challenges

State-of-the-art data-flow systems:

- Change queries? **Restart!**
- Memory footprint? **Grows!**
Noria

<table>
<thead>
<tr>
<th>Stories</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Votes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Frontend
Noria

Frontend
Change queries? Live.
Noria

➤ Change queries? **Live.**
Noria

- Change queries? **Live.**
- Memory footprint? **Bounded.**
Noria

- Change queries? **Live.**
- Memory footprint? **Bounded.**
Noria

- Change queries? **Live.**
- Memory footprint? **Bounded.**
- No global coordination.
New model:
Partially-stateful data-flow
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 (⊥).

Frontend
Partially-stateful data-flow

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

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

Lower memory footprint.
Partially-stateful data-flow

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

Lower memory footprint.
No need to update absent entries.
Partially-stateful data-flow

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

Lower memory footprint.
No need to update absent entries.
Enables live data-flow changes.
Partially-stateful data-flow: upqueries
Partially-stateful data-flow: upqueries

JOIN

FILTER

COUNT

Stories

Votes

??? Need to fill absent entry!

Frontend
**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.

Frontend
Partial state enables live data-flow changes

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

Frontend
High performance requires concurrency

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

Without global coordination!

Frontend
High performance requires concurrency

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

Without global coordination!
High performance requires concurrency

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

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

1. Concurrent upqueries and forward processing — races!

   Must maintain **correctness** under concurrency!
Challenges implementing partially-stateful data-flow

1. Concurrent upqueries and forward 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 **
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.
Upquery responses in total order with updates

**Goal:** upquery restores state as if present all along.
Upquery responses in total order with updates

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

resulting state 3
respects total order  ✔️
Upquery responses in total order with updates

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

resulting state **3** respects total order

resulting state **2** violates total order
Upquery responses in total order with updates

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

More complex cases: merged upquery responses, evictions *(Paper)*.
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.
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  (see Paper)
   Drop updates that touch absent state, future upquery repeats them.
Noria implementation
Noria implementation

```sql
/* base tables */
CREATE TABLE stories
(id int, author int, title text, url text);
CREATE TABLE votes (user int, story_id int);
CREATE TABLE users (id int, username text);

/* internal view: vote count per story */
CREATE INTERNAL VIEW VoteCount AS
SELECT story_id, COUNT(*) AS vcount
FROM votes GROUP BY story_id;

/* external view: story details */
CREATE VIEW StoriesWithVC AS
SELECT id, author, title, url, vcount
FROM stories
JOIN VoteCount ON VoteCount.story_id = stories.id
WHERE stories.id = ?;
```
Noria implementation

```sql
/* base tables */
CREATE TABLE stories
(id int, author int, title text, url text);
CREATE TABLE votes (user int, story_id int);
CREATE TABLE users (id int, username text);
/* internal view: vote count per story */
CREATE INTERNAL VIEW VoteCount AS
SELECT story_id, COUNT(*) AS vcount
FROM votes GROUP BY story_id;
/* external view: story details */
CREATE VIEW StoriesWithVC AS
SELECT id, author, title, url, vcount
FROM stories
JOIN VoteCount ON VoteCount.story_id = stories.id
WHERE stories.id = ?;
```
Noria implementation

```sql
/* base tables */
CREATE TABLE stories
(id int, author int, title text, url text);
CREATE TABLE votes (user int, story_id int);
CREATE TABLE users (id int, username text);

CREATE VIEW VoteCount AS
SELECT story_id, COUNT(*) AS vcount
FROM votes GROUP BY story_id;

CREATE VIEW StoriesWithVC AS
SELECT id, author, title, url, vcount
FROM stories
JOIN VoteCount ON VoteCount.story_id = stories.id
WHERE stories.id = ?;
```
Noria implementation

```sql
/* base tables */
CREATE TABLE stories
(id int, author int, title text, url text);
CREATE TABLE votes (user int, story_id int);
CREATE TABLE users (id int, username text);

CREATE INTERNAL VIEW VoteCount AS
SELECT story_id, COUNT(*) AS vcount
FROM votes GROUP BY story_id;

CREATE VIEW StoriesWithVC AS
SELECT id, author, title, url, vcount
FROM stories
JOIN VoteCount ON VoteCount.story_id = stories.id
WHERE stories.id = ?;
```

