Noria: dynamic, partially-stateful data-flow for high-performance web applications

Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, and Lara Timbó Araújo,
MIT CSAIL; **Martin Ek,** *Norwegian University of Science and Technology;*
Eddie Kohler, *Harvard University;* **M. Frans Kaashoek and Robert Morris,** *MIT CSAIL*

https://www.usenix.org/conference/osdi18/presentation/gjengset

This paper is included in the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18).

October 8–10, 2018 • Carlsbad, CA, USA

ISBN 978-1-939133-08-3

Open access to the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

Noria: dynamic, partially-stateful data-flow for high-performance web applications

Jon Gjengset * Malte Schwarzkopf * Jonathan Behrens Lara Timbó Araújo
Martin Ek† Eddie Kohler‡ M. Frans Kaashoek Robert Morris
MIT CSAIL † *Norwegian University of Science and Technology* ‡ *Harvard University*

Abstract

We introduce *partially-stateful data-flow*, a new streaming data-flow model that supports eviction and reconstruction of data-flow state on demand. By avoiding state explosion and supporting live changes to the data-flow graph, this model makes data-flow viable for building long-lived, low-latency applications, such as web applications. Our implementation, *Noria*, simplifies the backend infrastructure for read-heavy web applications while improving their performance.

A Noria application supplies a relational schema and a set of parameterized queries, which Noria compiles into a data-flow program that pre-computes results for reads and incrementally applies writes. Noria makes it easy to write high-performance applications without manual performance tuning or complex-to-maintain caching layers. Partial statefulness helps Noria limit its in-memory state without prior data-flow systems' restriction to windowed state, and helps Noria adapt its data-flow to schema and query changes while on-line. Unlike prior data-flow systems, Noria also shares state and computation across related queries, eliminating duplicate work.

On a real web application's queries, our prototype scales to 5× higher load than a hand-optimized MySQL baseline. Noria also outperforms a typical MySQL/memcached stack and the materialized views of a commercial database. It scales to tens of millions of reads and millions of writes per second over multiple servers, outperforming a state-of-the-art streaming data-flow system.

1 Introduction

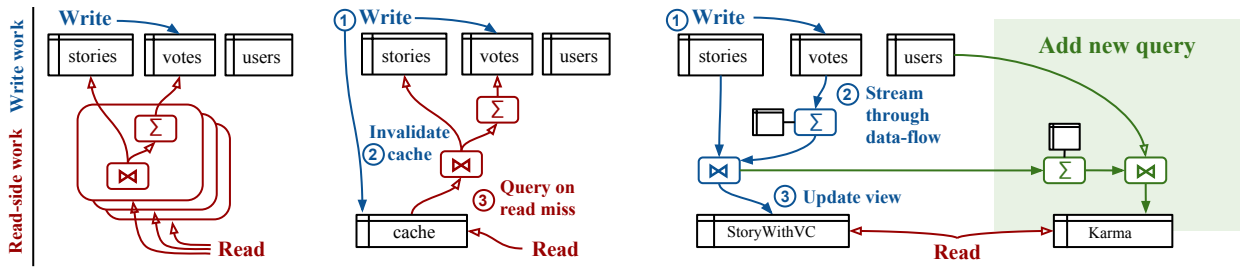
Web applications must serve many users at low latency. They respond to each user request using data queried from backend stores, usually relational databases. The vast majority of such store accesses are reads, and evaluating them as repeated queries over the normalized schema of a relational database is inefficient [54, 57]. Hence, many applications explicitly include pre-computed query results in their database schemas, or cache such results in separate key-value stores [8, 54]. For example, the Lobsters news aggregator [43] stores stories' computed vote counts and "hotness" in separate table columns to avoid re-computing them on every page load [42]. As each vote is reflected in several places, application logic must explicitly update computed columns every time a value changes. Hence, pre-computation complicates both application reads and writes. In general, developers must choose between convenient, but slow, "natural" relational queries (*e.g.*, with inline aggregations), and increased performance at the cost of application and deployment complexity (*e.g.*, due to caching).

Noria applications do not need to choose. Noria exposes a high-level query interface (SQL), but unlike in conventional systems, Noria accelerates the execution of even complex natural queries by answering with pre-computed results where possible. At its core, Noria runs a continuous, but dynamically changing, *data-flow computation* that combines the persistent store, the cache, and elements of application logic. Each write to Noria streams through a joint data-flow graph for the current queries and incrementally updates the cached, eventually-consistent internal state and query results.

Making this approach work for web applications is challenging. A naïve implementation might maintain unbounded pre-computed state, causing unacceptable space and time overhead, so Noria must limit its state size. Writes can update many pre-computed results, so Noria must ensure that writes are fast and avoid unnecessary work. Finally, since many web applications frequently change their queries [20, 61], Noria must accommodate changes without iterating over all data.

Existing data-flow systems either cannot perform fine-grained incremental updates to state [36, 52, 75], or limit the growth of operator state using "windowed" state (*e.g.*, this week's stories). This bounds their memory footprint but prohibits reading older data [11, 39, 46, 51]. Noria's data-flow operator state is *partial* instead of windowed, retaining only the subset of records that the application has queried. This is possible thanks to a new, *partially-stateful* data-flow model: when in need of missing state, operators request an *upquery* that derives the missing records from upstream state. Ensuring correctness with this model requires careful attention to invariants, as ordinary updates and upqueries can race. With-

* equal contribution



(a) Classic database operation with compute on reads.

(b) Two-tier stack with demand-filled cache [54, §2].

(c) Noria: stateful data-flow operators pre-compute data for reads incrementally; data-flow change supports new queries.

Figure 1: Overview of how current website backends and Noria process frontend reads and writes.

out care, such races could produce permanently incorrect state, and therefore incorrect cached query results.

The state that Noria keeps is similar to a materialized view, and its data-flow processing is akin to view maintenance [2, 37]. Noria demonstrates that, contrary to conventional wisdom, maintaining materialized views for all application queries is feasible. This is possible because partially-stateful operators can evict rarely-used state, and discard writes for that state, which reduces state size and write load. Noria further avoids redundant computation and state by jointly optimizing its queries to merge overlapping data-flow subgraphs.

Few existing streaming data-flow systems can change their queries and input schemas without downtime. For example, Naiad must re-start to accommodate changes, and Spark's Structured Streaming must restart from a checkpoint [18]. Noria, by contrast, adapts its data-flow to new queries without interrupting existing clients. It applies changes while retaining existing state and while remaining live for reads throughout. Writes from current clients see sub-second interruptions in the common case.

Noria's techniques remain compatible with traditional parallel and distributed data-flow, and allow Noria to parallelize and scale fine-grained, partially materialized view maintenance over multiple cores and machines.

In summary, Noria makes four principal contributions:

1. the partially-stateful data-flow model, its correctness invariants, and a conforming system design;
2. automatic merge-and-reuse techniques for data-flow subgraphs in joint data-flows over many queries, which reduce processing cost and state size;
3. near-instantaneous, dynamic transitions for data-flow graphs in response to changes to queries or schema without loss of existing state; and
4. a prototype implementation and an evaluation that demonstrates that practical web applications benefit from Noria's approach.

Our Noria prototype exposes a backwards-compatible MySQL protocol interface and can serve real web applications with minimal changes, although its benefits increase for Noria-optimized applications. When serving the Lobsters web application on a single Amazon EC2 VM, our prototype outperforms the default MySQL-based backend by 5× while simultaneously simplifying the application (§8.1). For a representative query, our prototype outperforms the widely-used MySQL/memcached stack and the materialized views of a commercial database by 2–10× (§8.2). It also scales the query to millions of writes and tens of millions of reads per second on a cluster of EC2 VMs, outperforming a state-of-the-art data-flow system, differential dataflow [46, 51] (§8.3). Finally, our prototype adapts the data-flow without any perceptible downtime for reads or writes when transitioning the same query to a modified version (§8.5).

Nevertheless, our current prototype has some limitations. It only guarantees eventual consistency; its eviction from partial state is randomized; it is inefficient for sharded queries that require shuffles in the data-flow; and it lacks support for some SQL keywords. We plan to address these limitations in future work.

2 Background

We now explain how current website backends and Noria process data. Figure 1 shows an overview.

Many web applications use a **relational database** to store and query data (Figure 1a). Page views generate database queries that frequently require complex computation, and the query load tends to be read-heavy. Across one month of traffic data from a HotCRP site and the production deployment of Lobsters [32], 88% to 97% of queries are reads (`SELECT` queries), and these reads consume 88% of total query execution time in HotCRP. Since read performance is important, application developers often manually optimize it. For example, Lobsters stores individual votes for stories in a `votes` table, but also stores per-story vote counts as a column in the `stories` table. This speeds up read queries of vote counts, but "de-normalizes" the schema and complicates vote writes, which must update the derived counts.

Websites often deploy an **in-memory key-value cache** (like Redis, memcached, or TAO [8]) to speed up common-case read queries (Figure 1b). Such a cache avoids re-evaluating the query when the underlying records are unchanged. However, the application must invalidate or replace cache entries as the records change. This process is error-prone and requires complex application-side logic [37, 48, 57, 64]. For example, developers must carefully avoid performance collapse due to "thundering herds" (*viz.*, many database queries issued just after an invalidation) [54, 57]. Since the cache can return stale records, reads are eventually-consistent.

Some sites use **stream-processing systems** [13, 39] to maintain results for queries whose re-execution over all past data is infeasible. One major problem for these systems is that they must maintain *state* at some operators, such as aggregations. To avoid unbounded growth, existing systems "window" this state by limiting it to the most recent records. This makes it difficult for a stream processor to serve the general queries needed for websites, which need to access older as well as recent state. Moreover, stream processors are less flexible than a database that can execute any relational query on its schema: introducing a new query often requires a restart.

