Noria: a new take on fast web application backends

Writing web applications that tolerate high load is difficult. The reason is that the backend storage system that the application relies on -- typically a relational database, like MySQL -- can easily become a serious bottleneck with many clients. Each page view typically involves ten or more database queries, which each take CPU time on the database servers to evaluate. To avoid such slow database interactions and reduce load on the database, applications often introduce caches (like memcached or Redis) that store already-computed query results for fast common case access. These caches, however, impose significant application complexity, as the application must query, invalidate, and maintain them [1]. Surely there has to be a better way?

At OSDI 2018, we presented Noria [2], a new web application backend that delivers the same fast reads as an in-memory cache in front of the database, but without the application having to manage the cache. Even better, Noria still accepts SQL queries and allows changes to the queries without extra effort, just like a database. Noria performs well: it serves up to 14M requests/second on a single server, and supports a 5x higher load than issuing carefully hand-tuned queries to MySQL.

Data-flow for high performance

At first glance, Noria is similar to a database as it processes SQL queries. However, instead of evaluating queries on-the-fly, as a traditional database would, the application registers long-term queries with Noria for repeated use. Queries contain free parameters that the application specifies when it actually executes its reads, similar to the interface provided by prepared SQL statements. From the pre-specified queries, Noria constructs a *data-flow graph* that *continuously* and *incrementally* evaluates the queries when the underlying data changes.

Data-flow processing was initially invented in the 1970s for circuit design, but has recently been adopted for large-scale parallel data-processing, for example in systems like Dryad [4], Naiad [5], and TensorFlow [6]. In data-flow, the system represents computations as a graph whose vertices are data-flow *operators* and whose edges carry *updates* between the operators. When an operator receives an update on and incoming edge, it processes the update (possibly consulting internal *state* that it keeps) and emits zero or more updates of its own on all its outgoing edges. This graph representation is appealing, as it makes the computation's dependencies explicit: update propagation across different edges and processing at different vertices can happen in parallel. Therefore, data-flow processing is well-suited to scaling across multiple CPU cores and servers.

In Noria, the data-flow graph connects classic database tables at its inputs to *materialized views* at its leaves. The intervening operators proactively execute the application's queries for each change to the tables. Noria generates the data-flow from SQL queries using a process similar to database query planning. Noria then serves all reads directly from the materialized views in memory, which makes reads as fast as reading from a cache. When the records in a table change (e.g., in response to a client insert or update), Noria feeds updates through the data-flow to modify the materialized views as necessary.

The idea of materialized views has been around for decades, and some commercial and research databases support them. However, existing implementations lack the flexibility and performance that web applications require.

Noria's approach effectively flips the database query model on its head: instead of executing queries in response to reads, Noria executes them in response to *writes*. Reads are simple lookups into materialized

state, which makes them (much) faster by moving work from reads to writes. Modern web applications are generally read-heavy, so this trade-off makes sense for them. Furthermore, since Noria takes care of making reads fast even for complex SQL queries, the developer no longer needs to write error-prone, complex cache-maintenance code, or tune their queries for fast execution. They can simply issue the SQL queries they wish, inline aggregations and all, and Noria does the rest.

