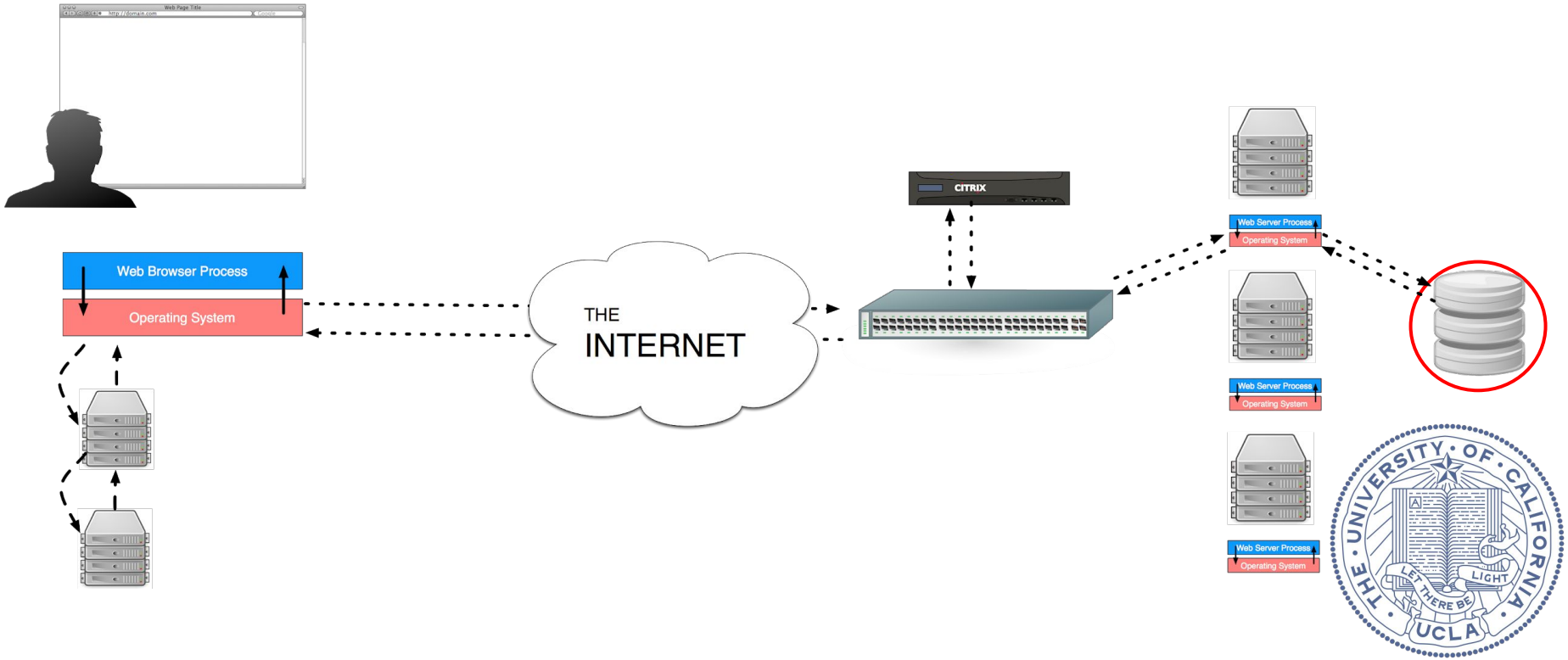
CS 188

Scalable Internet Services

Andrew Mutz

October 31, 2019

For Today


Web Browser Process
Operating System
THE
INTERNET
CITRIX
Web Server Process
Operating System
Web Server Process
Operating System
Web Server Process
Operating System
THE UNIVERSITY OF CALIFORNIA
LET THERE BE LIGHT
UCLA

Motivation

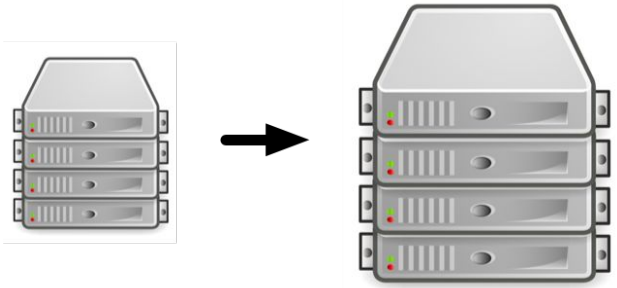
You've got your application running on EC2. It is becoming increasingly popular and performance is degrading.

What do you do?

Motivation

You've got your application running on EC2. It is becoming increasingly popular and performance is degrading.

What do you do?



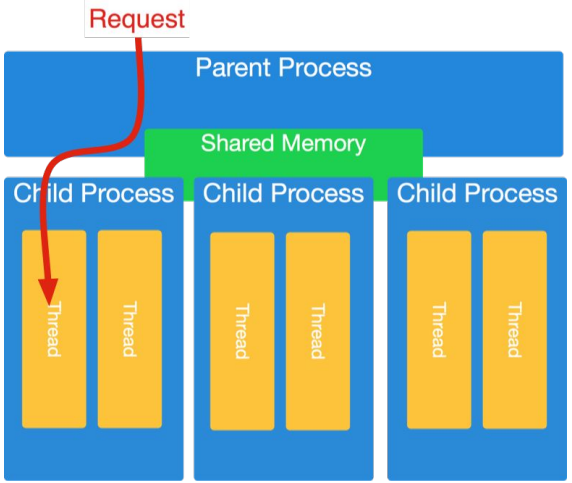
You've increased instance sizes, and that buys some time. But as popularity grows, there are no larger instances to buy.

Motivation

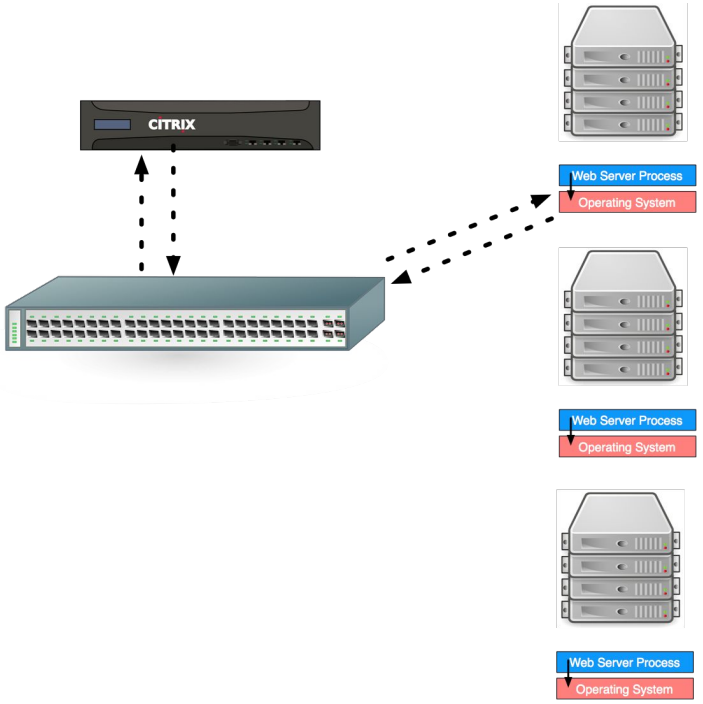
You've got your application running on EC2. It is becoming increasingly popular and performance is degrading.

What do you do?

You've deployed application servers that can serve many requests simultaneously, and that helped. But as popularity grows, it's not enough.



Motivation

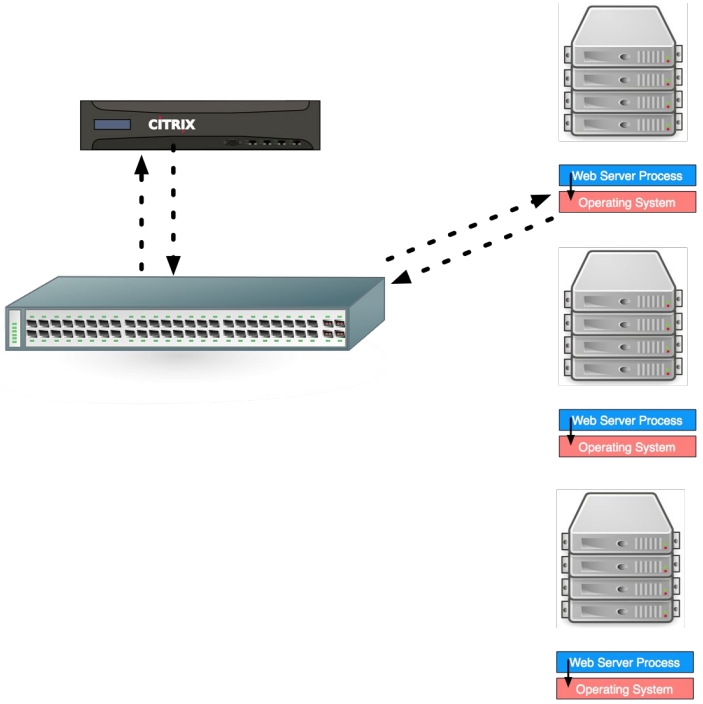


You've got your application running on EC2. It is becoming increasingly popular and performance is degrading.

What do you do?

You've set up a load balancer and spread load across many servers. But as popularity grows, it's not enough.

Motivation



You've got your application running on EC2. It is becoming increasingly popular and performance is degrading.

What do you do?

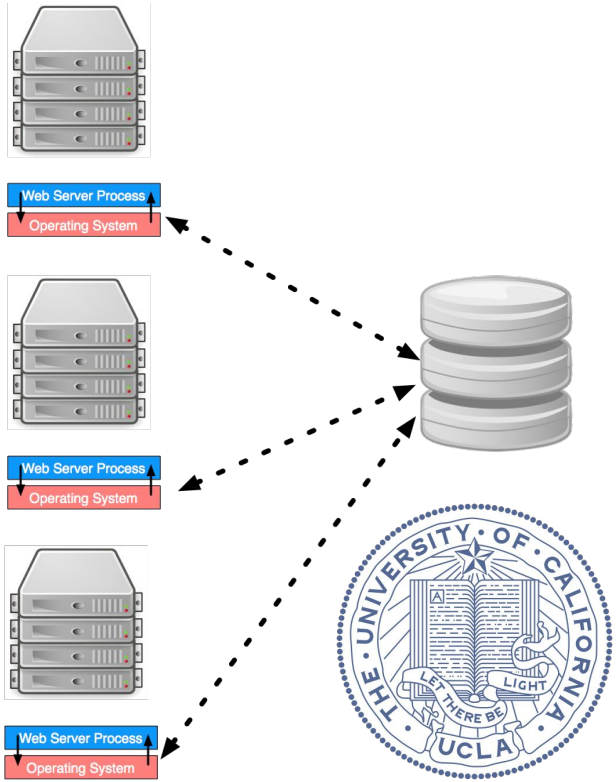
You've set up HTTP caching correctly so unnecessary requests never happen. You're caching heavyweight database operations. But as popularity grows, it's not enough.