Noria, as shown in Figure 1c, combines the best of these worlds. It supports the fast reads of key-value caches, the efficient updates and parallelism of streaming data-flow, and, like a classic database, supports changing queries and base table schemas without downtime.

3 Noria design

Noria is a stateful, dynamic, parallel, and distributed data-flow system designed for the storage, query processing, and caching needs of typical web applications.

3.1 Target applications and deployment

Noria targets read-heavy applications that tolerate eventual consistency. Many web applications fit this model: they accept the eventual consistency imposed by caches that make common-case reads fast [15, 19, 54, 72]. Noria's current design primarily targets relational operators, rather than the iterative or graph computations that are the focus of other data-flow systems [46, 51], and processes structured records in tabular form [12, 16]. Large blobs (*e.g.*, videos, PDF files) are best stored in external blob stores [7, 24, 50] and referenced by Noria's records.

Noria runs on one or more multicore servers that communicate with clients and with one another using RPCs. A Noria deployment stores both *base tables* and *derived views*. Roughly, base tables contain the data typically stored persistently, and derived views hold data an application might choose to cache. Compared to conventional database use, Noria base tables might be smaller, as Noria derives data that an application may otherwise precompute and store in base tables for performance. Views, by contrast, will likely be larger than a typical cache footprint, because Noria derives more data, including some intermediate results. Noria stores base tables persistently on disk, either on one server or sharded across multiple servers, but stores views in server memory. The application's working set in these views should fit in memory for good performance, but Noria reduces memory use by only materializing records that are actually read, and by evicting infrequently-accessed data.

```
/* 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.

3.2 Programming interface

Applications interact with Noria via an interface that resembles parameterized SQL queries. The application supplies a *Noria program*, which registers base tables and views with parameters supplied by the application when it retrieves data. Figure 2 shows an example Noria program for a Lobsters-like news aggregator application (? is a parameter). The Noria program includes base table definitions, *internal* views used as shorthands in other expressions, and *external* views that the application later queries. Internally, Noria instantiates a data-flow to continuously process the application's writes through this program, which in turn maintains the external views.

To retrieve data, the application supplies Noria with an external view identifier (*e.g.*, `StoriesWithVC`) and one or more sets of parameter values. Noria then responds with the records in the view that match those values. To modify records in base tables, the application performs insertions, updates, and deletions, similar to a SQL database. Noria applies these changes to the appropriate base tables and updates dependent views.

The application may change its Noria program to add new views, to modify or remove existing views, and to adapt base table schemas. Noria expects such changes to be common and aims to complete them quickly. This contrasts with most previous data-flow systems, which lack support for efficient changes without downtime.

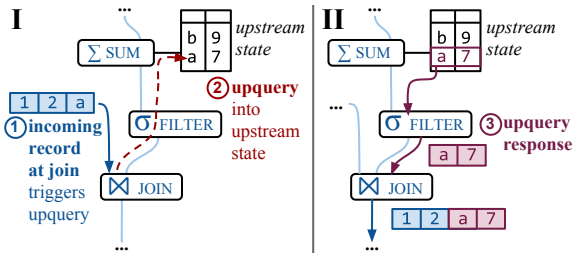


Figure 3: Noria's data-flow operators can query into upstream state: a join issues an upquery (**I**) to retrieve a record from upstream state to produce a join result (**II**).

In addition to its native SQL-based query interface, Noria provides an implementation of the MySQL binary protocol, which allows existing applications that use prepared statements against a MySQL database to interact with Noria without further changes. The adapter turns ad-hoc queries and prepared SQL statements into writes to base tables, reads from external views, and incrementally effects Noria program changes. Noria supports much, but not all, SQL syntax. We discuss the experience of building and porting applications in §7.

3.3 Data-flow execution

Noria's data-flow is a directed acyclic graph of relational operators such as aggregations, joins, and filters. Base tables are the roots of this graph, and external views form the leaves. Noria extends the graph with new base tables, operators, and views as the application adds new queries.

When an application write arrives, Noria applies it to a durable base table and injects it into the data-flow as an *update*. Operators process the update and emit derived updates to their children; eventually updates reach and modify the external views. Updates are *deltas* [46, 60] that can add to, modify, and remove from downstream state. For example, a count operator emits deltas that indicate how the count for a key has changed; a join may emit an update that installs new rows in downstream state; and a deletion from a base table generates a "negative" update that revokes derived records. Negative updates remove entries when Noria applies them to state, and retain their negative "sign" when combined with other records (*e.g.*, through joins). Negative updates hold exactly the same values as the positives they revoke and thus follow the same data-flow paths.

Noria supports *stateless* and *stateful* operators. Stateless operators, such as filters and projections, need no context to process updates; stateful operators, such as count, min/max, and top-*k*, maintain state to avoid inefficient re-computation of aggregate values from scratch. Stateful operators, like external views, keep one or more indexes to speed up operation. Noria adds indexes based on *indexing obligations* imposed by operator semantics; for example, an operator that aggregates votes by user ID requires a user ID index to process new votes efficiently.

In most stream processors, join operators keep a windowed cache of their inputs [3, 76], allowing an update arriving at one input to join with all relevant state from the other. In Noria, joins instead perform *upqueries*, which are requests for matching records from stateful ancestors (Figure 3): when an update arrives at one join input, the join looks up the relevant state by querying its other inputs. This reduces Noria's space overhead, since joins often need not store duplicate state, but requires care in the presence of concurrent updates, an issue further discussed in §4. Upqueries also impose indexing obligations that Noria detects and satisfies.

3.4 Consistency semantics

To achieve high parallel processing performance, Noria's data-flow avoids global progress tracking or coordination. An update injected by a base table takes time to propagate through the data-flow, and the update may appear in different views at different times. Noria operators and the contents of its external views are *eventually-consistent*. Eventual consistency is attractive for performance and scalability, and is sufficient for many web applications [15, 54, 72].

Noria does ensure that if writes quiesce, all external views eventually hold results that are the same as if the queries had been executed directly against the base table data. Making this work correctly requires some care. Like most data-flow systems, Noria requires that operators are deterministic functions over their own state and the inputs from their ancestors. In addition, Noria must avoid races between updates and upqueries; avoid reordering updates on the same data-flow path; and resolve races between related updates that arrive independently at multi-ancestor operators via different data-flow paths. Consider an `OR` that combines filters using a union operator, or a join between data-flow paths connected to the same base table: such operators' final output (and state) must be commutative over the order in which updates arrive at their inputs. The standard relational operators Noria supports have this property.

Web applications sometimes rely on database transactions, *e.g.*, to atomically update pre-computed values. Noria approach's is compatible with basic, optimistically-concurrent multi-statement transactions, but Noria also often obviates the need for them. For example, Lobsters uses transactions only to avoid write-write conflicts on vote counts and stories' "hotness" scores. A multi-statement transaction is required only because baseline Lobsters pre-computes hotness for performance. Noria instead computes hotness in the data-flow, which avoids write-write conflicts without a transaction, albeit at the cost of eventual consistency for reads. We

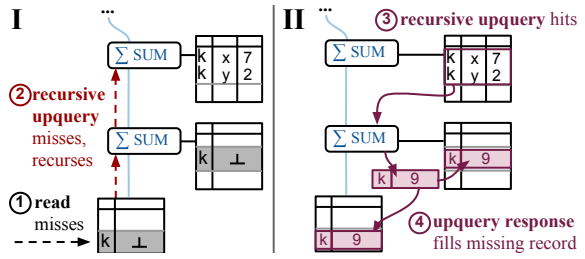


Figure 4: A partially-stateful view sends a *recursive upquery* to derive evicted state (⊥) for key *k* from upstream state (**I**); the response fills the missing state (**II**).

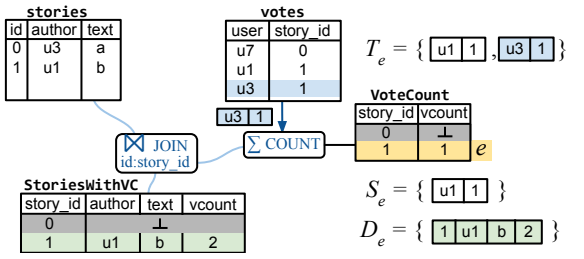


Figure 5: Definitions for partial state entry *e* (yellow) in `VoteCount`: an in-flight update from `votes` (blue) is in T_e, but not yet in S_e; the entry in `StoriesWithVC` is key-descendant from *e* via `story_id` (green).

omit further discussion of transactions with Noria in this paper; we plan to describe them in future work.

3.5 Challenges

An efficient Noria design faces two key challenges: first, it must limit the size of its state and views (§4); and second, changes to the Noria program must adapt the data-flow without downtime in serving clients (§5).

4 Partially-stateful data-flow

Noria must limit the size of its views, as the state for an application with many queries could exceed available memory and become too expensive to maintain.

The *partially-stateful data-flow model* lets operators maintain only a subset of their state. This concept of partial materialization is well-known for materialized views in databases [79, 80], but novel to data-flow systems. Partial state reduces memory use, allows eviction of rarely-used state, and relieves operators from maintaining state that is never read. Partially-stateful data-flow generalizes beyond Noria, but we highlight specific design choices that help Noria achieve its goals.

Partial state introduces new data-flow messages to Noria. *Eviction notices* flow forward along the update data-flow path; they indicate that some state entries will no longer be updated. Operators drop updates that would affect these evicted state entries without further processing or forwarding. When Noria needs to read from evicted state—for instance, when the application reads state evicted from an external view—Noria re-computes that state. This process sends *recursive upqueries* to the relevant ancestors in the graph (Figure 4). An ancestor that handles such an upquery computes the desired value (possibly after sending its own upqueries), then forwards a response that follows the data-flow path to the querying operator. When the upquery response eventually arrives, Noria uses it to populate the evicted entry. After the evicted entry has been filled, subsequent updates through the data-flow keep it up-to-date until it is evicted again.

