YouTube grew incredibly fast, to over 100 million video views per day, with only a handful of people responsible for scaling the site. How did they manage to deliver all that video to all those users? And how have they evolved since being acquired by Google?

Information Sources

1. Google Video

Platform

1. Apache
2. Python
3. Linux (SuSe)
4. MySQL
5. psyco, a dynamic python>C compiler
6. lighttpd for video instead of Apache

What's Inside?

The Stats

1. Supports the delivery of over 100 million videos per day.
2. Founded 2/2005
3. 3/2006 30 million video views/day
4. 7/2006 100 million video views/day
5. 2 sysadmins, 2 scalability software architects
6. 2 feature developers, 2 network engineers, 1 DBA

Recipe for handling rapid growth

```
while (true)
{
identify_and_fix_bottlenecks();
drink();
sleep();
notice_new_bottleneck();
}
```

This loop runs many times a day.

Web Servers

1. NetScalar is used for load balancing and caching static content.
2. Run Apache with mod_fast_cgi.
3. Requests are routed for handling by a Python application server.

4. Application server talks to various databases and other informations sources to get all the data and formats the html page.
5. Can usually scale web tier by adding more machines.
6. The Python web code is usually NOT the bottleneck, it spends most of its time blocked on RPCs.
7. Python allows rapid flexible development and deployment. This is critical given the competition they face.
8. Usually less than 100 ms page service times.
9. Use psyco, a dynamic python>C compiler that uses a JIT compiler approach to optimize inner loops.
10. For high CPU intensive activities like encryption, they use C extensions.
11. Some pregenerated cached HTML for expensive to render blocks.
12. Row level caching in the database.
13. Fully formed Python objects are cached.
14. Some data are calculated and sent to each application so the values are cached in local memory. This is an underused strategy. The fastest cache is in your application server and it doesn't take much time to send precalculated data to all your servers. Just have an agent that watches for changes, precalculates, and sends.

Video Serving

Costs include bandwidth, hardware, and power consumption.

Each video hosted by a minicluster. Each video is served by more than one machine.

- Using a a cluster means:
  More disks serving content which means more speed.
  Headroom. If a machine goes down others can take over.
  There are online backups.
- Servers use the lighttpd web server for video:
  Apache had too much overhead.
  Uses epoll to wait on multiple fds.
  Switched from single process to multiple process configuration to handle more connections.
- Most popular content is moved to a CDN (content delivery network):
  CDNs replicate content in multiple places. There's a better chance of content being closer to the user, with fewer hops, and content will run over a more friendly network.
  CDN machines mostly serve out of memory because the content is so popular there's little thrashing of content into and out of memory.
- Less popular content (120 views per day) uses YouTube servers in various colo sites.
  There's a long tail effect. A video may have a few plays, but lots of videos are being played. Random disks blocks are being accessed.
  Caching doesn't do a lot of good in this scenario, so spending money on more cache may not make sense. This is a very interesting point. If you have a long tail product

caching won't always be your performance savior.
Tune RAID controller and pay attention to other lower level issues to help.
Tune memory on each machine so there's not too much and not too little.

- **Serving Video Key Points**

1. Keep it simple and cheap.
2. Keep a simple network path. Not too many devices between content and users. Routers, switches, and other appliances may not be able to keep up with so much load.
3. Use commodity hardware. More expensive hardware gets the more expensive everything else gets too (support contracts). You are also less likely find help on the net.
4. Use simple common tools. They use most tools build into Linux and layer on top of those.
5. Handle random seeks well (SATA, tweaks).

