If this text is too small to read, move closer!

Real World MySQL Performance Tuning

Slides at `http://develooper.com/talks/`

Jay Pipes
MySQL AB

(Jays slides are in a different PDF, see the url above)

Ask Bjørn Hansen
Develooper LLC

r12

Hello.

- I'm Ask Bjørn Hansen
- This is Jay Pipes
- 83* brilliant° tips to make your website keep working past X requests/transactions per T time
  - Requiring minimal extra work! (or money)
  - Concepts applicable to ~all languages and platforms!

* Estimate, your mileage may vary

° Well, a lot of them are pretty good

Construction Ahead!

- Conflicting advice ahead
- Not everything here is applicable to everything
- Ways to "think scalable" rather than end-all-be-all solutions
- (Jay has lots of "solutions"!)

Questions ...

- How many ...
- ... are using Perl? PHP? Python? Java? Ruby?
- 3.23? 4.0? 4.1? 5.0? 5.1?
- MyISAM? InnoDB? Other?
- Are primarily "programmers" vs "DBAs"
- Replication? Cluster? Partitioning?
- Enterprise? Community?
- PostgreSQL? Oracle? SQL Server? Other?

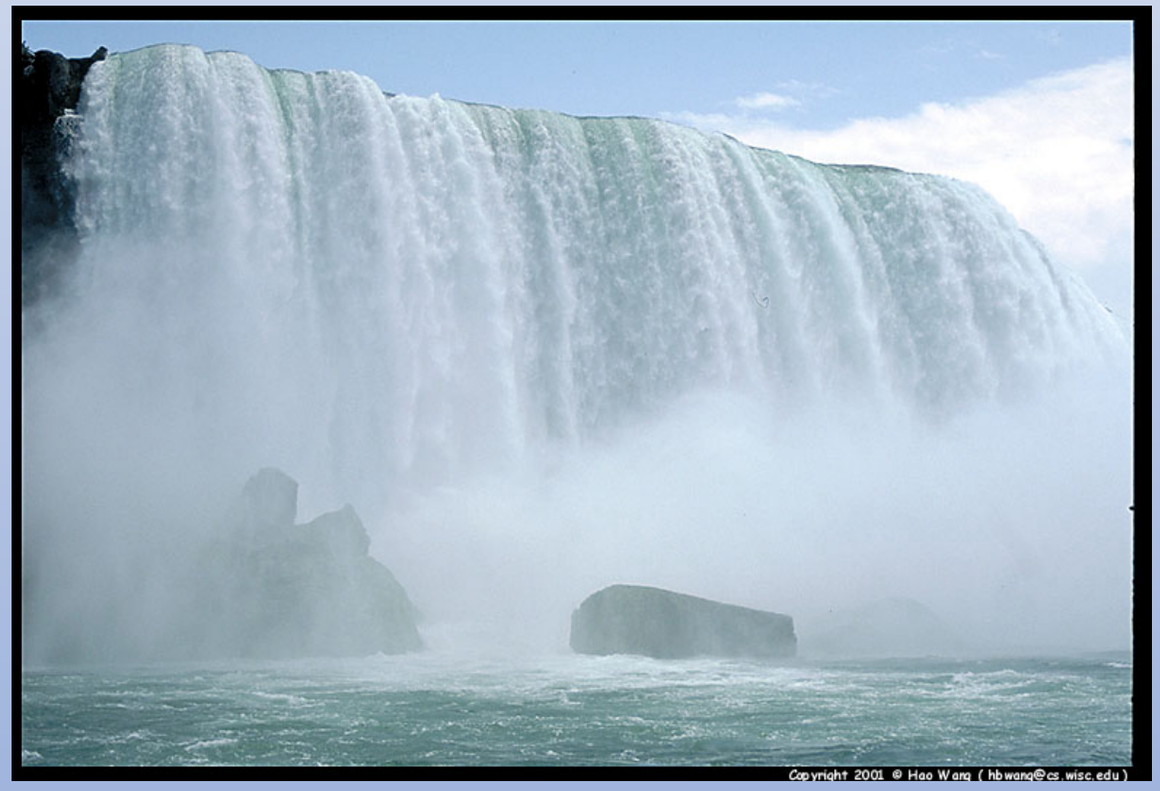
- The first, last and only lesson:

- **Think Horizontal!**

- Everything in your architecture, not just the front end web servers

- Micro optimizations and other implementation details — Bzzzzt! Boring! (that's Jays stuff – I suggest a bathroom break...)

(blah blah blah, we'll get to the cool stuff in a moment!)



Benchmarking techniques

- Scalability isn't the same as processing time
  - Not “how fast” but “how many”
  - Test “force”, not speed. Think amps, not voltage
  - Test *scalability*, not just “performance”
- Use a realistic load
  - Test with "slow clients"
- Testing “how fast” is ok when optimizing implementation details (code snippets, sql queries, server settings)

Vertical scaling

- "Get a bigger server"
- "Use faster CPUs"
- Can only help so much (with bad scale/$ value)
- A server twice as fast is more than twice as expensive
- Super computers are horizontally scaled!

Horizontal scaling

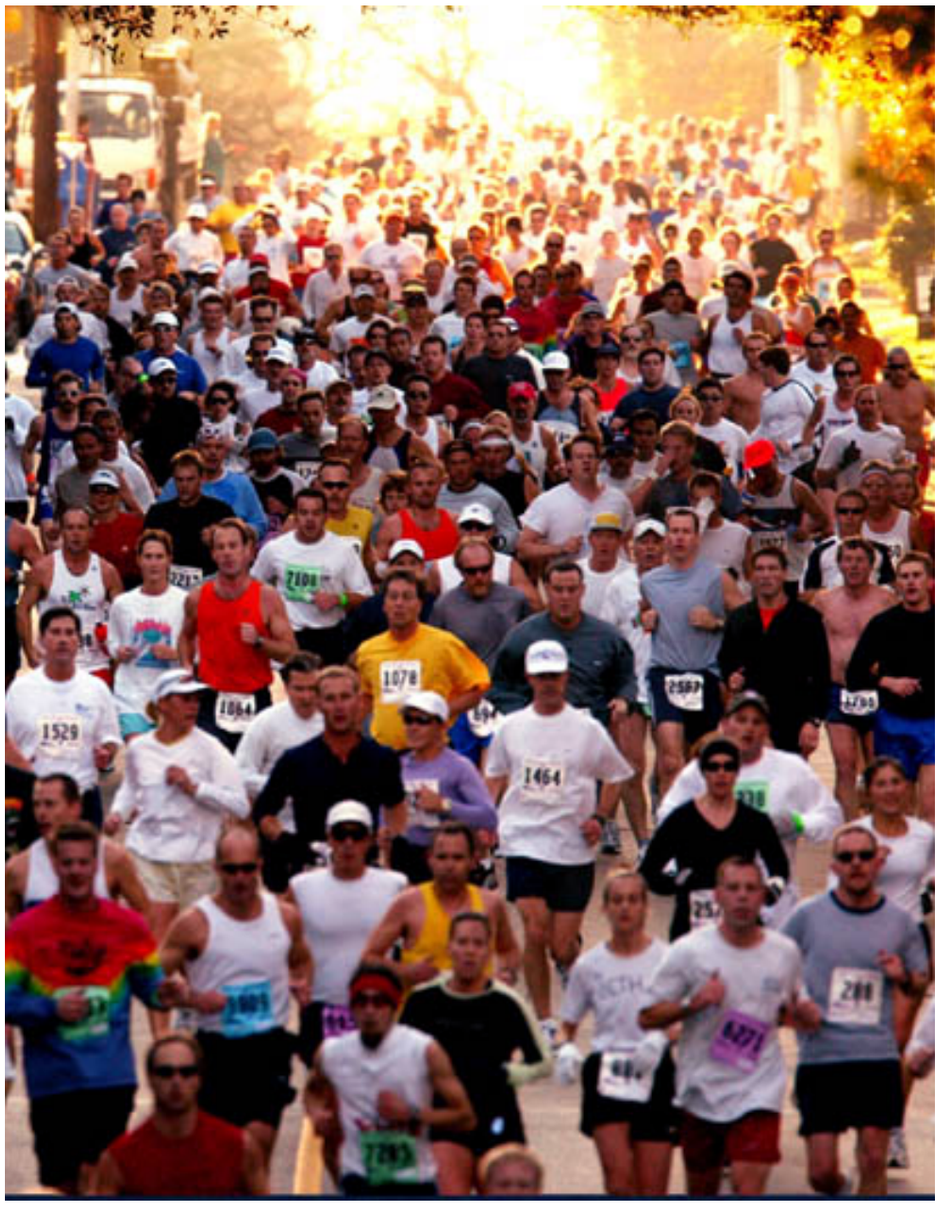
- "Just add another box" (or another thousand or ...)
- Good to great ...
  - **Implementation**, scale your system **a few** times
  - **Architecture**, scale dozens or **hundreds** of times
- Get the big picture right first, do micro optimizations later

Scalable Application Servers

Don't paint yourself into a corner from the start

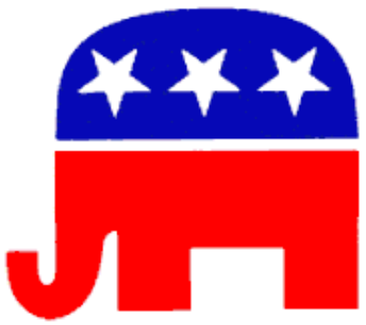
Run Many of Them

- For your application...
- Avoid having *The Server* for anything
- Everything should (be able to) run on any number of boxes



Stateless vs Stateful

- "Shared Nothing"
- Don't keep state within the application server (or at least be Really Careful)
- Do you use PHP or mod_perl (or something else that's running in Apache HTTPD)?
  - You get that for free! (usually)



I did all this – it's still slow!