For correctness, upqueries must produce eventually-consistent results. For performance, Noria should continue to process updates—including updates to the waiting operator—while (possibly slow) upqueries are in flight. These requirements complicate the design.

4.1 Data-flow model and invariants

We first describe high-level correctness invariants of Noria's partially-stateful data-flow. These invariants ensure that Noria remains eventually-consistent and never returns results contaminated by duplicate, missing, or spurious updates. Since Noria allows operators to execute in parallel to take advantage of multicore processors, these invariants must hold in the presence of concurrent updates and eviction notices. The invariants concern *state entries*, where a state entry models one record in one operator or view. Data-flow implementations derive state entry values from input records, possibly after multiple steps. For ease of expression, we model a state entry as the multiset of input records that produced that entry's value. Noria's eventual consistency requires that each state entry's contents approach the ideal set of input records that would produce the most up-to-date value.

Given some state entry *e*, we define:

- T_e is the set of all input records received so far that, in a correct implementation of the data-flow graph, would be used to compute *e*.
- S_e is either the multiset of input records *actually* used to compute in *e*, or ⊥, which represents an evicted entry. We use a multiset so the model can represent potential bugs such as duplicate updates.
- D_e is the set of *key-descendant entries* of *e*. These are entries of operators downstream of *e* in the data-flow that depend on *e* through key lookup.

T_e and S_e are time-dependent, whereas the dependencies represented in D_e can be determined from the data-flow graph. If *e* is the `VoteCount` entry for some story in Figure 5, then T_e contains all input votes ever received for that story; S_e contains the updates represented in its `vcount`; and D_e includes its `StoriesWithVC` entry.

Correctness of partially-stateful data-flow relies on ensuring these invariants:

1. **Update completeness:** if $S_e \neq \bot$, then either all updates in $T_e - S_e$ are in flight toward e, or an eviction notice for e is in flight toward e.
2. **No spurious or duplicate updates:** $S_e \subseteq T_e$.
3. **Descendant eviction:** if $S_e = \bot$, then for all $d \in D_e$, either $S_d = \bot$, or an eviction notice for d is in flight toward d's operator.
4. **Eventual consistency:** if T_e stops growing, then eventually either $S_e = T_e$ or $S_e = \bot$.

We now explain the mechanisms that Noria uses to realize this data-flow model and maintain the invariants.

4.2 Update ordering

Noria uses update ordering to ensure eventual consistency without global data-flow coordination. Each operator totally orders all updates and upquery requests it receives for an entry; and, critically, the downstream data-flow ensures that all updates and upquery responses from that entry are processed by all consumers in that order. Thus, if the operator orders update u_1 before u_2, then every downstream consumer likewise processes updates derived from u_1 before those derived from u_2. Noria data-flows can split and merge (*e.g.*, at joins), but update ordering and operator commutativity ensure that the eventual result is correct independent of processing order.

4.3 Join upqueries

Join operators use upqueries (§3.3): when an update arrives at one input, the join upqueries its other input for the corresponding records, and combines them with the update. Join upqueries reach the next upstream stateful operator, which computes a snapshot of the requested state entry and forwards it along the data-flow to the querying join. Intermediate operators process the response as appropriate. Unlike normal updates, upquery responses follow the single path back to the querying operator without forking. Upquery responses also commute neither with each other nor with previous updates. This introduces a problem for join update processing, since *every* such update requires an upquery that produces non-commutative results, yet must produce an update that *does* commute.

Noria achieves this by ensuring that no updates are in flight between the upstream stateful operator and the join when a join upquery occurs. To do so, Noria limits the scope of each join upquery to an operator chain processed by a single thread. Noria executes updates on other operator chains in parallel with join upqueries.

This introduces a trade-off between parallelism and state duplication: join processing must stay within a single operator chain, so copies of upstream state may be required in each operator chain that contains a join.

4.4 Eviction and recursive upqueries

Evicted state introduces new challenges for Noria's data-flow. If the application requests evicted state, Noria must use recursive upqueries to fill it in. Moreover, operators now encounter evicted state when they handle updates. These factors influence the Noria design in several ways.

First and simplest, Noria operators drop updates that encounter evicted entries. This reduces the time spent processing updates downstream, but necessitates the descendant eviction invariant: operators downstream of an evicted entry never see updates for that entry, so they must evict their own dependent entries lest they remain permanently out of date.

Second, recursive upqueries now occasionally cascade up in the data-flow until they encounter the necessary state—in the worst case, up to base tables. Responses then flow forward to the querying operator. Upquery results are snapshots of operator state, and do not commute with updates. For unbranched chains, update ordering (§4.2) and the fact that updates to evicted state are dropped ensure that the requested upquery response is processed before any update for the evicted state.

Recursive upqueries of branching subgraphs, such as joins, are more complex. A join operator must emit a single correct response for each upquery it receives, even if it must make one or more recursive upqueries of its own to produce the needed state. Combining the upqueries' results directly would be incorrect: those upqueries execute independently, and updates can arrive between their responses. Joins thus issue recursive upqueries, but compute the final result exclusively with *join* upqueries once the recursive upqueries complete (multiple rounds of recursive upqueries may be required). These join upqueries execute within a single operator chain and exclude concurrent updates. Noria supports other branching operators, such as unions, which obey the same rules as joins.

Finally, a join upquery performed during update processing may encounter evicted state. In this case, Noria chooses to drop the update and evict dependent entries downstream; Noria statically analyzes the graph to compute the required eviction notices. There is a trade-off here: computing the missing entry could avoid future upqueries. Noria chooses to evict to avoid blocking the write path while filling in the missing state.

Such evictions are rare, but they can occur. For example, imagine a version of Figure 2 that adds `AuthorVotes`, which aggregates `VoteCount` by `stories.author`, and the following system state:

- `stories[id=1]` has `author=Elena`.
- `VoteCount[story_id=1]` has `vcount=8`.
- `AuthorVotes[author=Elena]` has `vcount=8`.
- `stories[id=2]` has `author=Bob`.
- `VoteCount[story_id=2]` is evicted.

Now imagine that an update changes story 2's author to Elena. When this update arrives at the join for `AuthorVotes`, that join operator upqueries for `VoteCount[story_id=2]`, which is evicted. As a result,

Noria sends an eviction notice for Elena—whose number of votes has changed—to `AuthorVotes`.

4.5 Partial and full state

Noria makes state partial whenever it can service upqueries using efficient index lookups. If Noria would have to scan the full state of an upstream operator to satisfy upqueries, Noria disables partial state for that operator. This may happen because every downstream record depends on all upstream ones—consider *e.g.*, the top 20 stories by vote count. In addition, the descendant eviction invariant implies that partial-state operators cannot have full-state descendants.

Partial-state operators in Noria start out fully evicted and are gradually and lazily populated by upqueries. As we show next, this choice has important consequences for Noria's ability to transition the data-flow efficiently.

5 Dynamic data-flow

Application queries evolve over time, so Noria's dynamic data-flow represents a continuously-changing set of SQL expressions. Existing data-flow systems run separate data-flows for each expression, initialize new operators with empty state and reflect only new writes, or require restarting from a checkpoint. Changes to the Noria program instead adapt the data-flow dynamically.

Given new or removed expressions, Noria *transitions* the data-flow to reflect the changes. Noria first plans the transition, reusing operators and state of existing expressions where possible (§5.1). It then incrementally applies these changes to the data-flow, taking care to maintain its correctness invariants (§5.2). Once both steps complete, the application can use new tables and queries.

The key challenges for transitions are to avoid unnecessary state duplication and to continue processing reads and writes throughout. Operator reuse and partial state help Noria address these challenges.

5.1 Determining data-flow changes

To initiate a transition, the application provides Noria with sets of added and removed expressions. Noria then computes required changes to the currently-running data-flow. This process resembles traditional database query planning, but produces a long-term *joint* data-flow across *all* expressions in the Noria program. This allows Noria to reuse existing operators for efficiency: if two queries include the same join, the data-flow contains it only once.

