NORIA DYNAMIC DATA FLOW FOR WEBAPPS

High performance through partial statefulness

By J. Gjengset, M. Schwarzkopf, J. Behrens, L. T. Araujo, M. Ek, E. Kohler, M. F. Kaashoek and R. Morris

THE PROBLEM

- Web services: high throughput, low latency
- Only eventual consistency needed, see CAP theorem
- Changes may need to be rolled out quickly and painlessly
- Many more reads than writes
- Relational structure of data
- ! Relational database is too slow
- ! Need to add some form of caching

CAP Theorem [1998]:
A distributed data store cannot achieve all three of the following:

1. Consistency
2. Availability
3. Partition Tolerance

SOLUTION NO. 1: REDIS

- Key-value store. Examples: Redis, Memcached.
- Good for rapid access to precomputed queries
- But there is a problem on write operations: we must invalidate or replace cache
- ! Cache invalidation is hard to get done right (“thundering herds”)
- ! Cache invalidation might be slow
- Eventual consistency

SOLUTION NO. 2: STREAM PROCESSING

- Examples: Twitter Heron, internal Facebook tooling
- Keeps results for queries that use lots of old data
- But use only recent records
- ! Need to keep a complete state for some operators
- ! Typically need a restart on change to query structure

PROBLEMS:

- ! Invalidation is hard to get done right (“thundering herds”)
- ! Invalidation might be slow
- ! Need to keep state for some operators
- ! Typically need a restart on change to query structure

SOLUTIONS:

- ! Invalidation is hard to get done right (“thundering herds”): **outsource to Noria**
- ! Invalidation might be slow: **make it asynchronous**
- ! Need to keep state for some operators: **partial state - evict unused data, compute on demand**
- ! Typically need a restart on change to query structure: **clever transitions**

NORIA

- Stateful
- Dynamic
- Parallel
- Distributed

CONTRIBUTIONS:

- Data flow model with *partial statefulness* and *upquery* capability
- A module for transitioning a dataflow graph on schema change
- Implementation in Rust and performance assessment

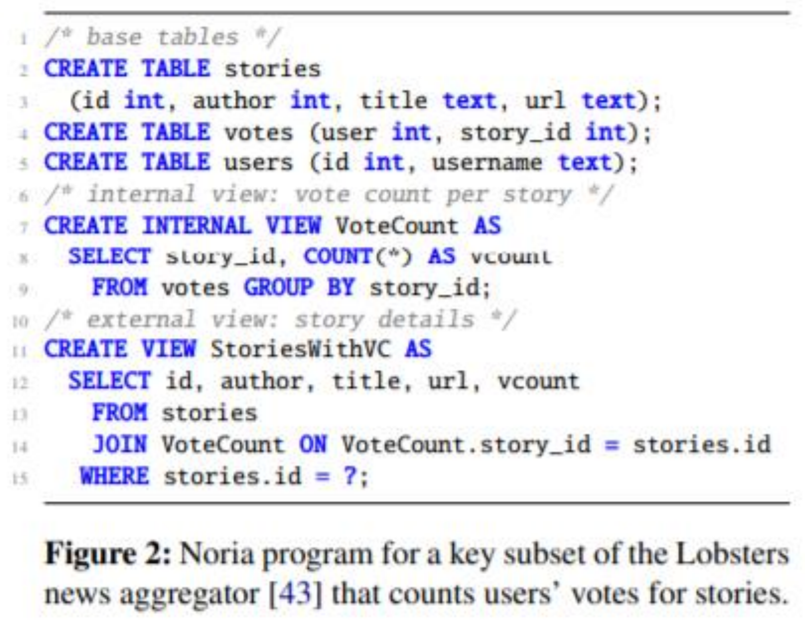
CONTRIBUTIONS (1):

- Data flow model with *partial statefulness* and *upquery* capability
- A module for transitioning a dataflow graph on schema change
- Implementation in Rust and performance assessment

BUILDING BLOCKS OF NORIA

- Base tables: store raw data; no redundancy
- Internal views: helpers for external views
- External views: a predefined set of "standard" queries

- A *Noria Program* is a schema of base tables and views
- Similar to the database schema, supports SQL
- But can be changed on-the-run



```
/* 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 = ?;
```

Figure 2: Noria program for a key subset of the Lobsters news aggregator [43] that counts users' votes for stories.

