Real World Web Scalability

Ask Bjørn Hansen
Developer LLC

O'REILLY
OPENSOURCE
CONVENTION
AUGUST 1-5 2005 • OREGON CONVENTION CENTER • PORTLAND, OREGON

Hello.

- 28 brilliant methods to make your website keep working past $goal requests/transactions/sales per second/hour/day
    - Requiring minimal extra work! (or money)
    - Applicable to most languages and platforms!
- All for the low low fee of $49.95!!!
    - Only available *TODAY* from 1-800-SCALE!

- The Single Most Important Thing:

- # Think Horizontal!

- Not just many servers side by side, the system has to be designed horizontal

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

- The single *least* important thing
- Micro optimizations and other implementation details
- Save money on servers
- But it doesn't make you scale!
- And this talk is all about scaling

Benchmarking techniques

- Scalability isn't the same as processing time
  - Not “how fast” but “how many”
  - Force, not speed. Amps, not voltage
  - Test *scalability*, not just performance
- Use a realistic load
- Test with "slow clients"

Vertical scaling

- “Get a bigger server”
- “Use faster CPUs”
- Can only help so much (with bad scale/$ value)

Vertical scaling

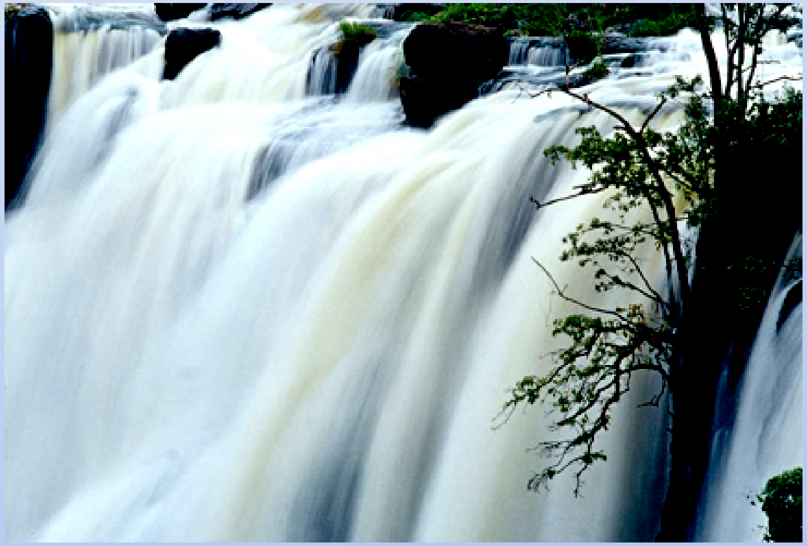
- A server twice as fast is more than twice as expensive
- That only allows for small time scaling
- Even super computers are horizontally scaled now

Typical scaling bottlenecks

- Single machine “app server”
  - Put your application horizontally
- Databases
- Session stores
- Network equipment

It’s the architecture

- Good to great ...
  - Implementation, scale a few times
  - Architecture, scale dozens or hundreds of times
- Get the big picture right first, do micro optimizations later

