Baseline architecture

- Starting point: The Node.js Roary implementation of Exercise 4.
  For simplicity, the service worker is ignored in this exercise. Also a slight modification was made (different and more performat sqlite3 library) but otherwise is the same as the Exercise 4 submission.

- Subsequently called the „old“ Node.js implementation. Can be found in the **node-roary-old.zip** file.

- Benchmarking and evaluation was done using the setup demonstrated in the **benchmark.zip** file. In summary: *wrk2* was used in combination with Lua scripts (called *benchmarks*) to generate various requsts and observe latency results.

Main evaluation results:

- Slow sqlite library was used and therefore replaced before implementing the new implementation to have better comparability (see: https://github.com/JoshuaWise/better-sqlite3/blob/master/docs/benchmark.md)

- Inefficient communication model: New messages/roars retrieved via polling

- A single Node.js process can sustain only a limited load due to the non-concurrent single-threaded nature.

For more details and some charts, see the Comparison section.

Considered and implemented concepts

- **Concept: HTTP2 and WebSockets, Implemented: No**
  This would arguably one of the most important concepts to implement for the Roary use case.

  HTTP2 can be used to server push the initial messages and also reduces or eliminates a lot of HTTP/1.1 problems.
  The WebSocket protocol allows for a full-duplex communication which doesn't follow the request-response cycle imposed by HTTP. This is very useful for a chat application like Roary because it allows to get rid of polling: When the server receives a new post, it simply broadcasts it to all other connected users. This completely eliminates polling and therefore reduces the load on the server a lot of the load on the server.

  There are multiple reason why this was not implemented (although it is possible): My current benchmarking framework is not setup for this. *wrk2* only supports HTTP/1.1 and no WebSockets. Also: In this case a completely different benchmarking model has to be used since communication is initiated by the server and there's basically no API to query Roars.

  Another reason is statelessness. Such a communication model is generally stateful and I did implement a stateless app. It is possible to do this stateless and I did consider implementing ithis concept via *socket.io*, benchmarking via *k6* and achieving statelessness via *socket.io-redis* but didn't have enough time for this (as it would require to completely change my setup an basically starting anew).

- **Concept: REST, Implemented: Yes**
  The REST concept is implemented. API paths for accessing Roars are resource-oriented and CRUD operations are implemented via HTTP methods.

  The main reason for this concept is that it supports a range of other important concepts which are more important such as statelessness, cacheability, etc.

- **Concept: statelessness, Implemented: Yes**
  The new Roary implementation via Node.js is completely statelss. The only piece of data which would make the application stateful is session data. This was solved using the *connect-redis* library.

The reason for statelessness is that is allows for much easer scaling, e.g. running multiple instances of the app to support concurrency.

- **Concept: Caching, Implemented: sort of**
  The Roary application doesn't actually have much to cache. Write operations are obviously not cacheable so the only read operation – fetching Roars – is the only cacheable operation. However, the data returned can change since new Roars might have been added so some state would have to be maintained in order to indicate a „dirty" cache which goes against statelessness. Therefore this operation is not cached.

  However, what is cached is the static *index.html* file. The nginx web server which serves this file is configured to return „302 Not Modified" responses so the browser has to use its cached version of the file.

- **Concept: Database Replication, Implemented: No**
  Database replication would actually be a very sensible thing to do since the benchmarks will show that write operations are slower and can't be as easily scaled up to avoid data inconsistencies. An approach with multiple leader databases which are synchronzied regulary would be a good choice here.

  I did consider this using *litesync*, a multi-language replication framework for SQLite, however it has some constraints which don't fit my database schema. Particulary, it doesn't support auto-increment keys which I use for primary keys. This would generally not be a problem (just use UUID or something like that) but the advantage of this approach is that I can use the primary keys as a sequence number of posts to poll only new posts. Without auto-increment the application would have to maintain an ID counter itself and synchronzie it across instances which is not ideal.

  Also, this might be easier using a full-blown DMBS such as PostgreSQL.

- **Concept: Database Sharding, Implemented: No**
  Sharding would allow multiple databases to exist and therefore avoid problems with too many accesses to a single database. However, this is not easily doable for the Roary app, since there is no easy way to create isolated key spaces, primarily because there is a relation across partitions (the like state of a Roar).

- **Concept: Application replication, Implemented: Yes**
  Since the Roary application is stateless, it is easy to create multiple instances for it which solves on of the main problems of the old implementation which is

a load too high for a single Node.js process and therefore implicitly adds concurrency since multiple instances can read data simultaneously. Of course this has no real use for writing data since some locking is still required. Database replication would be required for that.