An example: votes for news stories

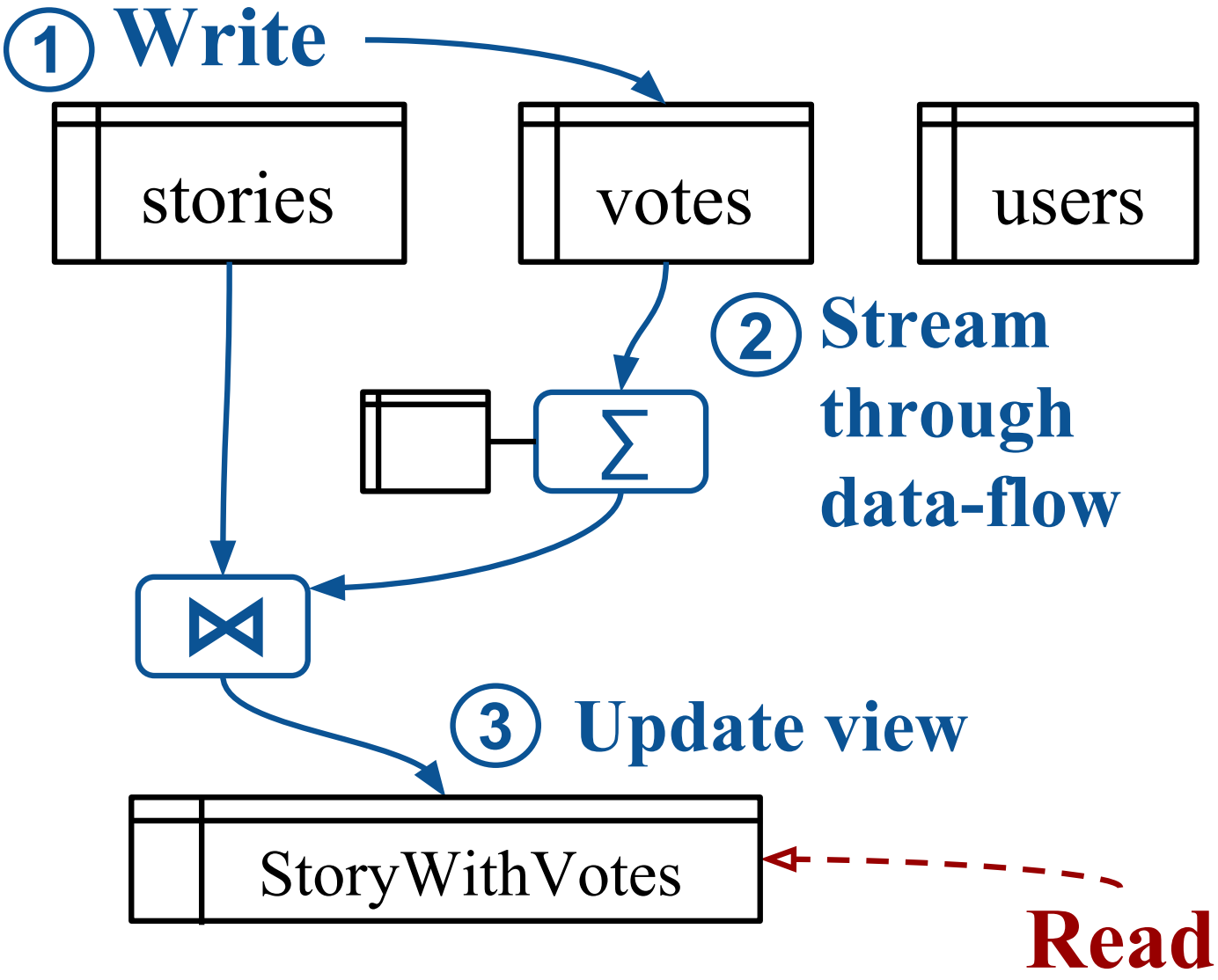

Figure 1a: *Example Noria data-flow for a query that counts the votes for each story in a news aggregator, and incrementally updates the count as new votes arrive (solid, blue). Reads hit materialized view (dashed, red).*

Let's take a look at how Noria executes a particular SQL query. Figure 1a above shows the data-flow that Noria constructs when given a query that counts the votes for each story in a news aggregator like HackerNews or lobste.rs. The query joins with the `stories` table to retrieve the story's details (title, author, etc.). When a client inserts a new vote (let's say for the story with the identifier `A`), an update enters the data-flow at the vertex that corresponds to the `votes` table. From there, the data-flow propagates the update to the aggregation vertex below, which looks up the *current* vote count for the new vote's story in internal state it maintains (say, 7). The count then updates the internal state to record that the vote count for that story is now

8, and emits an *update* to its children saying that the count for `A` is now 8, not 7. This update arrives at the join, which looks up `A`'s title in `stories`, and produces a new update that says `A`, whose title is "Space elevator nearly completed", now has a vote count of 8, not 7. That update finds its way to the materialized view `StoryWithVotes`, which Noria updates appropriately so that any subsequent read from it sees `A`'s vote count as 8. Here, we say that `StoryWithVotes` is *keyed* by the story's identifier. In general, the key for a view is dictated by a set of free parameters in the corresponding SQL query issued by the application.