To plan a transition, Noria first translates each new expression into an extended *query graph* [21]. The query graph contains a node for each table or view in the expression, and an edge for every join or group-by clause. Noria uses query graphs to inexpensively reject many expressions from consideration [21, §3.4, 78, §3] and to quickly establish a set of *sharing candidates* for each new expression. The sharing candidates are existing expressions that likely overlap with the new expression. Next, Noria generates a verbose intermediate representation (IR), which splits the new expression into more fine-grained operators. This simplifies common subexpression detection, and allows Noria to efficiently merge the new IR with the cached IR of the sharing candidates.

For each sharing candidate, Noria reorders joins in the new IR to match the candidate when possible to maximize re-use opportunities. It then traverses the candidate's IR in topological order from the base tables. For each operator, Noria searches for a matching operator (or clique of operators) in the new IR. A match represents a reusable subexpression, and Noria splices the two IRs together at the deepest matches.

This process continues until Noria has considered all identified reuse candidates, producing a final, merged IR.

5.2 Data-flow transition

The combined final IRs of all current expressions represent the transition's *target* data-flow. Noria must add any operator in the final IR that does not already exist in the data-flow. To do so, Noria first informs existing operators of index obligations (§3.3) incurred by new operators that they must construct indexes for. Noria then walks the target data-flow in topological order and inserts each new operator into the running data-flow and bootstraps its state. Finally, after installing new operators and deleting removed queries' external views, Noria removes obsolete operators and state from the data-flow.

Bootstrapping operator state. When Noria adds a new stateful operator, it must ensure that the operator starts with the correct state. Partially-stateful operators and views start processing immediately. They are initially empty and bootstrap via upqueries in response to application reads during normal operation, amortizing the bootstrapping work over time. Fully-stateful operators are initially marked as "inactive", which causes them to ignore all incoming updates. Noria then executes a special, large upquery for *all* keys on behalf of the fully-stateful operator. Once the last upquery response has arrived, Noria activates the operator for update processing and moves on to the next new operator.

Base table changes. As applications evolve, developers often add or remove base table columns [17]. This affects *existing* operators in the data-flow: new updates from the base table may now lack values that existing operators expect. Noria could rebuild the data-flow or transform the existing base table state to effect such a change, but this would be inefficient for large base tables. Instead, Noria base tables internally track all columns that have existed in the table's schema, including those that have been deleted. When a base table processes an application write, it automatically injects default values for missing

columns (but does not store them). This permits queries for different base table schemas to coexist in the data-flow graph, and makes most base table changes cheap.

6 Implementation

Our Noria prototype implementation consists of 45k lines of Rust and can operate both on a single server and across a cluster of servers. Applications interface with Noria either through native Rust bindings, using JSON over HTTP, or through a MySQL protocol adapter.

6.1 Persistent data storage

Noria persists base tables in RocksDB [66], a high-performance key-value store based on log-structured merge (LSM) trees. Batches of application updates are synchronously flushed into RocksDB's log before Noria acknowledges them and admits them into the data-flow; a background thread asynchronously merges log entries into the LSM trees. Each base table index forms a RocksDB "column family". For base tables with non-unique indexes, Noria uses RocksDB's ordered iterators to efficiently retrieve all rows for an index key [14, 67].

Persistence reduces Noria's write throughput by about 5% over in-memory base tables. Reads are not greatly impacted when an application's working set fits in memory: only occasional upqueries access RocksDB, and these add $<$ 1ms of additional latency on a fast SSD.

6.2 Parallel processing

Noria shards the data-flow and allows concurrent reads and writes with minimal synchronization for parallelism.

Sharding. Noria processes updates in parallel on a cluster by hash-partitioning each operator on a key and assigning shards to different servers. Each machine runs a Noria *instance*, a process that contains a complete copy of the data-flow graph, but holds state only for its shards of each operator. When an operator with one hash partitioning links to an operator with a different partitioning, Noria inserts "shuffle" operators that perform inter-shard transfers over TCP connections. Upqueries across shuffle operators are expensive since they must contact all ancestor shards. This limits scalability, but allows operators below a shuffle to maintain partial state.

Multicore parallelism. Noria achieves multicore parallelism within each server in two ways: a server can handle multiple shards by running multiple Noria instances, and each instance runs multiple threads to process its shard. Each instance has two thread pools: *data-flow workers* process updates within the data-flow graph, and *read handlers* handle reads from external views.

At most one data-flow worker executes updates for each data-flow operator at a time. This arrangement yields CPU parallelism among different operators, and also allows lock-free processing within each operator. There are typically fewer data-flow workers than operators in the data-flow graph, so Noria multiplexes operator work across the worker threads. Within one instance, Noria schedules chains of operators with the same key as a unit. This reduces queueing and inter-core data movement at operator boundaries. It also allows Noria to optimize some upqueries: an upquery within a chain can simply access the ancestor's data synchronously, without worry of contamination from in-flight updates (§4.3).

Read handlers process clients' RPCs to read from external views. They must access the view with low latency and high concurrency, even while a data-flow worker is applying updates to the view. To minimize synchronization, Noria uses double-buffered hash tables for external views [27]: the data-flow worker updates one table while read handlers read the other, and an atomic pointer swap exposes new writes. This trades space and timeliness for performance: with skewed key popularity distributions, it can improve read throughput by 10$\times$ over a single-buffered hash table with bucket-level locks.

6.3 Distributed operation

A Noria controller process manages distributed instances on a cluster of servers, and informs them of changes to the data-flow graph and of shard assignments. Noria elects the controller and persists its state via ZooKeeper [34]. Clients discover the controller via ZooKeeper, and obtain long-lived read and write handles to send requests directly to instances.

Noria handles failures by rebuilding the data-flow. If the controller fails, Noria elects a new controller that restores the data-flow graph. It then streams the persistent base table data from RocksDB to rebuild fully-stateful operators and views. Partial operators are instead populated through on-demand upqueries. If individual instances fail, Noria rebuilds only the affected operators.

6.4 MySQL adapter

Our prototype includes an implementation of the MySQL binary protocol in a dedicated stateless adapter that appears as a standard MySQL server to the application. This adapter allows developers to easily run existing applications on Noria. The adapter transparently translates prepared statements and ad-hoc queries into transitions on Noria's data-flow, and applies reads and writes using Noria's API behind the scenes. Its SQL support is sufficiently complete to run some unmodified web applications (*e.g.*, JConf [74] written in Django [22]), and to run Lobsters with minimal syntax adaptation.

6.5 Limitations

Our current prototype has some limitations that we plan to address in future work; none of them are fundamental. First, it only shards by hash partitioning on a single column, and resharding requires sending updates through

a single instance, which limits scalability. Second, it re-computes data-flow state on failure; recovering from snapshots or data-flow replicas would be more efficient (*e.g.*, using selective rollback [35]). And third, it does not currently support range indices or multi-column joins.

7 Applications

This section discusses our experiences with developing Noria applications. Noria aims to simplify the development of high-performance web applications; several aspects of our implementation help it achieve that goal.

First, applications written for a MySQL database can use Noria directly via its MySQL adapter, provided they generate parameterized SQL queries (for instance, via libraries like PHP Data Objects [69] or Python's MySQL connector [55, §10.6.8]). Porting typically proceeds in three steps. First, the developer points the application at the Noria MySQL adapter instead of a MySQL server and imports existing data into Noria from database dumps. The application will immediately see performance improvements for read queries that formerly ran substantial in-line compute. Though the MySQL adapter even supports ad-hoc read queries (it transitions the data-flow as required to support each query), the most benefit will be seen for frequently-reused queries. Second, the developer creates views for computations that the MySQL application manually materialized, such as the per-story vote count in Lobsters. These views coexist with the manual materializations, and allow existing queries to continue to work as the developer updates the write path so that it no longer manually updates derived views and caches. Third, the developer incrementally rewrites their application to rely on natural views and remove manual write optimizations. These changes gradually increase application performance as the developer removes now-unnecessary complexity from the application's read and write paths.

The porting process is not burdensome. We ported a PHP web application for college room ballots—developed by one of the authors and used production for a decade—to Noria; the process took two evenings, and required changes to four queries. We also used the MySQL adapter to port the Lobsters application's queries to Noria; the result is a focus of our evaluation.