Motivation

You've got your application running on EC2. It is becoming increasingly popular and performance is degrading.

What do you do?

As you keep adding application servers, your database struggling to keep up. You've used EXPLAIN to detect missing indices and add them. You've detected slow queries and fixed them. But as popularity grows, it's not enough.

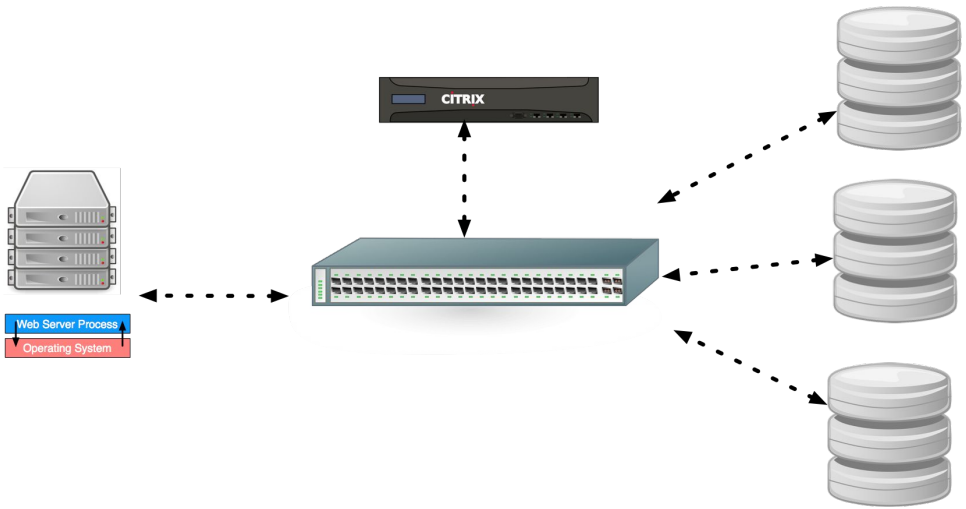


Motivation

Your relational database is still on one machine and that machine is having trouble keeping up with increased load.

So what do we do?

Motivation

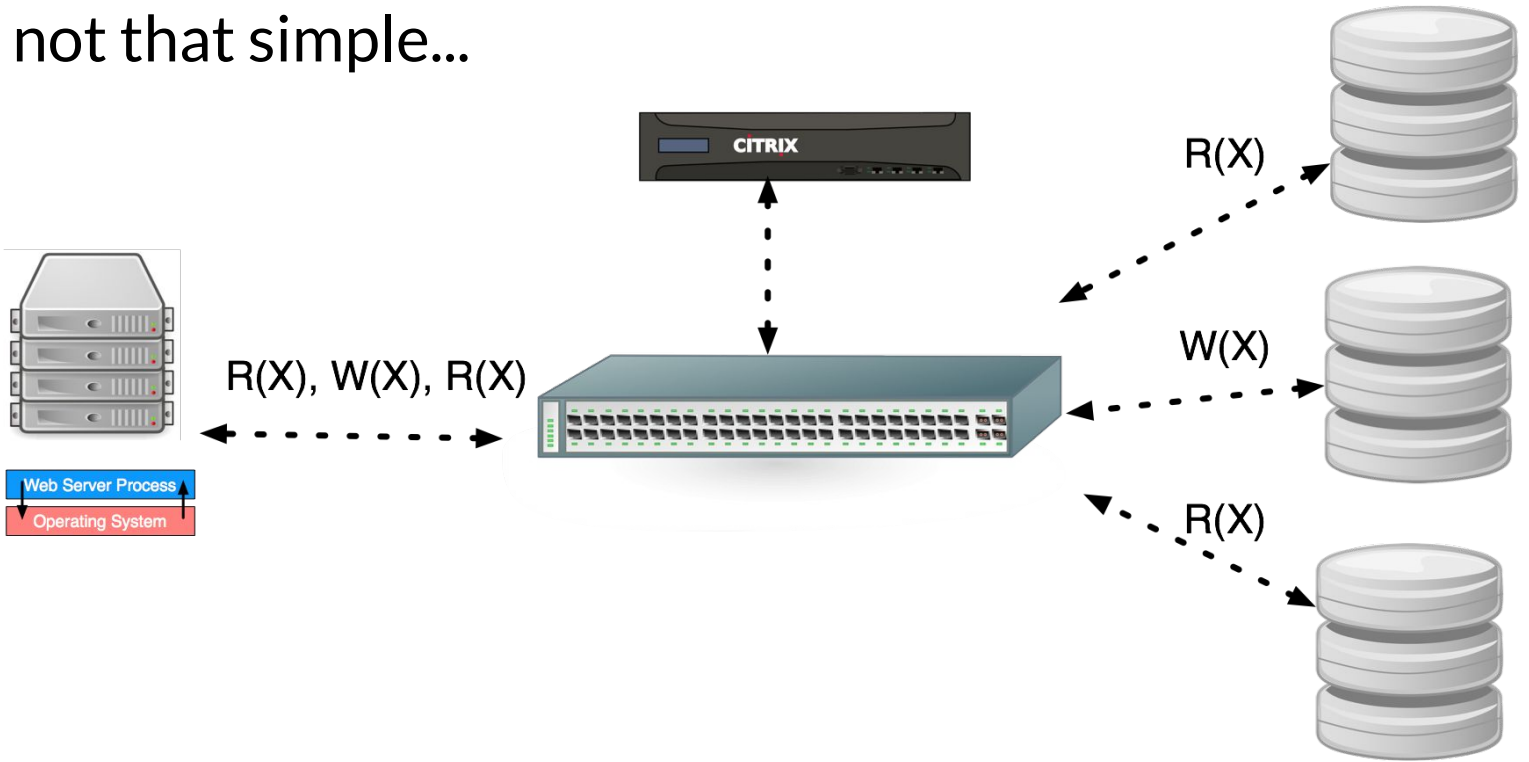


In the past we've been able to scale horizontally by adding machines.

Can we just put a load balancer in front of the database and spread load across many databases?

Motivation

It’s not that simple...



Motivation

Relational databases are hard to scale.

In future lectures, we will discuss scaling the data layer using non-relational data stores.

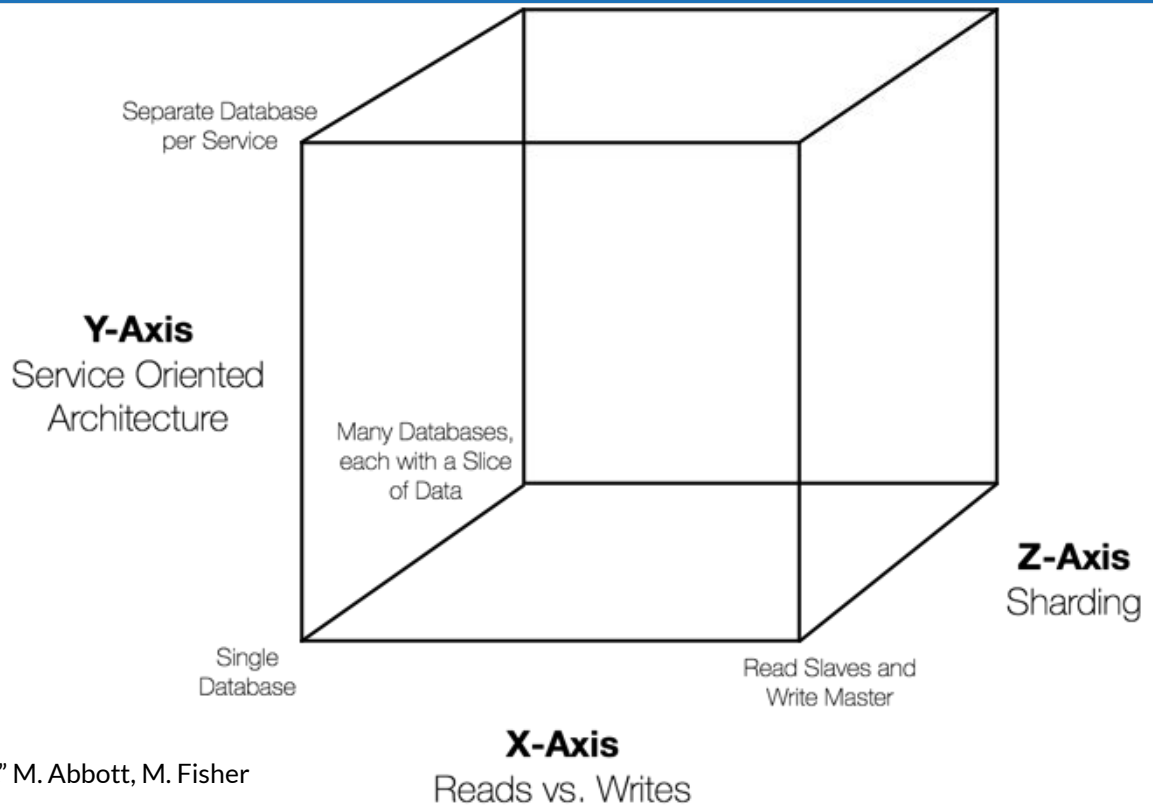
Today we will talk about what we **can** do to scale with relational databases.

Today's Agenda

Each technique breaks up a RDBMS in a different manner:

- Sharding
- Service Oriented Architectures
- Distinguishing Reads from Writes

AKF Cube



Source: “Scalability Rules,” M. Abbott, M. Fisher

Sharding

We want to take a single database and break it up into multiple databases

- We still want everything to work.

