

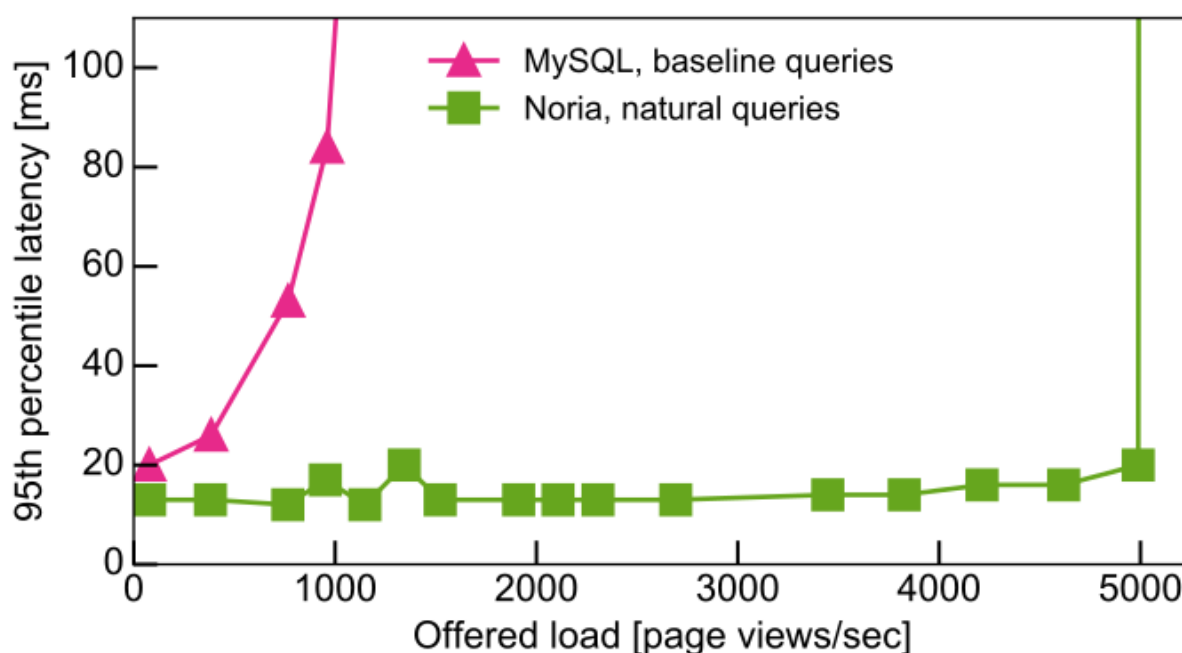
---

## Noria: data-flow for high-performance web applications

 Azure Pipelines partially succeeded  docs failing  crates.io v0.6.0  docs failing  Azure Pipelines partially succeeded

Noria is a new streaming data-flow system designed to act as a fast storage backend for read-heavy web applications based on Jon Gjengset's Phd Thesis, as well as this paper from OSDI'18. It acts like a database, but precomputes and caches relational query results so that reads are blazingly fast. Noria automatically keeps cached results up-to-date as the underlying data, stored in persistent *base tables*, change. Noria uses partially-stateful data-flow to reduce memory overhead, and supports dynamic, runtime data-flow and query change.

Noria comes with a MySQL adapter that implements the binary MySQL protocol. This lets any application that currently talks to MySQL or MariaDB switch to Noria with minimal effort. For example, running a Lobsters-like workload that issues the equivalent SQL queries to the real Lobsters website, Noria improves throughput supported by 5x:



At a high level, Noria takes a set of parameterized SQL queries (think prepared statements), and produces a data-flow program that maintains materialized views for the output of those queries. Reads now become fast lookups directly into these materialized views, as if the value had been directly cached in memcached. The views are then kept up-to-date incrementally through the data-flow, which yields high write throughput.

---

## Running Noria

Like most databases, Noria follows a server-client model where many clients connect to a (potentially distributed) server. The server in this case is the `noria-server` binary, and must be started before clients can connect. Noria also uses Apache ZooKeeper to announce the location of its servers, so ZooKeeper must be running.

You (currently) need nightly Rust to build `noria-server`. This will be arranged for automatically if you're using `rustup.rs`. To build `noria-server`, run

```
1 $ cargo build --release --bin noria-server
```

You may need to install some dependencies for the above to work:

- clang
- libclang-dev
- libssl-dev
- liblz4-dev
- build-essential

To start a long-running `noria-server` instance, ensure that ZooKeeper is running, and then run:

```
1 $ cargo r --release --bin noria-server -- --deployment myapp --no-reuse
  --address 172.16.0.19 --shards 0
```

`myapp` here is a *deployment*. Many `noria-server` instances can operate in a single deployment at the same time, and will share the workload between them. Workers in the same deployment automatically elect a leader and discovery each other via ZooKeeper.

