Scalable News Feeds – MongoDB vs Stream

Stream is a special purpose database & API built for news feeds and activity streams. Today we'll compare it to MongoDB, a general purpose object-oriented database. Traditionally, companies have leveraged Redis or Cassandra for building scalable news feeds. Twitter's feed, for instance, is based on Redis. Instagram started with Redis, switched to Cassandra and recently wrote their own in-house storage layer.

Over the past years, MongoDB has skyrocketed in terms of popularity. Because of this, some teams have chosen to build their feeds on top of MongoDB.

This blog post will analyze the features and performance of a news feed built on top of MongoDB and compare to the same functionality built on Stream. The MongoDB code and benchmark is open source and available on Github (mongodb-activity-feed) We'll compare the MongoDB based technology to Stream.

1. News Feeds with Stream

Stream Test Setup

For the purpose of this benchmark, we'll use the relatively small $899/mo. plan offered by Stream. Note that there are larger plans available tailored for high volume customers. For example, several of Stream's largest customers have more than 100 million users on our custom Enterprise plans.

Benefits of Stream

Scalable and engaging news feeds are extremely hard to build. The founders of Stream went through that experience at their previous startup. Don't take our word for it though, here's what Tim from Dubsmash has to say about it:

> *Building feed technology is quite a challenge. We made the mistake of trying to build it ourselves… and we were just stuck. From a feature perspective, we couldn't build something as robust as what Stream can provide. I'm simply amazed at how certain features, such as ranking, work. –Tim Specht, Founder/CTO at Dubmash*

Dubsmash is not alone in that experience. Twitter's fail whale, Tumblr's technical debt, Facebook's slow down, the struggle at Hyves, Friendster, the list goes on...

The main benefits of using Stream for your news feed are:

- **Time to Market**: Get your feeds up and running in hours instead of months.
- **Focus:** Focus on product development instead of spending time building, maintaining, hosting and improving feed technology.
- **User Engagement**: Ranking, Aggregation, Real-time, and Personalization enable you to improve user engagement and retention within your application.
- **No Firefighting**: An experienced team makes sure that your feeds stay up and running. Historically our uptime has been above 99.99%.
- **Growth**: Build your business without worrying about the scalability of your feed infrastructure.
- **Performance**: Our API typically returns data in 12ms, allowing your feeds to be lightning fast.

Luke Chesser from Unsplash does a great job of explaining why it's so important to keep your team focused in his blog post: "Scaling Unsplash with a small team".

2. News Feed with MongoDB & CRDTs

MongoDB is a popular distributed NoSQL database optimized for ease of use. MongoDB's focus on ease of use comes with some costs. A great technical discussion about some of the tradeoffs MongoDB makes is detailed in Jepsen for version 3.4.0.

The open source MongoDB activity feed repo uses:

- Unordered bulk writes
- Conflict-free replicated data types
- Bull + Redis for Queuing

MongoDB Test Setup

For the benchmark we're using Node v10.8.0, MongoDB v4.0 and the following infrastructure which will be hosted on AWS:

1. MongoDB cluster running on MongoDB Atlas (M50, standard CPU, 2640 max IOPS, Mongo 4.0 = $2203/mo.+ other charges for backups, data transfer etc).
2. Redis as a task broker, highly available (cache.m4.large * 2 = $228.40/mo.+ other charges for backups, data transfer etc.)
3. Socket.IO for real-time websockets (c5.large * 2 = $124.44/mo.)
4. 3 servers to run task processing with Bull (c5.large * 3 = $186.66/mo.)
5. 2 API server to access the feed (c5.large * 2 = $124.44/mo.)

The above prices are based on the AWS rates in US-east per 8/18. The total monthly cost of the MongoDB setup: $2,865/mo. + some extra fees for backups, data transfer, load balancers etc. In our experience with running MongoDB in production, these extra fees increase the price by roughly 15% so the total price will be roughly $3,300/mo..

In addition to the hosting cost, there is of course also the hidden cost of setting up the infrastructure, monitoring, maintaining it and waking up when it fails. It's especially important to have monitoring in place for the asynchronous job queuing infrastructure. You'll also want to add some sort of throttling to prevent a spike in writes from breaking reads on your MongoDB cluster.