- Optimizations and good micro-practices are necessary, of course
- But don't confuse what is what!
- Know when you are optimizing
- Know when you need to step back and rethink "the big picture"

Sessions

"The key to be stateless"

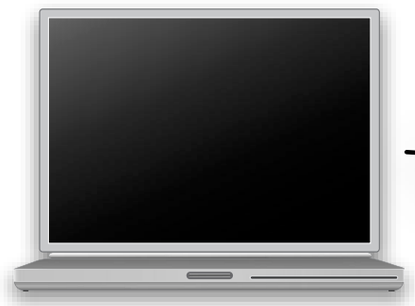
or

"What goes where"

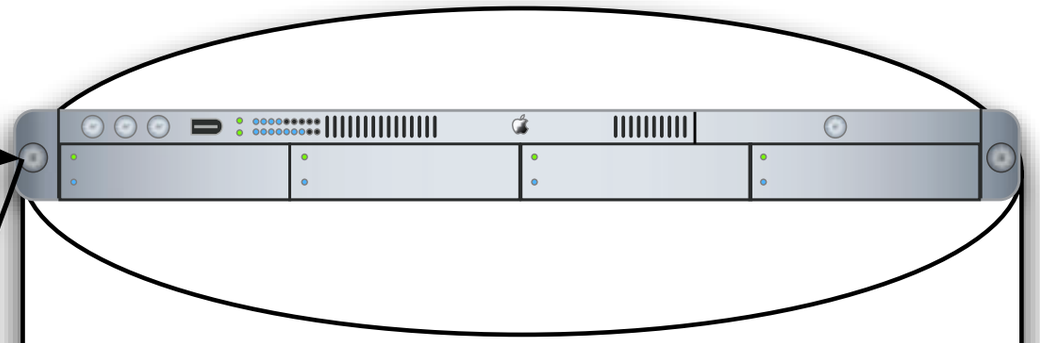
No Local Storage

- Ever! Not even as a quick hack.
- Storing session (or other state information) "on the server" *doesn't work*.
- *"But my load balancer can do 'sticky sessions'"*
  - The web isn't "session based", it's one short request after another – deal with it

Evil Session



Cookie: session_id=12345

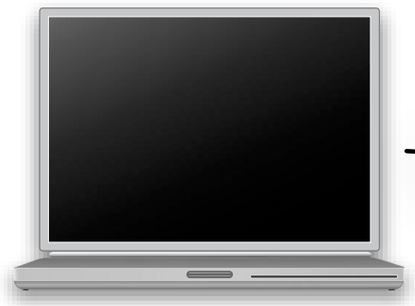


Web/application server
with local
Session store

```
...
12345 => {
    user =>
      { username => 'joe',
        email => 'joe@example.com',
        id => 987,
    },
    shopping_cart => { ... },
    last_viewed_items => { ... },
    background_color => 'blue',
},
12346 => { ... },
....
```

What's wrong with this?

Evil Session



Cookie: session_id=12345

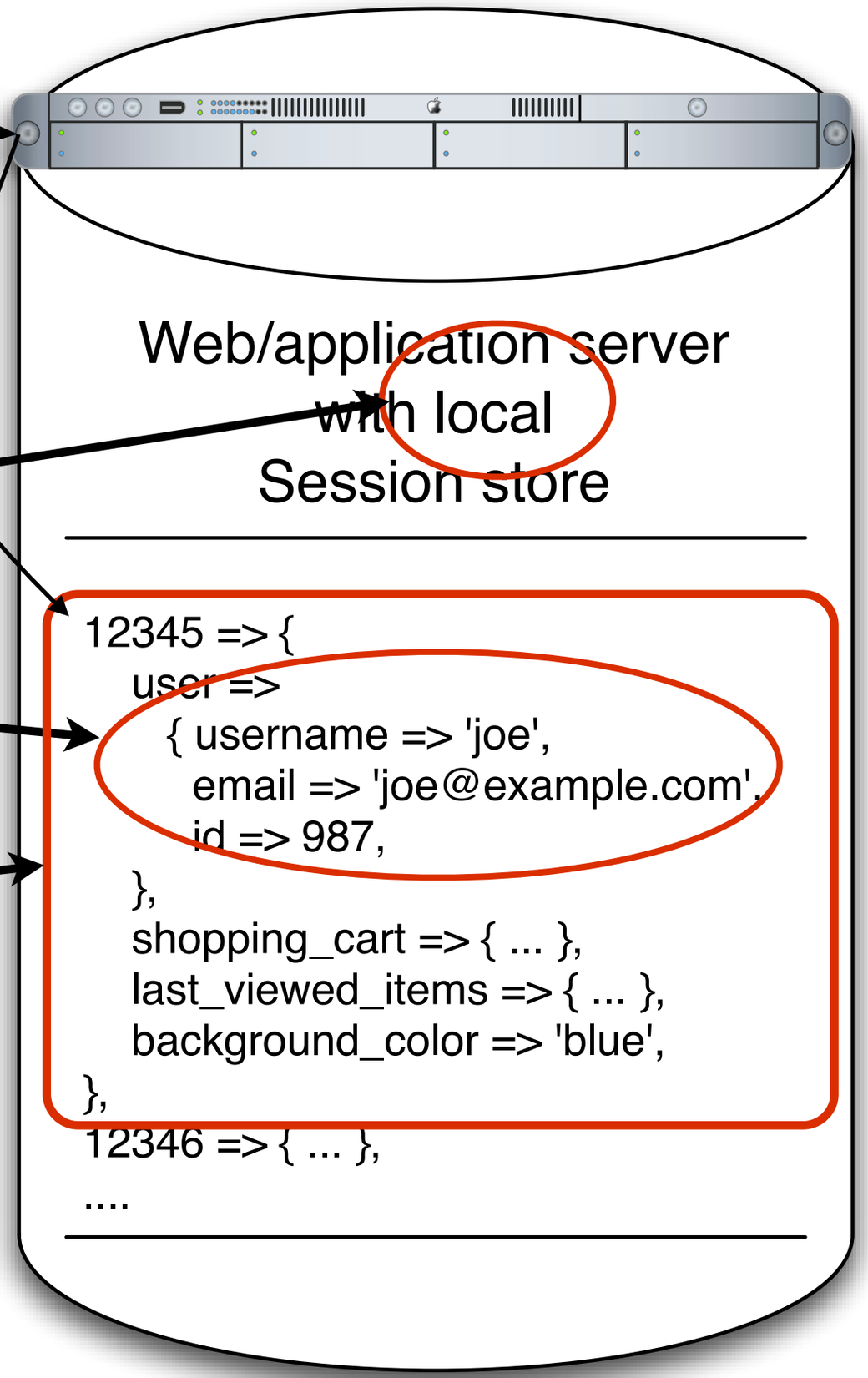
Easy to guess cookie id

Saving state on one server!

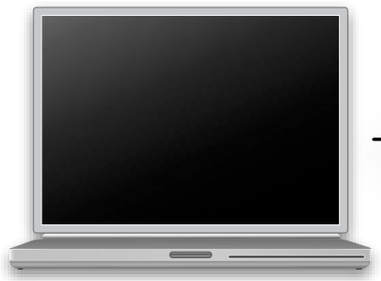
Duplicate data from a DB table

Big blob of junk!

What's wrong with this?



Good Session!



Cookie: sid=seh568fzkj5k09z;
user=987-65abc;
bg_color=blue;
cart=...;

Web/application server

Database(s)

Users
987 =>
{ username => 'joe',
email => 'joe@example.com',
},
...

Shopping Carts
...

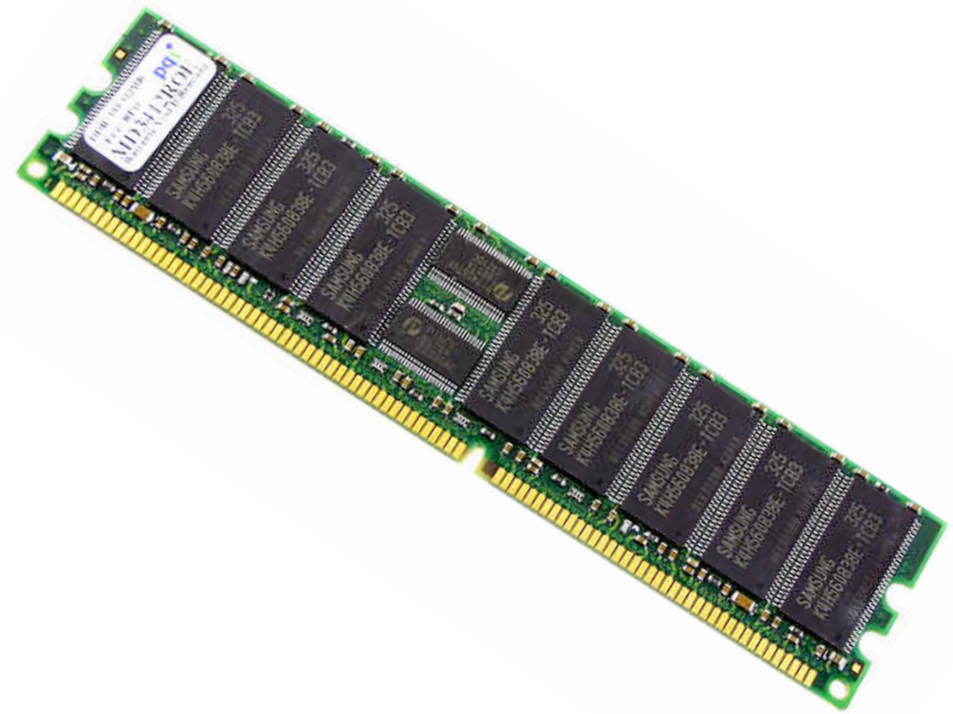
- Stateless web server!
- Important data in database
- Individual expiration on session objects
- Small data items in cookies

memcached cache

seh568fzkj5k09z =>
{ last_viewed_items => {...},
... other "junk" ...
},
....

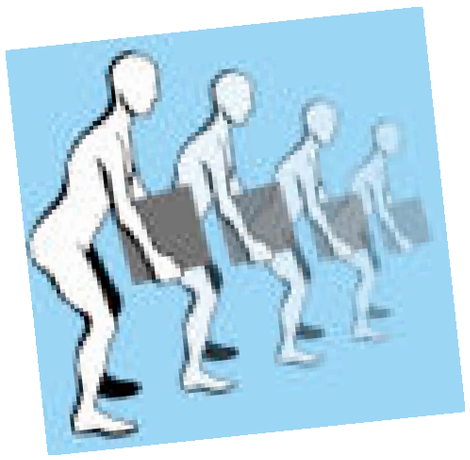
Safe cookies

- Worried about manipulated cookies?
- Use checksums and timestamps to validate them!
  - `cookie=1/value/1123157440/ABCD1234`
  - `cookie=1/user::987/cart::943/ts::1123.../EFGH9876`
  - `cookie=$cookie_format_version`
    `/$key::$value[/$key::$value]`
    `/ts::$timestamp`
    `/$md5`
- Encrypt them if you must (rarely worth the trouble and CPU cycles)



Caching

How to not do all that work again and again and again...