Developing native Noria applications can be even easier. We developed a simple web application to show the results of our continuous integration (CI) tests for Noria. The CI system stores its results in Noria, and the web application displays performance results and aggregate statistics. Since we developed directly for Noria, we were not tempted to cache intermediate results or apply other manual optimizations, and could use aggregations and joins in queries without fear that performance would suffer as a result (*e.g.*, due to aggregations over the long commit history). Most application updates reduced to single-table inserts, deletes, or updates.

Limitations. Though applications traditionally use parameterized queries to avoid SQL injection attacks and cache query plans, Noria parameterized queries also build materialized views. An application with many distinct parameterized queries can thus end up with more views than necessary. The developer can correct this by adding shared views. Our prototype does not yet support update and delete operations conditioned on non-primary key columns, and lacks support for parameterized range queries (*e.g.*, `age > ?`), which some applications need. Planned support for range indexes and an extended base table implementation will address these limitations.

8 Evaluation

We evaluated our Noria prototype using backend workloads generated from the production Lobsters web application, as well as using individual queries. Our experiments seek to answer the following questions:

1. What performance gains does Noria deliver for a typical database-backed web application? (§8.1)
2. How does Noria perform compared to a MySQL/memcached stack, the materialized views of a commercial database, and an idealized cache-only deployment? (§8.2)
3. Given a scalable workload, how does our prototype utilize multiple servers, and how does it compare to a state-of-the-art data-flow system? (§8.3)
4. What space overhead does Noria's data-flow state impose, and how does Noria perform with limited memory and partial state? (§8.4)
5. Can Noria data-flows adapt to new queries and input schema changes without downtime? (§8.5)

Setup. In all experiments, Noria and other storage backends run on an Amazon EC2 c5.4xlarge instance with 16 vCPUs; clients run on separate c5.4xlarge instances unless stated otherwise. Our setup is "partially open-loop": clients generate load according to a Poisson distribution of interarrival-times and have a limited number of backend requests outstanding, queueing additional requests. This ensures that clients maintain the measurement frequency even during periods of high latency [45]. Our test harness measures offered request throughput and "sojourn time" [62], which is the delay from request generation until a response returns from the backend.

8.1 Application performance: Lobsters

We first evaluate Noria's performance on a realistic web application workload to answer two questions:

1. Do Noria's fast reads help it outperform a conventional database on a real application workload, even on a hand-optimized application?

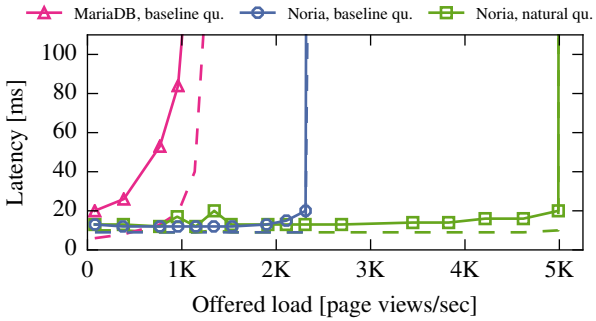


Figure 6: Noria scales Lobsters to a 5× higher load than MariaDB (2.3× with baseline queries) at sub-100ms 95%ile latency (dashed: median). MariaDB is limited by read computation, while Noria becomes write-bound.

2. Can Noria preserve good performance for an application without hand optimization?

Our workload models production Lobsters traffic. The benchmark emulates authenticated Lobsters users visiting different pages according to the access frequencies and popularity distributions in the production workload [32]. Lobsters is a Ruby-on-Rails application, but our benchmark generates database operations directly in order to eliminate Rails overhead. We seed the database with 9.2k users, 40k stories and 120k comments—the size of the real Lobsters deployment—and run increasing request loads to push the different setups to their limits.

The baseline queries include the Lobsters developers' optimizations, which manually materialize and maintain aggregate values like vote counts to reduce read-side work. We also developed "natural" queries that produce the same results using Noria data-flow to compute aggregations rather than manual optimizations. We compare MariaDB (a community-developed MySQL fork; v10.1.34) with Noria using baseline queries, and then to Noria using natural queries (both via Noria's MySQL adapter). We configured MariaDB to use a thread pool, to avoid flushing to disk after transactions, and to store the database on a ramdisk to remove overheads unrelated to query execution. With the baseline queries, the median page view executes 11 queries; this reduces to eight with natural queries. This experiment uses an m5.24xlarge EC2 instance for the CPU-intensive clients.

Figure 6 shows the results as throughput-latency curves. An ideal system would show as a horizontal line with low latency; in reality, each setup hits a "hockey stick" once it fails to keep up with the offered load. MariaDB scales to 1,000 pages/second, after which it saturates all 16 CPU cores with read-side computation (*e.g.*, for per-page notification counts [33]). Noria running the same baseline queries scales to a 2.3× higher offered load, since its incremental write-side processing avoids redundant re-computation on reads.

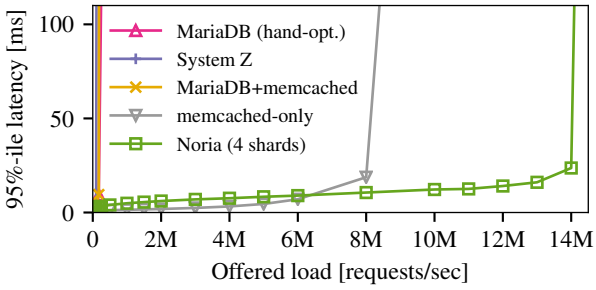
The baseline queries manually pre-compute aggregates. MariaDB requires this for performance: without the pre-computation, it supports just 20 pages/sec. Noria instead maintains pre-computed aggregates in its data-flow. This allows us to include the aggregations directly in the queries, which normalizes the base table schema, reduces write load, and avoids bugs due to missed updates to pre-computed values. With all aggregate computation moved into Noria's data-flow ("natural queries"), throughput scales higher still, to 5,000 pages/second (5× MariaDB). Eliminating application pre-computation reduces overall write load and compacts the data-flow, which lets Noria parallelize it more effectively.

The result is that Noria achieves both good performance and natural, robust queries. We observed similar benefits with other applications (*e.g.*, a synthetic TPC-W-like workload), which we omit for space.

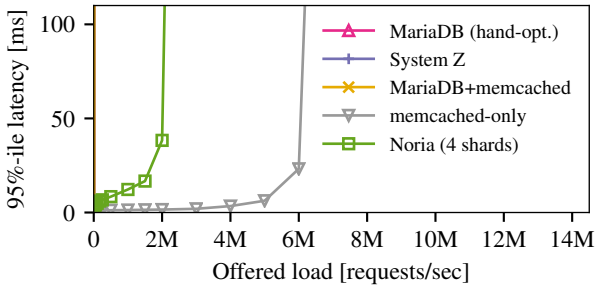
8.2 In-depth performance comparison

We compare to alternative systems using a subset of Lobsters. This restriction gives us better control over workload properties, while capturing the aspects of web workloads that motivated the Noria design. We use one kind of write, inserting a vote, and one read query, `StoriesWithVC` from Figure 2. This read query fetches stories and their vote counts; 85% of page views in production Lobsters are for pages that execute this query.

We compare five single-server deployments that all have access to the same resources, but differ in how they store and calculate the per-story vote count. **MariaDB** uses the baseline Lobsters approach of pre-computing and storing vote counts in a column of the Lobsters `stories` table. **System Z**, a commercial database with materialized view support, uses an incrementally-maintained materialized view defined similarly to `StoriesWithVC`; we use System Z to compare database view maintenance with Noria's data-flow-based approach. MariaDB and System Z run at the fastest transactional isolation level ("read uncommitted") and are configured to keep data in memory. **MariaDB+memcached** adds a demand-filled memcached (v1.5.6) cache [54] to MariaDB that caches `StoryWithVC` entries. This reduces read load on MariaDB, but complicates application code even beyond pre-computation: writes must invalidate the cache and reads must sometimes populate it. We also measure **memcached-only** without a relational backend. This setup offers good performance, but is unrealistic: it does not store individual votes or stories, is not persistent, and cannot prevent double-voting. It helps us estimate how a backend that serves all reads from memory and does minimal work for writes might perform. Finally, we measure **Noria** sharded four ways on `stories.id`, with the remaining 12 cores serving reads.



(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).

Figure 7: A Lobsters subset (Figure 2) benchmarked on Noria hand-optimized MariaDB, System Z's materialized views, a MariaDB/memcached setup, and on memcached only, all with Zipf-distributed ($s = 1.08$) reads and votes.

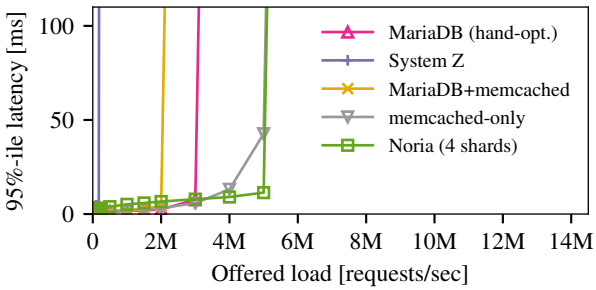


Figure 8: For a uniformly-distributed, read-heavy (95%/5%) workload on Figure 2, Noria performs similarly to the (unrealistic) memcached-only setup.

Noria uses natural queries; other systems except System Z manually pre-compute vote counts.

Clients read and insert votes for randomly-chosen stories; we measure the 95th-percentile latency for each offered load. Before measurement begins, we populate the `stories` table with 500k records and perform 40 seconds of warmup using the same workload as the benchmark itself. Absolute throughput is higher in these experiments because the data-flow only contains a single query and clients batch reads and writes for up to 1ms.

Figure 7 shows results for a **skewed workload** similar to Lobsters', with story popularity following a Zipfian distribution ($s = 1.08$). With 95% reads, Noria outperforms all other systems, including the unrealistic cache-only deployment (Figure 7a). Most updates write votes for popular stories, which creates write contention problems in MariaDB and System Z. The MariaDB+memcached setup performs equally poorly: on memcached invalidations for popular keys, multiple clients miss and a "thundering herd" of clients simultaneously issues database queries [54, §3.2.1]. memcached on its own scales, but Noria outperforms it (despite doing more work) since Noria's lockless views avoid contention for popular keys. Noria scales to 14M requests/second with four shards. Noria also handles a **write-heavy workload** (50% writes) well (Figure 7b): although absolute performance has dropped, Noria still outperforms all other systems apart from the cache-only setup. This is because sharding allows data-parallel write processing, which helps Noria scale to 2M requests/second.

With a (less-realistic) **uniform workload**, other systems come closer to Noria's 5M requests/second (Figure 8). System Z does better than before, but suffers from slow writes to the materialized view. MariaDB+memcached, perhaps surprisingly, performs worse than MariaDB, which scales to 3M requests/second: the reason lies in the extra work (and RPCs) the application must perform for invalidations. This illustrates that a look-aside cache only helps if it avoid expensive queries; a write-through cache avoids invalidation overheads, but would still perform worse than the idealized memcached-only setup (and thus, than Noria).

