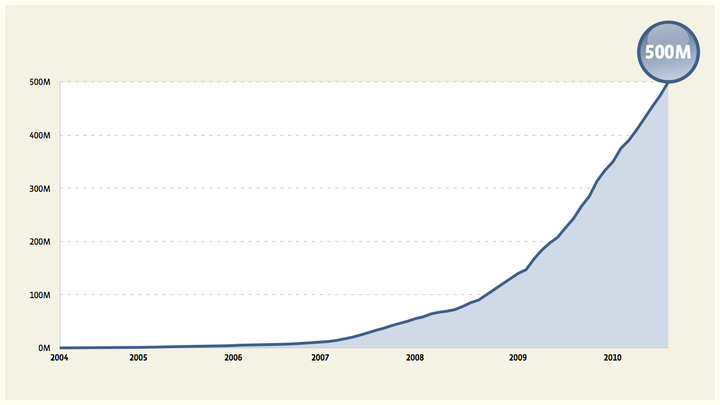
Today we hit an important milestone for Facebook - half a billion users. It's particularly exciting to those of us in engineering and operations who build the systems to handle this massive growth. I started at Facebook four years ago when we had seven million users (which seemed like a really big number at the time) and the technical challenges along the way have been just as crazy as you might imagine.  
A few of the big numbers we deal with:  
\* 500 million active users  
\* 100 billion hits per day  
\* 50 billion photos  
\* 2 trillion objects cached, with hundreds of millions of requests per second  
\* 130TB of logs every day  
Over the years we've written on this page about a number of the technical solutions we've used to handle these numbers. Today, I'd like to step back and talk about some of the general ways we think about scaling, and some of the principles we use to tackle scaling problems. Like Facebook itself, these principles involve both technology and people. In fact, only a couple of the principles below are entirely technical. At the end of the day it's people who build these systems and run them, and our best tools for scaling them are engineering and operations teams that can handle anything. The scaling statistic I'm most proud of is that we have over 1 million users per engineer, and this number has been steadily increasing.  


**Scale Horizontally**

This isn't at all novel but it's really important. If something is increasing exponentially, the only sensible way to deal with it is to get it spread across arbitrarily many machines. Remember, there are only three numbers in computer science: 0, 1, and n.  
For example, consider a database for users that can't handle the load. We could break it into two functions -- say, accounts and profiles -- and put them on different databases. This would get us through the day but it would be a lot of work and it only buys twice the capacity. We'd have to start the next one as soon as we were done, and the next one would be harder. Instead, we can take some extra time and write the code to handle the case where two users aren't on the same database. This is probably even more work than splitting the application code in half, but it continues to pay off for a very long time.  
Note that this doesn't improve efficiency, in fact it will probably make it worse. Efficiency is important but we think of it as a separate project from scaling. Scaling usually hurts efficiency, and efficiency projects rarely give you enough improvement to have a big effect on scaling.

**Move Fast**

If you look at the graph of our growth you'll notice that there's no point where it's flat. We never get to sit back and take a deep breath, pat ourselves on the back, and think about what we might do next time. Every week we have our biggest day ever.  
We of course have a pretty good idea of where the graph is headed, but at every level of scale there are surprises. The best way we have to deal with these surprises is to have engineering and operations teams that are flexible, and can deal with problems quickly. Moving fast also allows us to try a lot more things, to see which ones actually work out in practice. We've found that maintaining this flexibility is far more important than any particular technical decision.

**Change Incrementally**

The best way we've found to keep moving fast is to make lots of small changes and measure what happens with all of them. This doesn't mean we don't do big things, it just means that whenever possible we break them into as many distinct parts as possible. This is the opposite of many development philosophies that try to batch changes.  
Even when we have something that can't be broken apart in terms of functionality we try to roll it out incrementally. This could mean a few users or a few machines at a time, or even building a new system entirely in parallel with the old and slowly shifting traffic across while we measure the effects.  
The thing that's great about incremental changes is that you know as quickly as possible when something isn't going as you expected. This ends up, counter-intuitively, making it easier to keep the system stable.  
When there's a problem in production, by far the hardest part of correcting it is figuring out what went wrong. It’s a lot easier to figure out what went wrong when you've only made one change. In a more traditional model where you have weeks or months of changes going into effect together, figuring out which one caused the problem can be a nightmare.

**Measure Everything**

Making lots of small changes and watching what happens only works if you're actually able to watch what happens. At Facebook we collect an enormous amount of data - any particular server exports tens or hundreds of metrics that can be graphed. This isn't just system level things like CPU and memory, it’s also application level statistics to understand why things are happening.  
It's important that the statistics are from the real production machines that are having the problems, when they're having the problems - the really interesting things only show up in production. The stats also have to come from all machines, because a lot of important effects are hidden by averages and only show up in distributions, in particular 95th or 99th percentile.  
A number of the tools we built for collecting and analyzing this data have been released as open source, including [hive](https://www.facebook.com/note.php?note_id=409881258919) and [scribe](https://www.facebook.com/note.php?note_id=409881258919).

**Small, Independent Teams**

When I started at Facebook I was one of two people working on photos. That was pretty crazy, but now we're a "big" company. We have three people working on photos. Each of those three people knows photos inside and out, and can make decisions about it. So when something needs to change in photos they get it done quickly and correctly.  


**Control and Responsibility**

None of the previous principles work without operations and engineering teams that work together seamlessly, and can handle problems together as a team. This is much easier said than done, but we have one general principle that's enormously helpful:  
The person responsible for something must have control over it.  
This seems terribly obvious but it's often not the case. The classic example is someone deploying code that someone else wrote. The person deploying the code feels responsible for it but the person who wrote it actually has control over it. This puts the person deploying the code in a very tough spot - their only options are to refuse to release the code or risk being held responsible for a problem, so they have a strong incentive to say no. On the other side, if the person who wrote the code doesn't feel responsible for whether or not it works, it's pretty likely to not work.  
At Facebook we push code to the site every day, and the person who wrote that code is there to take responsibility for it. Seeing something you created be used by half a billion people is awe inspiring and humbling. Seeing it break is even more humbling. The best way we know of to get great software to these 500 million people is to have a person who understands the importance of what they're doing make a good decision about something they understand and control.

**Beyond 500 million**

We’re very proud to have built a site that 500 million people want to use, and a site that still works with 500 million people using it. But this is really just the start. We’re hoping that in the not too distant future there will be another 500 million people, and that these principles will help us overcome whatever new challenges arise on the way there.  
*Bobby, a director of engineering, has a much different appreciation of large numbers than he did four years ago.*