We may have relations that our application expects to JOIN across

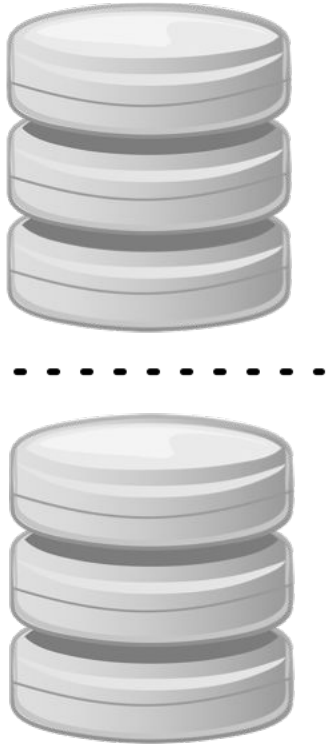
Sharding

Shard, noun:

- a piece of broken ceramic, metal, glass, or rock, typically having sharp edges.
- synonyms:
  - fragment, sliver, splinter, shiver, chip, piece, bit, particle

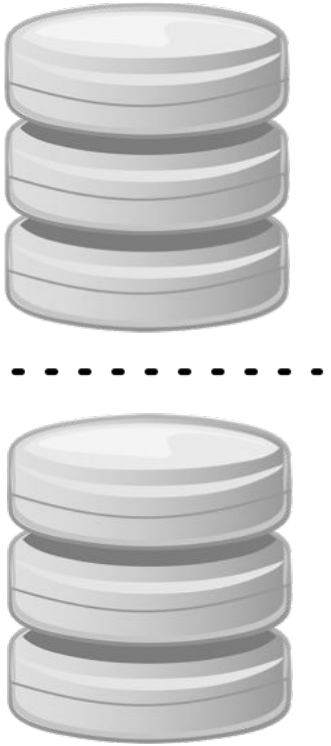
Also referred to as “horizontal partitioning” or “podding”.

Sharding

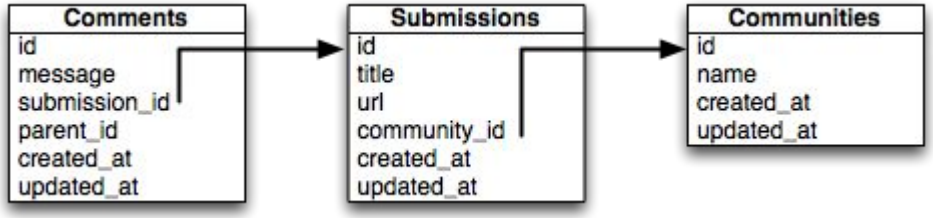


- To *Shard* a database is to find some partition that can divide your data into groups that are not related.
- When they have been separated into these shards, you will never JOIN across them.
- If you ever need to perform operations that cross shards, you must do it at the application level

Sharding



Partitioning your database in a way you will never JOIN across is easier said than done. Example:



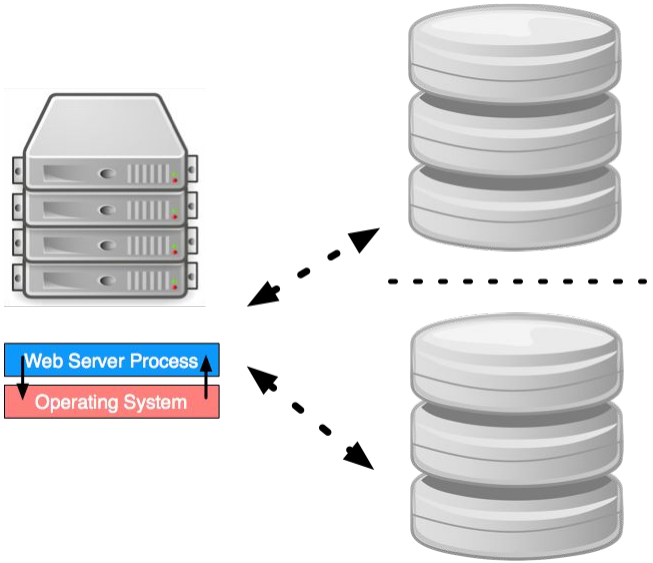
Sharding

Any particular DB join connects a small part of your database, but transitively they can connect everything.

Demo App:

- Any comment is only related to parent, children, and submission.
- Submissions can relate to each other through communities.
- Transitively, someone could create joins across these relations.

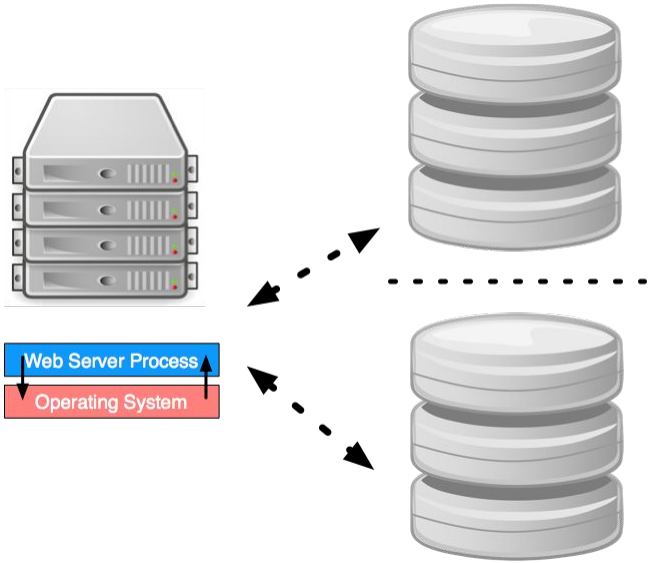
Sharding



When we shard we are taking data of the same type and splitting it. Ex:

- Splitting our Comments between multiple databases
- Not sharding: keeping comments and submissions in separate DBs.

Sharding



If we have different groups of data sitting in different databases, we need some sort of mapping mechanism to figure out where things are.

Where & how should we do this?

Sharding

At the app-server layer?

- How would we implement this?

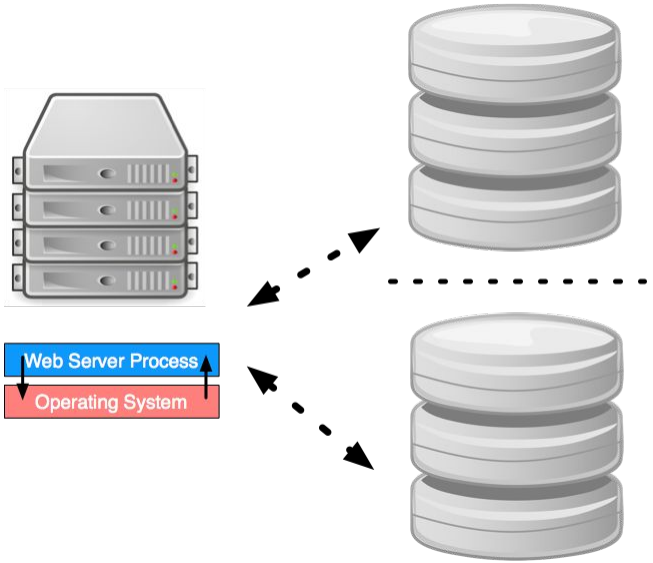
At the load balancer?

- How would we implement this?

Across multiple load balancers?

- How would we implement this?

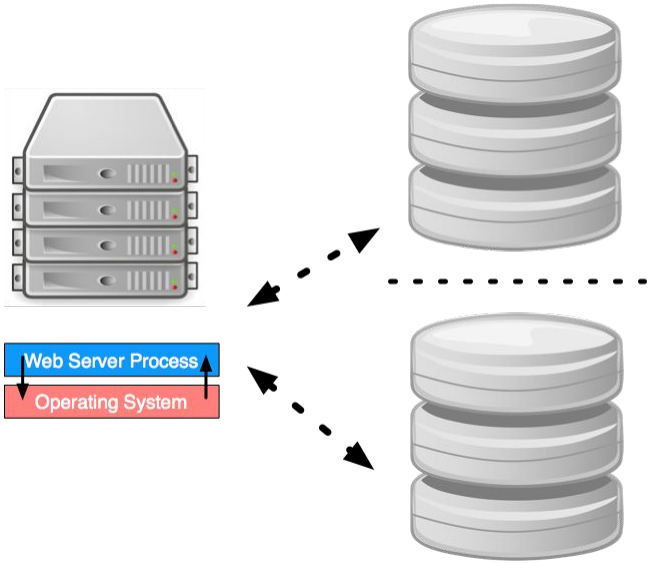
Sharding



At the App Server

- App server has configuration that tells it where each database is and how to map data to database.
- Mapping can be arbitrarily complex.
- Mapping can be stored in a database.

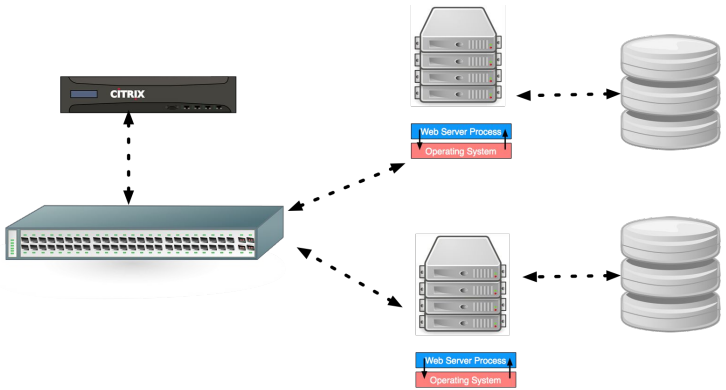
Sharding



At the App Server

```
if user.account_name.starts_with?(‘a’)
  database.connect(‘shard1’)
elsif user.account_name.starts_with?(‘b’)
  database.connect(‘shard2’)
else
...
```

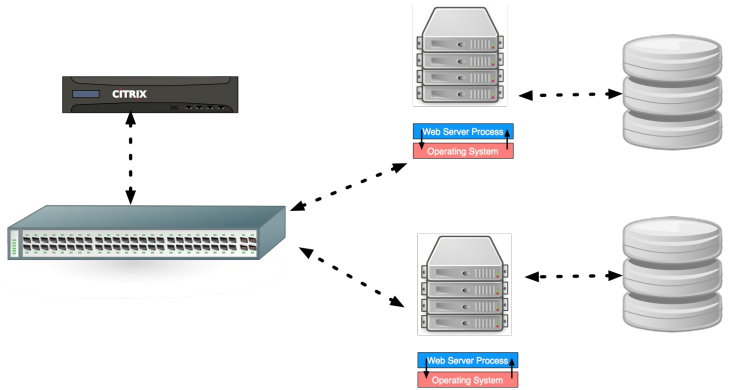
Sharding



At the load balancer

- Load balancer is configured to route requests to app servers that are talking to the right database.
- Mapping decision can only be based on information visible to LB
  - Resource
  - Cookie
  - Host header