MySQL adapter

Data-flow graph
Noria implementation

```
/* base tables */
CREATE TABLE stories
(id int, author int, title text, url text);
CREATE TABLE votes (user int, story_id int);
CREATE TABLE users (id int, username text);
/* internal view: vote count per story */
CREATE VIEW VoteCount AS
SELECT story_id, COUNT(*) AS vcount
FROM votes GROUP BY story_id;
/* external view: story details */
CREATE VIEW StoriesWithVC AS
SELECT id, author, title, url, vcount
FROM stories
JOIN VoteCount ON VoteCount.story_id = stories.id
WHERE stories.id = ?;
```

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

1. Can Noria improve a real web application’s performance?

2. How does Noria compare to alternative approaches?

3. Can Noria change queries without downtime?
1. Can Noria improve a real web application’s performance?

2. How does Noria compare to alternative approaches?

3. Can Noria change queries without downtime?

**Setup**  
Amazon EC2 c5.4xlarge instance (16 vCPUs)  
Open-loop clients, measuring latency & throughput
Evaluation

1. Can Noria improve a real web application’s performance?
2. How does Noria compare to alternative approaches?
3. Can Noria change queries without downtime?

**Setup**  Amazon EC2 c5.4xlarge instance (16 vCPUs)
Open-loop clients, measuring latency & throughput

multi-machine experiments comparison with differential dataflow  see Paper
Case study: Lobsters (http://lobste.rs)
Case study: Lobsters (http://lobste.rs)

- Ruby-on-Rails application with MySQL backend
Case study: Lobsters (http://lobste.rs)

- Ruby-on-Rails application with MySQL backend
- Hand-optimized by developers to pre-compute aggregations
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
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?

95th percentile latency [ms]

Offered load [page views/sec]
Can Noria improve Lobsters’ performance?

Offered load [page views/sec] 95th percentile latency [ms]

Better
Can Noria improve Lobsters’ performance?

![Graph showing the 95th percentile latency improvement for MySQL baseline queries with increasing offered load in page views per second. The graph indicates a better performance with Noria as the load increases.]
Can Noria improve Lobsters’ performance?

- MySQL, baseline queries
- Noria, natural queries

95th percentile latency [ms]

Better

Offered load [page views/sec]
Can Noria improve Lobsters’ performance?

Noria supports 5x MySQL’s throughput.
How does Noria compare to alternatives?
How does Noria compare to alternatives?

- Zipf-distributed story ID, 95% reads, 5% writes
- No TX, all in-memory
How does Noria compare to alternatives?

- Zipf-distributed story ID, 95% reads, 5% writes
- No TX, all in-memory
How does Noria compare to alternatives?

- Zipf-distributed story ID,
  95% reads, 5% writes
- No TX, all in-memory
How does Noria compare to alternatives?

- Zipf-distributed story ID, 95% reads, 5% writes
- No TX, all in-memory
How does Noria compare to alternatives?

- Zipf-distributed story ID, 95% reads, 5% writes
- No TX, all in-memory

Noria outperforms an in-memory key-value store and simplifies its interface.
Can Noria change queries without downtime?
Can Noria change queries without downtime?
Can Noria change queries without downtime?
Can Noria change queries without downtime?

Throughput [writes/sec]

-15 0 30 60 90

Time after transition start [sec]

new table & query added
Can Noria change queries without downtime?

![Graph showing throughput over time](image)

- New table & query added
Can Noria change queries without downtime?

- Zipf-distributed story ID, 95% reads;
- 2M existing votes at transition

Graph:
- Throughput [writes/sec] 0, 100K, 200K, 300K
- Time after transition start [sec] -15, 0, 30, 60, 90
- New table & query added
- Total write throughput
Can Noria change queries without downtime?

- Zipf-distributed story ID, 95% reads;
- 2M existing votes at transition
- Old view reads are live throughout
Can Noria change queries without downtime?

- Zipf-distributed story ID, 95% reads;
- 2M existing votes at transition
- Old view reads are live throughout
Can Noria change queries without downtime?

- Instantaneous transition, no downtime for writes
- 80% of reads from new view proceed without upquery after 1 second

- Zipf-distributed story ID, 95% reads;
- 2M existing votes at transition
- Old view reads are live throughout
Can Noria change queries without downtime?

Noria achieves downtime-free query change with partial state.

- Zipf-distributed story ID, 95% reads; 2M existing votes at transition
- Old view reads are live throughout
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!
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!

https://pdos.csail.mit.edu/noria

(see our demo at poster #37 today!)