Flickr is both my favorite bird and the web's leading photo sharing site. Flickr has an amazing challenge, they must handle a vast sea of ever expanding new content, ever increasing legions of users, and a constant stream of new features, all while providing excellent performance. How do they do it?

Platform

- PHP
- MySQL
- Shards
- Memcached for a caching layer.
- Squid in reverse-proxy for html and images.
- Linux (RedHat)
- Smarty for templating
- Perl
- PEAR for XML and Email parsing
- ImageMagick, for image processing
- Java, for the node service
- Apache
- SystemImager for deployment
- Ganglia for distributed system monitoring
- Subcon stores essential system configuration files in a subversion repository for easy deployment to machines in a cluster.
- Cvsup for distributing and updating collections of files across a network.

The Stats

- More than 4 billion queries per day.
- ~35M photos in squid cache (total)
- ~2M photos in squid's RAM
- ~470M photos, 4 or 5 sizes of each
- 38k req/sec to memcached (12M objects)
- 2 PB raw storage (consumed about ~1.5TB on Sunday
- Over 400,000 photos being added every day

The Architecture

- A pretty picture of Flickr's architecture can be found on this slide . A simple depiction is:
- Pair of ServerIron's
- Squid Caches
- Net App's
- PHP App Servers
- Storage Manager
- Master-master shards
- Dual Tree Central Database
- Memcached Cluster

- Big Search Engine

-The Dual Tree structure is a custom set of changes to MySQL that allows scaling by incrementally adding masters without a ring architecture. This allows cheaper scaling because you need less hardware as compared to master-master setups which always requires double the hardware. The central database includes data like the 'users' table, which includes primary user

keys (a few different IDs) and a pointer to which shard a users' data can be found on.

- Use dedicated servers for static content.
- Talks about how to support Unicode.
- Use a share nothing architecture.
- Everything (except photos) are stored in the database.
- Statelessness means they can bounce people around servers and it's easier to make their APIs.
- Scaled at first by replication, but that only helps with reads.
- Create a search farm by replicating the portion of the database they want to search.
- Use horizontal scaling so they just need to add more machines.
- Handle pictures emailed from users by parsing each email is it's delivered in PHP. Email is parsed for any photos.
- Earlier they suffered from Master-Slave lag. Too much load and they had a single point of failure.
- They needed the ability to make live maintenance, repair data, and so forth, without taking the site down.
- Lots of excellent material on capacity planning. Take a look in the Information Sources for more details.