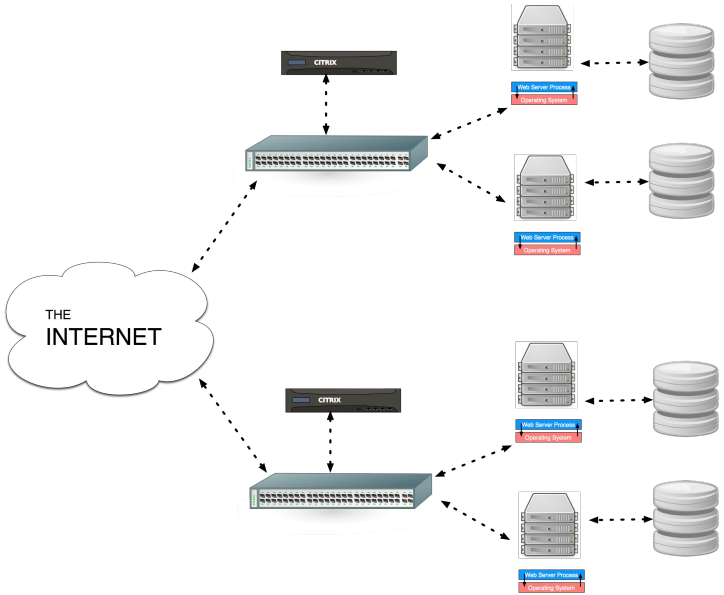
Sharding



At the load balancer

```
if connection.hostname.starts_with?(‘a’)
  send_request_to(‘pool1’)
elsif user.account_name.starts_with?(‘b’)
  send_request_to(‘pool2’)
else
...
```

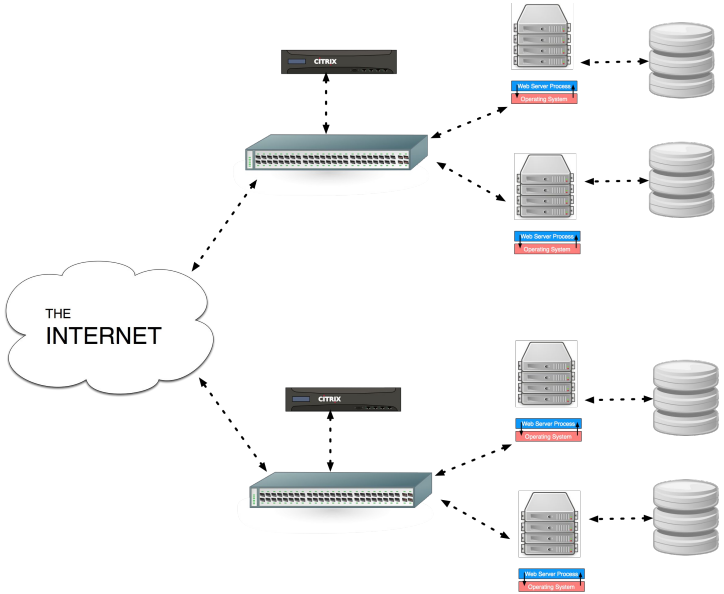
Sharding



Across Load Balancers

- DNS is configured to point to the correct load balancer for a given request.
- Examples:
  - na6.salesforce.com
  - user.github.io

Sharding



Across Load Balancers

```
$ORIGIN example.com.
customer1       IN  A     192.0.2.3
customer2       IN  A     192.0.2.4
...
```

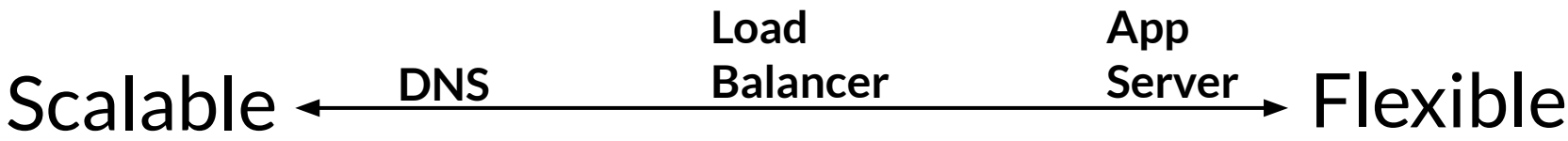
Sharding

What are the tradeoffs of each?

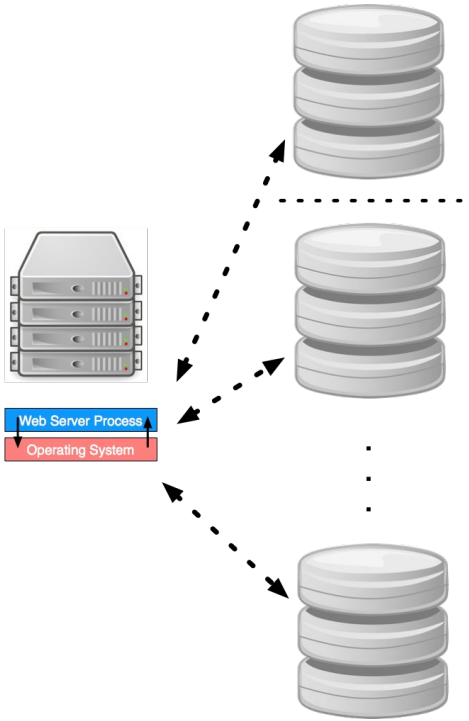
Sharding

What are the tradeoffs of each?

In general...



Sharding



- Ideally, the number of your partitions increase as usage of your application increases.
- Example:
  - If each customer's data can be partitioned from the rest, then doubling the number of customers doubles the number of shards.

Sharding

Simple Example:

- Gmail!
- How would we build this?

Sharding

Simple Example:

- Gmail!
- The data that represents my email needs no relations to the data representing other people's email
- When a request arrives, we apply some mapping function to determine which database it belongs to
- We can cleanly partition users into separate data stores
    - Scales cleanly as number of users increases

Sharding

Harder Example: The demo app.

- Users can create and view communities
- Users can create submissions in these communities
- Each submission has a tree of comments

How should we divide up (shard) this application?

Sharding

By User?

- Would be awkward, since logged in users will want to see submissions and comments by other users

By Submission?

- Users should be able to view many submissions when looking at a community

By Community?

- Maybe...

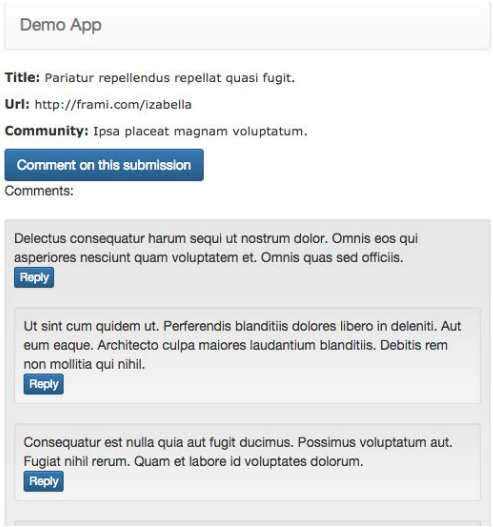
Sharding

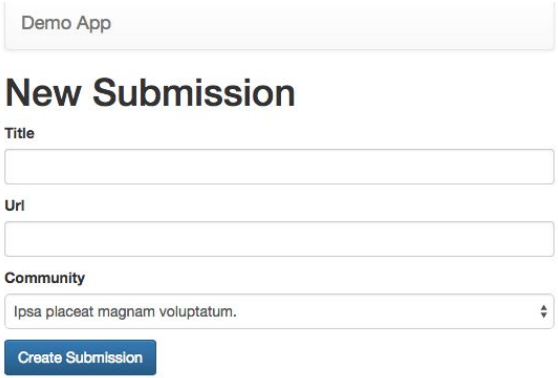
How could this work?

- Upon receiving a request, the app server would figure out which database it needed to speak to in order to serve this request
  - http://gardening.demoapp.com/
  - http://demoapp.com/gardening/
- After connecting to the correct database, it would issue SQL queries as normal

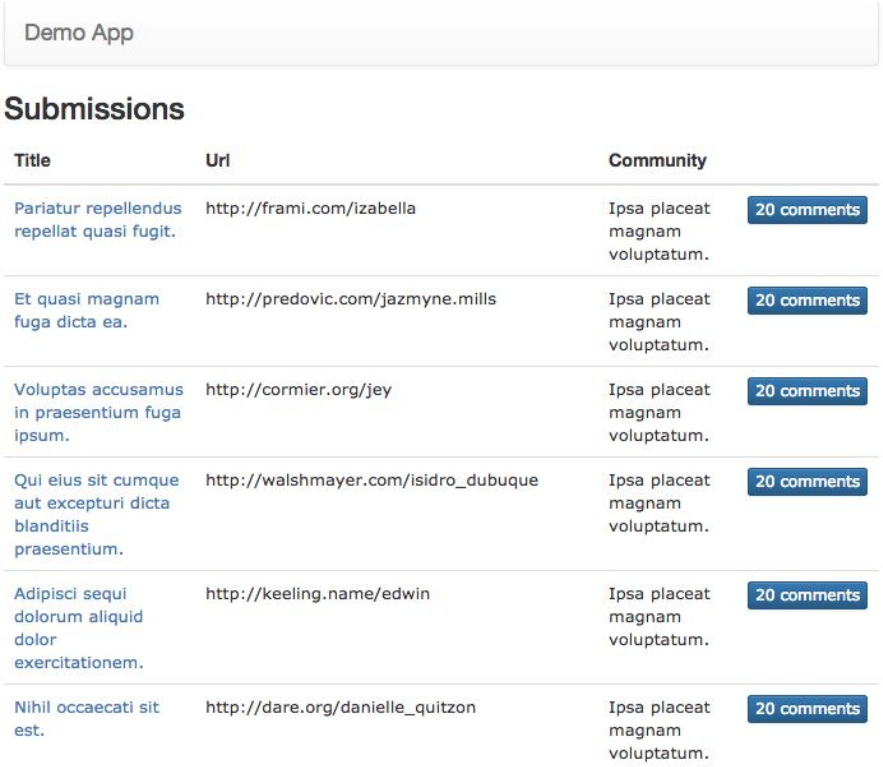
Sharding

Some parts of our user interface would work with community-based sharding





Sharding



Others would be more difficult:

- Global views of all submissions, across communities

Sharding

Solutions?

Sharding

Solutions:

- Modify the user interface so the difficult to shard page doesn't need to exist.
  - Maybe a semi-static list of communities and you need to dig down into each to see what is submitted.
- Construct that page using other methods
  - Create a sub-service to aggregate this data.