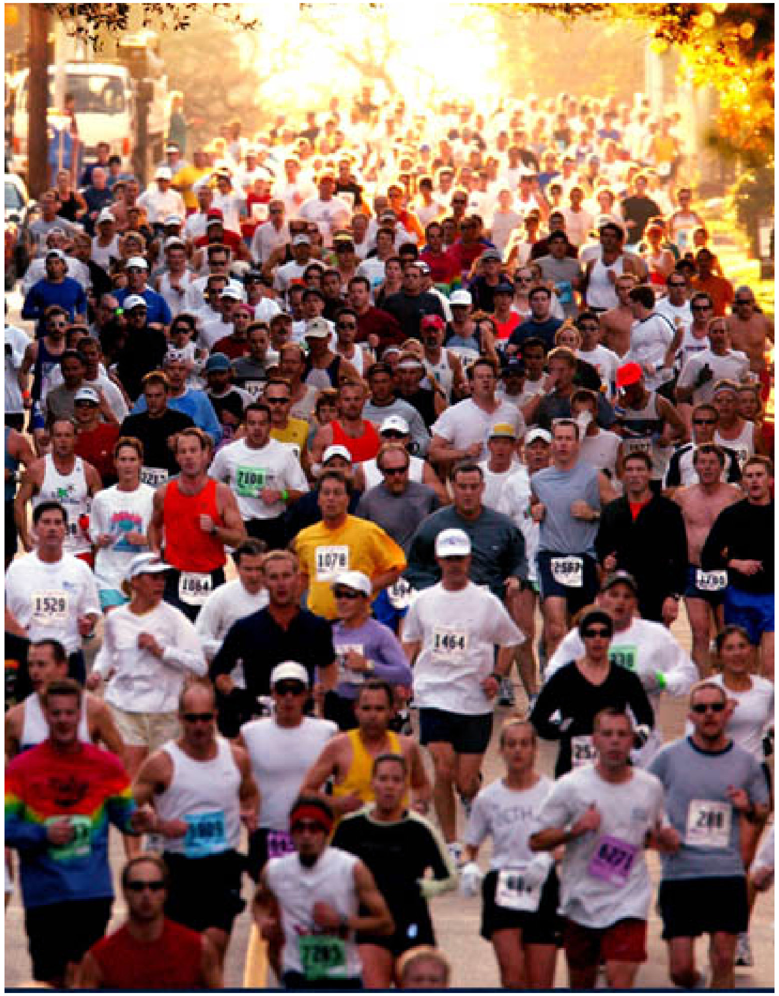


Scalable Application Servers

Don’t paint yourself into a corner from the start

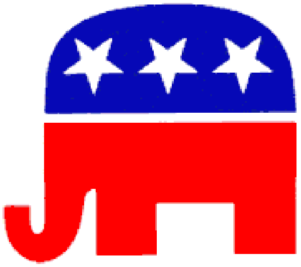
Run Many of Them

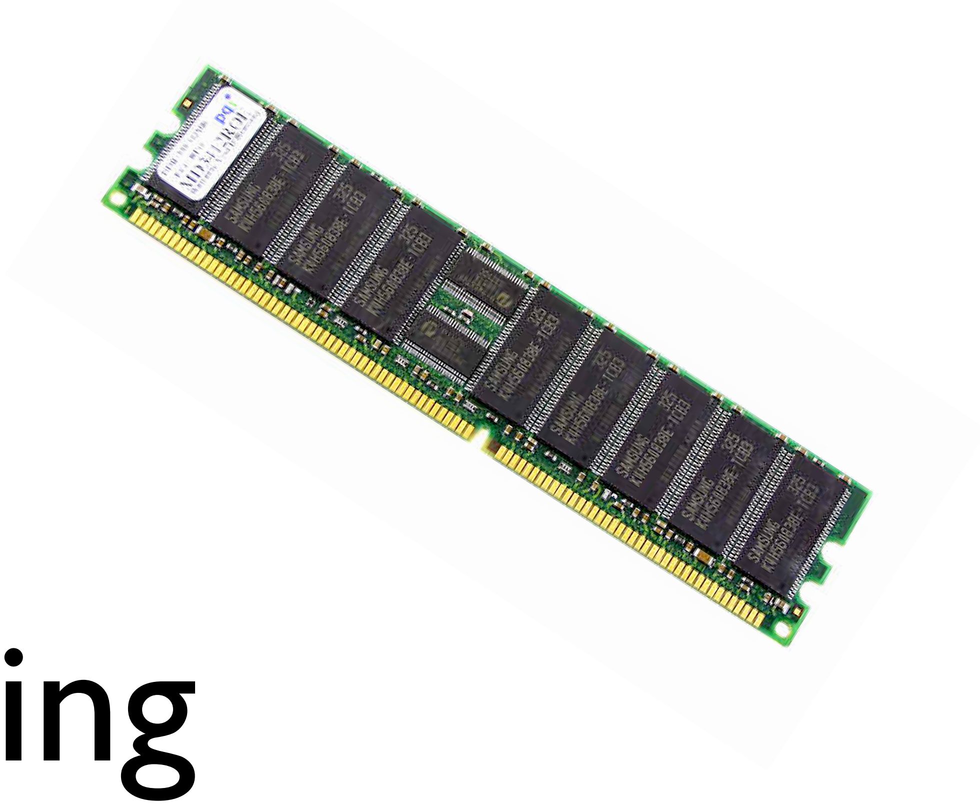
- For your application...
- Avoid having *The Server* for anything
- Everything should (be able to) run on any number of boxes



Stateless vs Stateful

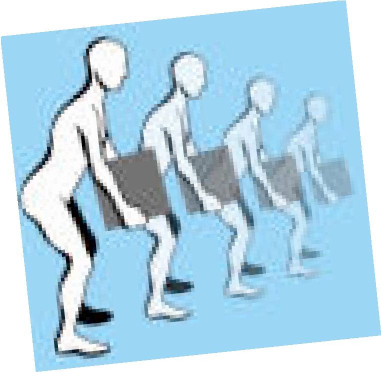
- Don’t keep state within the application server (or at least be Really Careful)
- Do you use PHP or mod_perl (or something else that’s running in Apache HTTPD)?
    - You get that for free! (usually)
    - “Shared Nothing”





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
  - or something dynamic per user

