13 scaling-related lessons I learned as a software engineer

2 Comments

Adam Pahlevi Baihaqi (https://www.techinasia.com/profile/adam-baihaqi)
1:58 AM at Dec 9, 2016 | 5 min read

213

(https://www.facebook.com/dialog/share?app_id=206930646126140&display=popup&href=https://www.techinasia.com/talk/13-scaling-lessons-learned-software-engineer&title=13%20scaling-related%20lessons%20I%20learned%20as%20a%20software%20engineer)

(https://twitter.com/intent/tweet?text=13%20scaling-related%20lessons%20I%20learned%20as%20a%20software%20engineer&url=https://www.techinasia.com/talk/13-scaling-lessons-learned-software-engineer)

(https://www.linkedin.com/shareArticle?mini=true&url=https://www.techinasia.com/talk/13-scaling-lessons-learned-software-engineer&title=13%20scaling-related%20lessons%20I%20learned%20as%20a%20software%20engi

%20software%20engineer&url=https://www.techinasia.c

A code artisan. Some of my writing has made into columns such as infoQ. I do enjoy and have been giving talks in Jakarta, KL, and Madrid the farthest. Kindle is by my side all the time. You can create a community post just like Adam Pahlevi here (/create/article).

Photo credit: artisticco / 123RF Stock Photo (http://www.123rf.com/profile_artisticco).

I work at Midtrans (http://midtrans.com), a payment gateway in Indonesia that is growing tremendously each month. While the growth is good for the business, it is a challenge for the engineering squad. As a result, scaling has become a problem for us.

We had our hardest hits in September and October which resulted in our outage. Since then, we've been gearing up to ensure the same scaling problems never happen again.

What did I learn?

(https:/

Treat it as invaluable, lifelong learning

The word stress is an understatement to describe the experience at the time. Our system, the same one used by other startups and companies, was down. A lot of us even had to stay at the office until the evening of the next day.

In such a calamity we were reminded, however, that it was just an experience. It won't happen every day — God forbid. Cool yourself, there is no better way to address the problem than with clear, unhurried thinking.

Estimate your system's points of failure

Problems are guaranteed to occur, but it is better for the engineering team to estimate when the next one will be. I believe it is possible to estimate your own system's durability just before an alarm goes off and even calculate the best time to address the issue.

Be prepared

Write events to the log, monitor the system, do a periodical health check, and please be sure that when the problems occur, there are tools at hand to track what is happening. Having no tools and logs will make you clueless to the possible issue at hand. Adding more tools may not hurt in the long run.

An architecture-first problem

As a Rails (Ruby on Rails) coder, I used to believe that it was the language and tools which made it hard to scale, and not the architecture.

The fact is scalability is not about how we write our software, but it is more about how we architect our system to be able to handle requests at scale. Don't forget that some banks are still using COBOL and RPG, and they can handle it with style. Yes, language does play a part in deciding and designing the architecture, but at the end of the day, nothing plays a greater role to scalability than a sound architecture.

It is hard and time-consuming

Why yes, it is hard and time-consuming. The key is to start small and make it work as soon as possible.

The business team needs to know the difficulty so when they have a new feature request, the first thing they think of is whether the application is stable enough to accept more expectations, or if other courses should be taken to make it more robust.

Monolithic may not be the way

Monolithic can be very costly in everyday engineering and when debugging. Case in point, it takes too much time to find out the real cause of an error. This is because the scope is wider and more interconnected than it should be.

Don't share databases

We know sharing databases with multiple applications is a bad practice. But these days, moving fast is preferred over orchestrating applications to work seamlessly throughout some public interfaces.

Until recently, it was a real pain for us. Some applications that were writing to the database blocked the others that were going to read it, and vice versa. Needless to say, such an effect is not desirable. Sharing databases also does not cater to unique indexing needs that belong exclusively to an app or service.

Replicate the database

Our first survival maneuver was replicating the database. We made an effort to separate them by their role: read and write. The read (known as "slave") is made to keep in sync with the write (known as "master") through certain strategies. By doing so, no matter how terrible read load is, it won't affect the write load, and vice versa.

Different applications no longer needed to read/write from the same database. It bought us some time, but this was definitely not the real savior.

Fix the bad queries

Bad queries do exist. This is the second database reliability fix.

We assigned a whole sprint to find out the bad queries and then find a way to address and make them better. In fact, we managed to speed up some request executions from seven minutes to one minute.

Do database archival and rotation

Each month, our system generates huge amounts of data as transactions come in. By doing soft archival and rotation, we keep hot data for certain months on disk and offload other months and years of history into an archive, retrievable only when needed.

Therefore, search queries will skip archived data, keeping the system performing no matter how long we run. This cannot be applied to all services, however; it is very important to separate the database per service first. For example, it doesn't make sense to rotate the user's account data.

Partitioning and sharding

On top of archival and rotation, the partitioning (or splitting) of the database is also essential in load balancing the requests coming into the database.