Making data-flow work for web applications

Naively adding new queries and initializing their data-flow state and materialized views may require Noria to compute a significant amount of state for the new query and induce downtime while it does so. More generally, if Noria always kept all state for all stateful internal data-flow operators and all its materialized views, its memory footprint would explode with many queries. Noria solves this problem by introducing *partially-stateful* data-flow. This new model in turn enables Noria to support *dynamic* materialized views, where the set of queries changes over time without requiring a system restart.

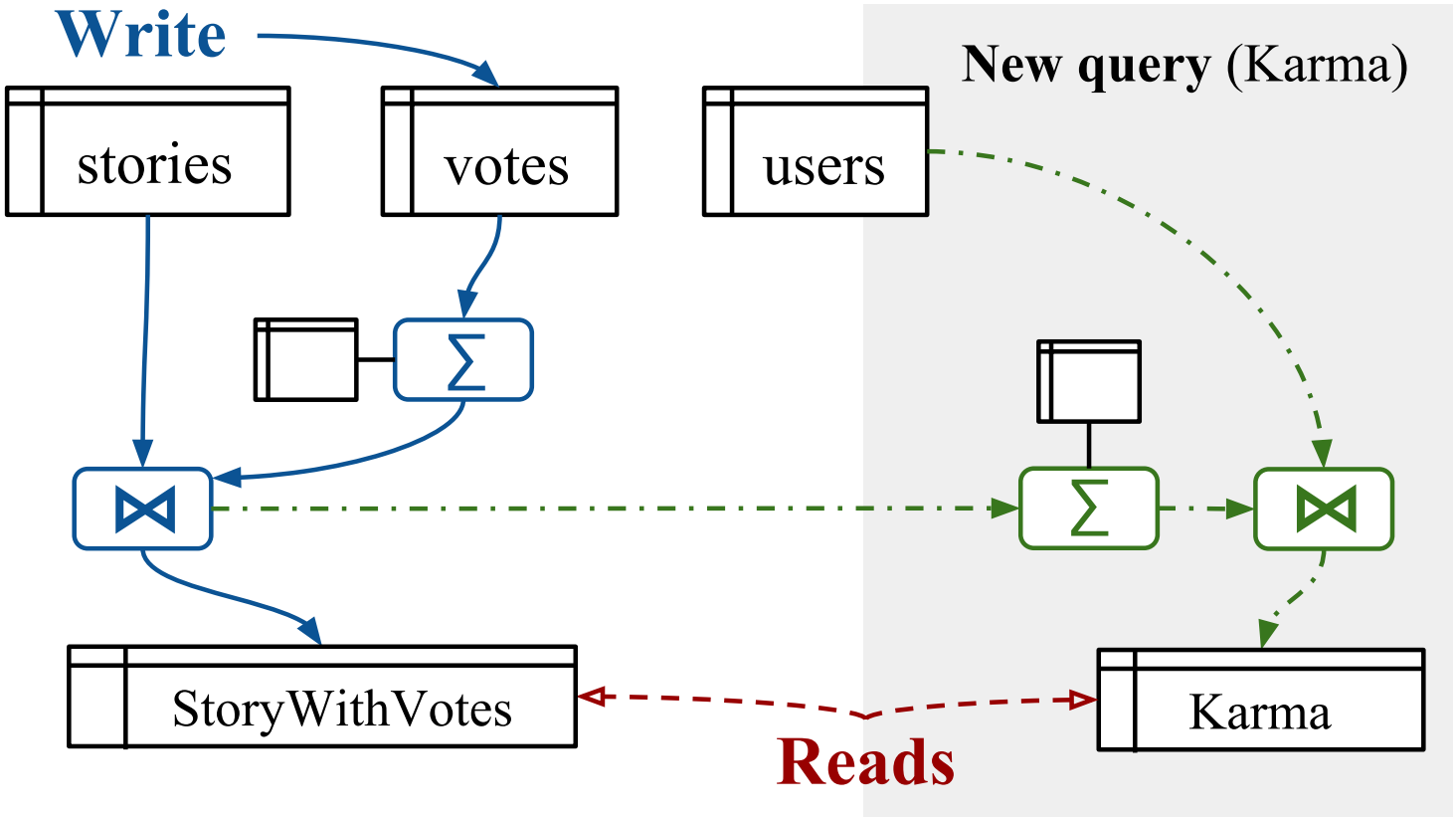

Figure 1b: *If the application adds another query to compute the "Karma" score for each user (the total votes received the user's stories), Noria dynamically adds the extra operators and materialized views needed to the running data-flow (dash-dot, green).*

Dynamic change. Figure 1b shows the data-flow from Figure 1a after the application adds a new `Karma` query (the shaded gray region). `Karma` computes the total votes for all stories posted by a given user. Notice that the data-flow path for `Karma` partially overlaps with that of `StoryWithVotes`. Noria realizes that it does not need to re-count all the votes, but can instead re-use the counts it already has. When the application first issues the `Karma` query, Noria extends the currently running data-flow to also include the extra data-flow operators needed for the new query, and a new materialized view for `Karma`. It then initializes the state needed by stateful data-flow operators and the materialized view before making the latter available for application reads. Reads of old views are unaffected by changes to the data-flow, as are writes to unconnected parts of the

data-flow. In combination with partial state, Noria makes the change instantaneous for writes as well.

Data-flow systems prior to Noria were designed for stream, graph and parallel "big data" processing, and cannot change the computation (i.e., queries) without restarting [6]. They must either keep all computed state in internal operator state and materialized views, or apply *windowing* to reduce computed state by throwing away old records. For web applications, neither is acceptable: the backend cannot be down when queries change, and must return complete results rather than ones based only on recent changes.

This brings us back to Noria's key idea: partially-stateful data-flow. Noria's data-flow changes on-the-fly in response to query changes, and keeps only a subset of state in memory, fetching missing data on-demand.