Cache **full** pages

- Front end cache (**mod_cache, squid**, ...)
  - Set Expires header to control cache times
- ***or*** Rewrite rule to generate page if the cached file doesn't exist

  -
    ```
    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")*

Cache **full** pages 2.0

- Cache full output **in the application**
- Include cookies etc etc in the "cache key"
- Fine tuned application level control
- The most flexible
  - "use cache when this, not when that"

Cache **partial** pages

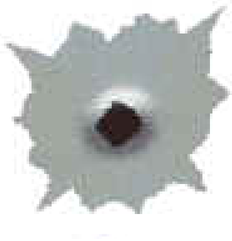
- Pre-generate static page "snippets"
  - Have the handler just assemble pieces ready to go
- Cache little page snippets (say the sidebar)
- Be careful, easy to spend more time managing the cache snippets than you save!
- "Regexp" dynamic content into an otherwise cached page

Cache **data**

- Cache data that's slow to query, fetch or calculate
- Generate page from the cached data
- This moves load to cache servers
  - (For better or worse)

Cache **hit-ratios**

- Start with things you hit all the time
- Look at database logs
- Don’t cache if you’ll spend more energy 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



Caching Tools

Where to cache?

(a couple of not so great ideas)

- 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

- Write into one or more cache tables
- 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

memcached

- LiveJournal's distributed caching system
  (also used at slashdot, wikipedia, etc etc)
- memory based
- run it on boxes with free memory
- no "master"
- simple protocol
  - perl, java, php, python, ruby, ...
- Linux 2.6 (epoll) or FreeBSD (kqueue)

Database scaling

How to avoid buying that gazillion dollar Sun box

~$4,000,000

~$5,000
*(= **800** for $4M!)*

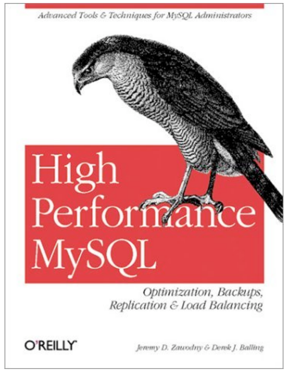
Be Simple

- Use MySQL
  - It’s fast and it’s easy to manage and tune
  - Easy to setup development environments

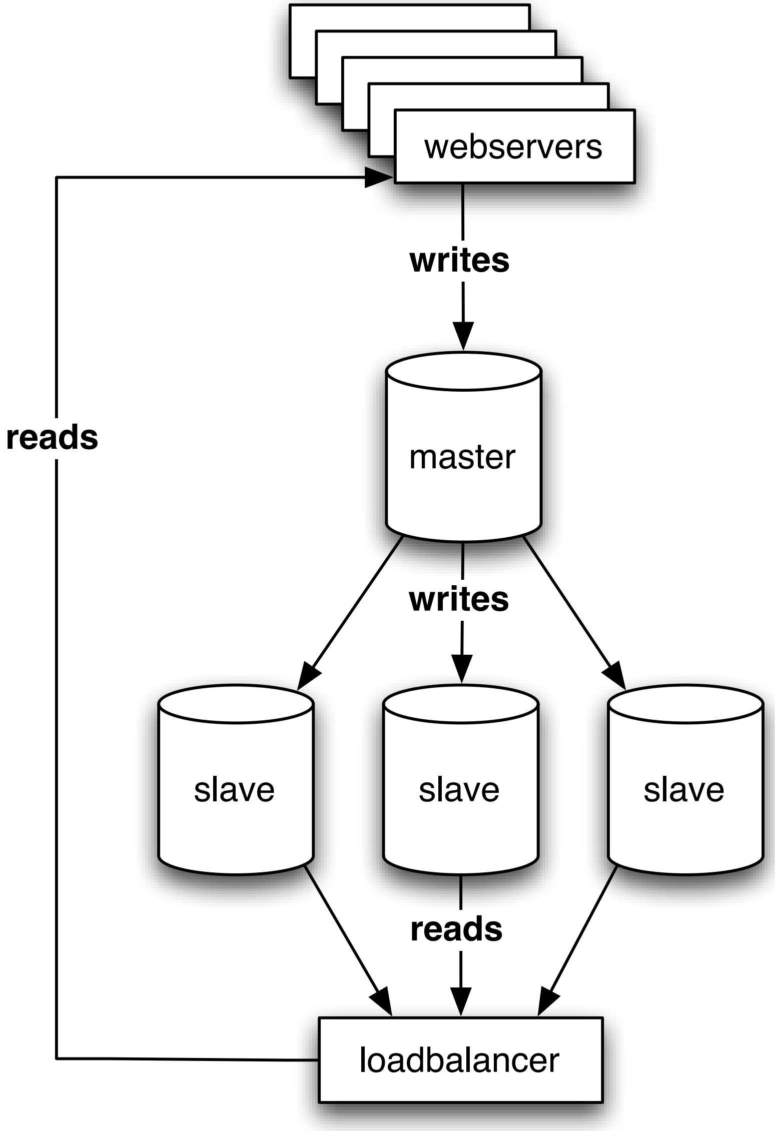
- *PostgreSQL is fast too :-)*

Basic Replication

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

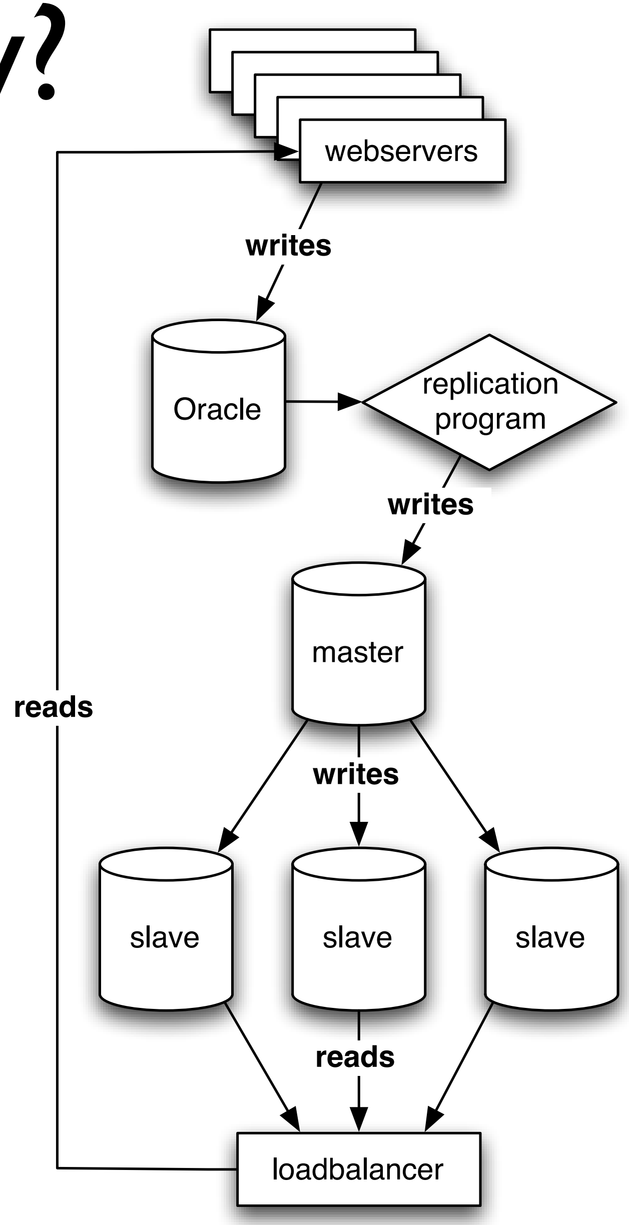

Advanced Tools & Techniques for MySQL Administrators
High Performance MySQL
Optimization, Backups, Replication & Load Balancing
O'REILLY
Jeremy D. Zawodny & Derek J. Balling

Lots more details in