Separately, we evaluated Noria's view maintenance against DBToaster [2, 53], a state-of-the-art materialized view maintenance system that compiles view definitions to native code. DBToaster (v2.2.3387) lacks support for persistent base tables, concurrent reads, or multicore parallelism—its only read operation snapshots entire views—but it does provide fast updates to materialized views. When we constrain Noria to only one shard and data-flow worker thread, we expect DBToaster to outperform it, since DBToaster's generated C++ code does close-to-minimal work to incrementally maintain the vote count. We measure the write throughput of 50M uniformly-distributed votes that update `StoriesWithVC` for 500k stories. Noria achieves 240k single-record writes/second for fully-populated state, and 1M writes/second for fully-evicted state. DBToaster only supports fully-populated state, and achieves 520k single-record writes/second. At the same time, Noria is more memory-efficient, using 6.2 GB of memory for base tables and all derived state, 36% of DBToaster's 17 GB.

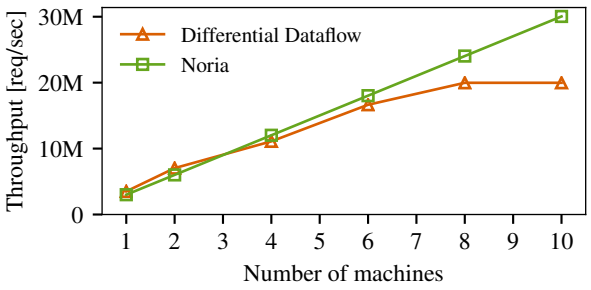


Figure 9: For a uniform 95%/5% workload, Noria scales to ten machines with sub-100ms 95th %tile latency by sharding the data-flow. Differential dataflow [44] scales less well due to its inter-worker coordination.

Additionally, Noria can process shards in parallel and use more machines to increase throughput.

8.3 Distribution over multiple servers

We next evaluate Noria's support for distributed operation. Can Noria effectively use multiple machines' resources given a scalable workload?

We evaluate the 95%-read Lobsters subset from §8.2 with two million stories. We shard the data-flow on `stories.id` and vary the number of machines from one to ten, with each machine hosting four shards. For a deployment with n Noria machines, we scale client load to $n\times$3M requests/second in a partially open-loop test harness. This arrangement achieves close to Noria's maximum load at sub-100ms 95th-percentile latency for two million stories on one machine. Load generators select stories uniformly at random, so the workload is perfectly shardable. The ideal result is a straight diagonal, with n machines achieving n times the throughput of a single one. Figure 9 shows that Noria achieves this and serves the full per-machine load at all points.

We also implemented this benchmark for a state-of-the-art Differential Dataflow (DD) implementation (v0.7) in Rust [44] based on Naiad and its earlier version of DD [46, 51]. Since DD lacks a client-facing RPC interface, we co-locate DD clients with workers; this does not disadvantage DD since load generation is cheap compared to RPC processing. DD uses 12 worker threads and four network threads per machine.

Figure 9 shows that Noria is competitive with DD on this benchmark. On one and two machines, DD supports a slightly higher per-machine load (3.5M requests/second *vs.* Noria's 3M) within our 95th-percentile latency budget of 100ms. Beyond four machines, however, DD fails to meet Noria's maximum per-machine load. Its supported throughput tails off to around 20M requests/sec at ten machines. This tail-off is due to DD's progress-tracking protocol, which coordinates between workers to expose writes atomically, and which imposes increasing overhead as the number of machines grows. DD amortizes this coordination by increasing its batch size, and consequently sees increased latency as throughput increases. Noria avoids such coordination and scales well, but offers only eventually-consistent reads.

8.4 State size

Noria relies on partial state to keep its memory footprint low. How much of Noria's state for Lobsters can be partial, and how does Noria perform when it evicts from partial state to meet a memory limit? We investigate these questions using the full Lobsters application, first at Lobsters production scale, and then at 10× scale.

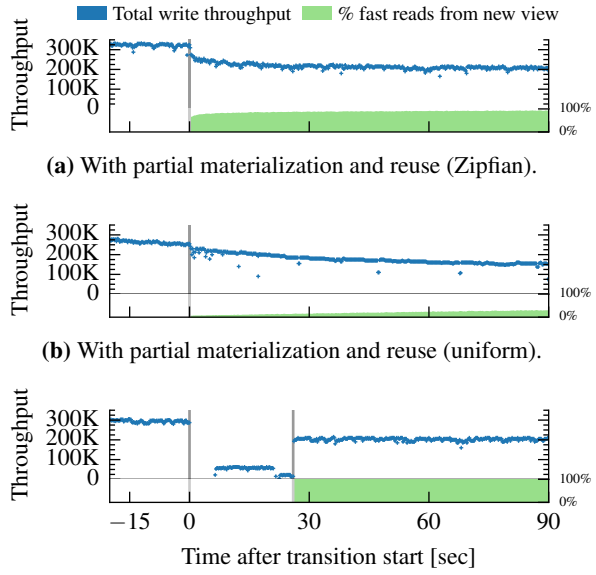
The Noria data-flow for the natural Lobsters queries has 235 operators, of which 60 of are stateful. With partial state disabled, *i.e.*, forcing all data-flow operators to keep full state, Noria needs 789 MB of in-memory state (8× the base table size of 137 MB). With partial state enabled, 35 of the stateful operators can use partial state; the remaining 25 are part of unparameterized views (*e.g.*, all stories on the front page) whose state Noria cannot make partial as they lack suitable keys. Together, the non-partial state occupies 73 MB: Noria's essential memory requirement for Lobsters therefore amounts to 9% of total state (adding an overhead of 53% of base table size). Noria can evict and re-compute the remaining 91% of state should it exceed a memory limit.

As for any cache, this memory limit should exceed the application's working set size to achieve low read latency and avoid thrashing of evictions and upqueries. For Lobsters, the working set size depends on the offered load, as higher load means a wider range of stories are read. We determine it by varying Noria's state size limit (and hence, eviction frequency) and measuring 95th-percentile read latency. With production-scale Lobsters data, Noria's working set contains 525 MB of state (60% of total, 3.8× base tables) at an offered load of 2,300 pages/second. However, with a few thousand users, the production Lobsters deployment is small. Our benchmark further understates its size as we use synthetic story and comment texts of a few bytes. Hence, we repeated this experiment with the Lobsters data scaled up by 10×. Noria meets sub-100ms 95th percentile latency at 2,300 pages/second if the memory limit exceeds the 2.6 GB working set (38% of 7 GB total state; 3× base tables).

These results suggest that Noria imposes a reasonable space overhead (around 3× base table size) for Lobsters, and that partial state is key to reducing the overhead.

8.5 Live data-flow adaptation

In a traditional database, query changes are easy and instantaneous. Can Noria's data-flow adaptation seamlessly transition to include new SQL expressions? The goal is for the transition to complete quickly, for write



(a) With partial materialization and reuse (Zipfian).

(b) With partial materialization and reuse (uniform).

(c) No reuse or partial materialization (Zipfian).

Figure 10: Reuse and partial state allow Noria to adapt the live data-flow. Gray lines delimit start and end of the transition (in (a) and (b), the transitions are almost instantaneous); the green shaded area shows the fraction of new view reads that require no upqueries. Reads from the old view (not shown) proceed at full speed throughout.

performance to remain stable, for reads from existing views to be unaffected, and for reads from newly-added views to quickly achieve low latency.

We test this by adding a modified version of the `StoriesWithVC` view to the Lobsters subset. This new view, `StoriesWithRatings`, uses numeric ratings stored in a `ratings` base table instead of votes. It also reflects old votes scaled to a rating. We first load an unsharded Noria with 2M stories and 30M votes, then transition to the new program. Once the transition finishes, clients perform "rating reads" from `StoriesWithRatings` and start writing to the new `ratings` table. Throughout the experiment, clients also read the `StoriesWithVC` view, and write to the `votes` table. We expect post-transition throughput to be reduced—the new data-flow graph is larger, with more tables and deeper paths—although removing the old view would increase throughput again. However, we hope that throughput and latency do not suffer greatly *during* the transition.

Figure 10a shows the transition with reuse and partial materialization enabled. The transition completes immediately: Noria creates the new operators and view as empty, and populates them on demand in response to reads. Due to the skewed read and write distributions, upqueries for only a few popular keys suffice for Noria to serve the majority of rating reads without recursive upqueries. Reuse is also crucial: without reusing `VoteCount`, Noria must upquery rating reads by recomputing from the base tables. This leads to slow upqueries for popular stories, as the data-flow must recount their votes. With reuse enabled, pre-computed vote counts satisfy the upqueries. The results also follow this pattern for a uniform workload (Figure 10b). Initially, most rating reads are slow, but fast reads increase as the partial state populates; write throughput is reduced because data-flow updates contend with upquery responses. Contention increases as more entries populate, since fewer updates hit evicted state.

Figure 10c shows the same transition (with a Zipfian workload), but with partial materialization and operator reuse disabled. Noria fully populates the `StoriesWithRatings` view and all internal stateful operators during the transition. It copies `votes` and `stories` to bootstrap the rating aggregation state, and then copies the resulting state again to initialize the new external view. Each copy stops write processing for several seconds, and Noria's state transfer to the new operators via the data-flow slows down concurrent writes. When transition completes after 25 seconds, the `StoriesWithRatings` view is fully materialized and all rating reads are fast. This illustrates that partial state and reuse are crucial for downtime-free data-flow transitions.

How often can Noria achieve a live transition in practice? In a separate analysis of query and schema changes in HotCRP and TPC-W, we found that Noria live-transitioned for over 95% of program changes. Existing approaches are less flexible: System Z must rebuild its materialized views on change; a memcached cluster must be carefully transitioned [54, §4.3]; DBToaster lacks support for query changes; and even relational databases pause writes during some schema updates.

8.6 Discussion

We evaluated Lobsters both at production scale and at 10× scale, but many web applications are much larger still. We believe that Noria can also support such applications. For applications with many queries, and consequently a large data-flow, Noria can assign shards of only some operators to each machine, sending cross-operator traffic over the network. Similarly, Noria can shard large base tables and operators with large state across machines. Efficient resharding and partitioning the data-flow to minimize network transfers are important future work for Noria to achieve truly large scale.

We also believe Noria is well suited for applications whose working sets change over time. Many large, real-world applications see such changing workloads; for instance, an old story may suddenly become popular. As

clients request such items, Noria's upqueries bring them into the working set, making subsequent reads fast.

9 Related work

Noria builds on considerable related work.