Partial state. Noria marks some keys in each data-flow state as *absent*, and recomputes them only when needed. To support such recomputation -- e.g., when a client reads an absent key from a materialized view -- Noria relies on *upqueries* through the data-flow. Upqueries allow a vertex to ask its ancestors to recompute the absent state the vertex needs in order to serve an application read. The upstream ancestors respond to an upquery with the records in their state that match the absent key or keys specified by the upquery, and the results percolate back down through the data-flow. Since upqueries allow vertices to recover absent state, Noria is free to evict infrequently accessed state to save memory. More important, Noria also uses absent state to create new materialized views and operators with initially empty state, relying on upqueries to fill the state on demand. This allows Noria to adapt to most query changes entirely without downtime -- all that is required is to bring up a set of empty data-flow operators. Absent state also speeds up regular processing, as updates for keys that are evicted, or that the application has never requested, can be discarded early.

Partial state and upqueries are conceptually simple, but making them always correct actually requires care. Intuitively, a partially-stateful data-flow is only correct if it always -- whether directly or via upqueries -- produces the same result for a client read that a classic data-flow with full state would have returned. However, ensuring this in the face of concurrent processing in the data-flow, and with upqueries that can race with "normal" updates traveling downstream that themselves may be contained in the eventual upquery response, is difficult. Noria ensures this property using a new data-flow model and extra invariants. Some of the challenges are:

- How do data-flow operators handle updates that encounter absent state? Consider the earlier count: if its state for story `A` is absent, how can the count operator produce `(A, 8)` as the emitted update?
- How does parallel processing of complex data-flows that fork and join still ensure that upquery responses always contain all the updates processed at the queried operator exactly once?
- How do operators that change the key column handle upqueries? For example, the sum operator added in Figure 1 may upquery the join on its incoming edge for a particular user, but that join is keyed by the story identifier column.
- How do multi-ancestor operators handle upqueries if state for the upquery key is available in one ancestor, but not in the other?

Our paper [2] gives the invariants that Noria must maintain to guarantee correct execution, and points out what goes wrong if these invariants are not properly maintained.