"High Performance MySQL"



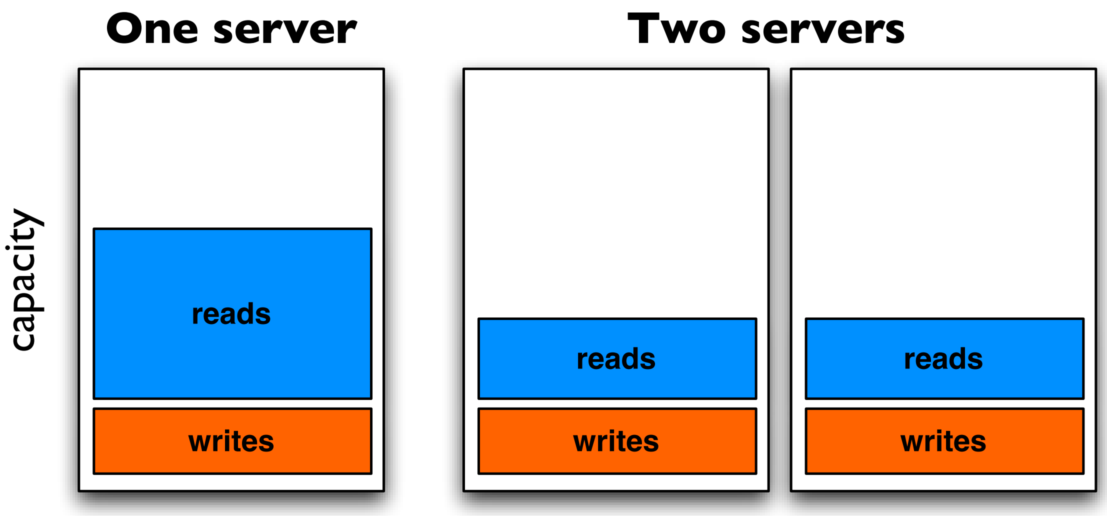
Running Oracle now?

- Replicate from Oracle to a MySQL cluster
- Use triggers to keep track of changed rows in Oracle
- Copy them to the MySQL master server with a replication program



Replication Scaling – Reads

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

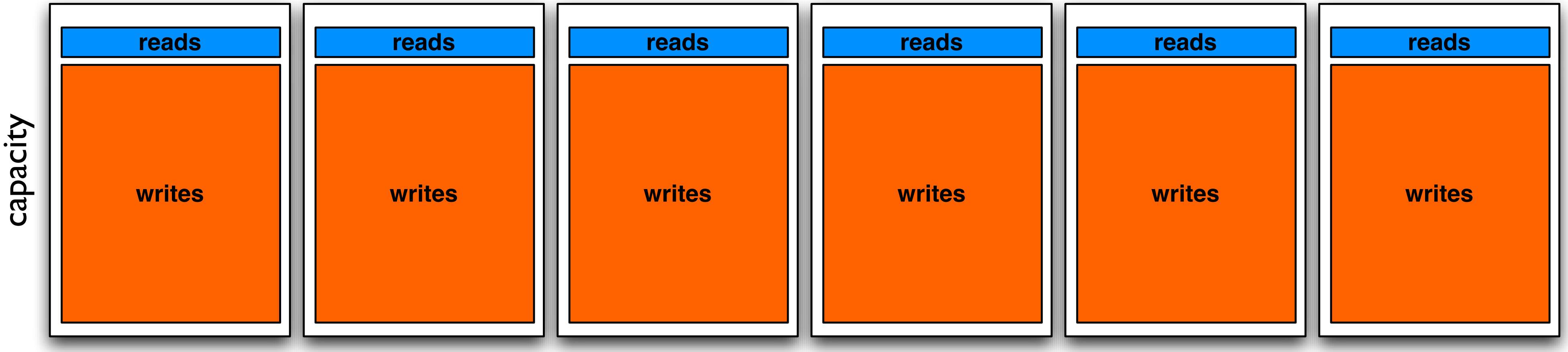


(thanks to Brad Fitzpatrick!)

Replication Scaling – Writes

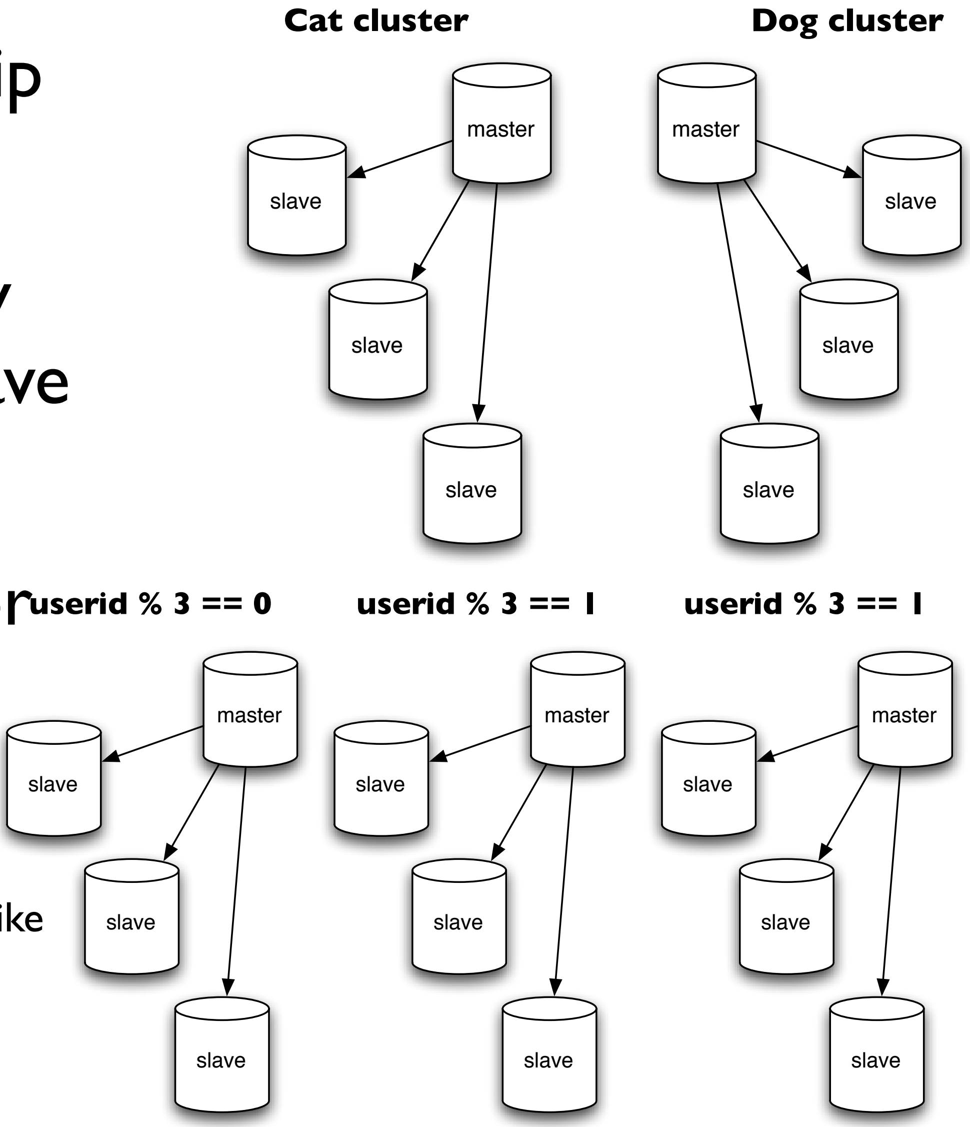
(aka when replication sucks)

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



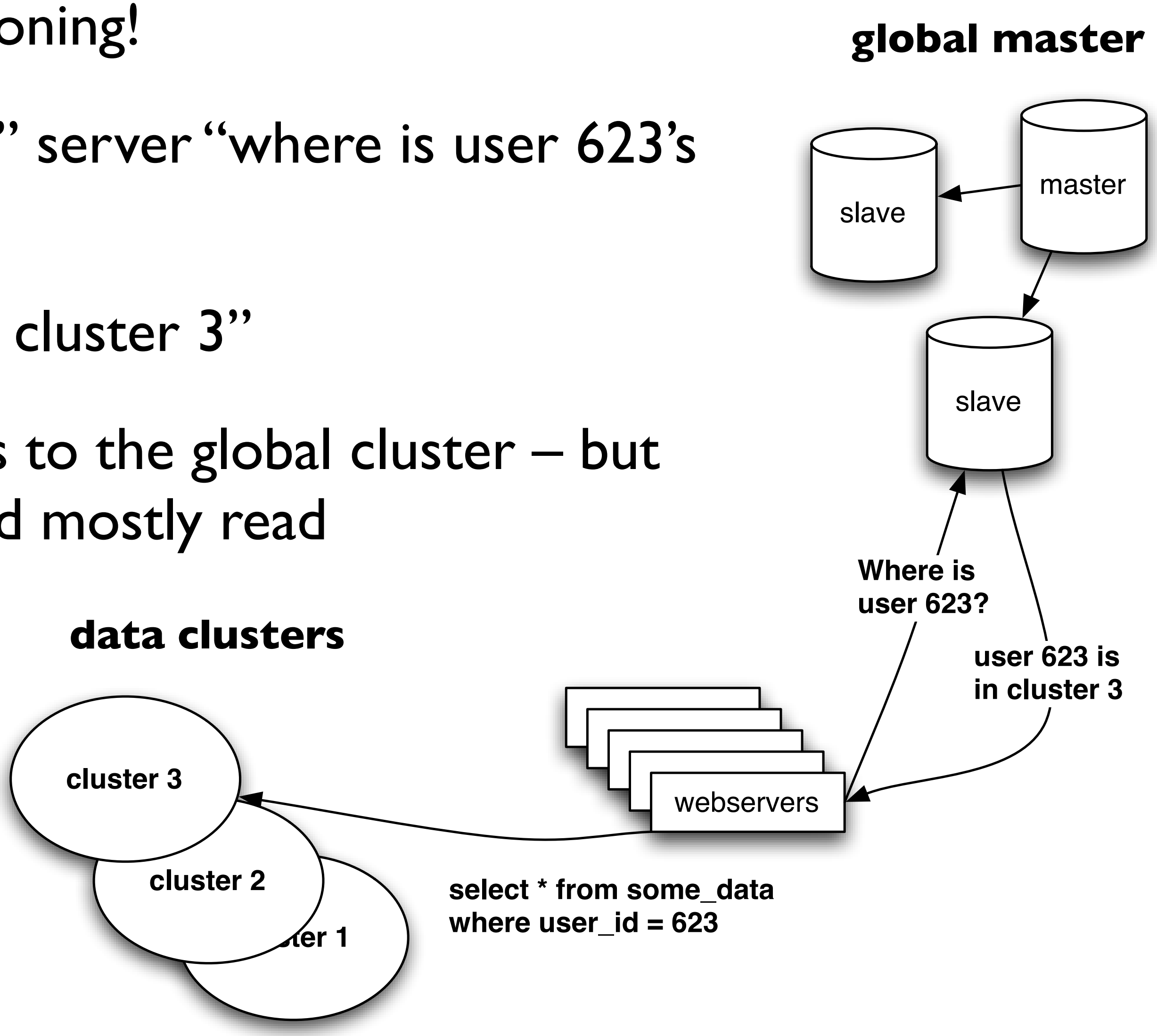
Partition your data

- 99% 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
- Split your data up in different clusters (don't do it like it's done in the illustration)



Cluster data with a master server

- Flexible partitioning!
- ask the "global" server "where is user 623's data?"
- "user 623 is at cluster 3"
- Lots of queries to the global cluster – but very simple and mostly read



Preload, -dump and -process

- Let the servers do as much as possible without touching the database directly