Source: Noria paper

CONCEPT: UPQUERY

- Noria builds a data flow graph
- Stateful and stateless vertices
- Parts of states are evicted in:
  - Writes
  - Running out of space – least used ones
- Eviction of children states runs in the background
- If query depends on unknown data, upquery is performed

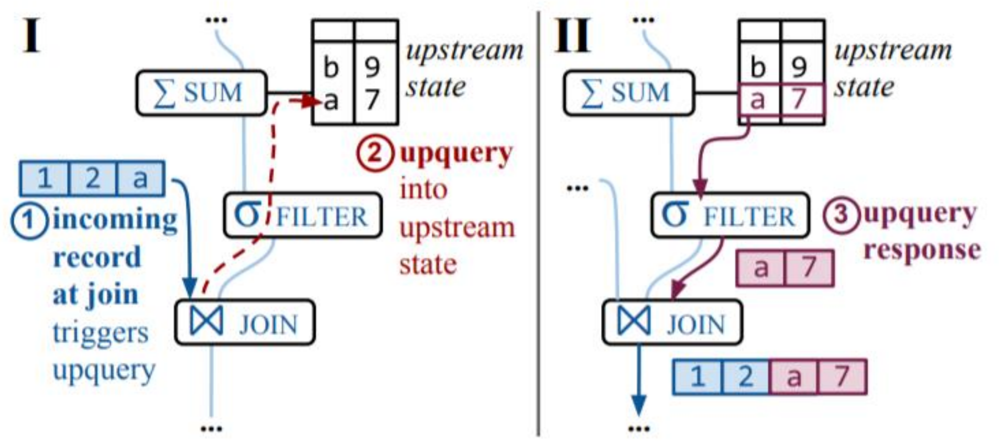
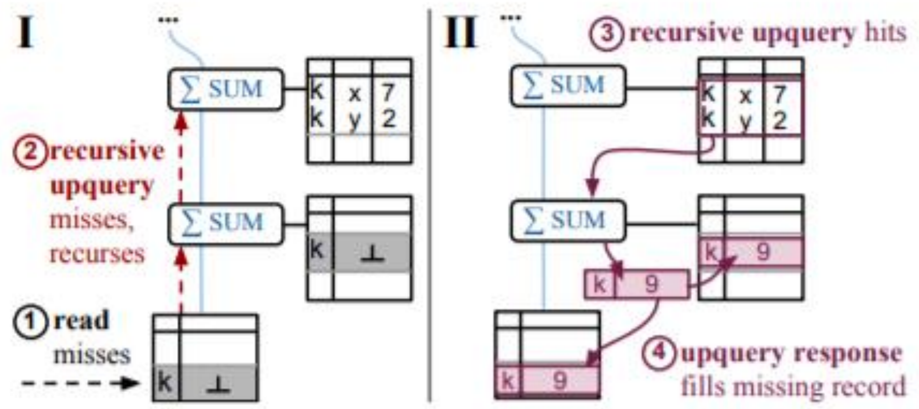


Figure 3: Noria's data-flow operators can query into upstream state: a join issues an upquery (**I**) to retrieve a record from upstream state to produce a join result (**II**).

Source: Noria paper

WHAT HAPPENS ON EVICTION?



I
II
SUM
1 read misses
2 recursive upquery misses, recurses
3 recursive upquery hits
4 upquery response fills missing record

Figure 4: A partially-stateful view sends a *recursive upquery* to derive evicted state (⊥) for key *k* from upstream state (**I**); the response fills the missing state (**II**).

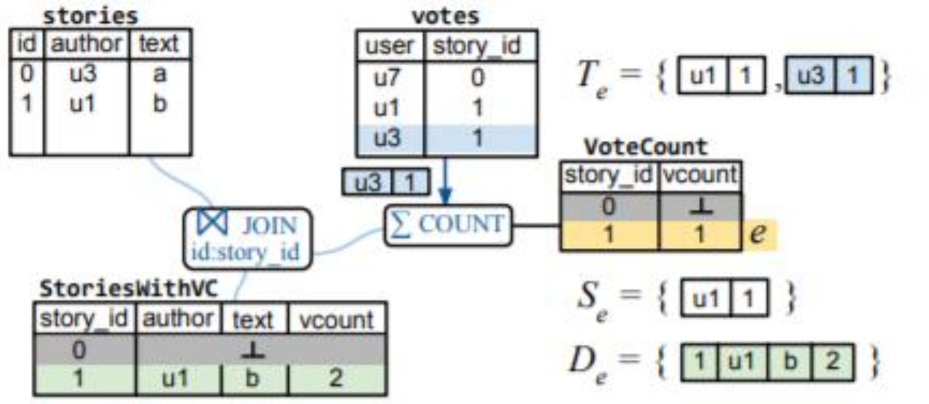


Figure 5: Definitions for partial state entry *e* (yellow) in `VoteCount`: an in-flight update from `votes` (blue) is in T_e, but not yet in S_e; the entry in `StoriesWithVC` is key-descendant from *e* via `story_id` (green).