Evaluating the Noria prototype

We implemented Noria in about 60,000 lines of Rust, along with a MySQL adapter that implements the MySQL binary protocol and makes Noria appear as a MySQL server to legacy applications. This way, Noria can support unmodified MySQL applications that use prepared statements (e.g., through PHP's PDO library). Noria supports sharding and partitioning the data-flow across cores and servers, and stores all base tables

durably in RocksDB [7]. It handles failures in the distributed system by re-creating those parts of the data-flow that a failure affects.

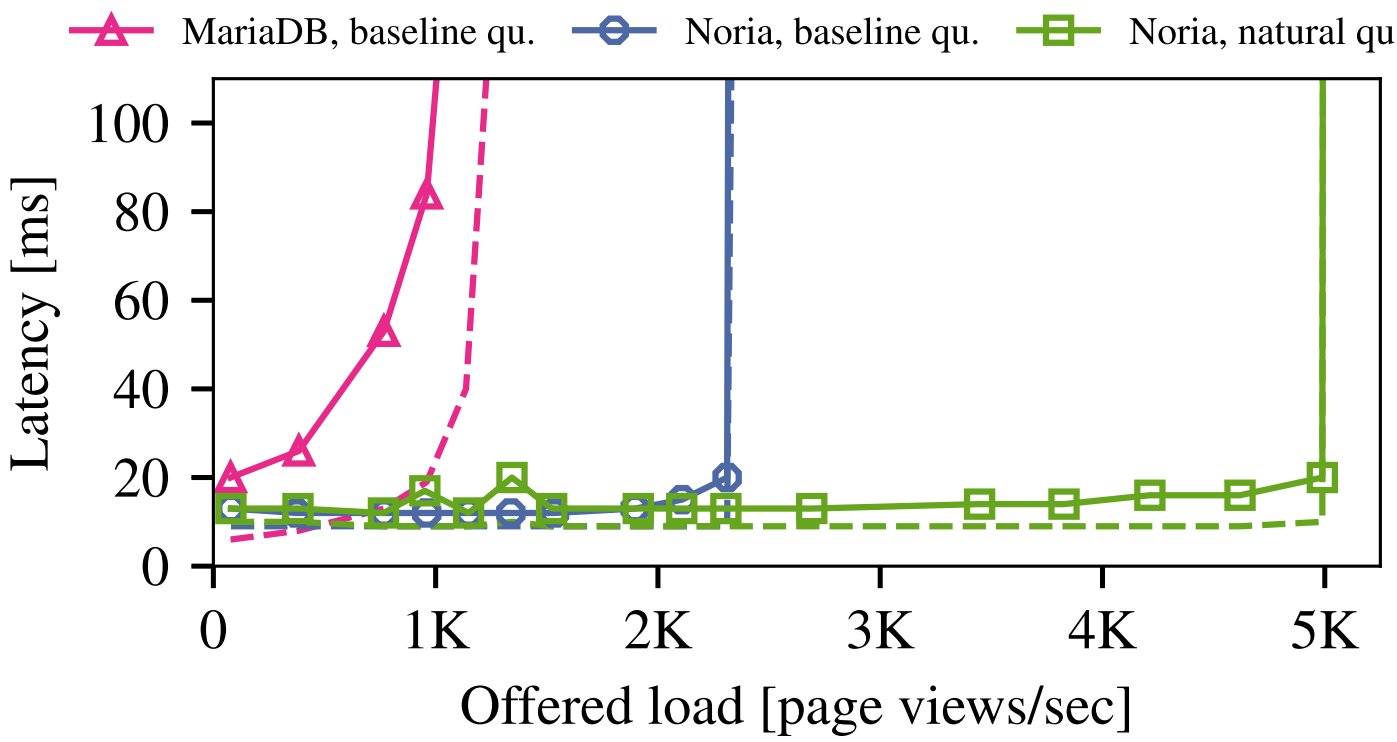

Figure 2: *Noria scales to a 5x higher load than MySQL for the lobste.rs website's workload while using queries free of hand-tuning (2.5x with the lobste.rs's developers' original queries). Solid line shows median, dashed is the 95th percentile.*

