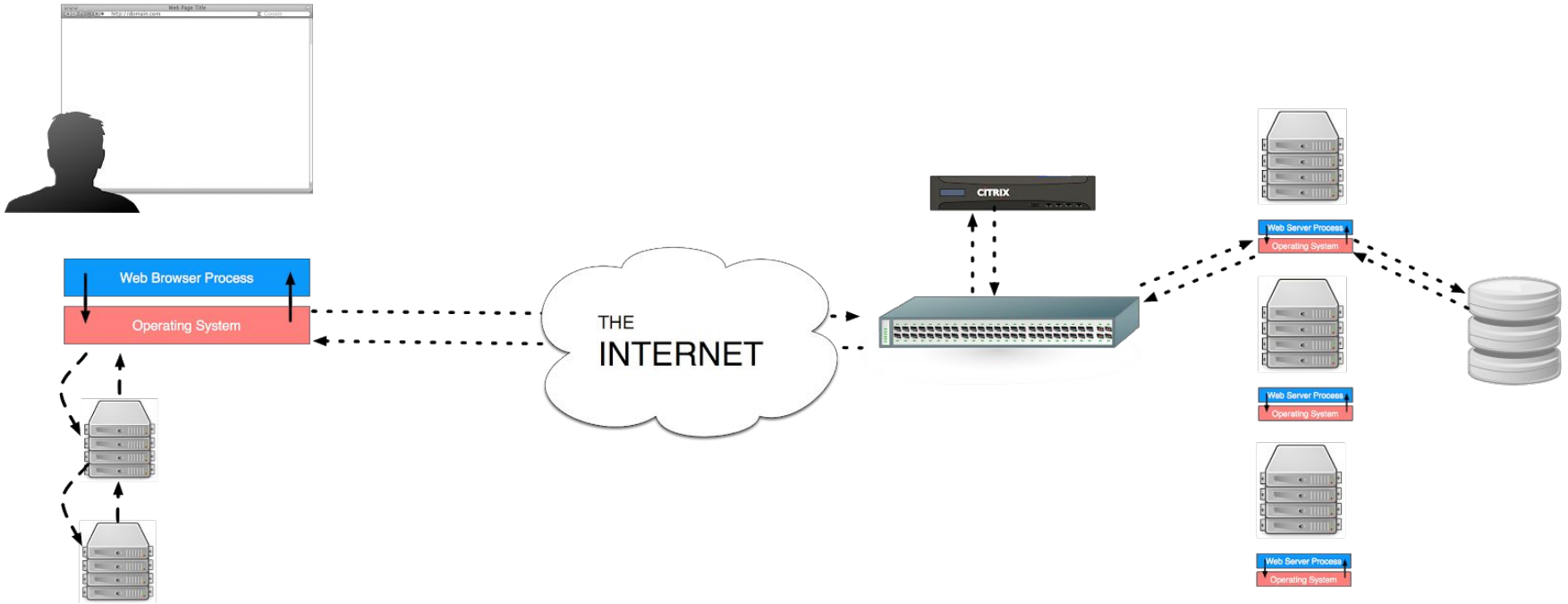
CS188
Scalable Internet Services

John Rothfels, 10/29/20

Motivation


Web Browser Process
Operating System
THE
INTERNET
CITRIX
Web Server Process
Operating System
Web Server Process
Operating System
Web Server Process
Operating System

Motivation

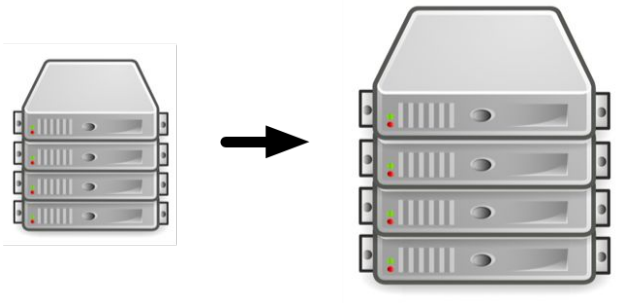
You've got your application up and running. It is becoming increasingly popular and performance is degrading.

What do you do?

Motivation

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

What do you do?



