Scalable Web Architectures: Common Patterns and Approaches

Scalable Web Architectures: Common Patterns and Approaches
So what is scalability?
Traffic growth
Dataset growth
Maintainability

Three goals of application architecture:
Scale
HA
Performance

Two kinds:
Vertical (get bigger)
Horizontal (get more)

Sometimes vertical scaling is right

Buying a bigger box is quick (ish)
Redesigning software is not

Running out of MySQL performance?
Spend months on data federation
Or, Just buy a ton more RAM

App Servers
Sessions!
(State)
Local sessions == bad
When they move == quite bad

Centralized sessions == good
No sessions at all == awesome!

Local Sessions
Stored on disk
PHP sessions

Stored in memory
Shared memory block (APC)

Bad!
Can’t move users
Can’t avoid hotspots
Not fault tolerant
If your load balancer has sticky sessions, you can still get hotspots
Depends on volume – fewer heavier users hurt more

Remote centralized sessions
Store in a central database
Or an in-memory cache

No porting around of session data
No need for sticky sessions
No hot spots

Need to be able to scale the data store
But we've pushed the issue down the stack

No sessions
Stash it all in a cookie!

Sign it for safety
$data = $user_id . '-' . $user_name;
$time = time();
$sig = sha1($secret . $time . $data);
$cookie = base64("$sig-$time-$data");

Timestamp means it's simple to expire it

Super slim sessions
If you need more than the cookie (login status, user id, username), then pull their account row from the DB
Or from the account cache

None of the drawbacks of sessions
Avoids the overhead of a query per page
Great for high-volume pages which need little personalization
Turns out you can stick quite a lot in a cookie too
Pack with base64 and it's easy to delimit fields

App servers
The Rasmus way
App server has 'shared nothing'
Responsibility pushed down the stack
Ooh, the stack

Other services scale similarly to web apps
That is, horizontally

The canonical examples:
Image conversion
Audio transcoding
Video transcoding
Web crawling
Compute!

Amazon
Let's talk about Amazon
S3 - Storage
EC2 – Compute! (XEN based)
SQS – Queueing

All horizontal

Cheap when small

Not cheap at scale

Load Balancer
Queuing
Parallelizable == easy!

If we can transcode/crawl in parallel, it's easy
But think about queuing
And asynchronous systems
The web ain't built for slow things
But still, a simple problem

Asynchronous system
Helps with peak periods

Database
More read power
Web apps typically have a read/write ratio of somewhere between 80/20 and 90/10
If we can scale read capacity, we can solve a lot of situations
MySQL replication!

Master-Slave Replication

Caching
Caching avoids needing to scale!
Or makes it cheaper

Getting more complicated…
Write-through cache
Write-back cache
Sideline cache
Easy to implement
Just add app logic

Need to manually invalidate cache
Well designed code makes it easy

Memcached
From Danga (LiveJournal)
http://www.danga.com/memcached/

HA Data
But what about HA?
SPOF

The key to HA is avoiding SPOFs
Identify
Eliminate

Some stuff is hard to solve
Fix it further up the tree
Dual DCs solves Router/Switch SPOF

Master-Master

Either hot/warm or hot/hot

Writes can go to either
But avoid collisions
No auto-inc columns for hot/hot
Bad for hot/warm too
Unless you have MySQL 5
But you can't rely on the ordering!
Design schema/access to avoid collisions
Hashing users to servers

Rings
Master-master is just a small ring
With 2 nodes

Bigger rings are possible
But not a mesh!
Each slave may only have a single master
Unless you build some kind of manual replication

Dual trees
Master-master is good for HA
But we can't scale out the reads (or writes!)

We often need to combine the read scaling with HA
We can simply combine the two models

There's a problem here
We need to always have 200% capacity to avoid a SPOF
400% for dual sites!
This costs too much

Solution is straight forward
Make sure clusters are bigger than 2

N+M
N+M
N = nodes needed to run the system
M = nodes we can afford to lose

Having M as big as N starts to suck
If we could make each node smaller, we can increase N while M stays constant
(We assume smaller nodes are cheaper)

Data Federation

At some point, you need more writes
This is tough
Each cluster of servers has limited write capacity

Just add more clusters!

Vertical partitioning
Divide tables into sets that never get joined

Split these sets onto different server clusters
Voila!

Simple things first
Logical limits
When you run out of non-joining groups
When a single table grows too large

Split up large tables, organized by some primary object
Usually users

Put all of a user's data on one 'cluster'
Or shard, or cell

Have one central cluster for lookups

Need more capacity?
Just add shards!
Don't assign to shards based on user_id!