To evaluate Noria's performance and check that it *actually* makes web applications faster and reduces their complexity, we wrote a workload generator that emulates the real production work-load seen by the news aggregator website lobste.rs. lobste.rs is a Ruby-on-Rails application backed by a MySQL database and the lobste.rs developers carefully hand-optimized its queries for performance. Our benchmark issues the same SQL queries as the real lobste.rs website, with the same frequency and popularity skew, using the MySQL binary protocol. We then run that against both MySQL directly (we use MariaDB v10.1.34, a GPLv2 community fork of MySQL) and against Noria, on a 16-core Amazon EC2 VM. Figure 2 plots the offered load on the *x*-axis (in page views per second; each page issues around 10 queries) and the achieved median and 95th percentile latency on the *y*-axis (so lower is better). At the point where each setup stops scaling -- for example because it saturates the server's CPU cores -- the latency curve forms a "hockey stick" that shoots up as the system cannot keep up with the load any more. The results indicate that Noria scales to a 2.5x--5x higher load than the MySQL baseline. For the initial result (blue line with circles, 2.5x improvement), we use the exact same queries as the lobste.rs developers. We then go a step further and remove all manual optimizations from the queries (green line with squares). For example, the original application keeps `upvotes` and `downvotes` columns in the `stories` table and updates them on every vote, so that read query evaluation avoids doing a `COUNT` over `votes`. This is effectively a hand-rolled "materialized view" of the vote count, but required the developers to customize the application to update this column whenever the vote count changes. In Noria, such hand-tuning is unnecessary. Indeed, removing the hand-optimizations from the queries, we see a 5x speed-up over MySQL. The difference here comes from the fact that by not having to maintain these auxiliary values in the base tables (but instead having Noria maintain them in the data-flow), we avoid an extra `UPDATE` query and parallelize the update processing.