Sharding

Sharding in Rails

- Nothing built in, but the gem “Octopus” performs this well (https://github.com/tchandy/octopus)

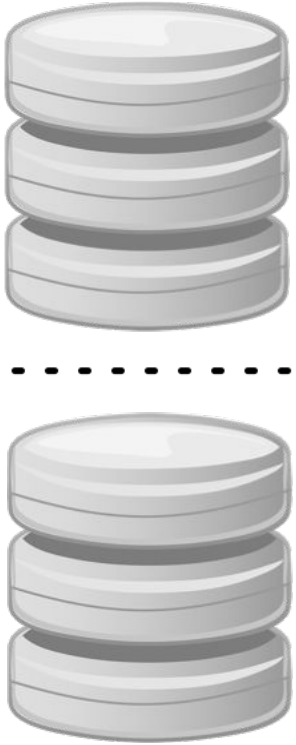
```
class ApplicationController < ActionController::Base
  around_filter :select_shard

  def select_shard(&block)
    Octopus.using(:brazil, &block)
  end
end
```

Sharding

- Strengths
  - If you genuinely have zero relations across shards, this scaling path is very powerful.
  - Works best when partitions grow with usage.
- Weaknesses:
  - Can inhibit feature development
    - Your app might be perfectly shardable today, but future features may change that.
  - Not easy to retroactively add to an application.
  - Transactions across shards don’t happen.
  - Consistent DB snapshots across shards don’t happen.

Service Oriented Architecture



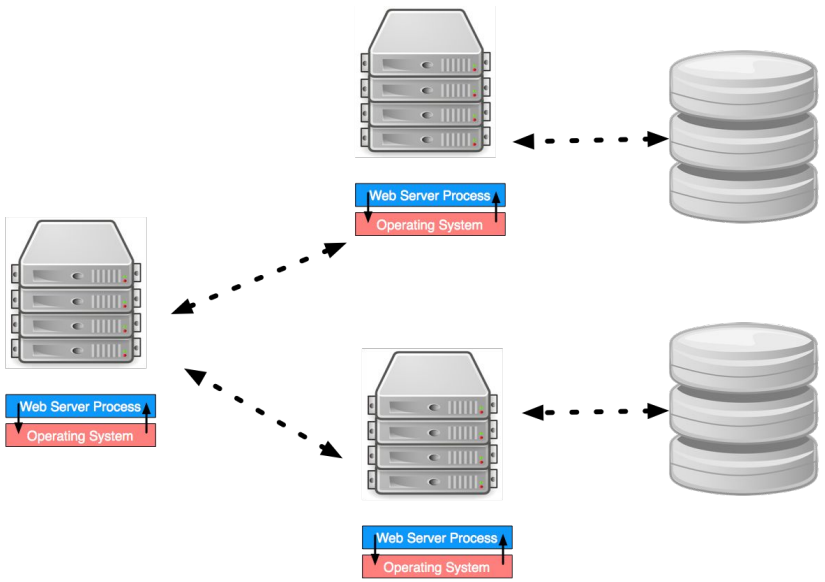
Sharding partitions data of the same type into separate, unrelated groups.

Service Oriented Architectures do the opposite:

- Partition data based on type and function.

Like sharding, no relations will cross these partitions.

Service Oriented Architecture

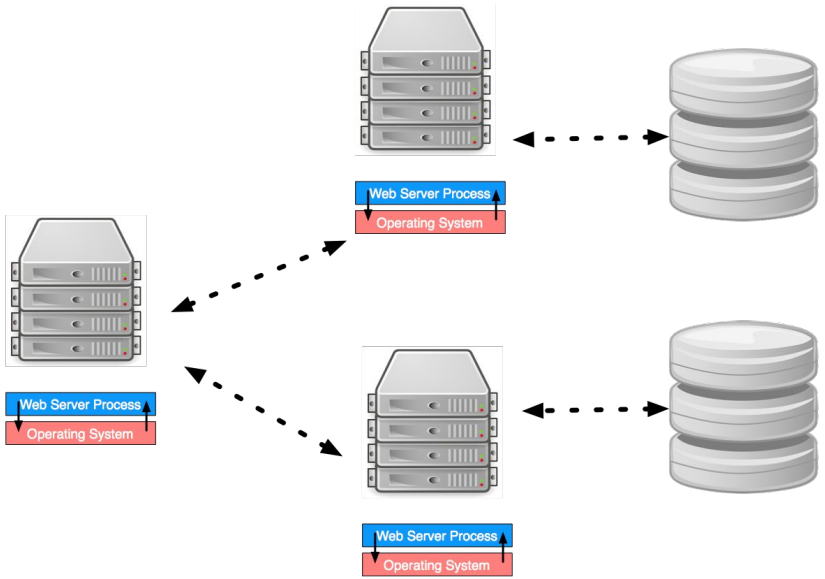


In addition to separating data by function, Service Oriented Architectures tend to encapsulate the data within mini-applications

- These applications are called services

When an app server needs data to satisfy a request, instead of speaking to a database, it will request data from another application server.

Service Oriented Architecture

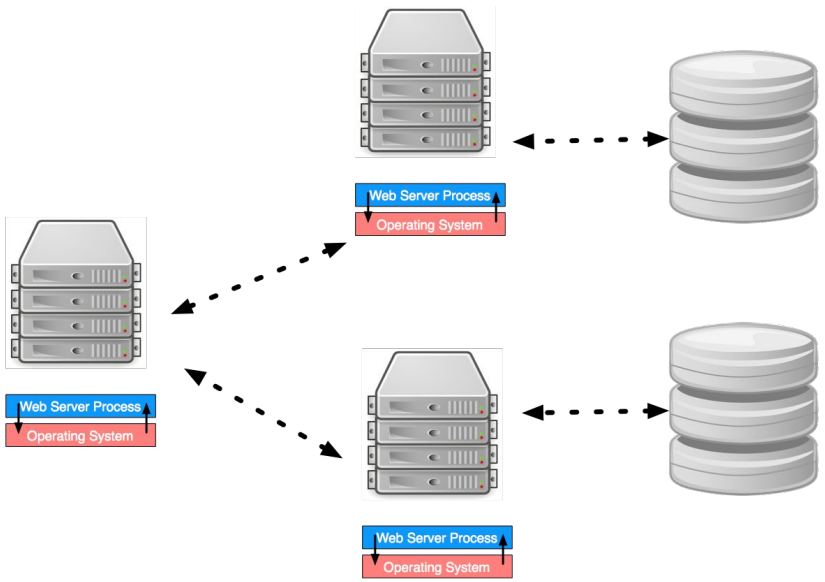


These services are broken out by logical function.

Examples:

- Users service that handles authentication and authorization.
- Billing service that handles credit cards and subscriptions
- Accounting subsystem that keeps track of invoices.

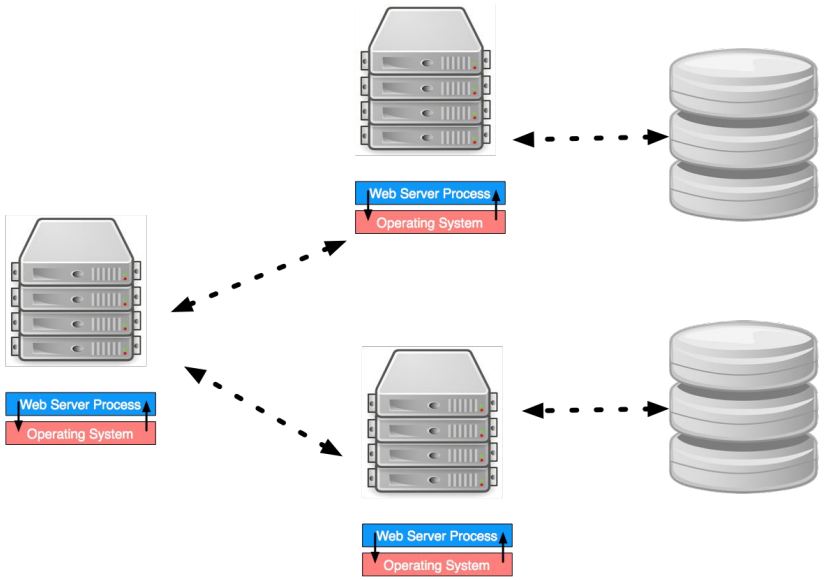
Service Oriented Architecture



Another difference between SOA and sharding is that a particular request generally reads from only one shard.

In SOA, a request generally consults many constituent services

Service Oriented Architecture



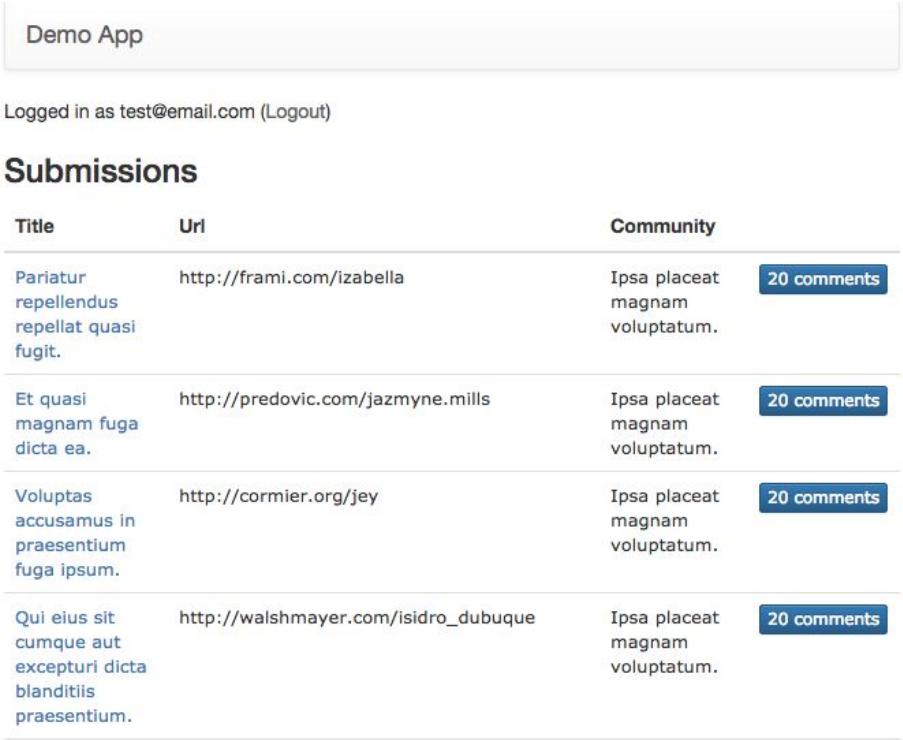
These architectures also have other benefits:

- Deployment of services is decoupled.
- A service interface can serve to encapsulate the work of a development team

Service Oriented Architecture

SOA Example: the Demo app.

How could we divide this up (into SOA)?