Data-flow systems excel at data-parallel computing [36, 51], including on streams, but cannot serve web applications directly. They only achieve low-latency incremental updates at the expense of windowed state (and incomplete results) or by keeping full state in memory. Noria's partially-stateful data-flow lifts this restriction. A few data-flow systems can reuse operators automatically: for example, Nectar [28] detects similar subexpressions in DryadLINQ programs, similar to Noria's automated operator reuse, using DryadLINQ-specific merge and rewrite rules. Support for dynamic changes to a running data-flow is more common: CIEL [52] dynamically extends batch-processing data-flows, as does Ray [58] for stateful "actor" operators' state transitions in reinforcement learning applications. Noria dynamically changes long-running, low-latency streaming computations by modifying the data-flow; unlike existing streaming data-flow systems like Naiad [51] or Spark Streaming [76], it has no need for a restart or recovery from a checkpoint.

Stream processing systems [3, 11, 39, 71, 76] often use data-flow, but usually have windowed state and static queries that process only new records. STREAM [6] identifies opportunities for operator reuse among static queries; Noria achieves similar reuse for dynamic queries. S-Store [47] lacks Noria's partial materialization and state reuse, but combines a classic database with a stream processing system using trigger-based view maintenance. S-Store enables transactional processing, a future goal for Noria.

Database **materialized views** [29, 41] were devised to cache expensive analytical query results. Commercial databases' materialized view support [1] is limited [49, 63] and views must usually be rebuilt on change. However, there is considerable research on incremental view maintenance in databases [30, 40, 41, 70, 77, 81]. Noria builds upon ideas from this work, but applies them in the context of a concurrent, stateful data-flow system for web applications. This requires efficient fine-grained access to views, solutions to new coordination problems and concurrency races, as well as inexpensive long-term adaptation as view definitions change. DBToaster [2, 53] supports incremental view maintenance under high write loads with generated recursive delta query implementations. Noria sees lower single-threaded performance, but supports parallel processing and changing queries; adding native-code generation to Noria might further improve its performance, but would complicate operator reuse. Pequod [37] and DBProxy [4] support **partial materialization** in response to client demand, although Pequod is limited to static queries, and unlike Noria, neither shares state nor processing across queries.

The problem of detecting shared subexpressions (§5.1) is a **multi-query optimization** (MQO) problem [21, 59, 78]. MQO tries to maximize sharing across a batch of expressions, with the freedom to rewrite any expression to suit the others. Like joint query processing systems [10, 25, 31], Noria faces the more restricted problem of mutating *new* expressions to increase their opportunity to share *existing* expressions in the data-flow.

A wide array of tools deal with websites' **query and schema transitions** [9, 23, 26, 56, 65]. Like Noria, they aim to transition backend stores without interruption in client service, but they require developers to manually configure complex "ghost tables" or binlog-following triggers. Base table schema changes increase complexity further [73]. Noria handles query changes transparently, and efficiently applies common base table schema changes by supporting many concurrent base table schemas. Most of its data-flow transitions are live for reads and writes without added complexity.

Finally, some open-source systems have experimented with flexible query and schema changes. Apache Kafka [5] achieves some flexibility in query and schema changes as used by the New York Times [68], and similar ideas were proposed as an extension proposal for Samza [38]. To our knowledge, however, no prior system achieves the performance and flexibility of Noria.

10 Conclusions

Noria is a web application backend that delivers high performance while allowing for simplified application logic. Partially-stateful data-flow is essential to achieving this goal: it allows fast reads, restricts Noria's memory footprint to state that is actually used, and enables live changes to the data-flow. In future work, we plan to add more flexible sharding, range indexes, and better eviction strategies.

Noria is open-source software and available at:

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

Acknowledgements

We thank Joana da Trindade and Nikhil Benesch for contributions to our implementation, as well as Frank McSherry for assisting with implementation and tuning of the differential dataflow benchmark. Jon Howell provided helpful feedback that much improved the paper, as did Ionel Gog, Frank McSherry, David DeWitt, Sam Madden, Amy Ousterhout, Tej Chajed, Anish Athalye, and the PDOS and Database groups at MIT. We are also grateful to the helpful comments we received from our anonymous reviewers, as well as from Wyatt Lloyd, our shepherd. This work was funded through NSF awards CSR-1301934, CSR-1704172, and CSR-1704376.

References

[1] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. "Automated Selection of Materialized Views and Indexes in SQL Databases". In: *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB)*. Cairo, Egypt, Sept. 2000, pages 496–505.

[2] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. "DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views". In: *Proceedings of the VLDB Endowment* 5.10 (June 2012), pages 968–979.

[3] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. "MillWheel: Fault-tolerant Stream Processing at Internet Scale". In: *Proceedings of the VLDB Endowment* 6.11 (Aug. 2013), pages 1033–1044.

[4] Khalil Amiri, Sanghyun Park, Renu Tewari, and Sriram Padmanabhan. "DBProxy: a dynamic data cache for web applications". In: *Proceedings of the 19th International Conference on Data Engineering (ICDE)*. Mar. 2003, pages 821–831.

[5] Apache Software Foundation. *Apache Kafka: a distributed streaming platform*. URL: http://kafka.apache.org/ (visited on 09/14/2017).

[6] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. "STREAM: The Stanford Data Stream Management System". In: *Data Stream Management: Processing High-Speed Data Streams*. Edited by Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Berlin/Heidelberg, Germany: Springer, 2016, pages 317–336.

[7] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. "Finding a Needle in Haystack: Facebook's Photo Storage". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. Vancouver, British Columbia, Canada, Oct. 2010, pages 1–8.

[8] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. "TAO: Facebook's Distributed Data Store for the Social Graph". In: *Proceedings of the USENIX Annual Technical Conference*. San Jose, California, USA, June 2013, pages 49–60.

[9] Mark Callaghan. *Online Schema Change for MySQL*. URL: https://www.facebook.com/note.php?note_id=430801045932 (visited on 02/01/2017).

[10] George Candea, Neoklis Polyzotis, and Radek Vingralek. "A Scalable, Predictable Join Operator for Highly Concurrent Data Warehouses". In: *Proceedings of the VLDB Endowment* 2.1 (Aug. 2009), pages 277–288.

[11] Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas. "Apache Flink: Stream and batch processing in a single engine". In: *IEEE Data Engineering* 38.4 (Dec. 2015).

[12] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. "Bigtable: A Distributed Storage System for Structured Data". In: *Proceedings of the 7th USENIX Symposium on Operating System Design and Implementation (OSDI)*. Seattle, Washington, USA, Nov. 2006.

[13] Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. "Realtime Data Processing at Facebook". In: *Proceedings of the 2016 SIGMOD International Conference on Management of Data*. San Francisco, California, USA, 2016, pages 1087–1098.

[14] CockroachDB. *Structured data encoding in CockroachDB SQL*. Jan. 2018. URL: https://github.com/cockroachdb/cockroach/blob/master/docs/tech-notes/encoding.md (visited on 04/20/2018).

[15] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. "PNUTS: Yahoo!'s Hosted Data Serving Platform". In: *Proceedings of the VLDB Endowment* 1.2 (Aug. 2008), pages 1277–1288.

[16] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. "Spanner: Google's Globally Distributed Database". In: *ACM Transactions on Computer Systems* 31.3 (Aug. 2013), 8:1–8:22.

[17] Carlo A. Curino, Letizia Tanca, Hyun J. Moon, and Carlo Zaniolo. "Schema Evolution in Wikipedia: toward a Web Information System Benchmark". In: *Proceedings of the International Conference on Enterprise Information Systems (ICEIS)*. June 2008.

[18] Databricks, Inc. *Structured Streaming in Production – Recover after changes in a streaming query*. URL: https://docs.databricks.com/spark/latest/structured-streaming/production.html#recover-after-changes-in-a-streaming-query (visited on 09/06/2018).

[19] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. "Dynamo: Amazon's Highly Available Key-value Store". In: *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. Stevenson, Washington, USA, Oct. 2007, pages 205–220.

[20] Dror G. Feitelson, Eitan Frachtenberg, and Kent L. Beck. "Development and Deployment at Facebook". In: *IEEE Internet Computing* 17.4 (July 2013), pages 8–17.

[21] Sheldon Finkelstein. "Common Expression Analysis in Database Applications". In: *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data*. Orlando, Florida, USA, June 1982, pages 235–245.

[22] Django Software Foundation. *Django: The Web framework for perfectionists with deadlines*. Mar. 2018. URL: https://www.djangoproject.com/ (visited on 03/20/2018).

[23] Matt Freels. *TableMigrator*. URL: https://github.com/freels/table_migrator (visited on 02/01/2017).

[24] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google File System". In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*. Bolton Landing, NY, USA, Oct. 2003, pages 29–43.

[25] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. "SharedDB: Killing One Thousand Queries with One Stone". In: *Proceedings of the VLDB Endowment* 5.6 (Feb. 2012), pages 526–537.

[26] GitHub, Inc. *gh-ost: GitHub's online schema migration for MySQL*. URL: https://github.com/github/gh-ost (visited on 02/01/2017).

[27] Jon Gjengset. *evmap: A lock-free, eventually consistent, concurrent multi-value map*. URL: https://github.com/jonhoo/rust-evmap (visited on 09/13/2018).

[28] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. "Nectar: Automatic Management of Data and Computation in Datacenters". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. Vancouver, British Columbia, Canada, 2010, pages 75–88.