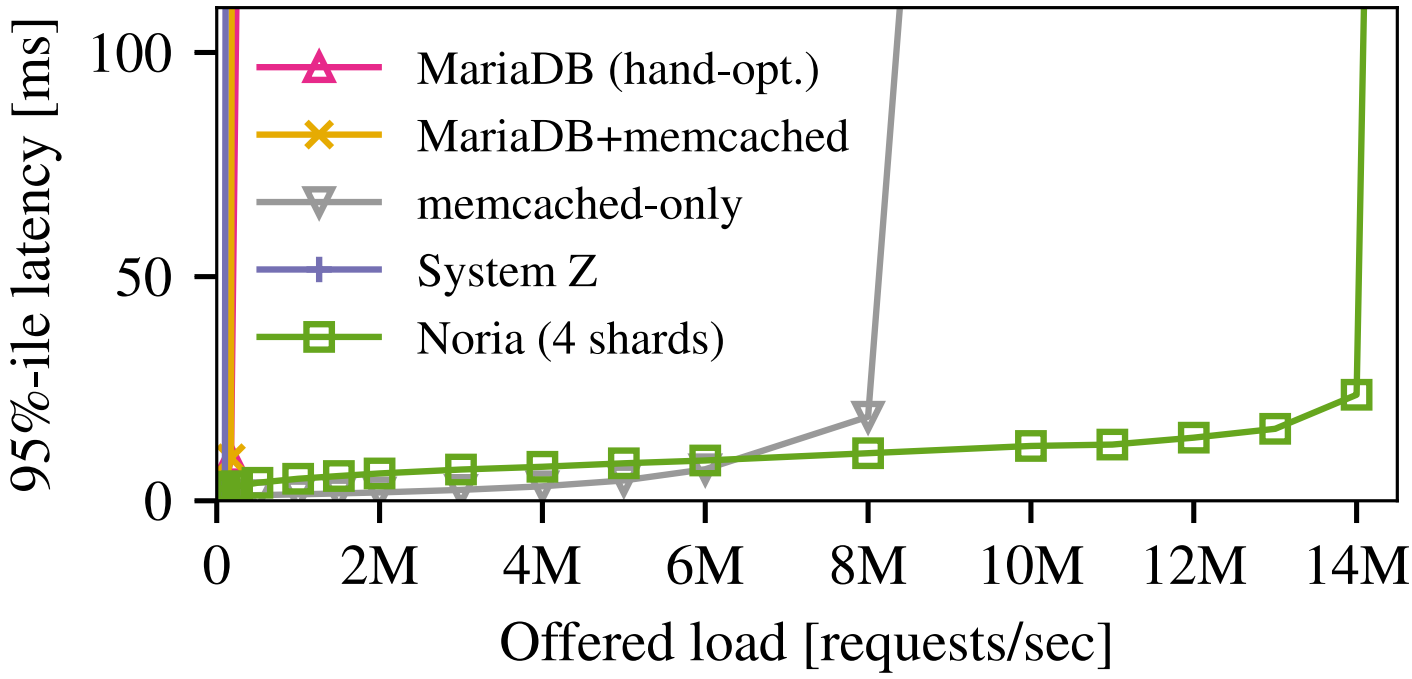

Figure 3: *Noria supports 14 million requests/second for a read-heavy (95% reads) workload, while other systems achieve only 200,000 requests/second -- with the exception of an unrealistic memcached-only setup that does strictly less work, but still underperforms Noria.*

To quantify how much Noria improves performance over existing approaches, we choose a single, common query (the join of `stories` with vote counts) and issue that same query against a number of common web backend setups. Here, 95% of the requests are reads, and 5% are new votes, and we use a similar, skewed popularity distribution as the real lobste.rs site observes. We benchmark MariaDB; System Z, a commercial database that supports materialized views; MariaDB with a memcached look-aside cache; as well as "memcached-only", an unrealistic deployment where the application stores vote counts directly in memcached without any database interactions; and Noria with four-way sharding for parallel processing. All systems run entirely in-memory to avoid measuring the I/O layer performance, and we set the databases to avoid transactions and use the lowest isolation level. Figure 3 again shows that Noria performs well: while the database-based systems do not scale beyond 200,000 requests/second, Noria scales all the way to 14 million requests/second. The unrealistic memcached-only deployment, for comparison, scales to 8 million requests/second, but then saturates the cores of the server. Noria outperforms memcached because it uses a more efficient, lock-free data structure to serve reads, but this is not fundamental (memcached could use the same data structure). Noria's high performance comes because reads directly hit the materialized view, and because it processes writes efficiently through the sharded, partially-stateful, incremental data-flow.

When to use Noria

Noria is designed for web applications that are read-heavy, and which can tolerate eventual consistency. The ubiquity of caches in modern web application stacks suggest that eventual consistency is often sufficient, although we are also working on ideas for high-performance transactions on Noria. Noria also obviates the need for transactions in some cases. The lobste.rs developers, for example, only use transactions to ensure that a story's vote count is incremented atomically with the vote being stored. Noria maintains the vote count internally in the data-flow, so this transaction is no longer necessary.

Noria primarily targets applications whose working set fits in memory when sharded and partitioned across many servers. Old records in base tables are only on disk, but applications that regularly need to access the full data set (e.g., fulltext search) would need additional support to work well in Noria.

How to use Noria

Noria is open-source and available at https://pdos.csail.mit.edu/noria. In many cases, you should only need to start up the Noria MySQL adapter, point your application at it instead of MySQL, and turn off all your caches. Noria will take care of the rest. The Noria prototype is research code and still in development, but we enjoy hearing how it works for other people!

References

[1]: 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.

[2]: Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. "Noria: dynamic, partially-stateful data-flow for high-performance web applications". In: *Proceedings of 13th USENIX conference on Operating Systems Design and Implementation (OSDI)*, October 2018, pages 213--231.

[3]: 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)*, April 2013, pages 385--398.

[4]: Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. "Dryad: distributed data-parallel programs from sequential building blocks". *SIGOPS Operating Systems Review* 41, 3 (March 2007), pages 59--72.

[5]: Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. "Naiad: a timely dataflow system". In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. November 2013, pages 439--455.

[6]: Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, and others. "TensorFlow: a system for large-scale machine learning". In: *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation (OSDI)*. November 2016, pages 265--283.

[7]: RocksDB, https://rocksdb.org/