Generate **Static** Pages

- Ultimate Performance: Make all pages static
- Generate them from templates nightly or when updated
- Doesn't work well if you have millions of pages or page variations

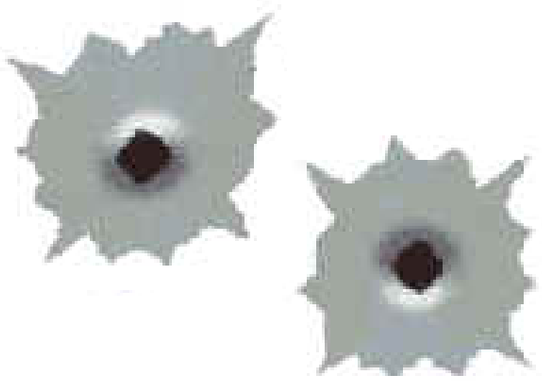
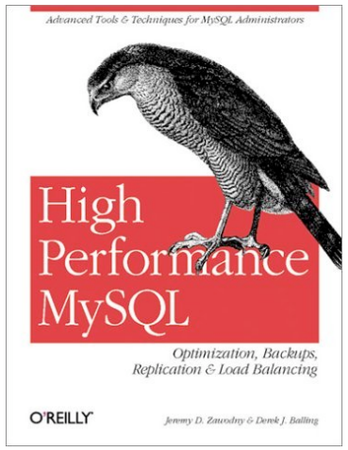
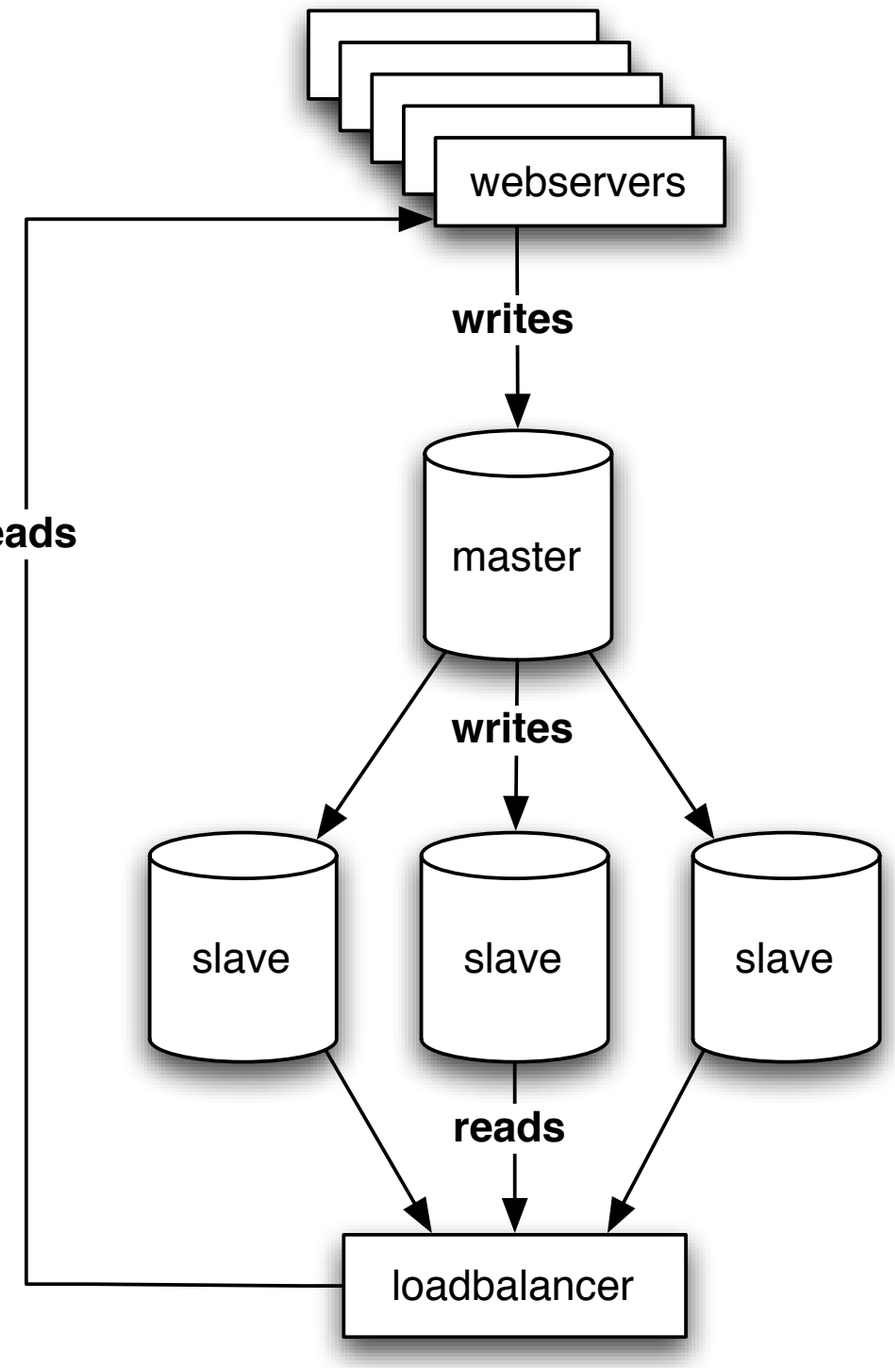
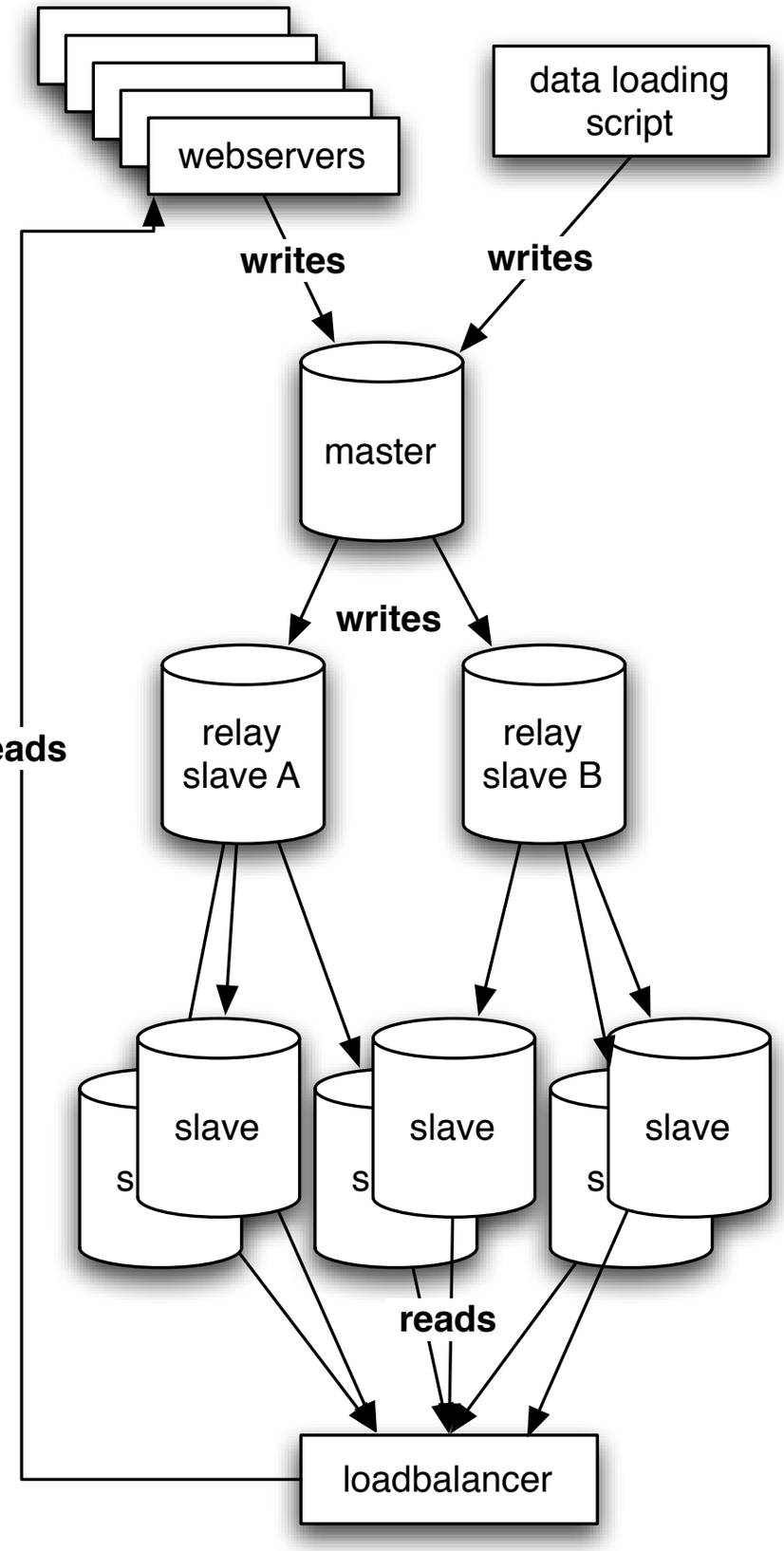