New architecture

- Application state is moved out of process memory to a *Redis* store. In particular, session data is stored in a *Redis* store. No other application state exists.
- Request routes changed to fit the REST model
- *pm2* (https://pm2.keymetrics.io/) is used as process manager to manage instances of the Node.js application
- *nginx* (https://www.nginx.com/) is used as a load balancer and to serve static files. It acts as a reverse proxy to the individual Node.js instances.
- Subsequently called the „new“ Node.js implementation. Can be found in the **node-roary-new.zip** file.

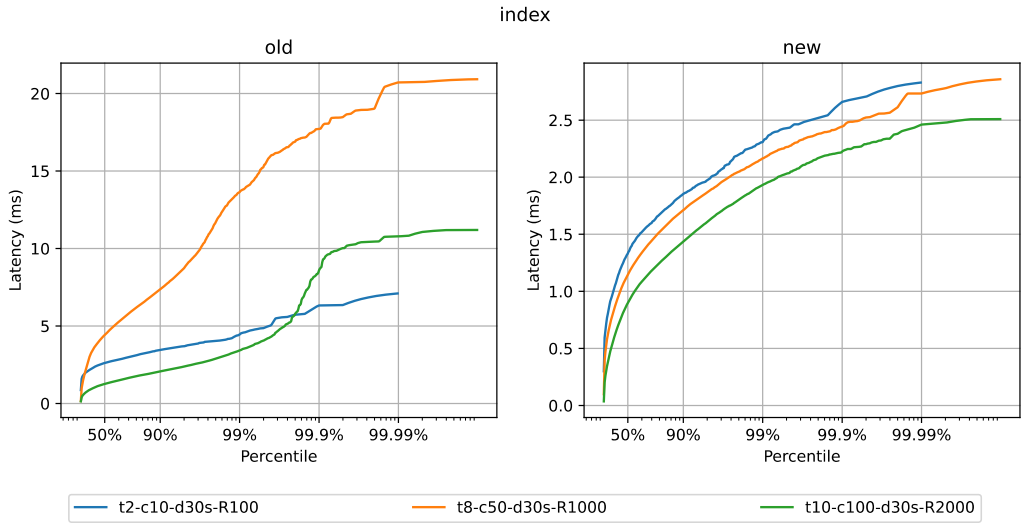
Main evaluation results:

- The multi-instance approach (and therefore multiple database connections) has significantly speed up read requests.
- Moderate improvement for write requests since some locking is required no matter the amount of readers.
- Caching and using nginx has improved loading the frontend *index.html* page.

For more details and some charts, see the Comparison section.

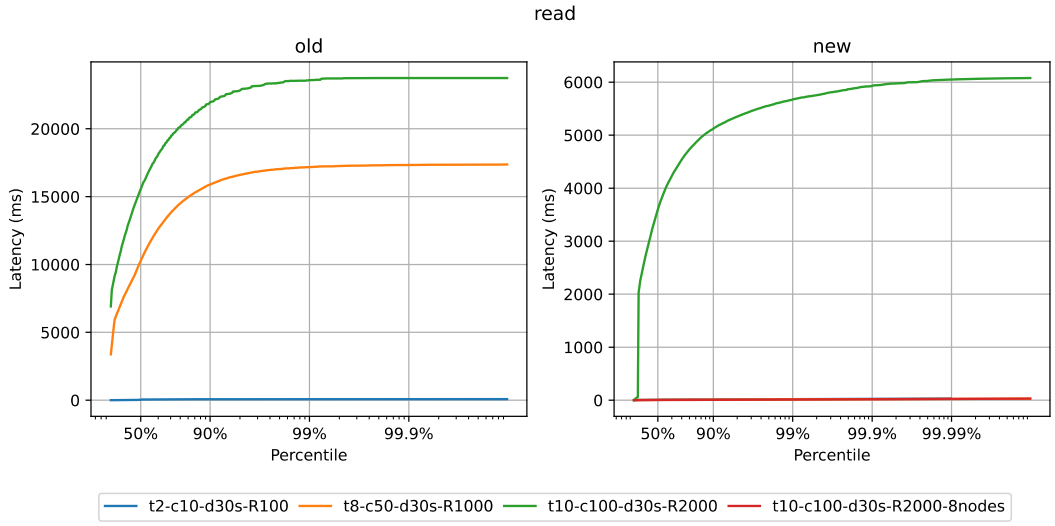
Benchmark comparison and results discussion

1. **index** benchmark (loading the frontend *index.html*)