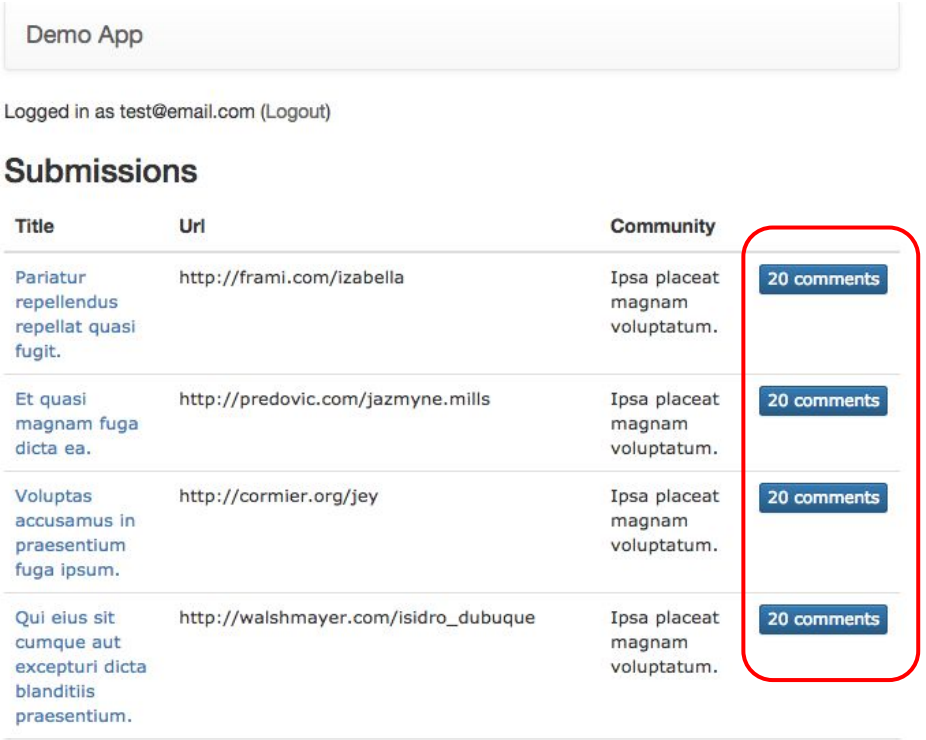


Service Oriented Architecture

SOA Example: the Demo app.

How could we divide this up (into SOA)?

Comments service can keep track of comments and replies for each submission.

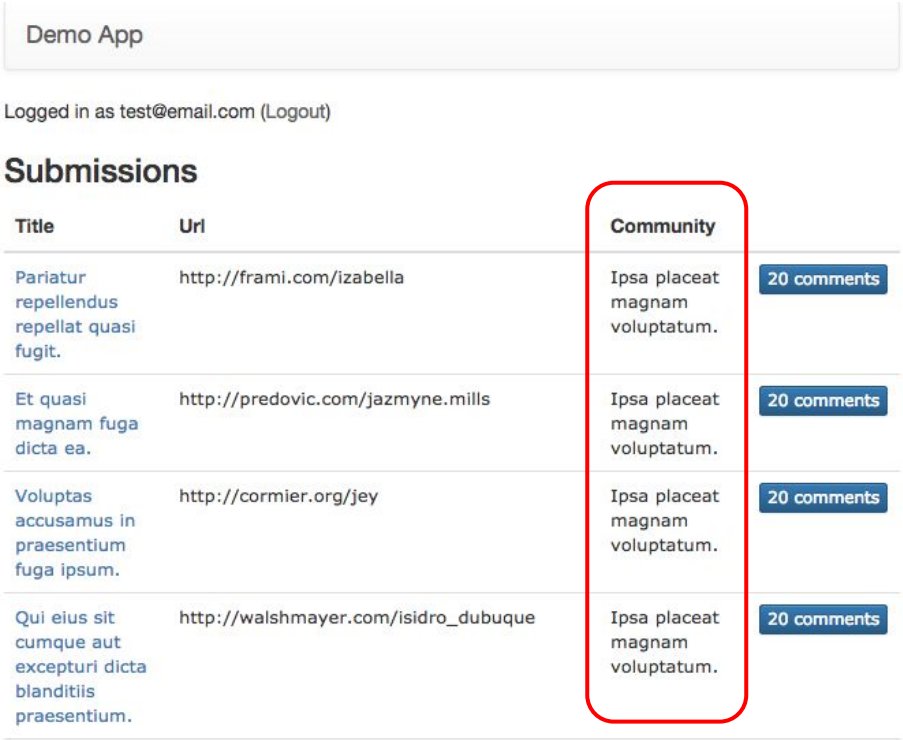


Service Oriented Architecture

SOA Example: the Demo app.

How could we divide this up (into SOA)?

Communities service can keep track of the list of communities, and who created them.

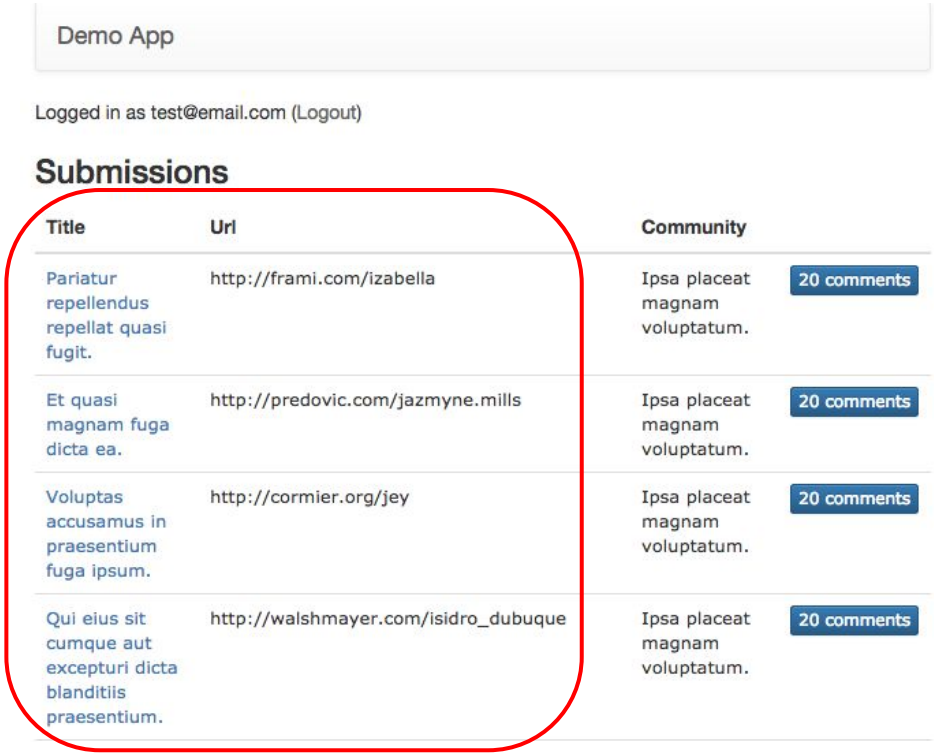


Service Oriented Architecture

SOA Example: the Demo app.

How could we divide this up (into SOA)?

A Submissions service could keep track of the links that had been submitted.

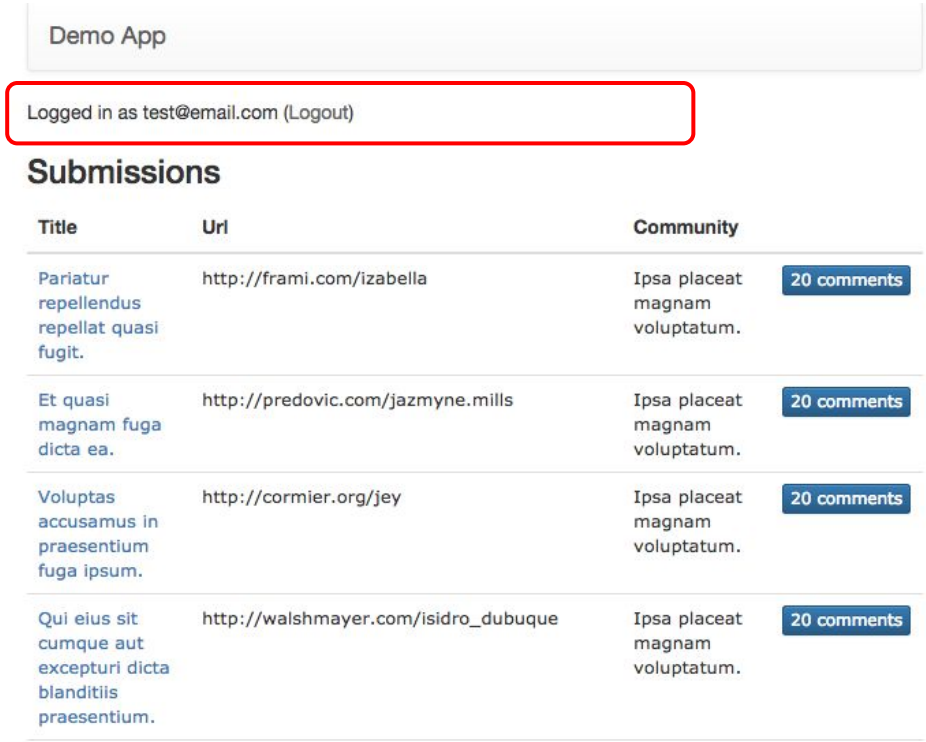


Service Oriented Architecture

SOA Example: the Demo app.

How could we divide this up (into SOA)?

A users service can keep track of users management.



Service Oriented Architecture

Example code change:

```
class CommunitiesController < ApplicationController
  def create
    if current_user.allowed_to_create_community?
    success = Community.create(params)
    if success
      render :show
    else
      render :new
    end
  end
end
```

```
class CommunitiesController < ApplicationController
  def create
    user= UsersService.get_user_from_session(cookie)
    if user.allowed_to_create_community?
      title = params['title']
      community = params['community']
      success= CommunitiesService.create_submission(
        title, community, user_id)
      if success
        render :show
      else
        render :new
      end
    end
  end
end
```

Service Oriented Architecture

SOA Implementation

- SOA these days is commonly implemented by JSON over HTTP. RESTful APIs are common.

- Why JSON/HTTP/REST?
  - Easily constructed: Rails makes you do work to **not** have a JSON API.
  - Easily discovered: HTTP and JSON are both very readable. Documentation can be very light when the API is readable.
  - Tech stack is shared.

Service Oriented Architecture

SOA Implementation

- JSON/HTTP/REST disadvantages: performance.
- For high-performance internal APIs, use
  - Google Protocol Buffers
  - Apache Thrift. (from Facebook)
- These options are generally more strongly typed (they need an encoding format) and need more documentation.