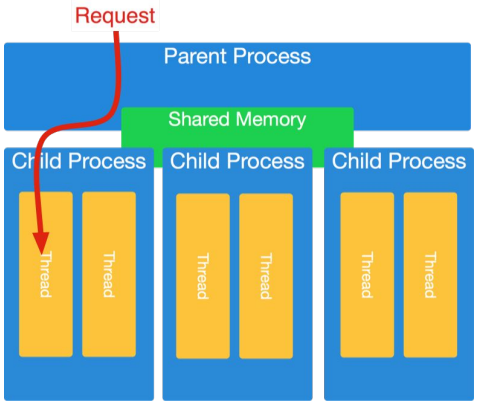
You've increased your server size, and that buys some time. But as popularity grows, there are no larger computers to buy. 😢

ℹ️ This is called **vertical scaling**.

Motivation

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

What do you do?

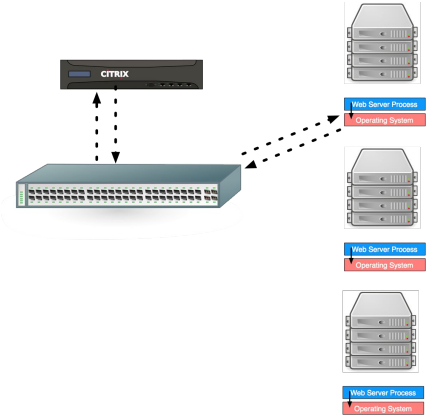


You've deployed application servers that can serve many requests simultaneously, and that helped. But as popularity grows, it's not enough. 😢

Motivation

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

What do you do?



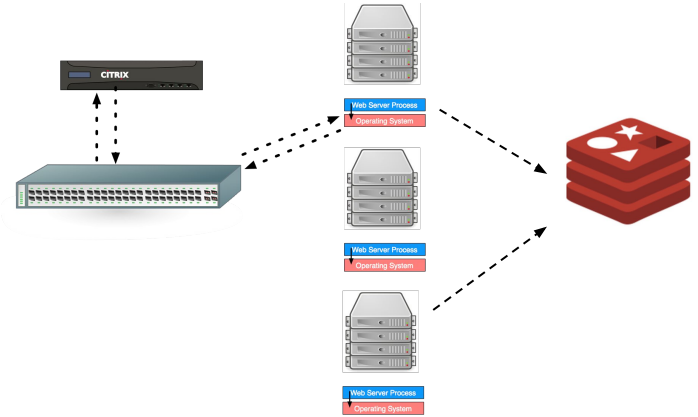
You've set up a load balancer and spread load across many servers. But as popularity grows, it's not enough. 😢

ℹ️ This is called **horizontal scaling**.

Motivation

You've got your application up and running. 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 still not enough. 😢

Motivation

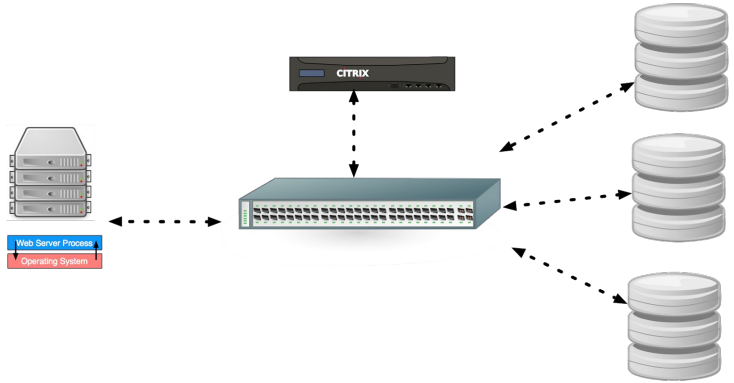
Your relational database is still on one machine and that machine is having trouble keeping up with increased load.

So what do we do?