[29] Himanshu Gupta and Inderpal Singh Mumick. "Selection of views to materialize in a data warehouse". In: *IEEE Transactions on Knowledge and Data Engineering* 17.1 (Jan. 2005), pages 24–43.

[30] Himanshu Gupta and Inderpal Singh Mumick. "Incremental Maintenance of Aggregate and Outerjoin Expressions". In: *Information Systems* 31.6 (Sept. 2006), pages 435–464.

[31] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. "QPipe: A Simultaneously Pipelined Relational Query Engine". In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. Baltimore, Maryland, USA, June 2005, pages 383–394.

[32] Peter Bhat Harkins. *Lobste.rs access pattern statistics for research purposes*. Mar. 2018. URL: https://lobste.rs/s/cqnzl5/lobste_rs_access_pattern_statistics_for#c_hj0r1b (visited on 03/12/2018).

[33] Peter Bhat Harkins. *replying_comments view in Lobsters*. Feb. 2018. URL: https://github.com/lobsters/lobsters/blob/640f2cdca10cc737aa627dbdf0bbe398b81b497f/db/views/replying_comments_v06.sql (visited on 04/20/2018).

[34] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. "ZooKeeper: Wait-free Coordination for Internet-scale Systems". In: *Proceedings of the USENIX Annual Technical Conference*. Boston, Massachusetts, USA, June 2010, pages 149–158.

[35] Michael Isard and Martín Abadi. "Falkirk Wheel: Rollback Recovery for Dataflow Systems". In: *CoRR* abs/1503.08877 (2015).

[36] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. "Dryad: Distributed Data-parallel Programs from Sequential Building Blocks". In: *Proceedings of the 2nd ACM SIGOPS European Conference on Computer Systems (EuroSys)*. Lisbon, Portugal, Mar. 2007, pages 59–72.

[37] Bryan Kate, Eddie Kohler, Michael S. Kester, Neha Narula, Yandong Mao, and Robert Morris. "Easy Freshness with Pequod Cache Joins". In: *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Seattle, Washington, USA, Apr. 2014, pages 415–428.

[38] Martin Kleppmann. *Turning the database inside-out with Apache Samza*. Mar. 2015. URL: https://martin.kleppmann.com/2015/03/04/turning-the-database-inside-out.html (visited on 05/09/2016).

[39] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. "Twitter Heron: Stream Processing at Scale". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. Melbourne, Victoria, Australia, May 2015, pages 239–250.

[40] Per-Åke Larson and Jingren Zhou. "Efficient Maintenance of Materialized Outer-Join Views". In: *Proceedings of the 23rd International Conference on Data Engineering (ICDE)*. Apr. 2007, pages 56–65.

[41] Ki Yong Lee and Myoung Ho Kim. "Optimizing the Incremental Maintenance of Multiple Join Views". In: *Proceedings of the 8th ACM International Workshop on Data Warehousing and OLAP (DOLAP)*. Bremen, Germany, Nov. 2005, pages 107–113.

[42] Lobsters Developers. *Lobsters Database Schema (schema.rb)*. Apr. 2018. URL: https://github.com/lobsters/lobsters/blob/93fe0fdd74028cf678134d6d112ae084d8fdd928/db/schema.rb#L145-L148 (visited on 04/23/2018).

[43] Lobsters Developers. *Lobsters News Aggregator*. Mar. 2018. URL: https://lobste.rs (visited on 03/02/2018).

[44] Frank McSherry. *Differential Dataflow in Rust*. URL: https://crates.io/crates/differential-dataflow (visited on 01/15/2017).

[45] Frank McSherry. *Throughput and Latency in Differential Dataflow: open-loop measurements*. Aug. 2017. URL: https://github.com/frankmcsherry/blog/blob/master/posts/2017-07-24.md#addendum-open-loop-measurements-2017-08-14 (visited on 04/13/2018).

[46] Frank McSherry, Derek G. Murray, Rebecca Isaacs, and Michael Isard. "Differential dataflow". In: *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR)*. Asilomar, California, USA, Jan. 2013.

[47] John Meehan, Nesime Tatbul, Stan Zdonik, Cansu Aslantas, Ugur Cetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, Andrew Pavlo, Michael Stonebraker, Kristin Tufte, and Hao Wang. "S-Store: Streaming Meets Transaction Processing". In: *Proceedings of the VLDB Endowment* 8.13 (Sept. 2015), pages 2134–2145.

[48] Jhonny Mertz and Ingrid Nunes. "Understanding Application-Level Caching in Web Applications: A Comprehensive Introduction and Survey of State-of-the-Art Approaches". In: *ACM Computing Surveys* 50.6 (Nov. 2017), 98:1–98:34.

[49] Microsoft, Inc. *Create Indexed Views – Additional Requirements*. SQL Server Documentation. URL: https://docs.microsoft.com/en-us/sql/relational-databases/views/create-indexed-views#additional-requirements (visited on 04/16/2017).

[50] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, and Sanjeev Kumar. "f4: Facebook's Warm BLOB Storage System". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. Broomfield, Colorado, USA, Oct. 2014, pages 383–398.

[51] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. Farmington, Pennsylvania, USA, Nov. 2013, pages 439–455.

[52] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. "CIEL: a universal execution engine for distributed data-flow computing". In: *Proceedings of the 8th USENIX Symposium on Networked System Design and Implementation (NSDI)*. Boston, Massachusetts, USA, Mar. 2011, pages 113–126.

[53] Milos Nikolic, Mohammad Dashti, and Christoph Koch. "How to Win a Hot Dog Eating Contest: Distributed Incremental View Maintenance with Batch Updates". In: *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. San Francisco, California, USA, 2016, pages 511–526.

[54] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. "Scaling Memcache at Facebook". In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. Lombard, Illinois, USA, Apr. 2013, pages 385–398.

[55] Oracle Corp. *MySQL Connector/Python Developer Guide*. URL: https://dev.mysql.com/doc/connector-python/en/connector-python-api-mysqlcursorprepared.html (visited on 09/05/2018).

[56] Percona LLC. *pt-online-schema-change*. URL: https://www.percona.com/doc/percona-toolkit/2.2/pt-online-schema-change.html (visited on 02/01/2017).

[57] Dan R. K. Ports, Austin T. Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. "Transactional Consistency and Automatic Management in an Application Data Cache". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. Vancouver, British Columbia, Canada, 2010, pages 279–292.

[58] "Ray: A Distributed Framework for Emerging AI Applications". In: *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, California, USA, Oct. 2018.

[59] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. "Efficient and Extensible Algorithms for Multi Query Optimization". In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. Dallas, Texas, USA, May 2000, pages 249–260.

[60] Kenneth Salem, Kevin Beyer, Bruce Lindsay, and Roberta Cochrane. "How to Roll a Join: Asynchronous Incremental View Maintenance". In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. Dallas, Texas, USA, 2000, pages 129–140.

[61] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. "Continuous Deployment at Facebook and OANDA". In: *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. Austin, Texas, USA, 2016, pages 21–30.

[62] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. "Open Versus Closed: A Cautionary Tale". In: *Proceedings of the 3rd USENIX Conference on Networked Systems Design and Implementation (NSDI)*. San Jose, California, USA, 2006, pages 239–252.

[63] Jes Schultz Borland. *What You Can (and Can't) Do With Indexed Views*. Brent Ozar Unlimited Blog. URL: https://www.brentozar.com/archive/2013/11/what-you-can-and-cant-do-with-indexed-views/ (visited on 04/16/2017).

[64] Ziv Scully and Adam Chlipala. "A Program Optimization for Automatic Database Result Caching". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. Paris, France, 2017, pages 271–284.

[65] SoundCloud Ltd. *Large Hadron Migrator*. URL: https://github.com/soundcloud/lhm (visited on 02/01/2017).

[66] Facebook Open Source. *A persistent key-value store for fast storage environments*. Apr. 2018. URL: http://rocksdb.org/ (visited on 04/20/2018).

[67] Facebook Open Source. *MyRocks data dictionary format*. Apr. 2018. URL: https://github.com/facebook/mysql-5.6/wiki/MyRocks-data-dictionary-format (visited on 04/20/2018).

[68] Boerge Svingen. *Publishing with Apache Kafka at The New York Times*. Confluent, Inc. blog. Sept. 2017. URL: https://www.confluent.io/blog/publishing-apache-kafka-new-york-times/ (visited on 09/14/2017).

[69] The PHP Group. *PHP Data Objects*. URL: http://php.net/manual/en/book.pdo.php (visited on 09/05/2018).

[70] Frank W. Tompa and Joseph A. Blakeley. "Maintaining Materialized Views Without Accessing Base Data". In: *Information Systems* 13.4 (Oct. 1988), pages 393–406.

[71] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. "Storm@Twitter". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. Snowbird, Utah, USA, June 2014, pages 147–156.

[72] Werner Vogels. "Eventually Consistent". In: *Communications of the ACM* 52.1 (Jan. 2009), pages 40–44.

[73] Jacqueline Xu. *Online migrations at scale*. Stripe engineering blog. URL: https://stripe.com/blog/online-migrations (visited on 02/01/2017).

[74] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. "Precise, Dynamic Information Flow for Database-backed Applications". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Santa Barbara, California, USA, June 2016, pages 631–647.

[75] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing". In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. San Jose, California, USA, Apr. 2012, pages 15–28.

[76] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. "Discretized Streams: Fault-tolerant Streaming Computation at Scale". In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. Farmington, Pennsylvania, USA, Nov. 2013, pages 423–438.

[77] Jingren Zhou, Per-Åke Larson, and Hicham G. Elmongui. "Lazy Maintenance of Materialized Views". In: *Proceedings of the 33rd International Conference on Very Large Data Bases*. Vienna, Austria, Sept. 2007, pages 231–242.

[78] Jingren Zhou, Per-Ake Larson, Johann-Christoph Freytag, and Wolfgang Lehner. "Efficient Exploitation of Similar Subexpressions for Query Processing". In: *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Beijing, China, 2007, pages 533–544.

[79] Jingren Zhou, Per-Åke Larson, and Jonathan Goldstein. *Partially Materialized Views*. Technical report MSR-TR-2005-77. Microsoft Research, June 2005.

[80] Jingren Zhou, Per-Åke Larson, Jonathan Goldstein, and Luping Ding. "Dynamic Materialized Views". In: *Proceedings of the 23rd International Conference on Data Engineering (ICDE)*. Istanbul, Turkey, Apr. 2007, pages 526–535.

[81] Yue Zhuge, Héctor García-Molina, Joachim Hammer, and Jennifer Widom. "View Maintenance in a Warehousing Environment". In: *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*. San Jose, California, USA, May 1995, pages 316–327.