Service Oriented Architecture

Amazon famously uses this approach extensively.
2002 memo from Jeff Bezos:

- All teams will henceforth expose their data and functionality through service interfaces.
- Teams must communicate with each other through these interfaces.
- There will be no other form of inter-process communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.
- It doesn't matter what technology they use.
- All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.
- **Anyone who doesn't do this will be fired. Thank you; have a nice day!**

Service Oriented Architecture

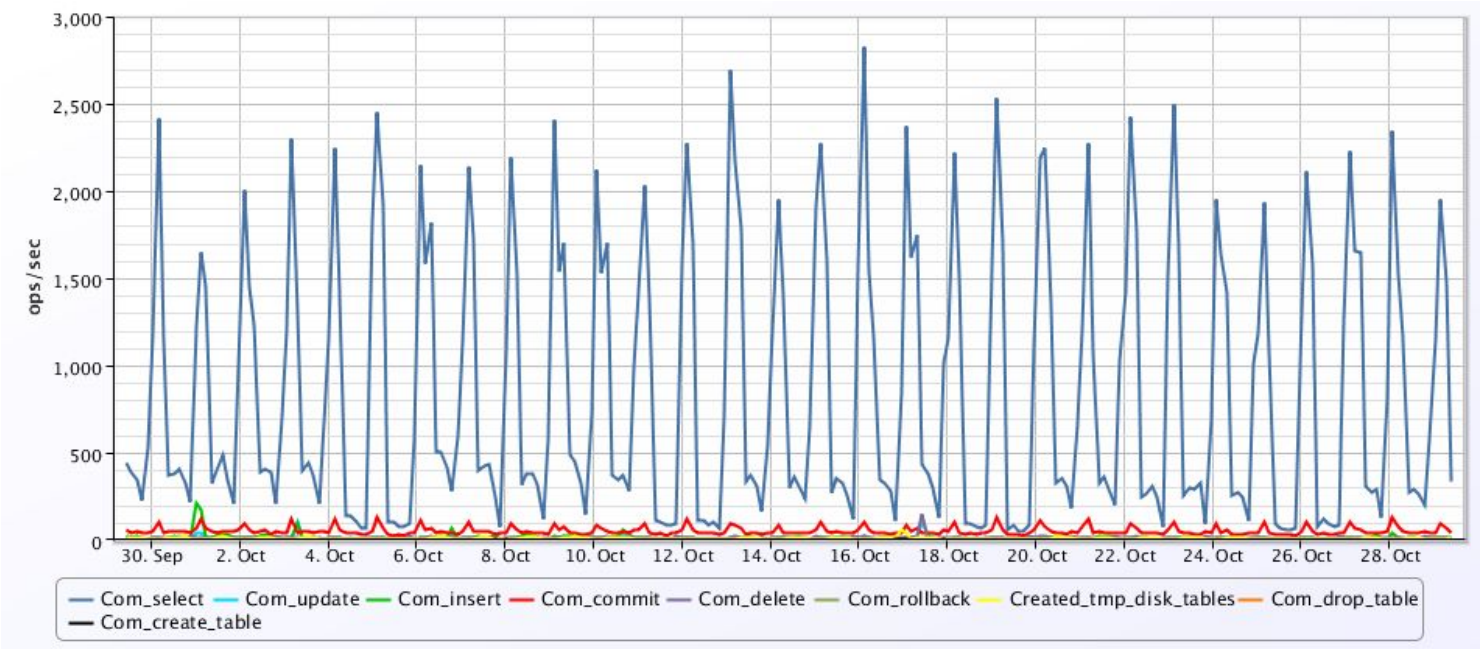
Strengths

- Other benefits: small, encapsulated codebases.
- Scales well as application size scales.
- Scales well as team size scales.

Weaknesses:

- Doesn't necessarily scale as the number of users increases.
- Transactions across services don't happen.

Reads vs. Writes



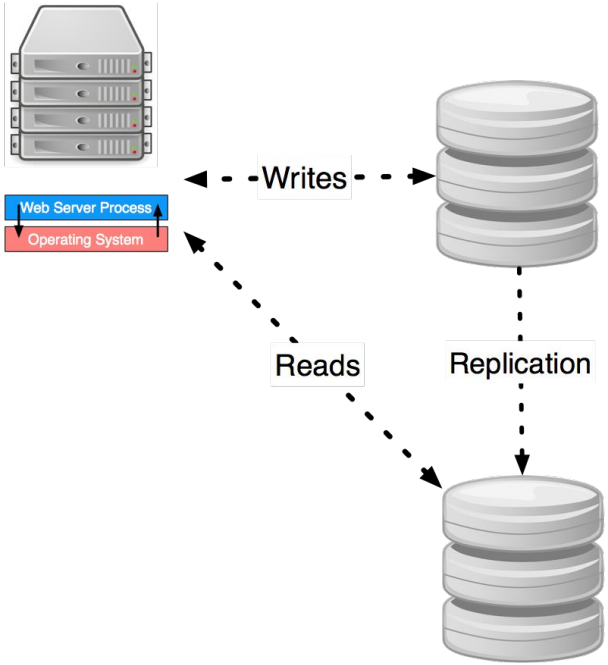
Reads vs. Writes

A relational database is hard to horizontally scale.

A read-only copy of a database is easy to horizontally scale:

- Set up a separate machine to act as a “read slave”
- Whenever any transaction commits to the “master” database, send it over to the slave and apply it.

Reads vs. Writes

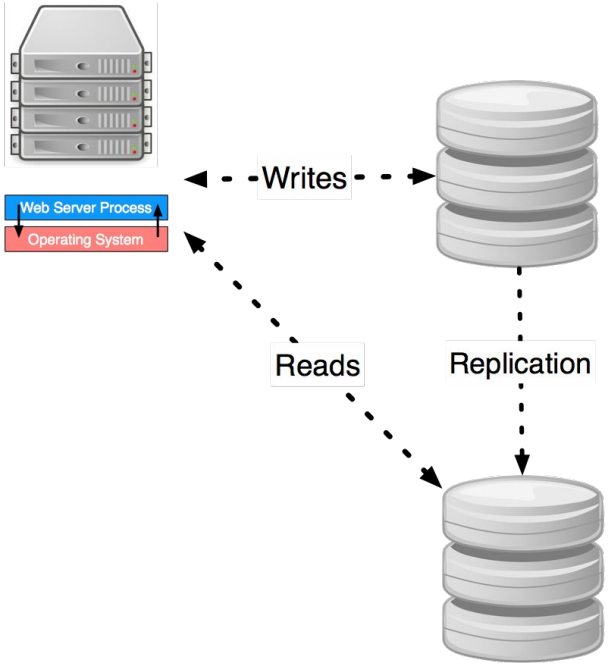


For most web applications, reads are much more common than writes.

If most traffic is read-only, and scaling read-only copies is easy, lets send our read traffic to a read-slave.

Slaves can be chained.

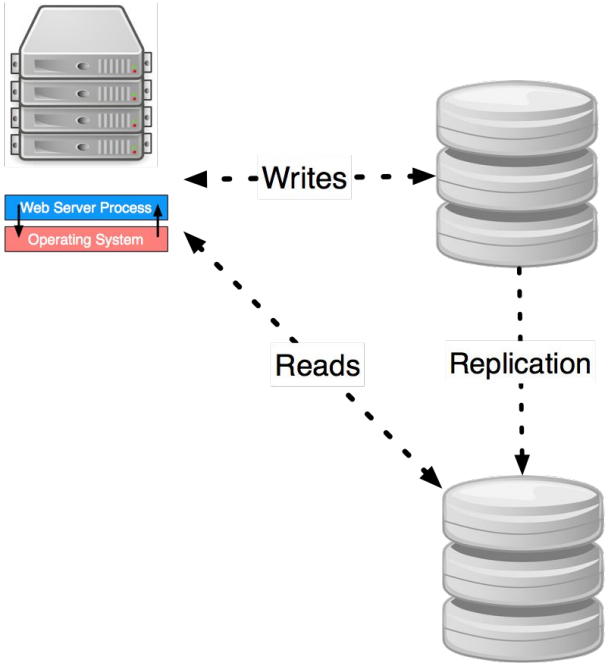
Reads vs. Writes



Replication can either be synchronous or asynchronous:

- **Synchronous**: When a transaction is committed to master, master sends transaction to slaves and waits until it is applied.
- **Asynchronous**: When a transaction is committed to master, master sends transaction to slaves but does not wait until it is applied.

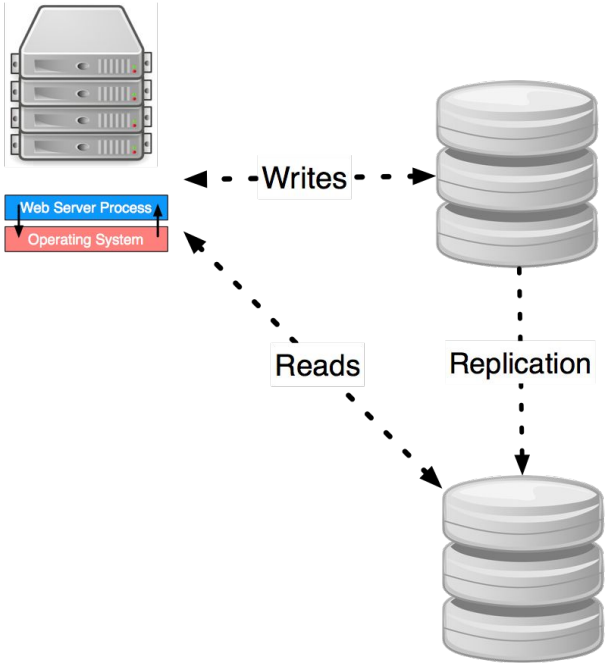
Reads vs. Writes



What are the advantages of waiting until it is applied to all slaves?

What are the disadvantages of waiting until it is applied to all slaves?

Reads vs. Writes



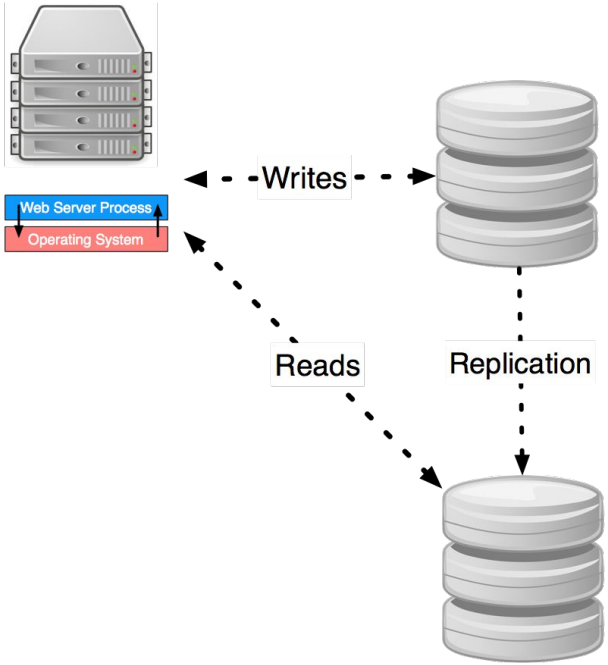
What are the advantages of waiting until it is applied to all slaves?