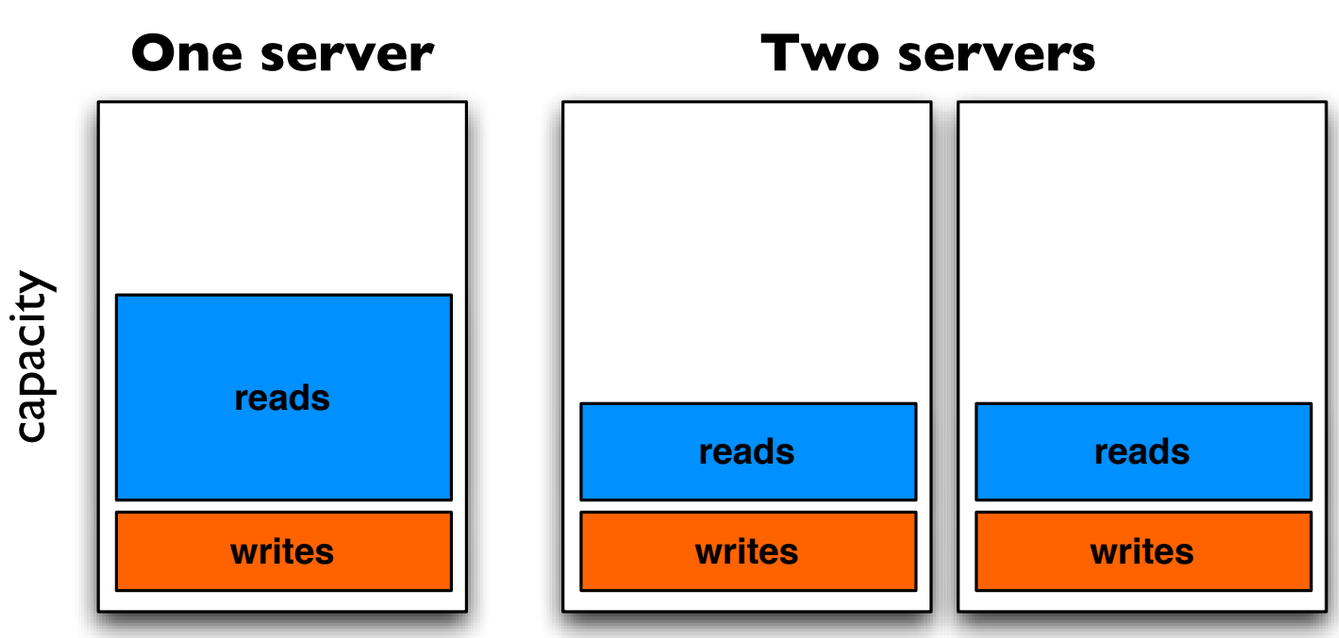
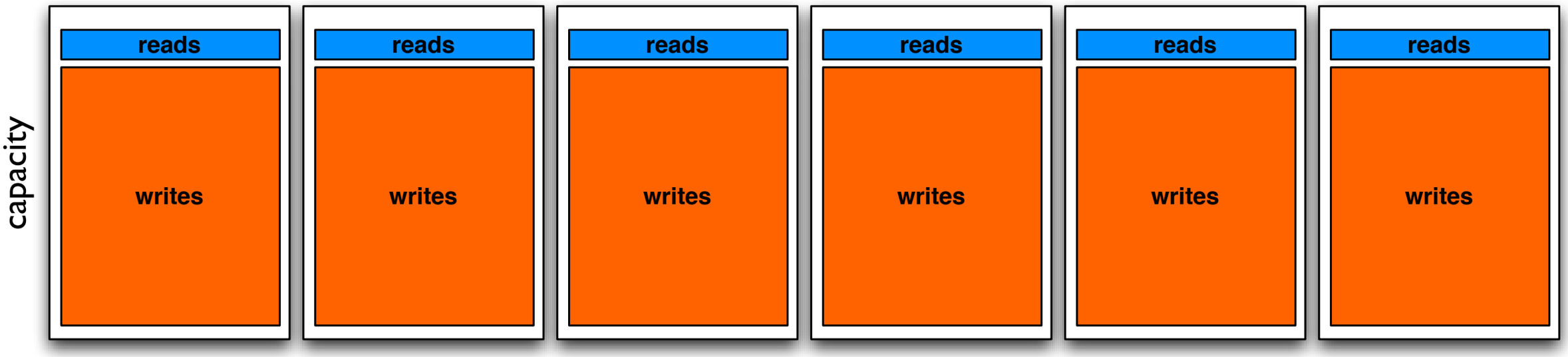
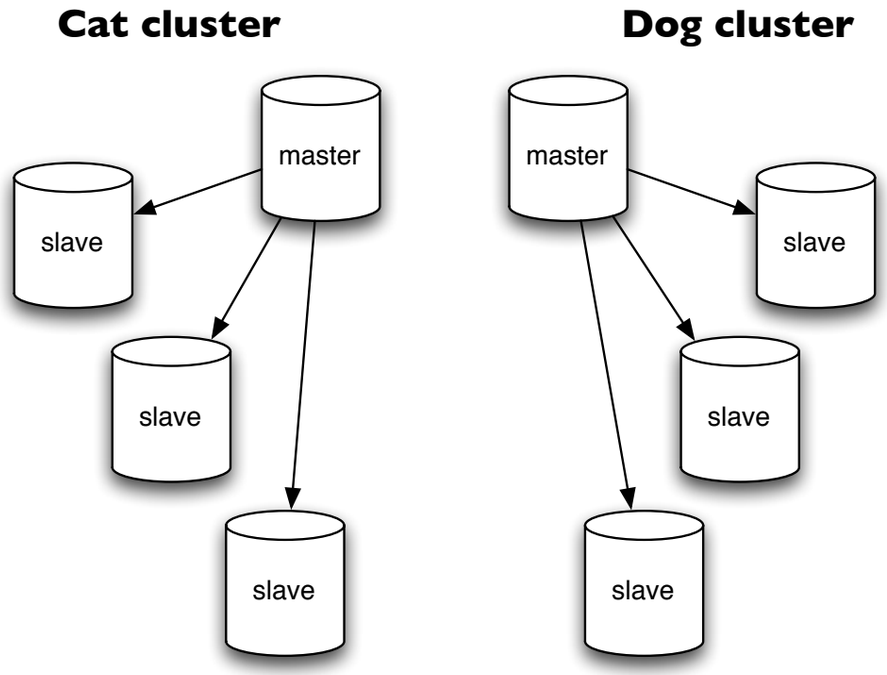
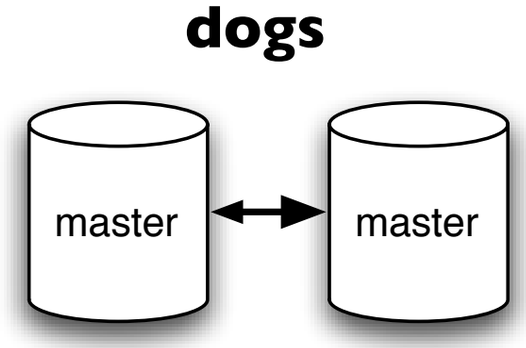
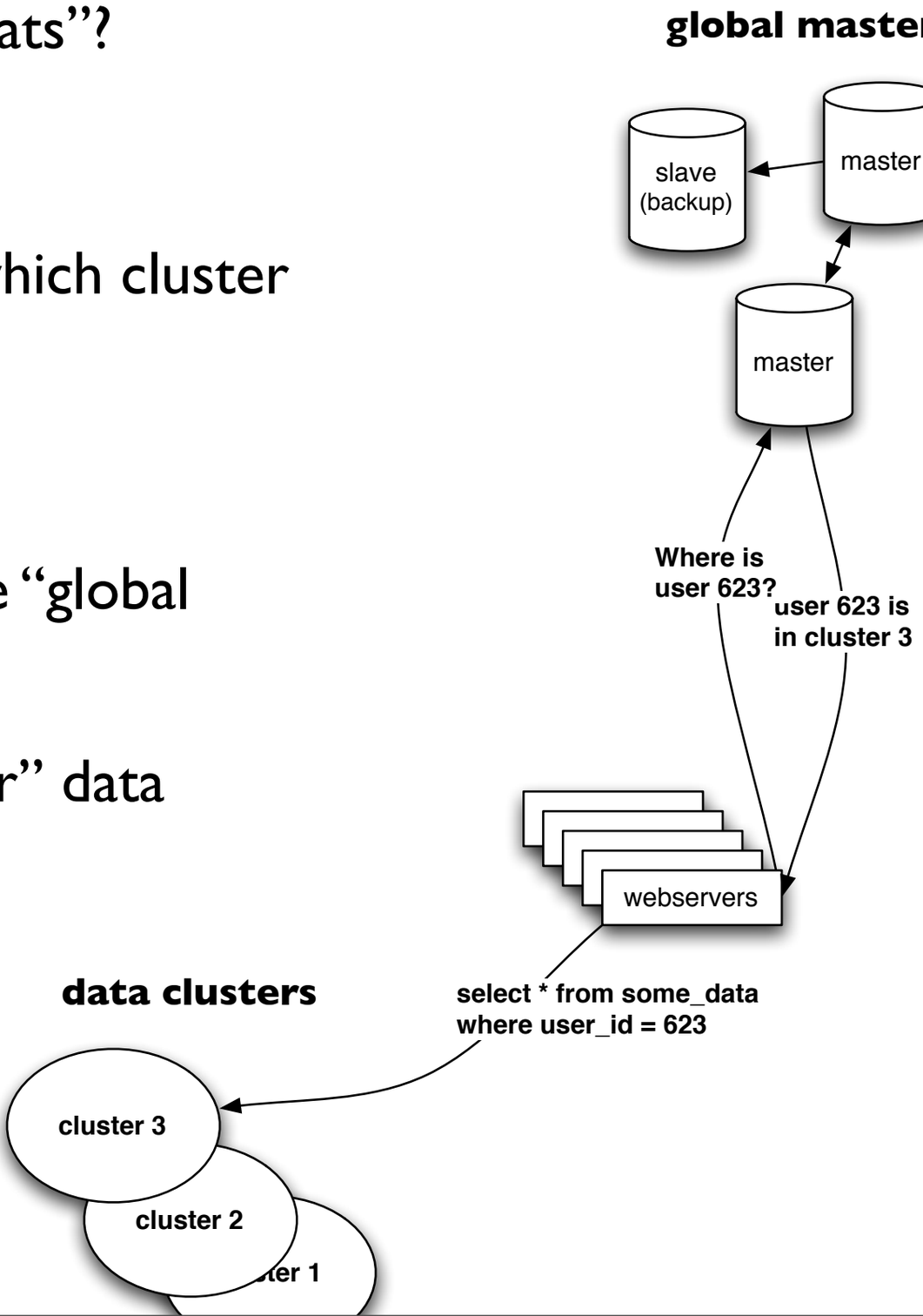
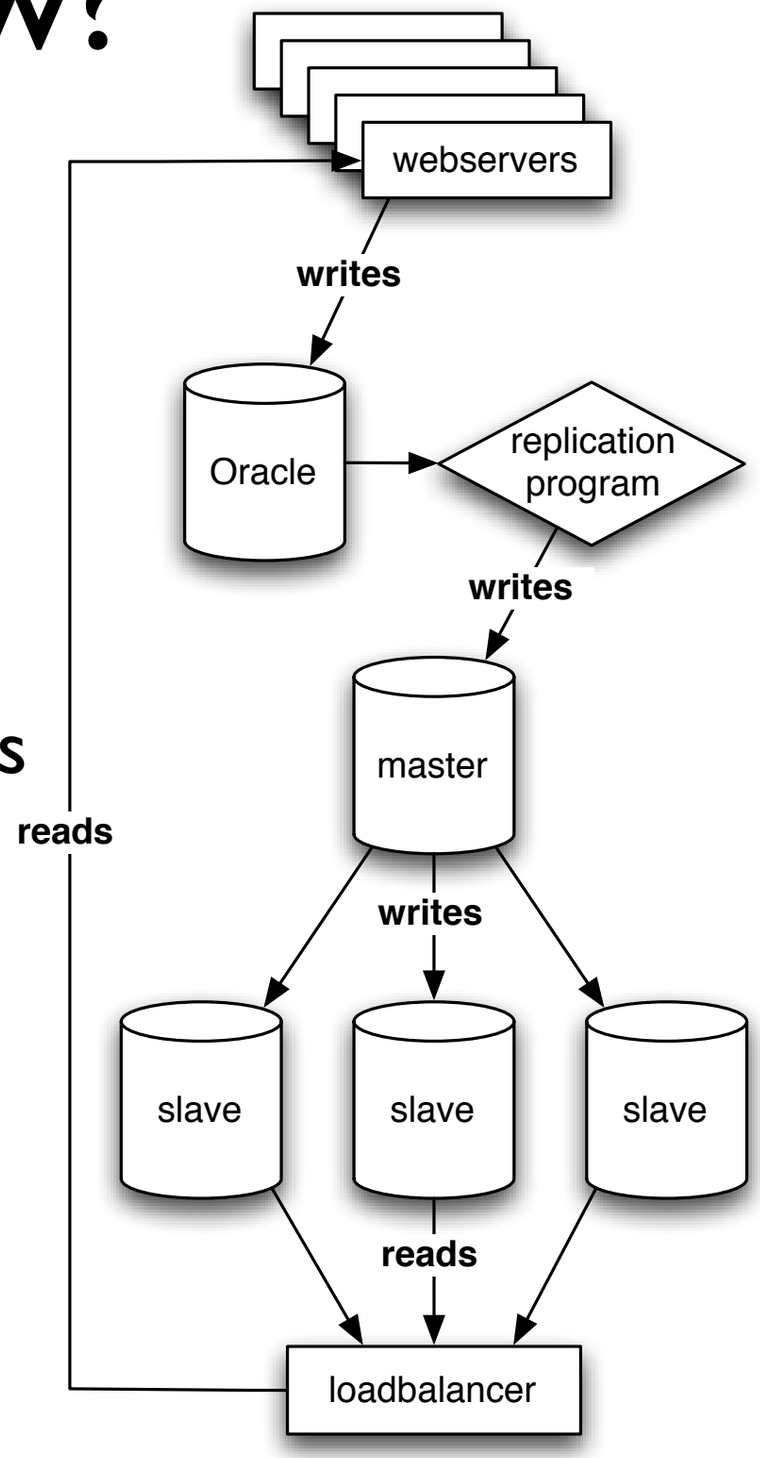
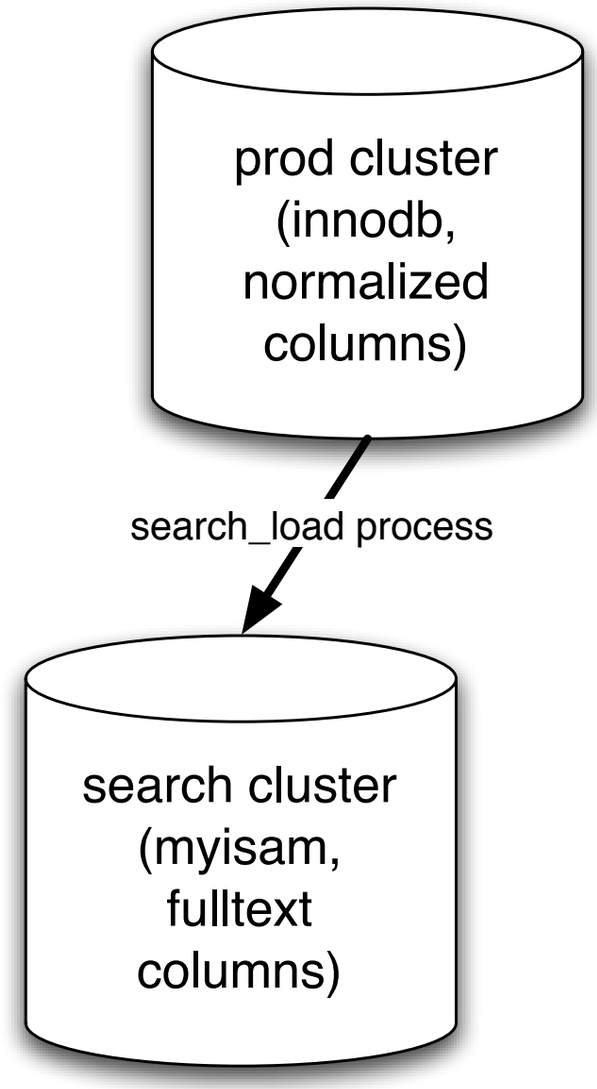
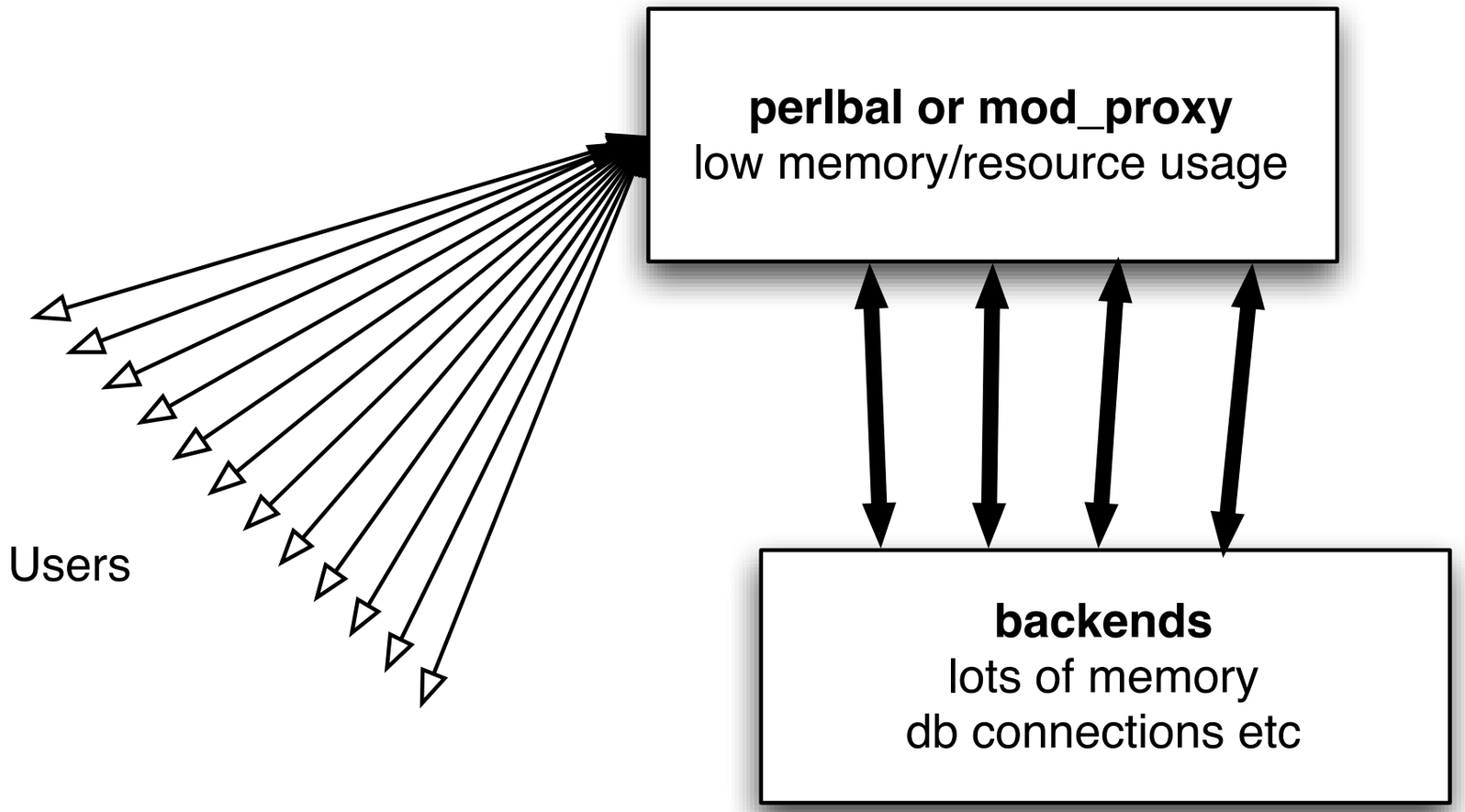
Cache **full** pages