Source: Noria paper

REQUIREMENTS FOR EVENTUAL CONSISTENCY

- Operators are deterministic wrt. own state and ancestor inputs
- No race conditions between writes and upqueries
- No reordering on the same path
- No race conditions between separate updates arriving on a single operator

CONTRIBUTIONS (2):

- Data flow model with *partial statefulness* and *upquery* capability
- A module for transitioning a dataflow graph on schema change
- Implementation in Rust and performance assessment

WHAT IF OUR SCHEMA CHANGES?

- A transition from old to new state of computational graph necessary
- Schema change is planned and careful
- Aimed at reusing operators and states
- When schema change concludes, old model is purged.

ADAPTABILITY

- Supports most (but not all) SQL operators
- A standalone tool able to replace a relational database and cache.
- Plug-in design, easy to adapt into an existing application.

CONTRIBUTIONS (3):

- Data flow model with *partial statefulness* and *upquery* capability
- A module for transitioning a dataflow graph on schema change
- Implementation in Rust and performance assessment

LATENCY V. MARIADB

- Tested on Lobsters news aggregator
- 2-10x faster than relational databases
- More write-bound than relational DBs
- Requires ample memory
- Much lower average latency

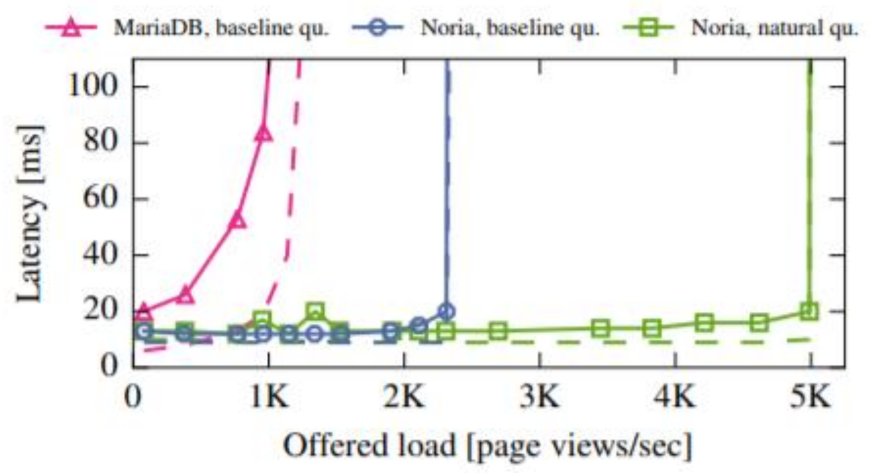
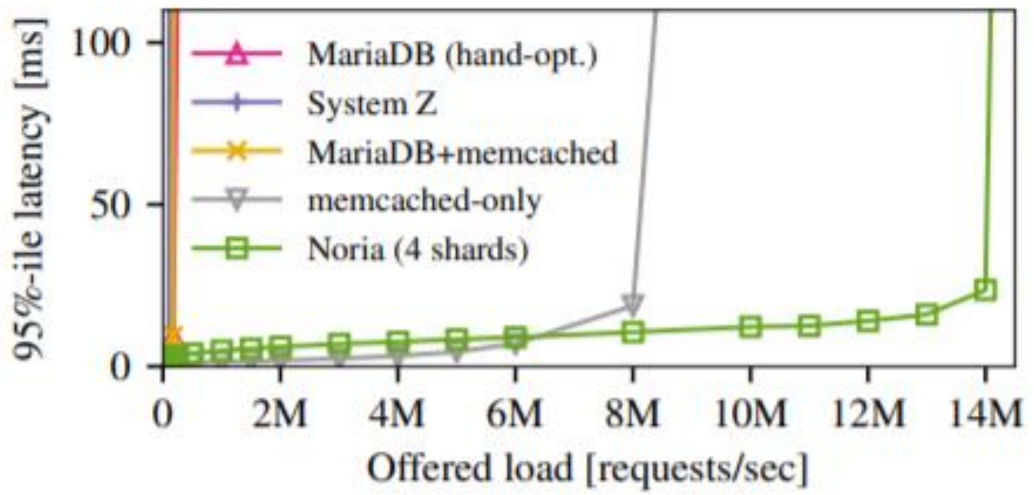