  - Data structures in memory – ultimate 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
- Denormalized summary tables
  - Just tell the DBA to bugger off

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 say 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!

Stored Procedures Evil

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

Reconsider Persistent DB Connections

- DB connection = thread = memory
- With lots of caching you might not need the main database that often
- MySQL connections are fast
- (unless you use Oracle!)
  - Commercial connection pooling products
- Newer glibc make MySQL use less memory

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 extra procedures to validate and process data maybe not so good
- Don’t be afraid of de-normalized data (call them summary tables and the DBAs won’t notice)

Sessions

All the things that you shouldn't put there …

WARNING: Confusion lies ahead.

Session storage

- Session data should be light
- Put the user_id in the session
  - Not the user record
- Put the shopping_cart_id in the session
  - Not the contents of the cart!

The Golden Session Balance

- If it’s important save it structured in a “proper” database table
- If it’s not important save it in a cookie or memcached or some such

Use cookies

- Make everything you want to store in the session fit in a cookie or three
- You shouldn’t put much in the session anyway!
- Keep things stateless on the server

Safe cookies

- Worried about manipulated cookies?
- Use checksums and timestamps to validate them
  - `cookie=1/value/1123157440/ABCD1234`
  - `cookie=1/key:value/ts:1123.../EFGH9876`
- Encrypt them if you must, but you shouldn't put something secret in the session anyway!

Use your resources wisely

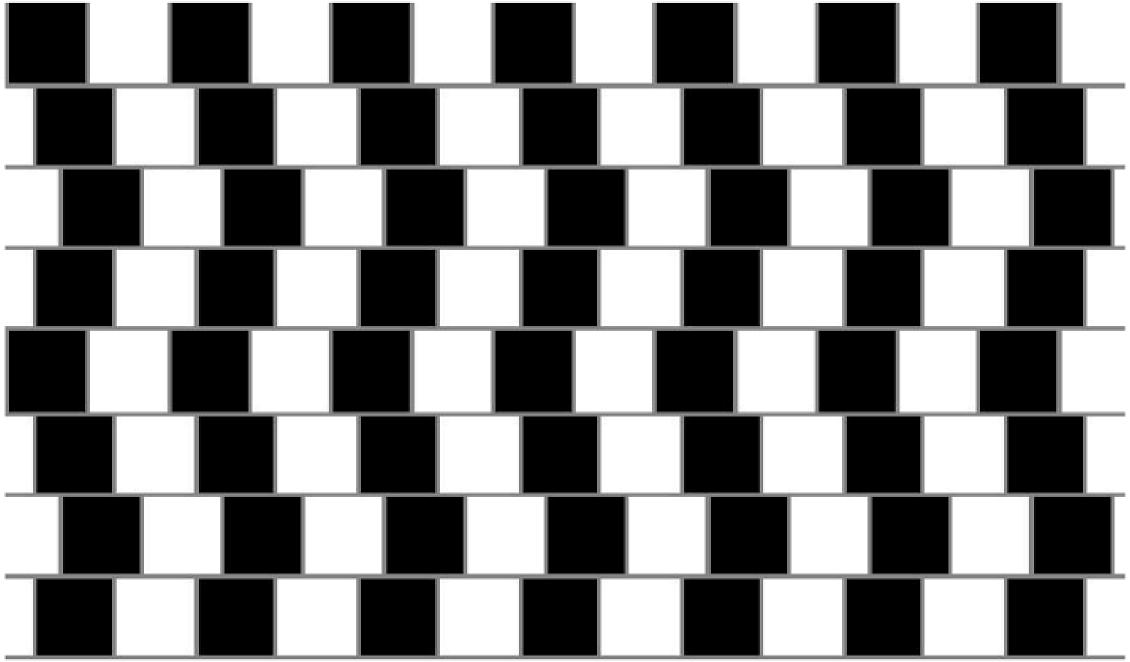
don't implode when things run warm

Resource management

- Balance how you use the hardware
  - Use memory to save CPU or IO
  - Balance your resource use (CPU vs RAM vs IO)
- Don’t swap memory to disk. Ever.

Parallelize work

- 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

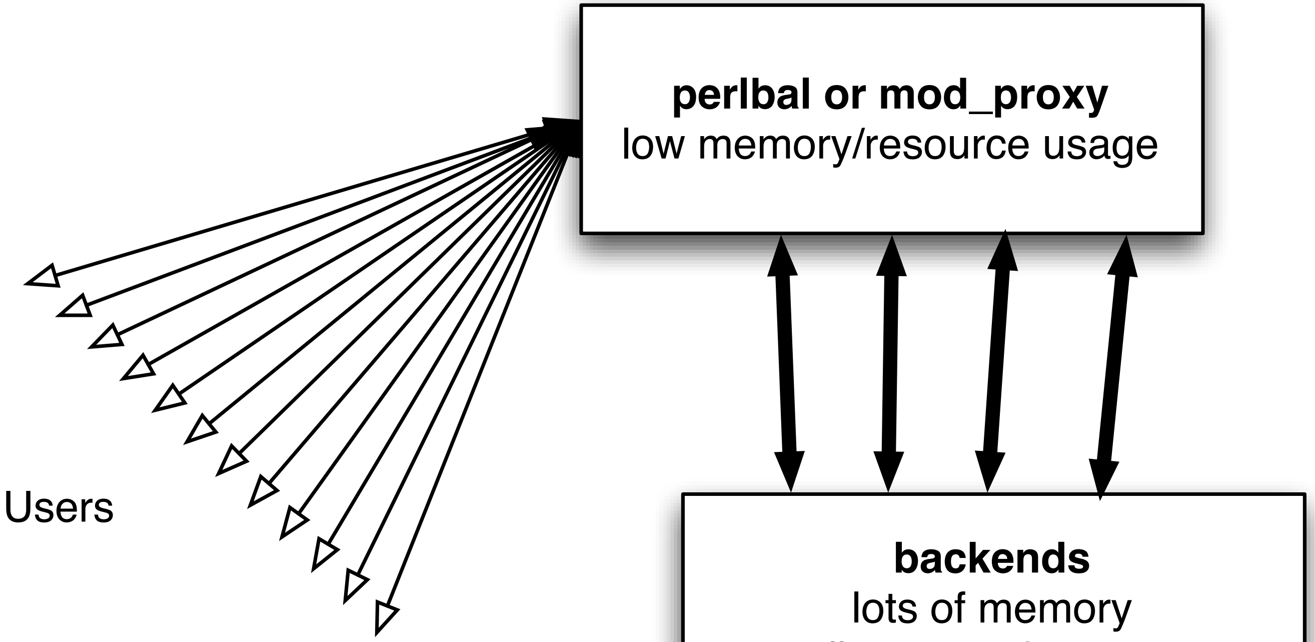


Are the horizontal lines parallel or do they slope?

Use light processes for light tasks

- Thin proxies servers or threads for "network buffers"
- Goes between the user and your heavier backend application