(or responses if it's an API)

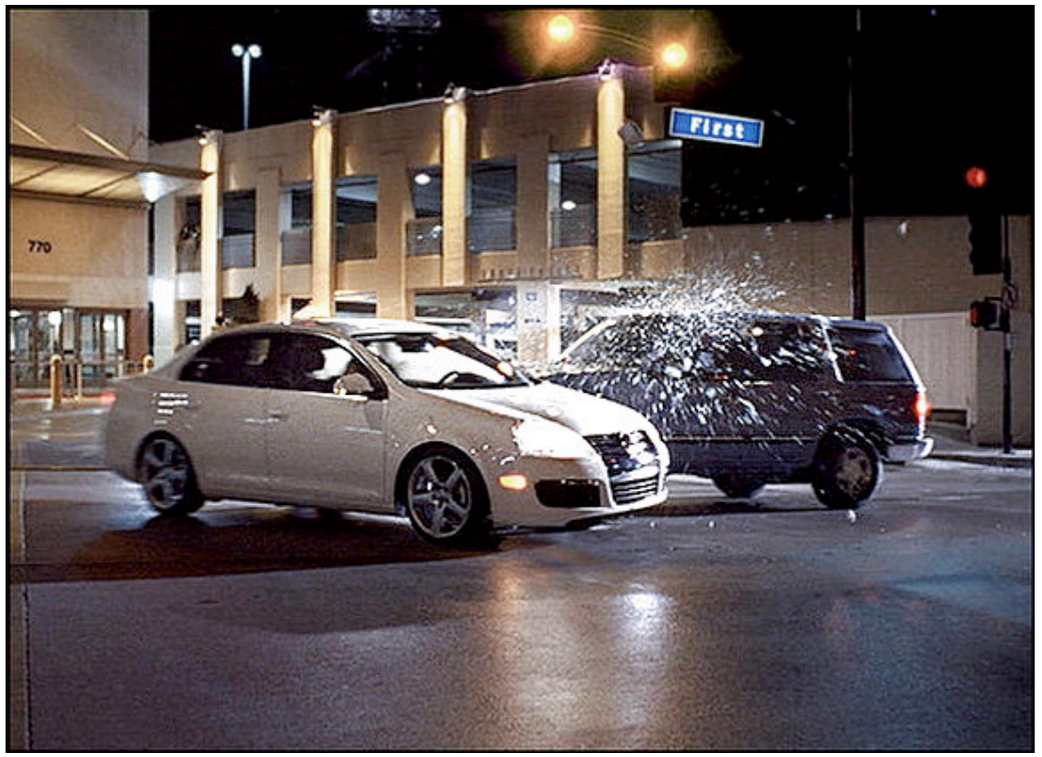
- Cache full output **in the application**
- Include cookies etc. in the "cache key"
- Fine tuned application level control
- The most flexible
  - "use cache when this, not when that"
  - Use regular expressions to insert customized content into the cached page
  -

Cache **full** pages 2

- Front end cache (**mod_cache, squid**, **Varnish***...) stores generated content
  - Set Expires header to control cache times
- ***or*** Rewrite rule to generate page if the cached file doesn't exist (this is what Rails does) – only scales to one server
  - ```
    RewriteCond %{REQUEST_FILENAME} !-s
    RewriteCond %{REQUEST_FILENAME}/index.html !-s
    RewriteRule (^/.*)  /dynamic_handler/$1 [PT]
    ```
- Still doesn't work for dynamic content per user *("6 items in your cart")*
- Great for caching "dynamic" images!

*This will be one of the cool tools in this field very soon
    ```

Cache **partial** pages

- Pre-generate static page "snippets"
  (this is what my.yahoo.com does or used to do...)
  - Have the handler just assemble pieces ready to go
- Cache little page snippets (say the sidebar)
- Be careful, easy to spend more time managing the cache snippets than you save!
- "Regexp" dynamic content into an otherwise cached page

Cache **data**

- Cache data that's slow to query, fetch or calculate
- Generate page from the cached data
- Use the same data to generate API responses!
- Moves load to cache servers
  - (For better or worse)
- Good for slow data used across many pages
  ("todays bestsellers in $category")

Cache **hit-ratios**

- Start with things you hit all the time
- Look at database logs
- Don’t cache if you’ll need more effort writing to the cache than you save
- Do cache if it’ll help you when that one single page gets a million hits in a few hours (one out of two hundred thousand pages on the digg frontpage)



Caching Tools

Where to put the cache data ...

A couple of bad ideas

Don't do this!

- Process memory (`$cache{foo}`)
  - Not shared!
- Shared memory? Local file system?
  - Limited to one machine (likewise for a file system cache)
  - Some implementations are really fast
- MySQL query cache
  - Flushed on each update
  - Nice if it helps; don't depend on it

MySQL cache table

- Write into one or more cache tables
- id is the “cache key”
- type is the “namespace”
- metadata for things like headers for cached http responses
- purge_key to make it easier to delete data from the cache