For resource leveling as time goes on, we want to be able to move objects between shards
Maybe – not everyone does this
'Lockable' objects

The wordpress.com approach
Hash users into one of n buckets
Where n is a power of 2

Put all the buckets on one server

When you run out of capacity, split the buckets across two servers
Then you run out of capacity, split the buckets across four servers
Etc

Data federation
Heterogeneous hardware is fine
Just give a larger/smaller proportion of objects depending on hardware

Bigger/faster hardware for paying users
A common approach
Can also allocate faster app servers via magic cookies at the LB

Downsides
Need to keep stuff in the right place
App logic gets more complicated
More clusters to manage
Backups, etc
More database connections needed per page
Proxy can solve this, but complicated
The dual table issue
Avoid walking the shards!

Data federation is how large applications are scaled
It's hard, but not impossible

Good software design makes it easier
Abstraction!

Master-master pairs for shards give us HA

Master-master trees work for central cluster (many reads, few writes)

Multi-site HA
Having multiple datacenters is hard
Not just with MySQL

Hot/warm with MySQL slaved setup
But manual (reconfig on failure)

Hot/hot with master-master
But dangerous (each site has a SPOF)

Hot/hot with sync/async manual replication
But tough (big engineering task)

Serving Files
Serving lots of files is not too tough
Just buy lots of machines and load balance!

We're IO bound – need more spindles!
But keeping many copies of data in sync is hard
And sometimes we have other per-request overhead (like auth)

Serving out of memory is fast!
And our caching proxies can have disks too
Fast or otherwise
More spindles is better
We stay in sync automatically

We can parallelize it!
50 cache servers gives us 50 times the serving rate of the origin server
Assuming the working set is small enough to fit in memory in the cache cluster

Dealing with invalidation is tricky

We can prod the cache servers directly to clear stuff out
Scales badly – need to clear asset from every server – doesn't work well for 100 caches

We can change the URLs of modified resources
And let the old ones drop out cache naturally
Or prod them out, for sensitive data

Good approach!
Avoids browser cache staleness
Hello Akamai (and other CDNs)
Read more:
http://www.thinkvitamin.com/features/webapps/serving-javascript-fast

High overhead serving
What if you need to authenticate your asset serving?
Private photos
Private data
Subscriber-only files

Two main approaches
Proxies w/ tokens
Path translation

Perlbal backhanding
Perlbal can do redirection magic
Client sends request to Perbal
Perlbl plugin verifies user credentials
token, cookies, whatever
tokens avoid data-store access
Perlbal goes to pick up the file from elsewhere
Transparent to user

Doesn't keep database around while serving
Doesn't keep app server around while serving
User doesn't find out how to access asset directly

Permission URLs
But why bother!?
If we bake the auth into the URL then it saves the auth step
We can do the auth on the web app servers when creating HTML
Just need some magic to translate to paths
We don't want paths to be guessable

Downsides
URL gives permission for life
Unless you bake in tokens
Tokens tend to be non-expirable
We don't want to track every token
Too much overhead
But can still expire

Upsides
It works
Scales nicely

Storing Files

Storing files is easy!
Get a big disk
Get a bigger disk
Uh oh!

Horizontal scaling is the key
Again
NFS
Stateful == Sucks
Hard mounts vs Soft mounts, INTR

SMB / CIFS / Samba
Turn off MSRPC & WINS (NetBOIS NS)
Stateful but degrades gracefully

HTTP
Stateless == Yay!
Just use Apache
Volumes are limited in total size
Except (in theory) under ZFS & others

Sometimes we need multiple volumes for performance reasons
When using RAID with single/dual parity

At some point, we need multiple volumes
Volumes are limited in total size
Except (in theory) under ZFS & others

Sometimes we need multiple volumes for performance reasons
When using RAID with single/dual parity

At some point, we need multiple volumes

Further down the road, a single host will be too small
Total throughput of machine becomes an issue
Even physical space can start to matter
So we need to be able to use multiple hosts

HA is important for assets too
We can back stuff up
But we tend to want hot redundancy

RAID is good
RAID 5 is cheap, RAID 10 is fast
But whole machines can fail
So we stick assets on multiple machines

In this case, we can ignore RAID
In failure case, we serve from alternative source
But need to weigh up the rebuild time and effort against the risk
Store more than 2 copies?

Self repairing systems
When something fails, repairing can be a pain
RAID rebuilds by itself, but machine replication doesn't

The big appliances self heal
NetApp, StorEdge, etc

So does MogileFS (reaper)

Field Work
Flickr
Because I know it

LiveJournal
Because everyone copies it