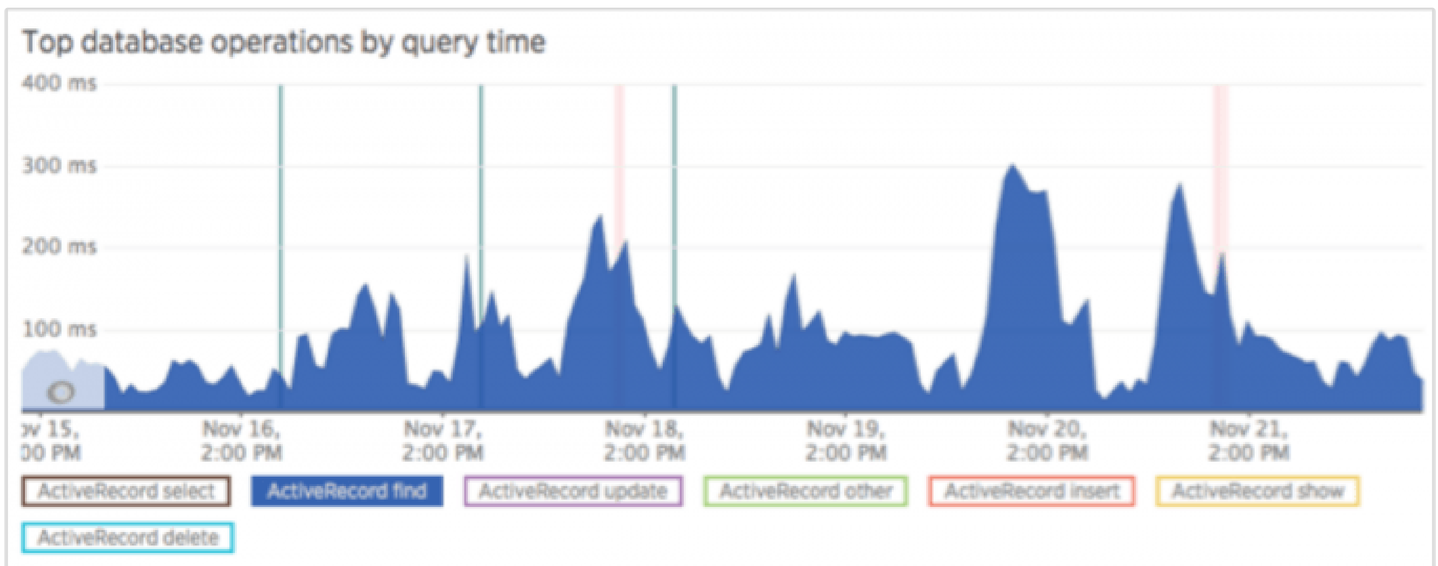
In short, partitioning or sharding allows access to the same table to be done in parallel rather than queue up. This is because the table is now split into various smaller chunks of its own.

Use databases less

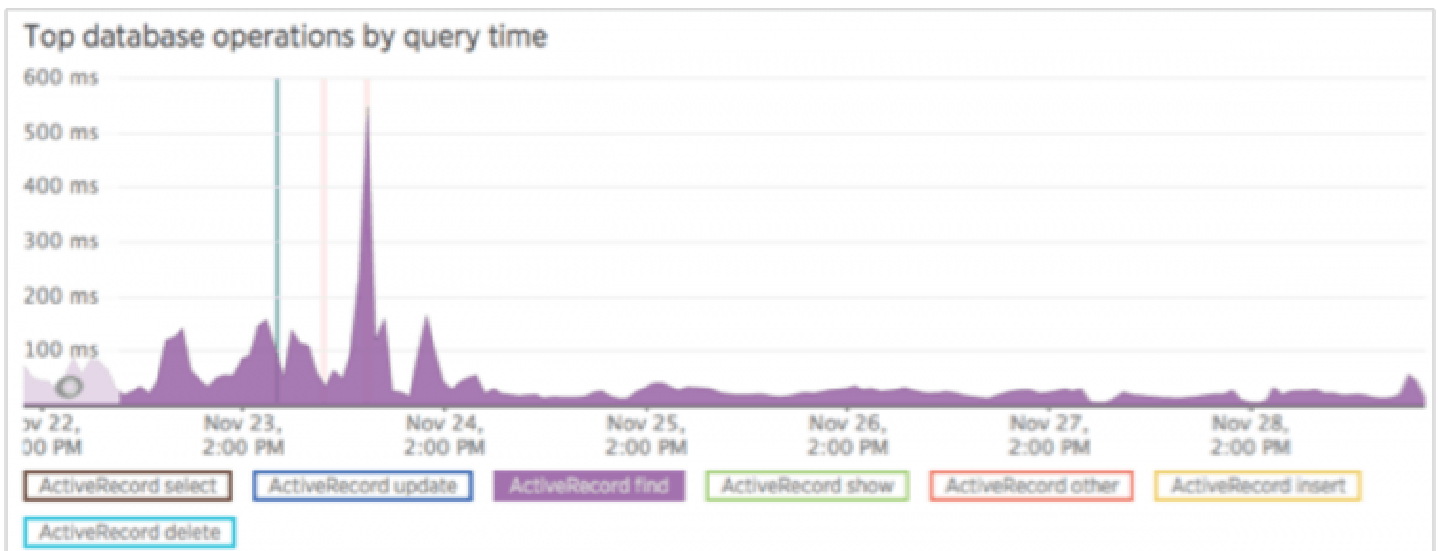
Saving 10 milliseconds makes a huge impact. That's why you need caching.

By doing careful caching, we see less access directed to the database. This makes the system much more efficient and keeps everyone happy.

? (https:/



Before caching.



After caching.

Precalculation will also help a lot, especially in our case where calculation-heavy pages are opened (eg. dashboard page). So rather than serving data off the database, pre-calculating it and saving the results into the user's local storage allowed the dashboard to render super fast.

Essentially, anything that can be done without involving the database should be considered. One example is moving from database to Redis for processing background jobs.

Don't finger-point

It is very easy to blame someone in the team when a problem does happen. Well, don't.

There is no advantage at all in doing so, other than boosting your own ego. Remember that all of this is for the sake of learning; everyone has the same share of responsibility.

Editing by Jaclyn Teng

(And yes, we're serious about ethics and transparency. More information here (/statement-of-ethics).)

? (https:/