```
CREATE TABLE `cache` (
  `id` varchar(128) NOT NULL,
  `type` varchar(128) NOT NULL default '',
  `created` timestamp NOT NULL,
  `purge_key` varchar(64) default NULL,
  `data` mediumblob NOT NULL,
  `metadata` mediumblob,
  `serialized` tinyint(1) NOT NULL default '0',
  `expires` datetime NOT NULL,
  PRIMARY KEY  (`id`,`type`),
  KEY `expire_idx` (`expire`),
  KEY `purge_idx` (`purge_key`)
) ENGINE=InnoDB
```

MySQL Cache Fails

- Scaling and availability issues
  - How do you load balance?
    - How do you deal with a cache box going away?
- Partition the cache to spread the write load
- Use Spread to *write* to the cache and distribute configuration

MySQL Cache Scales

- Persistence
- Most of the usual “scale the database” tricks apply
- Partitioning
- Master-Master replication for availability
- .... more on those things in a moment
- memcached scheme for partitioning and fail-over

memcached

- LiveJournal's distributed caching system
  (also used at slashdot, wikipedia, etc etc)
- memory based
- Linux 2.6 (epoll) or FreeBSD (kqueue)
  - Low overhead for many many connections
- Run it on boxes with free memory
- No "master"
- Simple lightweight protocol
  - perl, java, php, python, ruby, ...
- Performance (roughly) similar to a MySQL cache
- Scaling and high-availability is "built-in"

Database scaling

How to avoid buying that gazillion dollar Sun box

~$3,500,000
Vertical

~$2,000
*(= **1750** for $3.5M!)*
Horizontal

Be Simple

- Use MySQL (5+!)
  - It's fast and it's easy to manage and tune
  - Easy to setup development environments
  - Other DBs can be faster at certain complex queries but are harder to tune – and MySQL is catching up fast!
- Avoid making your schema too complicated
- Ignore some of the upcoming advice until you REALLY need it!

- *PostgreSQL is fast too :-)*

Replication

More data more places!
Share the ~~love~~ load

Basic Replication

- Good Great for read intensive applications
- Write to one master
- Read from many slaves



Lots more details in
“High Performance MySQL”



Relay slave replication

- Running out of bandwidth on the master?
- Replicating to multiple data centers?
- A “replication slave” can be master to other slaves
- Almost any possible replication scenario can be setup (circular, star replication, ...)



Replication Scaling – Reads

- Reading scales well with replication
- Great for (mostly) read-only applications



(thanks to Brad Fitzpatrick!)

Replication Scaling – Writes

(aka when replication sucks)

- Writing doesn't scale with replication
- All servers needs to do the same writes



Partition the data

Divide and Conquer!

or

Web 2.0 Buzzword Compliant!

Now free with purchase of milk!!

Partition your data

- 96% read application? Skip this step...
- Solution to the too many writes problem: Don't have all data on all servers
- Use a separate cluster for different data sets



The Write Web!

- Replication too slow? Don't have replication slaves!
- Use a (fake) **master-master** setup and partition / shard the data!
- Simple redundancy!
- No latency from commit to data being available
- Don't bother with fancy 2 or 3 phase commits
  - (Make each "main object" (user, product, ...) always use the same master – as long as it's available)



Partition with a global master server

- Can't divide data up in "dogs" and "cats"?
- Flexible partitioning!
- The "global" server keeps track of which cluster has the data for user "623"
- Get all PKs from the global master
- Only auto_increment columns in the "global master"
- Aggressively cache the "global master" data (memcached)
  - and/or use MySQL Cluster (ndb)



Hacks!

Don't be afraid of the data-duplication monster

Summary tables!

- Find queries that do things with COUNT(*) and GROUP BY and create tables with the results!
  - Data loading process updates both tables
  - or hourly/daily/... updates
- Variation: Duplicate data in a different "partition"
  - Data affecting both a "user" and a "group" goes in both the "user" and the "group" partition (Flickr does this)

Summary databases!

- Don't just create summary tables
- Use summary databases!
- Copy the data into special databases optimized for special queries
  - full text searches
  - index with both cats and dogs
  - anything spanning all clusters
- Different databases for different latency requirements (RSS feeds from replicated slave DB)

“Manual” replication

- Save data to multiple “partitions”
- Application writes two places *or*
- last_updated/modified_on and deleted columns *or*
- Use triggers to add to “replication_queue” table
- Background program to copy data based on the queue table or the last_updated column
- Build summery tables or databases in this process
- Build star/spoke replication system

Make everything repeatable

- Script failed in the middle of the nightly processing job?
  (they will sooner or later, no matter what)
- How do you restart it?
- Build your "summary" and "load" scripts so they always can be run again! (and again and again)
- One "authoritative" copy of a data piece – summaries and copies are (re)created from there

a brief diversion ...

Running Oracle now?

- Move read operations to MySQL!
- Replicate from Oracle to a MySQL cluster with “manual replication”
- Use triggers to keep track of changed rows in Oracle
- Copy them to the MySQL master server with a replication program
- Good way to “sneak” MySQL in ...



More MySQL

Faster, faster, faster ....

Table Choice

- Short version:
  Use InnoDB, it's harder to make them fall over
- Long version:
  Use InnoDB except for
  - Big read-only tables (smaller, less IO)
  - High volume streaming tables (think logging)
    - Locked tables / `INSERT DELAYED`
  - Specialized engines for special needs
  - More engines in the future
  - For now: InnoDB

Multiple MySQL instances

- Run different MySQL instances for different workloads
  - Even when they share the same server anyway!
  - InnoDB vs MyISAM instance
- Move to separate hardware and replication easier
- Optimize MySQL for the particular workload
- Very easy to setup with the instance manager or mysqld_multi
- mysql.com init.d script supports the instance manager (don't use the redhat/fedora script!)



Asynchronous data loading

- Updating counts? Loading logs?
- Don't talk directly to the database, send updates through Spread (or whatever) to a daemon loading data
- Don't update for each request

```
update counts set count=count+1 where id=37
```

- Aggregate 1000 records or 2 minutes data and do fewer database changes

```
update counts set count=count+42 where id=37
```

- Being disconnected from the DB will let the frontend keep running if the DB is down!

Preload, -dump and -process

- Let the servers do as much as possible without touching the database directly
  - Data structures in memory – ultimate cache!
  - Dump never changing data structures to JS files for the client to cache
- Dump smaller read-only often accessed data sets to SQLite or BerkeleyDB and rsync to each webserver (or use NFS, but...)
  - Or a MySQL replica on each webserver

Stored Procedures Dangerous

- Not horizontal
- Bad:
  Work done in the database server (unless it's read-only and replicated)
- Good:
  Work done on one of the scalable web fronts
- Only do stored procedures if they save the database work (network-io work > SP work)

Reconsider Persistent DB Connections

- DB connection = thread = memory
- With partitioning all httpd processes talk to all DBs
- With lots of caching you might not need the main database that often

- MySQL connections are fast
- Always use persistent connections with Oracle!
  - Commercial connection pooling products
- pgsql, sybase, oracle? Need thousands of persistent connections?
  - In Perl the new DBD::Gofer can help with pooling!

InnoDB configuration

- `innodb_file_per_table`
  Splits your innodb data into a file per table instead of one big annoying file
  - Makes `optimize table `table`` clear unused space
- `innodb_buffer_pool_size=($MEM*0.80)`
- `innodb_flush_log_at_trx_commit` setting
- `innodb_log_file_size`
- `transaction-isolation = READ-COMMITTED`

Store Large Binary Objects

(aka how to store images)

- Meta-data table (name, size, ...)
- Store images either in the file system
  - meta data says "server '123', filename 'abc'"
  - (If you want this; use mogilefs or Amazon S3 for storage!)
- ***OR*** store images in other tables
  - Split data up so each table don't get bigger than ~4GB
- Include "last modified date" in meta data
  - Include it in your URLs if possible to optimize caching (/images/$timestamp/$id.jpg)

Three Application Edicts

- All Unicode all the time! (except when you really really never need it – urls, email addresses, ...)
  - For Perl: DBD::mysql is fixed!
  - It will never be easier to convert than now
- Everything uses UTC – it'll never be easier to change than now (format for local timezone on display)
- My new favorite feature
  - Make MySQL picky about bad input!
  - `SET sql_mode = 'STRICT_TRANS_TABLES';`

Don’t overwork the DB

- Databases don’t easily scale
- Don’t make the database do a ton of work
- Referential integrity is good
    - Tons of stored procedures to validate and process data not so much
- Don’t be too afraid of de-normalized data – sometimes it’s worth the tradeoffs (call them summary tables and the DBAs won’t notice)

Use light processes for light tasks

- Thin proxies servers or threads for “network buffers”
- Goes between the user and your heavier backend application

- Built-in load-balancing! (for Varnish, perlbal, ...)
- httpd with mod_proxy / mod_backhand
  - perlbal
    – more on that in a bit
  - Varnish, ~~squid~~, ~~pound~~, ...

Proxy illustration



Light processes

- Save memory and database connections
- This works spectacularly well. Really!
- Can also serve static files
- Avoid starting your main application as root
- Load balancing
- In particular important if your backend processes are “heavy”

Light processes

- Apache 2 makes it **Really Easy**

- ```
  ProxyPreserveHost On
  <VirtualHost *>
    ServerName combust.c2.askask.com
    ServerAlias *.c2.askask.com
    RewriteEngine on
    RewriteRule (.*) http://localhost:8230$1 [P]
  </VirtualHost>
  ```

- Easy to have different “backend environments” on one IP

- Backend setup (Apache 1.x)
  ```
  Listen 127.0.0.1:8230
  Port 80
  ```


  ```

High Availability



and Load Balancing
and Disaster Recovery

High Availability

- Automatically handle failures! (bad disks, failing fans, "oops, unplugged the wrong box", ...)
- For your app servers the load balancing system should take out "bad servers" (most do)
  - perlbal or Varnish can do this for http servers
- Easy-ish for things that can just "run on lots of boxes"

Make that service always work!