## Interacting with Noria

There are two primary ways to interact with Noria: through the Rust bindings or through the MySQL adapter. They both automatically locate the running worker through ZooKeeper (use `-z` if ZooKeeper is not running on `localhost:2181`).

### Rust bindings

The `noria` crate provides native Rust bindings to interact with `noria-server`. See the `noria` documentation for detailed instructions on how to use the library. You can also take a look at the example Noria program using Noria's client API. You can also see a self-contained version that embeds `noria-server` (and doesn't require ZooKeeper) in this example.

---

## MySQL adapter

We have built a MySQL adapter for Noria that accepts standard MySQL queries and speaks the MySQL protocol to make it easy to try Noria out for existing applications. Once the adapter is running (see its [README](#)), you should be able to point your application at `localhost:3306` to send queries to Noria. If your application crashes, this is a bug, and we would appreciate it if you open an issue. You may also want to try to disable automatic re-use (with `--no-reuse`) or sharding (with `--shards 0`) in case those are misbehaving.

## CLI and Web UI

You can manually inspect the data stored in Noria using any MySQL client (e.g., the `mysql` CLI), or use Noria's own web interface.

## Noria development

Noria is a large piece of software that spans many sub-crates and external tools (see links in the text above). Each sub-crate is responsible for a component of Noria's architecture, such as external API (`noria`), mapping SQL to data-flow (`server/mir`), and executing data-flow operators (`server/dataflow`). The code in `server/src/` is the glue that ties these pieces together by establishing materializations, scheduling data-flow work, orchestrating Noria program changes, handling failovers, etc.

`server/src/lib.rs` has a pretty extensive comment at the top of it that goes through how the Noria internals fit together at an implementation level. While it occasionally lags behind, especially following larger changes, it should serve to get you familiarized with the basic building blocks relatively quickly.

The sub-crates each serve a distinct role:

- `noria/`: everything that an external program communicating with Noria needs. This includes types used in RPCs as arguments/return types, as well as code for discovering Noria workers through ZooKeeper, establishing a connection to Noria through ZooKeeper, and invoking the various RPC exposed by the Noria controller (`server/src/controller.rs`). The `noria` sub-crate also contains a number of internal data-structures that must be shared between the client and the server like `DataType` (Noria's "value" type). These are annotated with `#[doc(hidden)]`, and should be easy to spot in `noria/src/lib.rs`.
- `applications/`: a collection of various Noria benchmarks. The most frequently used one is `vote`, which runs the vote benchmark from §8.2 of the OSDI paper. You can run it in a bunch of

---

different ways (`--help` should be useful), and with many different backends. The `localsoup` backend is the one that's easiest to get up and running with.

- `server/src/`: the Noria server, including high-level components such as RPC handling, domain scheduling, connection management, and all the controller operations (listening for heartbeats, handling failed workers, etc.). It contains two notable sub-crates:
  - `dataflow/`: the code that implements the internals of the data-flow graph. This includes implementations of the different operators (`ops/`), “special” operators like leaf views and sharders (`node/special/`), implementations of view storage (`state/`), and the code that coordinates execution of control, data, and backfill messages within a thread domain (`domain/`).
  - `mir/`: the code that implements Noria’s SQL-to-dataflow mapping. This includes resolving columns and keys, creating dataflow operators, and detecting reuse opportunities, and triggering migrations to make changes after new SQL queries have been added. @ms705 is the primary author of this particular subcrate, and it builds largely upon `nom-sql`.
  - `common/`: data-structures that are shared between the various `server` sub-crates.

To run the test suite, use:

```
1 $ cargo test
```

Build and open the documentation with:

```
1 $ cargo doc --open
```

Once `noria-server` is running, its API is available on port 6033 at the specified listen address.

Alternatively, you can discover Noria’s REST API listen address and port through ZooKeeper via this command:

```
1 $ cargo run --bin noria-zk -- \  
2   --show --deployment myapp \  
3   | grep external | cut -d' ' -f4
```

A basic graphical UI runs at `http://IP:PORT/graph.html` and shows the running data-flow graph. You can also deploy Noria’s more advanced web UI that serves the REST API endpoints in a human-digestible form and includes the graph visualization.

## License

Licensed under either of

- 
- Apache License, Version 2.0 (LICENSE-APACHE or <http://www.apache.org/licenses/LICENSE-2.0>)
  - MIT license (LICENSE-MIT or <http://opensource.org/licenses/MIT>)

at your option.

## **Contribution**

Unless you explicitly state otherwise, any contribution intentionally submitted for inclusion in the work by you, as defined in the Apache-2.0 license, shall be dual licensed as above, without any additional terms or conditions.