The Data Model

The MongoDB activity feed project uses 5 different schemas:

- **Activity**: The actual data of the activity
- **Activity Feed**: An operation log. This stores that an activity should be added to a certain feed. (index on feed, time and operationTime)
- **Feed Group**: A group of feeds, for example: user, notification, timeline etc. (unique on name)
- **Feed**: A specific feed, ie user:scott, notification:josh, timeline:tom, etc. (unique on group and feedID)
- **Follow**: A follow relationship between 2 feeds. (unique on source, target, index on target desc)

You might be wondering why we're using an operation log instead of just storing the activities. If you simply add and remove activities you run into problems with locking. IE. a user likes something, unlikes it and likes it again. If you're just adding and removing activities you end up with either:

- Incorrect state (something shows as liked when you unliked it)
- Locks (which reduce the write capacity of your system)

The best way to solve this is using CRDTs. Instead of adding or removing the activity, you an operation. The operation states that an activity should be added or removed from a feed together with the time of the operation. This approach prevents race conditions and doesn't require locking.

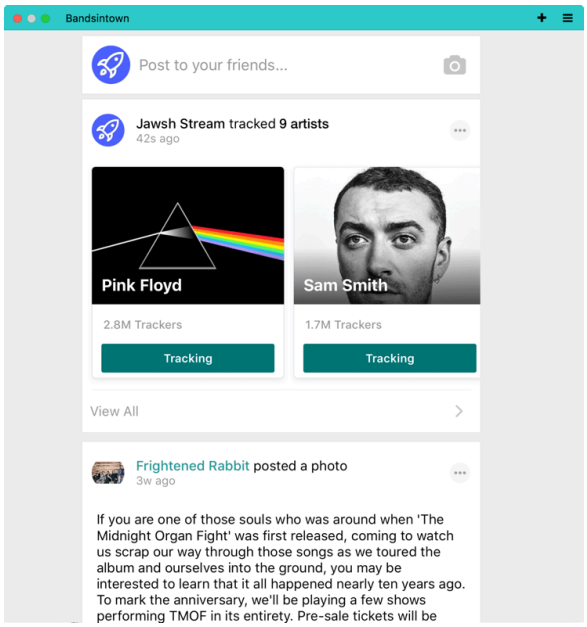
3. The Benchmark

Now that we've described both test setups it's time for a benchmark. Now let me start with a big disclaimer. A benchmark will never be 100% accurate. To have a fully accurate overview you should benchmark for your exact use case on your own hardware. That being said the numbers

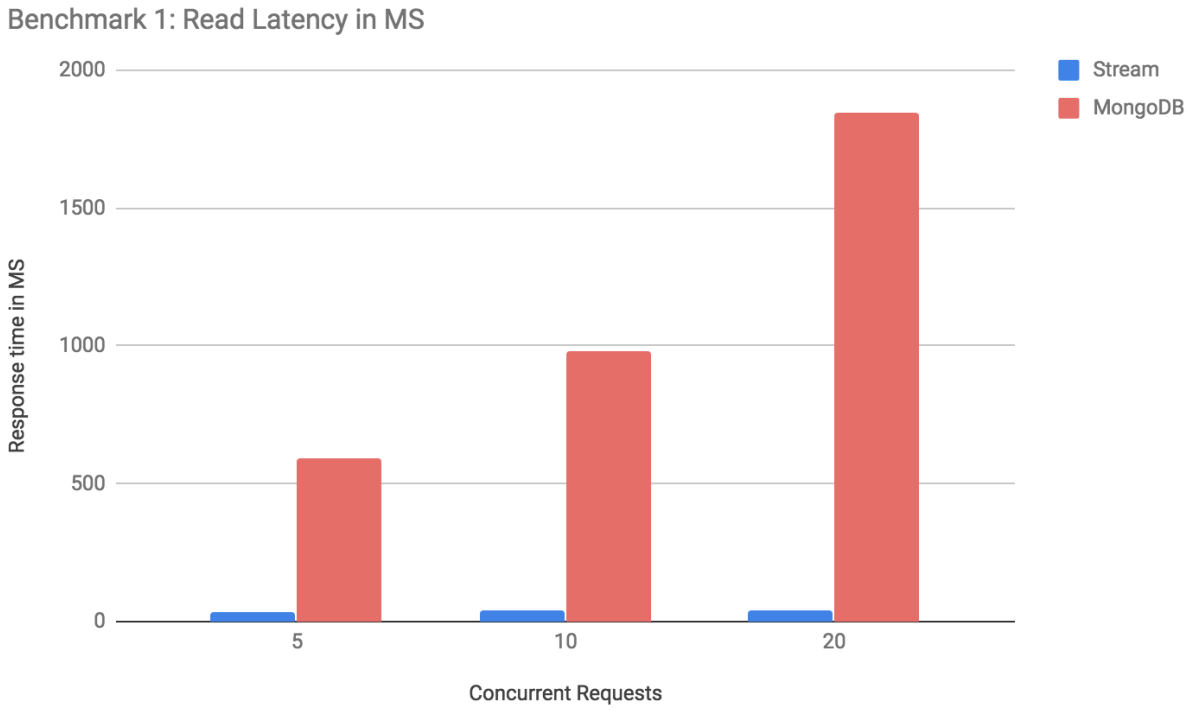
below will give you a decent approximation. The benchmark code is open source and available on GitHub.