- Consistency. Subsequent read requests will see changes.

What are the disadvantages of waiting until it is applied to all slaves?

- Performance. There may be many read slaves to apply changes to.

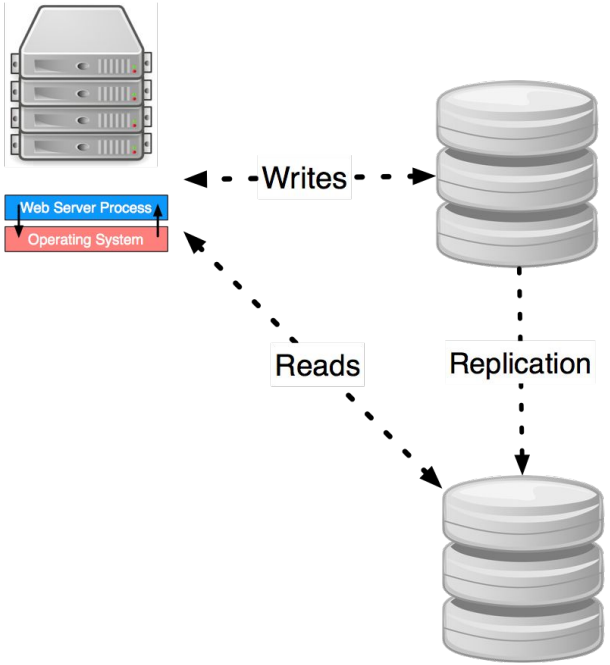
Reads vs. Writes



Statement vs. Block:

- Statement-level:
  - Similar to streaming the journal from the master to the slave
- Block-level:
  - Instead of sending the SQL statements to the slave, send the consequences of those statements
- Advantages of each?

Reads vs. Writes



Statement-level is faster, with a catch:

- A SQL statement is generally more compact than its consequences.

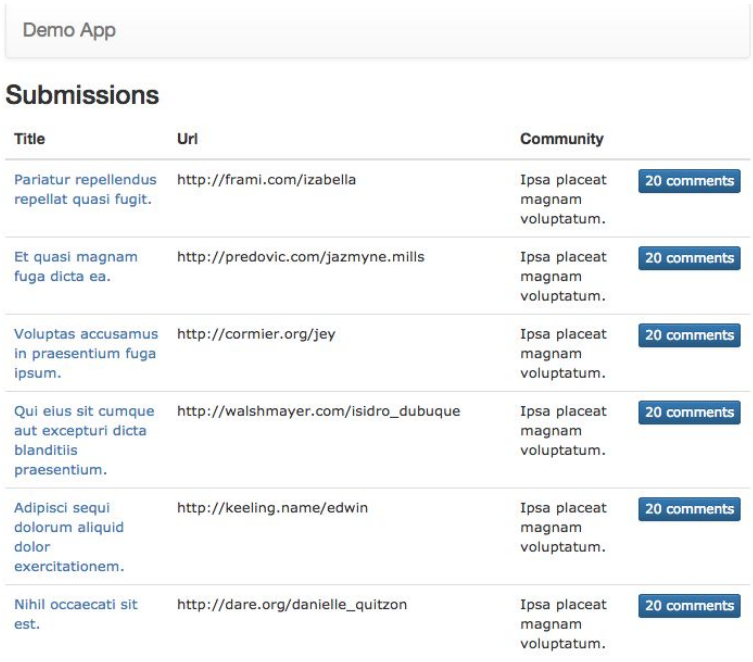
  Example: `UPDATE txns SET amount = 5`

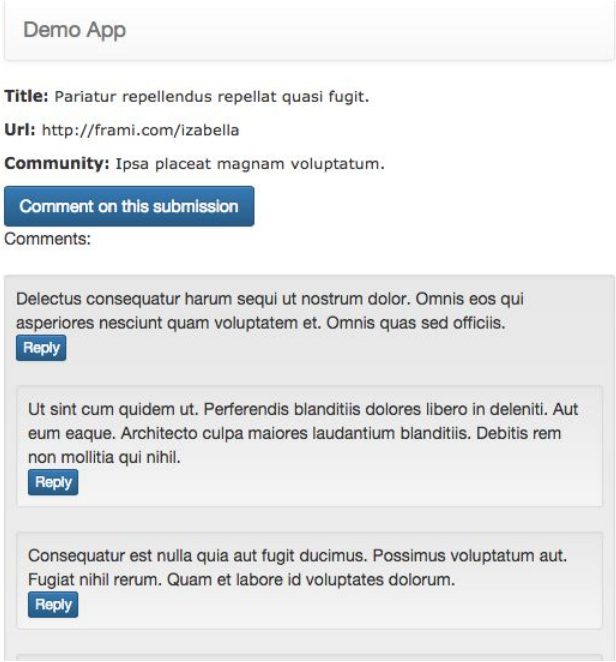
- SQL statements must now be deterministic

```
UPDATE txns
SET amount = 5,
    updated_at = NOW()
```

Reads vs. Writes

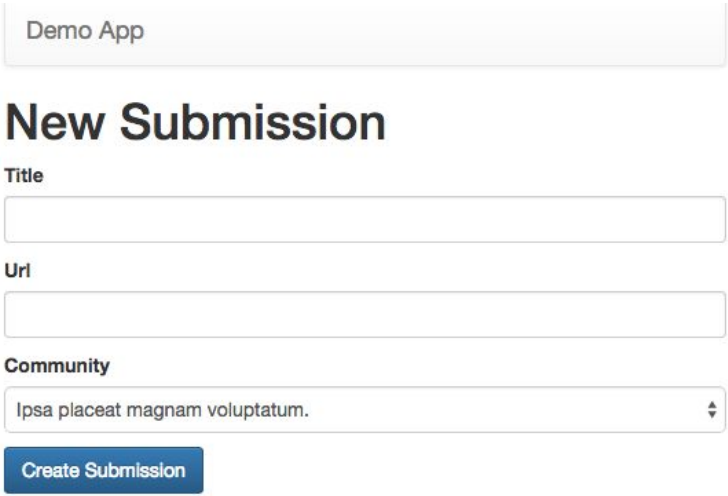
These responses could be served off read slaves



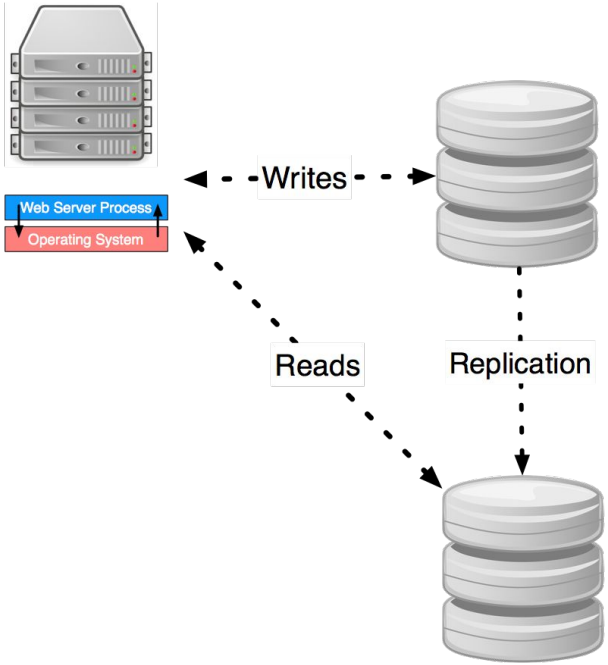


Reads vs. Writes

These would need to talk to the master database



Reads vs. Writes



No built in support in Rails, but the Octopus gem can do this for you:

```
Octopus.using(:read_slave) do
  num_users = User.count
end

Octopus.using(:master) do
  User.create(:name => "Mike")
end
```

Reads vs. Writes

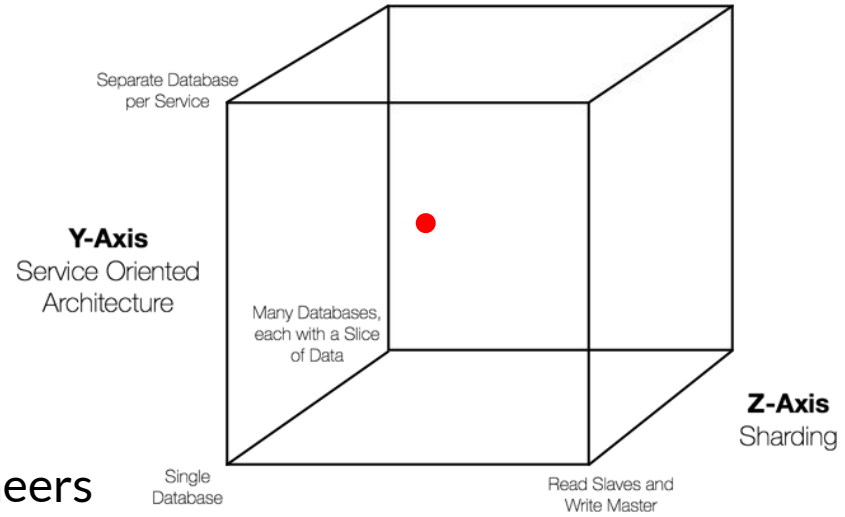
- Strengths:
  - For applications with a high read-to-write ratio, you can reduce the load on your master database significantly.

- Weaknesses:
  - Application developer needs to think about reads that affect writes vs. reads that don't affect writes.

AKF Cube at Appfolio

At Appfolio...

- High usage of sharding
  - Each customer in separate database.
- Medium use of SOA
  - Some functionality broken out into services, but more for scaling engineers than scaling load.
- Low use of Read vs. Write distinction
  - We have read slaves for backup and analysis.



Conclusion

Horizontal scaling of relational databases is hard.

There is no silver bullet, but by combining sharding, SOA and read slaves you can get very far.

For applications that need to scale writes beyond what RDBMS can offer, you need non-relational data stores.

For Next Time...

Tomorrow's lab: be ready to demo your application's features on AWS.

If you are still having problems deploying your application on AWS, be sure to post on piazza.