Serving Thumbnails

- Surprisingly difficult to do efficiently.
- There are a like 4 thumbnails for each video so there are a lot more thumbnails than videos.
- Thumbnails are hosted on just a few machines.
- Saw problems associated with serving a lot of small objects:
  Lots of disk seeks and problems with inode caches and page caches at OS level.
  Ran into per directory file limit. Ext3 in particular. Moved to a more hierarchical structure. Recent improvements in the 2.6 kernel may improve Ext3 large directory handling up to 100 times, yet storing lots of files in a file system is still not a good idea.
  A high number of requests/sec as web pages can display 60 thumbnails on page.
  Under such high loads Apache performed badly.
  Used squid (reverse proxy) in front of Apache. This worked for a while, but as load increased performance eventually decreased. Went from 300 requests/second to 20.
  Tried using lighttpd but with a single threaded it stalled. Run into problems with multiprocesses mode because they would each keep a separate cache.
  With so many images setting up a new machine took over 24 hours.
  Rebooting machine took 610 hours for cache to warm up to not go to disk.
- To solve all their problems they started using Google's BigTable, a distributed data store:
  Avoids small file problem because it clumps files together.
  Fast, fault tolerant. Assumes its working on a unreliable network.
  Lower latency because it uses a distributed multilevel cache. This cache works across different collocation sites.
  For more information on BigTable take a look at Google Architecture, GoogleTalk Architecture, and BigTable.

Databases

1. The Early Years
   Use MySQL to store meta data like users, tags, and descriptions.
   Served data off a monolithic RAID 10 Volume with 10 disks.

Living off credit cards so they leased hardware. When they needed more hardware to handle load it took a few days to order and get delivered.
They went through a common evolution: single server, went to a single master with multiple read slaves, then partitioned the database, and then settled on a sharding approach.
Suffered from replica lag. The master is multithreaded and runs on a large machine so it can handle a lot of work. Slaves are single threaded and usually run on lesser machines and replication is asynchronous, so the slaves can lag significantly behind the master.
Updates cause cache misses which goes to disk where slow I/O causes slow replication.
Using a replicating architecture you need to spend a lot of money for incremental bits of write performance.
One of their solutions was prioritize traffic by splitting the data into two clusters: a video watch pool and a general cluster. The idea is that people want to watch video so that function should get the most resources. The social networking features of YouTube are less important so they can be routed to a less capable cluster.

2. The later years:
   Went to database partitioning.
   Split into shards with users assigned to different shards.
   Spreads writes and reads.
   Much better cache locality which means less IO.
   Resulted in a 30% hardware reduction.
   Reduced replica lag to 0.
   Can now scale database almost arbitrarily.

Data Center Strategy

1. Used manage hosting providers at first. Living off credit cards so it was the only way.
2. Managed hosting can't scale with you. You can't control hardware or make favorable networking agreements.
3. So they went to a colocation arrangement. Now they can customize everything and negotiate their own contracts.
4. Use 5 or 6 data centers plus the CDN.
5. Videos come out of any data center. Not closest match or anything. If a video is popular enough it will move into the CDN.
6. Video bandwidth dependent, not really latency dependent. Can come from any colo.
7. For images latency matters, especially when you have 60 images on a page.
8. Images are replicated to different data centers using BigTable. Code looks at different metrics to know who is closest.

Lessons Learned

1. **Stall for time**. Creative and risky tricks can help you cope in the short term while you work out longer term solutions.

2. **Prioritize**. Know what's essential to your service and prioritize your resources and efforts around those priorities.
3. **Pick your battles**. Don't be afraid to outsource some essential services. YouTube uses a CDN to distribute their most popular content. Creating their own network would have taken too long and cost too much. You may have similar opportunities in your system. Take a look at Software as a Service for more ideas.
4. **Keep it simple!** Simplicity allows you to rearchitect more quickly so you can respond to problems. It's true that nobody really knows what simplicity is, but if you aren't afraid to make changes then that's a good sign simplicity is happening.
5. **Shard**. Sharding helps to isolate and constrain storage, CPU, memory, and IO. It's not just about getting more writes performance.
6. **Constant iteration on bottlenecks**:
   Software: DB, caching
   OS: disk I/O
   Hardware: memory, RAID
7. **You succeed as a team**. Have a good cross discipline team that understands the whole system and what's underneath the system. People who can set up printers, machines,