Benchmark 1 - Read Latency

Our first benchmark measures read latency on an aggregated feed. Aggregation is a common feature for news feeds that helps reduce noise. Here's an example of how Bandsintown, a large music app, uses aggregated feeds:



To prepare the benchmark we insert 3,000 activities into a feed. Next, we'll see how fast it is to read an aggregated feed at 5, 10, and 20 *concurrent* requests. The benchmark results for Stream and MongoDB are shown below:

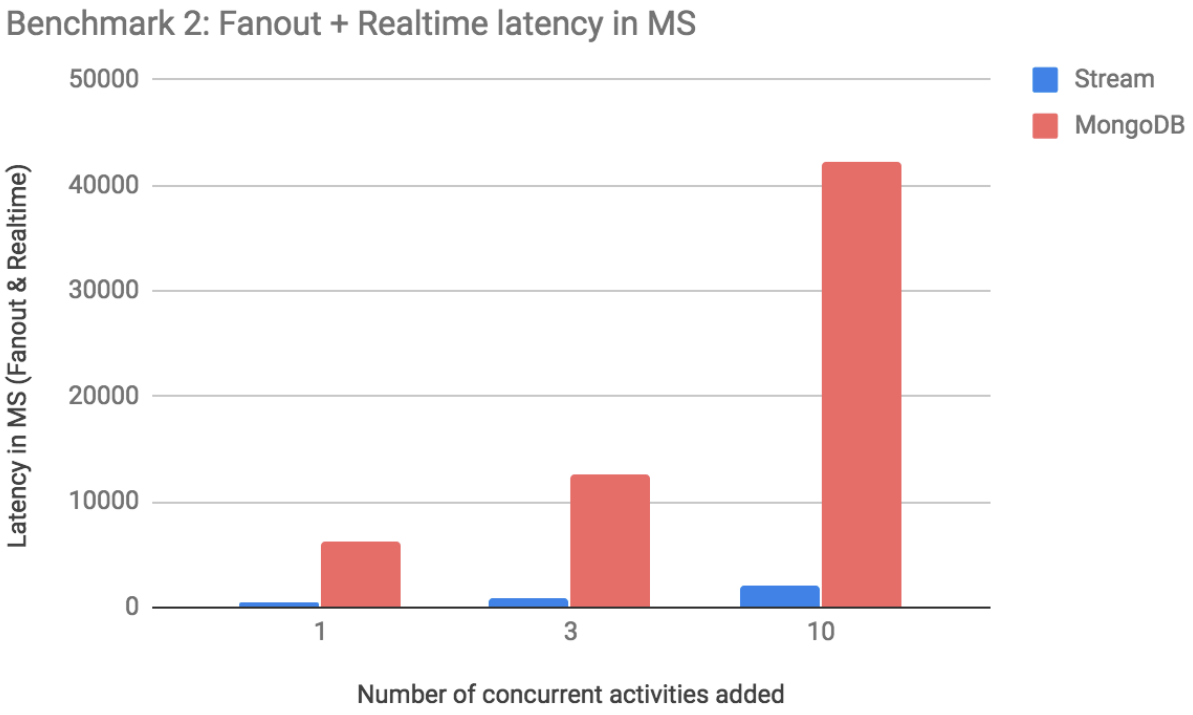


As you can see Stream is between 16 and 48 times faster for this specific benchmark.

Benchmark 2 - Fanout & Real-Time Latency

This second benchmark measures the fanout and real-time latency. The fanout is the process that writes an activity from one feed to all the feeds that follow that feed. The real-time latency is the part that notifies your browser (or mobile phone) that something changed in the feed.

We'll prepare this benchmark by having 20,000 users follow 1 popular feed. Then we'll write an activity to this feed (which now has 20k followers). The measurements are taken when publishing 1, 3 and 10 activities:



As you can see it takes Stream 1,978 ms to handle fanout & real-time for the 10 activities being distributed to 20k followers.