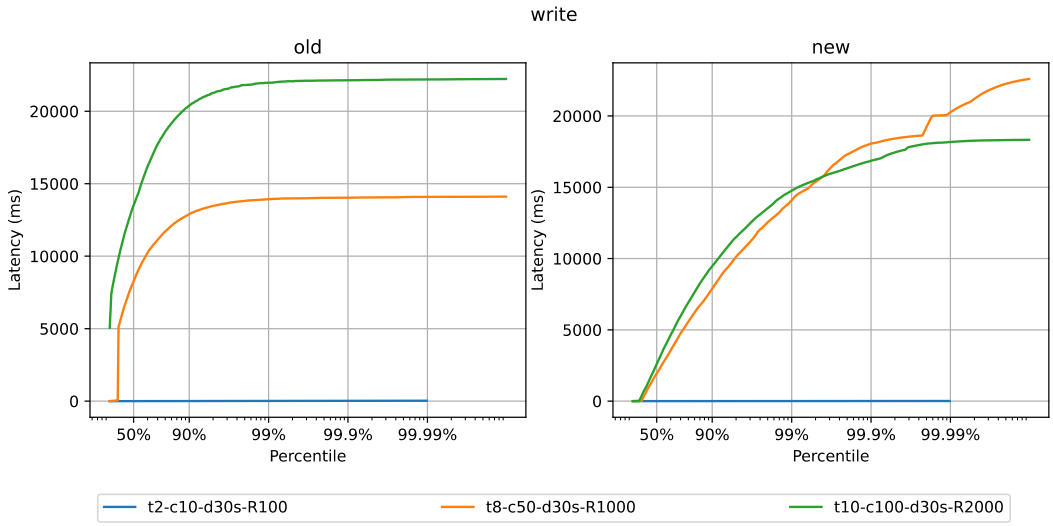
As can be seen in the graphs, the *nginx* response is consistenly faster although both times the latencies are very low. The new approach is also faster due to caching which results in 302 responses with no HTTP body to transmit.

2. **Read** benchmark (polling data from the server)



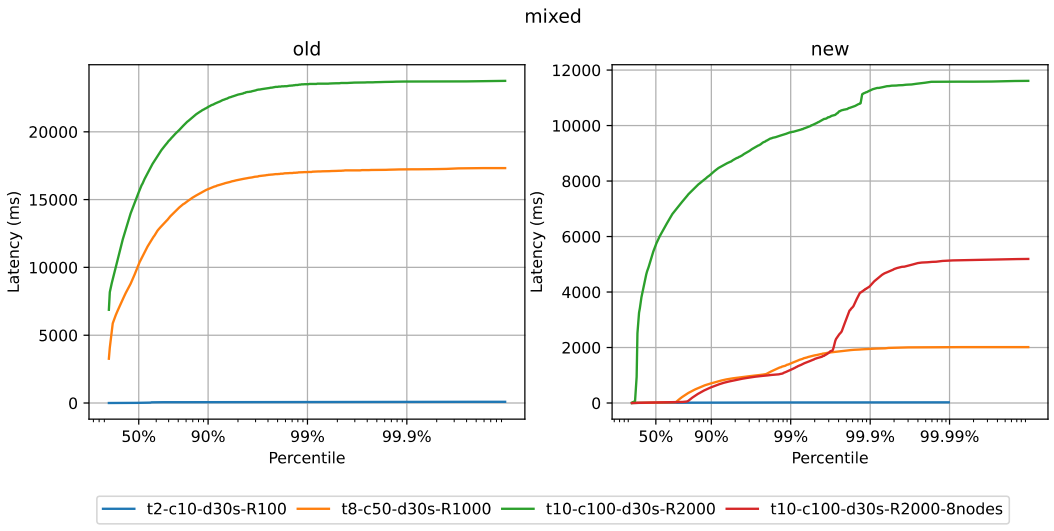
As can be seen in the left graph, the old implementation hits a limit somewhere before 1000R/s and the latencies become intolerable. The new implementation has no problems with 1000R/s but even here 2000R/s is too much. However, as seen by the red line, increasing the number of instances from 4 to 8 solves that problem and 2000R/s are handled without issues.

3. **Write** benchmark (sending a new Roar to the server)



As can be seen in the graphs, there is not much improvement for write requests since there can be only one writer on the database file no matter how many instances we have. This issue requires Database replication to be fixed. However, multiple instance do still have a positive effect: The curve is smoother because the load of all those write requests is distributed across multiple instances.

4. **Mixed** benchmark (90% reading and 10% writing)



As expected, we can see a mixture of the read and write results. The new implementation is consistently faster. For the default of 4 instances there is a latency hit just as with the read benchmark. When increased to 8 instances (or for less than 1000R/s), we can see the behaviour outlined in the write benchmark but to a lesser degree since we only have 10% writes). The performance is still much better than the old implementation but there needs to be another approach for writes to handle even more requests/sec.