- httpd with mod_proxy / mod_backhand
  - perlbal
    – new & improved, now with vhost support!
  - 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**

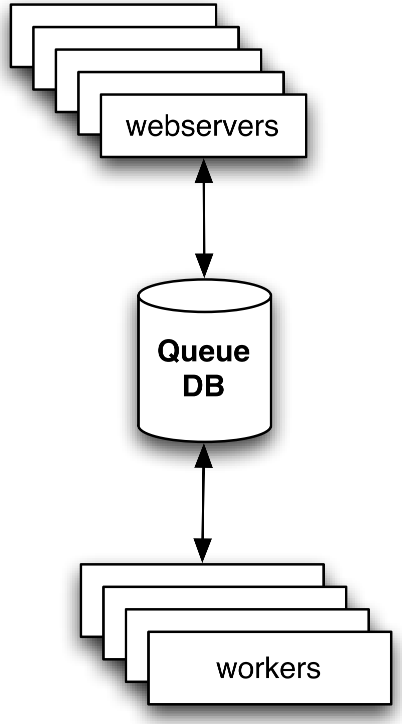
-
  ```
  ProxyPreserveHost On
  <VirtualHost *>
    ServerName combust.c2.askask.com
    ServerAlias *.c2.askask.com
    RewriteEngine on
    RewriteRule (.*) http://localhost:8230$1 [P]
  </VirtualHost>
  ```
- Easy to have different "backend environments" on one IP
- Backend setup (Apache 1.x)
  ```
  Listen 127.0.0.1:8230
  Port 80
  ```

Job queues

- Processing time too long for the user to wait?
- Can only do N jobs in parallel?
- Use queues (and an external worker process)
- IFRAMEs and AJAX can make this really spiffy

Job Queues

- Database "queue"
  - Webserver submits job
  - First available "worker" picks it up and returns the result to the queue
  - Webserver polls for status
- Other ways...
  - gearman
  - Spread
  - MQ / Java Messaging Service(?) / ...



remember

Think Horizontal!

– The End –

Questions?

Thank you!

ask@perl.org
ask@develooper.com

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