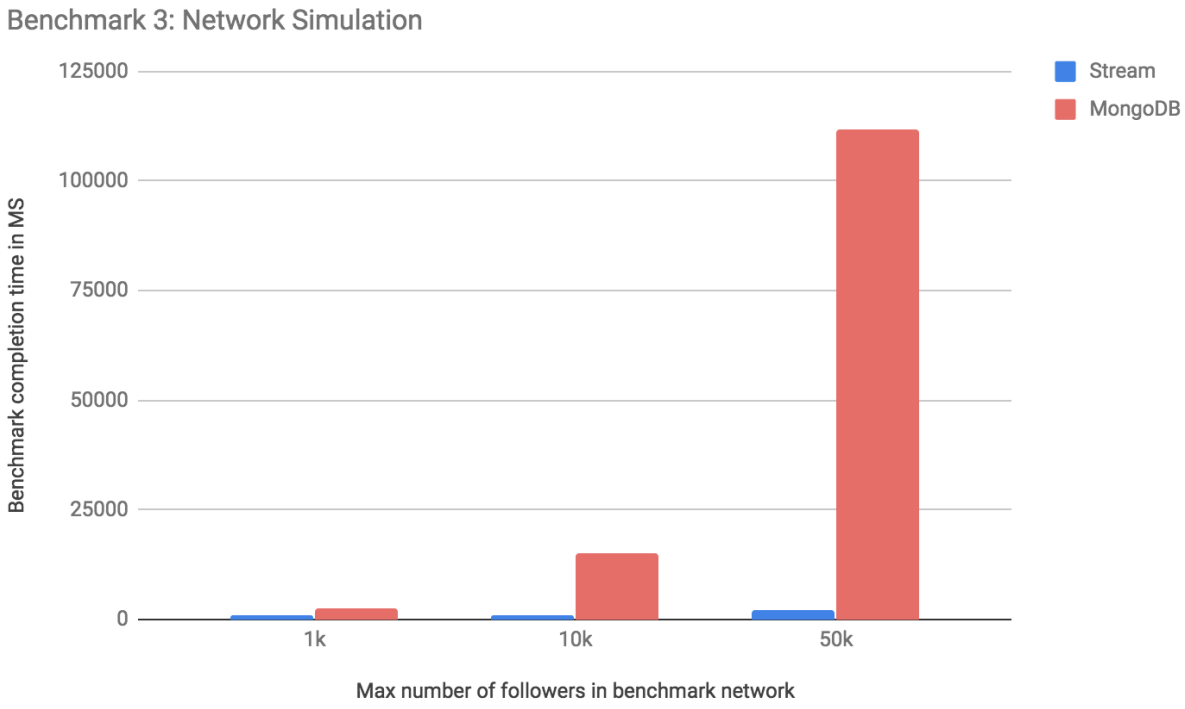
MongoDB, on the other hand, takes 42,319 ms to handle the fanout and real-time. Stream is roughly 21 times faster in this benchmark.

Benchmark 3 - Network Simulation

For this last benchmark, we'll get a bit closer to simulating a real social network. We're creating a social graph which looks like this:

- Account 0 to 50 are followed by N / 10^4 people (these people are the least popular)
- Account 50 to 70 are followed by N / 10^3 people
- Account 70 to 85 are followed by N / 10^2 people
- Account 85 to 95 are followed by N / 10 people
- Account 95 to 100 are followed by N people (these people are very popular)

In this benchmark, we'll add 1 activity to each of the 100 accounts and evaluate how much time it takes for all updates to propagate. We'll start out with N=1,000. Next, we'll increase N to 10k and 50k. This benchmark is a rough approximation but it takes into account that some feeds are very popular and most others are not.



Note how the MongoDB infrastructure starts struggling once the most popular feeds in the social graph have 50k followers. Given the same workload, Stream is capable of being 50 times faster than MongoDB.

Conclusion

As we've shown in this article it's possible to build a functional news feed with MongoDB. The code is available on GitHub.

While the MongoDB codebase is functional, it's not performant or cost-effective - it's simply not designed to be optimized for this specific use case. We ran these benchmarks with a MongoDB infrastructure that costs $3,300 a month and compared it against Stream's $899/mo. plan. The price for Stream is all-in. The MongoDB figure, however, does not include the cost of monitoring, maintenance, and firefighting for this infrastructure.

After running a benchmark for read, write and capacity we found the following results:

- **Benchmark 1 Read Performance**: Stream is **16-48** times faster
- **Benchmark 2 Fanout + Real-time latency**: Stream is **21** times faster
- **Benchmark 3 Social Network Simulation**: Stream is **50** times faster

Using a specialized database like Stream results in a large improvement to performance and scalability. Perhaps, more importantly, it helps you get to launch your app faster, focus on the product experience and gives you the tools you need to optimize user engagement.

To be clear this benchmark is not saying anything negative about MongoDB. It just shows the large difference between a special purpose solution compared to a general purpose tool such as MongoDB.

If you're curious about why Stream's news feed technology is so fast and efficient, you'll enjoy this blog post on Stackshare: How Stream uses RocksDB, Raft and Go to power the feeds for over 300 million users.

If you're curious about trying out Stream's API try out the API in your browser.