- Sometimes you need a service to always run, but on specific IP addresses
  - Load balancers (level 3 or level 7: perlbal/varnish/squid)
  - Routers
  - DNS servers
  - NFS servers
  - Anything that has failover or an alternate server – the IP needs to move (much faster than changing DNS)

Load balancing

- Key to horizontal scaling (duh)
- 1) All requests goes to the load balancer
  2) Load balancer picks a "real server"
- Hardware (lots of vendors)
  Coyote Point have relatively cheaper ones
- Linux Virtual Server
- Open/FreeBSD firewall rules (pf firewall pools)
  (no automatic failover, have to do that on the "real servers")

Load balancing 2

- Use a “level 3” (tcp connections only) tool to send traffic to your proxies
- Through the proxies do “level 7” (http) load balancing
- perlbal has some really good features for this!

perlbal

- Event based based for HTTP load balancing, web serving, and a mix of the two (see below).
- Practical fancy features like "multiplexing" keep-alive connections to both users and back-ends
- Everything can be configured or reconfigured on the fly
- If you configure your backends to only allow as many connections as they can handle (you should anyway!) perlbal with automatically balance the load "perfectly"
- Can actually give Perlbal a list of URLs to try. Perlbal will find one that's alive. Instant failover!
- `http://www.danga.com/perlbal/`

Varnish

- Modern high performance http accelerator
- Optimized as a “reverse cache”
- Whenever you would have used squid, give this a look
- v1.0 released recently with relatively few features but a solid framework
- Work on 2.0 will start in January
- Written by Poul-Henning Kamp, famed FreeBSD contributor
- BSD licensed, work is being paid by a norwegian newspaper
- `http://varnish.projects.linpro.no/`

Fail-over tools

"move that IP"

Buy a “hardware load balancer”

- Generally *Quite Expensive*
  - (Except on eBay)
- Not appropriate (cost-wise) until you have MANY servers
- If the feature list fits it “Just Works”
- ... but when we are starting out, what do we use?

wackamole

- Simple, just moves the IP(s)
- Can embed Perl so you can run Perl functions when IPs come and go
- Easy configuration format
- Setup “groups of IPs”
- Supports Linux, FreeBSD and Solaris
- Spread toolkit for communication
- Easy to troubleshoot (after you get Spread working...)
- http://www.backhand.org/wackamole/

Heartbeat

- Monitors and moves services (an IP address is “just a service”)
- v1 has simple but goofy configuration format
- v2 supports all sorts of groupings, larger clusters (up to 16 servers)
- Uses /etc/init.d type scripts for running services
- Maybe more complicated than you want your HA tools
- http://www.linux-ha.org/

Carp + pfsync

- Patent-free version of Ciscos "VRRP" (Virtual Router Redundancy Protocol)
- FreeBSD and OpenBSD only
- Carp (moves IPs) and pfsync (synchronizes firewall state)
- (awesome for routers and NAT boxes)
- Doesn't do any service checks, just moves IPs around

mysql master master replication manager

- mysql-master-master tool can do automatic failover!
- No shared disk
- Define potential “readers” and “writers”
- List of “application access” IPs
- Reconfigures replication
- Moves IPs
- `http://code.google.com/p/mysql-master-master/`
  `http://groups.google.com/group/mmm-devel/`

Suggested Configuration

- Open/FreeBSD routers with Carp+pfsync for firewalls
- A set of boxes with perlbal + wackamole on static "always up" HTTP enabled IPs
  - Trick on Linux: Allow the perlbal processes to bind to all IPs (no port number tricks or service reconfiguration or restarts!)

```
echo 1 > /proc/sys/net/ipv4/ip_nonlocal_bind
```

or

```
sysctl -w net.ipv4.ip_nonlocal_bind=1
```

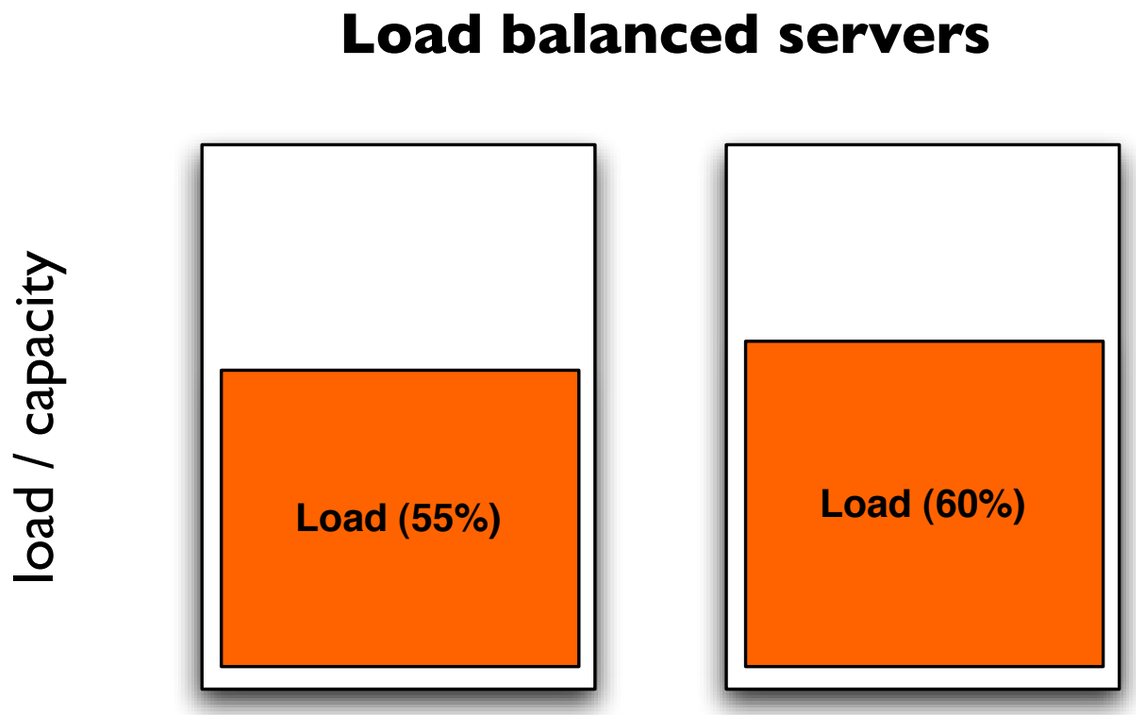
or

```
echo net.ipv4.ip_nonlocal_bind = 1 >> /etc/sysctl.conf
```

- Dumb regular http servers "behind" the perlbal ones
- wackamole for other services like DNS
- mmm for mysql fail-over

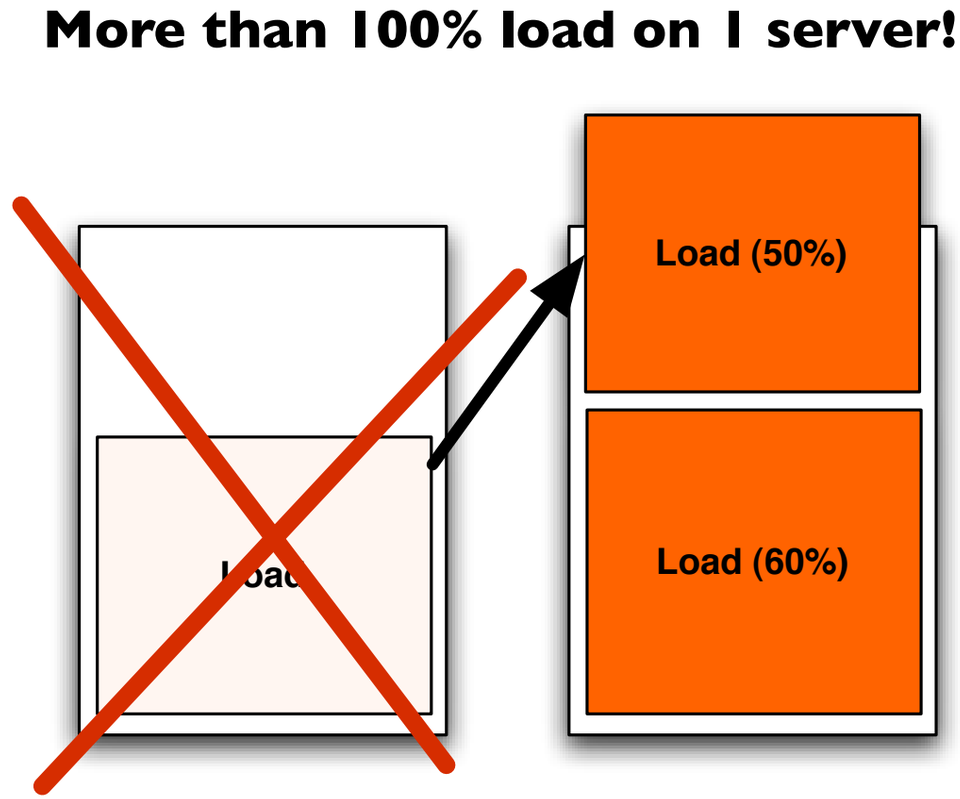
Redundancy fallacy!

- Don’t confuse load-balancing with redundancy
- What happens when one of these two fail?



Oops – no redundancy!

- Always have "n+1" capacity
- Consider have a "passive spare" (active/passive with two servers)
- Careful load monitoring!
  - Munin `http://munin.projects.linpro.no/`
  - MySQL Network
  - (ganglia, cacti, ...)



High availability
Shared storage

- NFS servers (for diskless servers, ...)
- Failover for database servers
- Traditionally either via fiber or SCSI connected to both servers
- Or NetApp filer boxes
- All expensive and smells like “the one big server”

Cheap high availability storage with DRBD

- Synchronizes a block device between two servers!
- “Network RAID1”
- Typically used in Active/Primary-Standby/Secondary setup
- If the active server goes down the secondary server will switch to primary, run fsck, mount the device and start the service (MySQL / NFS server / ...)
- The upcoming v0.8.0 can do writes on both servers at once – “shared disk semantics” (you need a filesystem on top that supports that, OCFS, GFS, ...)

Disaster Recovery

- Separate from “fail-over”
  (no disaster if we failed-over...)
- “All the “redundant” The network cables melted”
- “The datacenter got flooded”
- “The grumpy sysadmin sabotaged everything before he left”

Disaster Recovery Planning

- You won't be back up in 2 hours, but plan so you quickly will have an idea how long it will be
- Have a status update site / weblog
- Plans for getting hardware replacements
- Plans for getting running temporarily on rented "dedicated servers" (ev1servers, rackspace, ...)
- And ....

Backup your databse!

- Binary logs!
  - Keep track of "changes since the last snapshot"
- Use replication to Another Site
  (doesn't help on "for $table = @tables { truncate $table }")
- On small databases use mysqldump
  (or whatever similar tool your database comes with)

Backup Big Databases

- LVM snapshots (or ibbackup from Innobase / Oracle)
- InnoDB:
  Automatic recovery! (ooh, magic)
- MyISAM:
  Read Lock your database for a few seconds before making the snapshot
  (on MySQL do a "FLUSH TABLES" first (which might be slow) and then a "FLUSH TABLES WITH READ LOCK" right after)
- Sync the LVM snapshot elsewhere
- And then remove the snapshot!
- Bonus Optimization:
  Run the backup from a replication slave!

System Management

All Automation All The Time

or

How to manage 200 servers in your spare-time

Resource management

- If possible, only run one service per server (makes monitoring/ managing your capacity much easier)
- Balance how you use the hardware
  - Use memory to save CPU or IO
  - Balance your resource use (CPU vs RAM vs IO)
  - Extra memory on the app server? Run memcached!
  - Extra CPU + memory? Run an application server in a Xen box!
- Don't swap memory to disk. Ever.

Netboot your application servers!

- Definitely netboot the installation (you'll never buy another server with a tedious CD/DVD drive)
  - RHEL / Fedora: Kickstart + puppet = from box to all running in ~10 minutes
- Netboot application servers
- FreeBSD has awesome support for this
- Debian is supposed to
- Fedora Core ~~7~~ 8 looks like it will (RHEL5uX too?)

Keep a maintainable system!

- Configuration in SVN (or similar)
- Use tools to keep system configuration in sync
- Upcoming configuration management (and more) tools!
  - csync2 (librsync and sqlite based sync tool)
  - puppet (central server, rule system, ruby!)