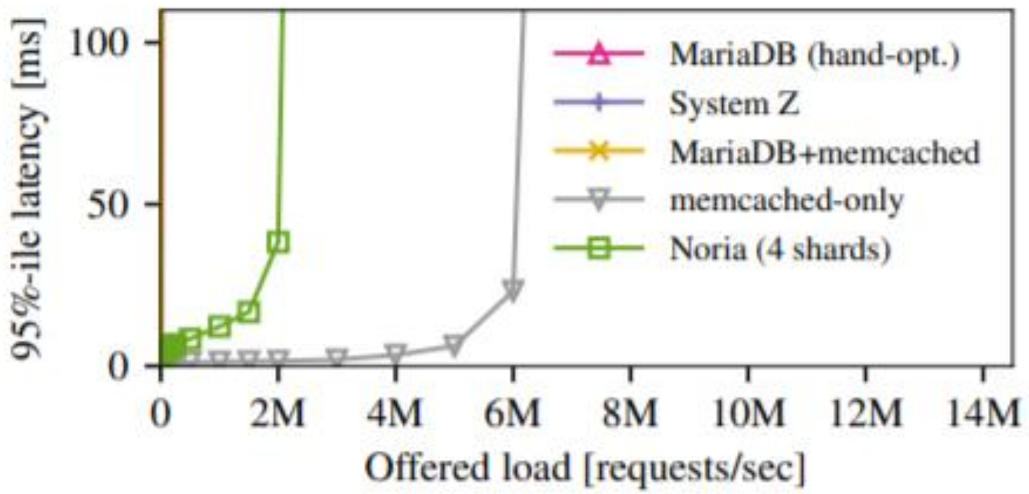
Figure 6: Noria scales Lobsters to a 5× higher load than MariaDB (2.3× with baseline queries) at sub-100ms 95%ile latency (dashed: median). MariaDB is limited by read computation, while Noria becomes write-bound.

Source: Noria paper

LATENCY WRT. READ/WRITE BALANCE



(a) Read-heavy workload (95%/5%): Noria outperforms all other systems (all but memcached at 100–200k requests/sec).



(b) Mixed read-write workload (50%/50%): Noria outperforms all systems but memcached (others are at 20k requests/sec).

Source: Noria paper

SCALABILITY

- Compared vs. Differential Dataflow, based on Naiad
- State size diminished due to partial state to ~20-40%
- Communication between nodes uses RPC
- Techniques: sharding, multicore parallelism

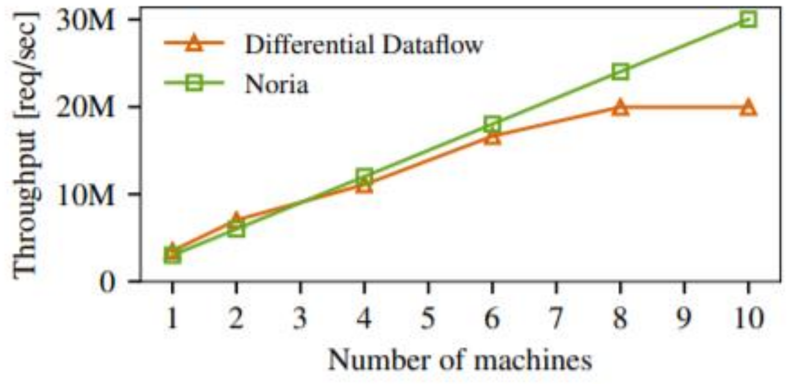


Figure 9: For a uniform 95%/5% workload, Noria scales to ten machines with sub-100ms 95th %tile latency by sharding the data-flow. Differential dataflow [44] scales less well due to its inter-worker coordination.

Source: Noria paper

ISSUES

- Only eventual consistency
- Randomized eviction from partial state – especially in low-memory settings
- Limited SQL support
- Slow response to some queries, especially writes

IS IT READY?

“Noria is most definitely still a research prototype, though I think the thing standing between where it is now and a production-ready version is mostly just engineering effort. We are a small team of researchers working on it, and we focus our efforts on the aspects of the system that are related to our ongoing research. There is relatively little room for spending lots of time on doing “production engineering” in the academic setting :)”

Source: interview with Jon Gjengset, https://notamonadtutorial.com/interview-with-norias-creator-a-promising-dataflow-database-implemented-in-rust-352e2c3d9d95

OUTCOMES

- Noria is a good and promising dataflow design
- Its adaptability to changes in data flow schema is novel
- It is open source
- It is easy to integrate into existing software
- But not yet production-ready