Keep software deployments easy

- Make upgrading the software a simple process
- Script database schema changes
- Keep configuration minimal
  - Servername (“www.example.com”)
  - Database names (“userdb = host=db1;db=users”;...”
  - If there’s a reasonable default, put the default in the code (for example)
  - “deployment_mode = devel / test / prod” lets you put reasonable defaults in code

Easy software deployment 2

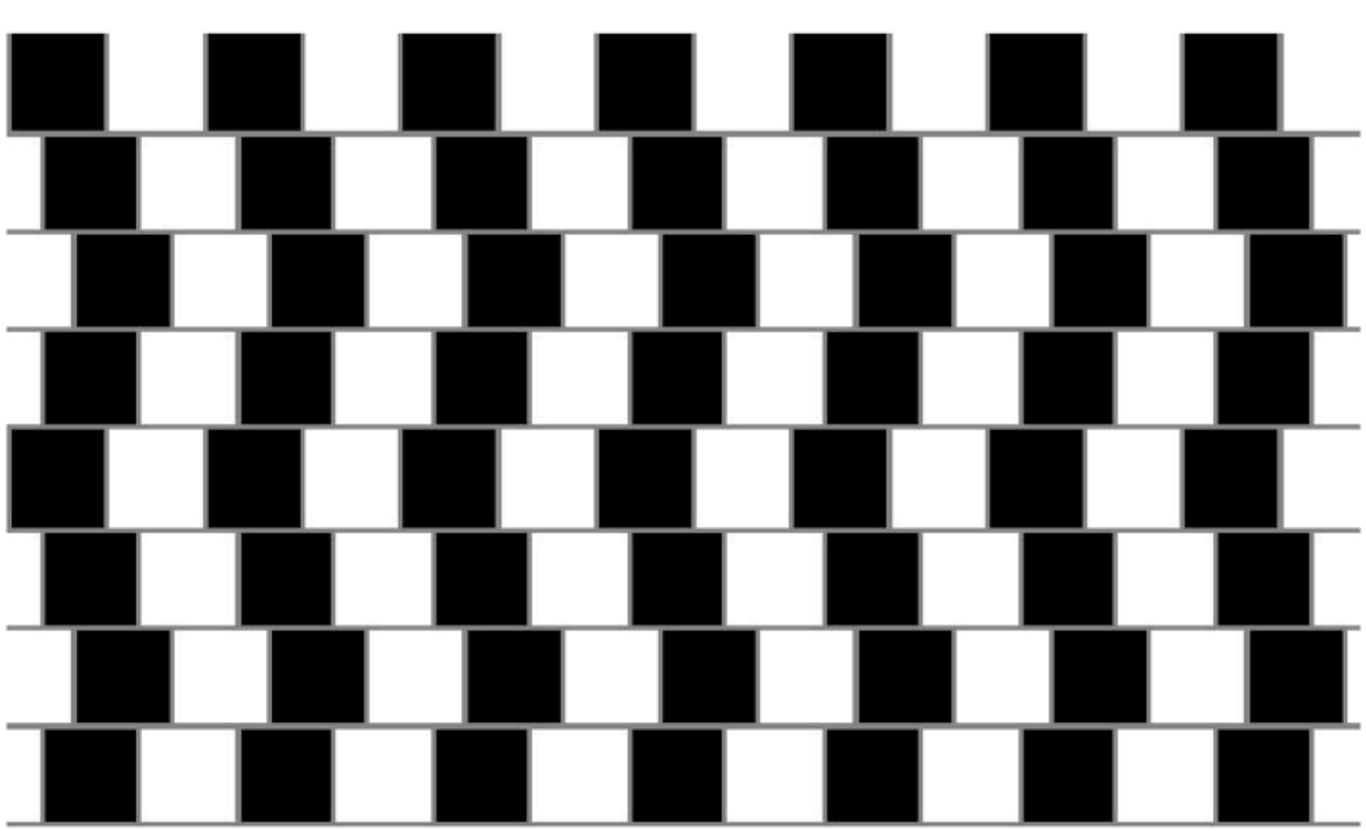
- How do you distribute your code to all the app servers?
- Use your source code repository (Subversion etc)! (tell your script to svn up to http://svn/branches/prod revision 123 and restart)
- .tar.gz to be unpacked on each server
- .rpm or .deb package
- NFS mount and symlinks
- No matter what: Make your test environment use the same mechanism as production and:
  Have it scripted!

Use your resources wisely

don't implode when things run warm

Work in parallel

- Split the work into smaller (but reasonable) pieces and run them on different boxes
- Send the sub-requests off as soon as possible, do something else and then retrieve the results



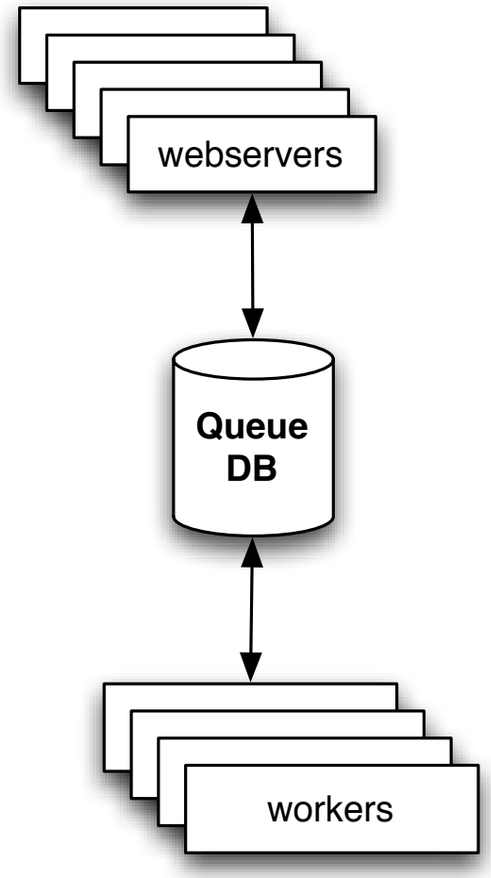
Are the horizontal lines parallel or do they slope?

Job queues

- Processing time too long for the user to wait?
- Can only process N requests / jobs in parallel?
- Use queues (and external worker processes)
- IFRAMEs and AJAX can make this really spiffy (tell the user "the wait time is 20 seconds")

Job queue tools

- Database "queue"
  - Dedicated queue table or just `processed_on` and `grabbed_on` columns
  - Webserver submits job
  - First available "worker" picks it up and returns the result to the queue
  - Webserver polls for status
- Other ways...
  - gearman
    `http://www.danga.com/gearman/`
  - Spread
  - MQ / Java Messaging Service(?) / ...
  - Upcoming CPAN module from Perrin Harkins



Log http requests

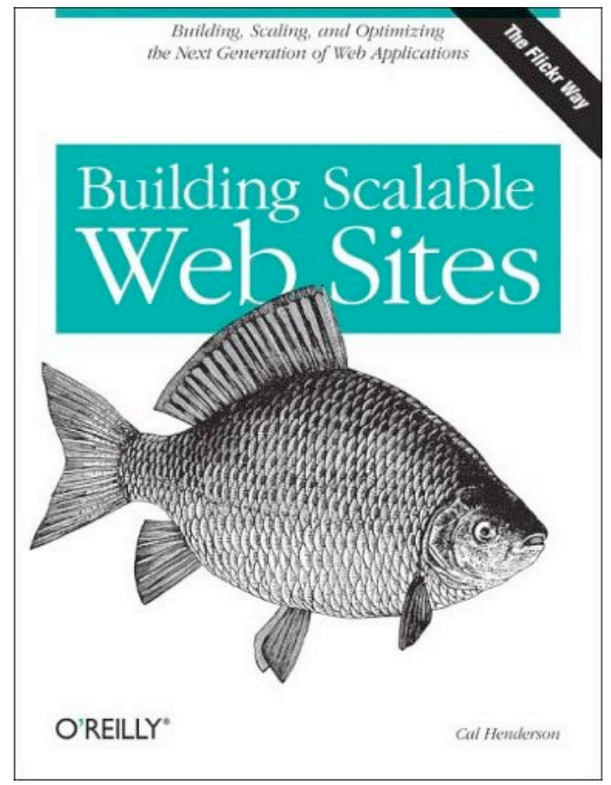
- Log slow http transactions to a database
  time, **response_time**, uri, remote_ip, user_agent, request_args, user, svn_branch_revision, log_reason (a "SET" column), ...
- Log 2% of all requests!
- Log all 4xx and 5xx requests
- Great for statistical analysis!
  - Which requests are slower
  - Is the site getting faster or slower?
- Time::HiRes in Perl, microseconds from `gettimeofday` system call

remember

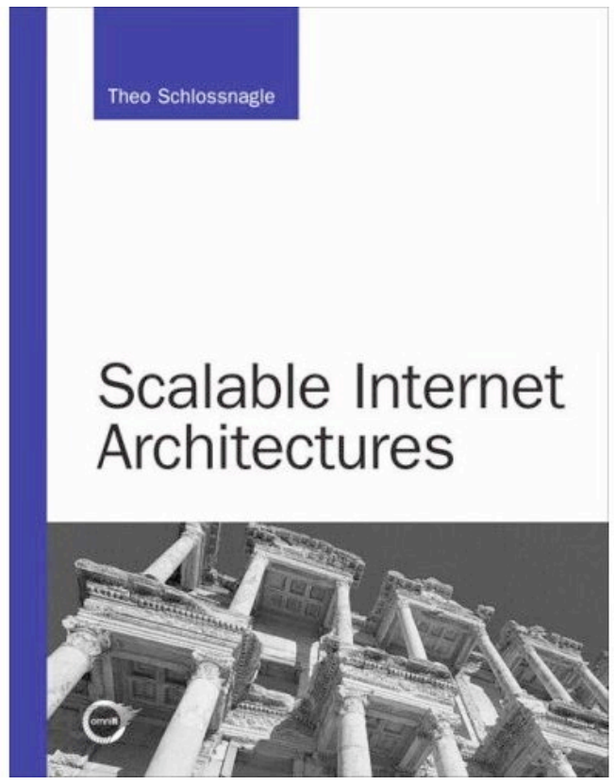
Think Horizontal!

Books!

- "Building Scalable Web Sites" by Cal Henderson of Flickr fame
  - Only $26 on Amazon! (But it's worth the $40 in from your local bookstore too)



- "Scalable Internet Architectures" by Theo Schlossnagle
  Teaching concepts with lots of examples
- "High Performance Web Sites: Rough Cuts Version" by Steve Souders
  Front end performance (not out yet)



Thanks!

- Direct and indirect help from ...
  - Cal Henderson, ~~Flickr~~ Yahoo!
  - Brad Fitzpatrick, ~~LiveJournal~~ SixApart
  - Kevin Scaldeferri, ~~Overture~~ Yahoo!
  - Perrin Harkins, Plus Three
  - David Wheeler
  - Tom Metro
  - Tim Bunce
  - Graham Barr
  - Vani Raja Hansen

– The End –

Questions?

Thank you!

More questions? Comments? Need consulting?

ask@develooper.com

`http://develooper.com/talks/`

(link to the second half of the tutorial on that page!)