Motivation

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

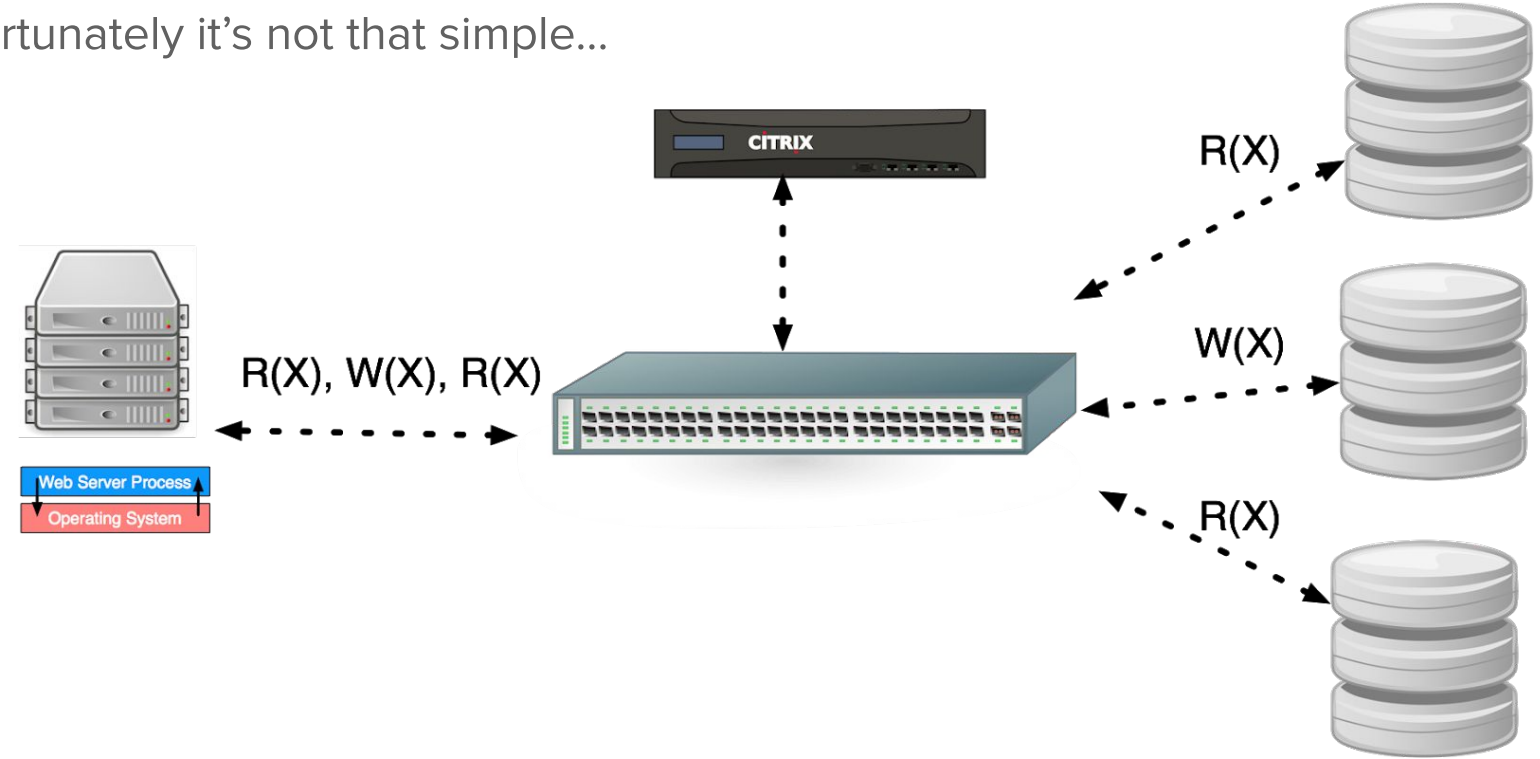
So what do we do?



💡 Split the database across multiple machines?

Motivation

Unfortunately it’s not that simple…



Scaling databases

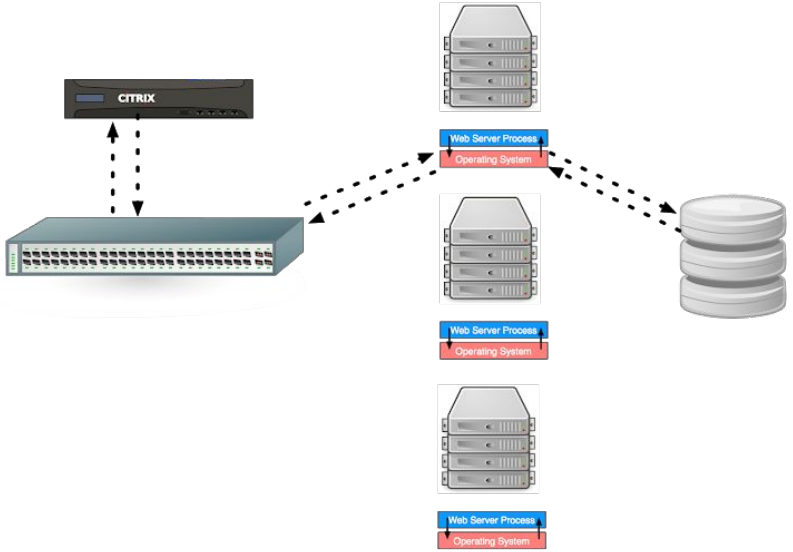
Scaling databases can be difficult.

Different databases have different scaling characteristics.

Scaling databases

So far we've assumed we have many stateless application servers.

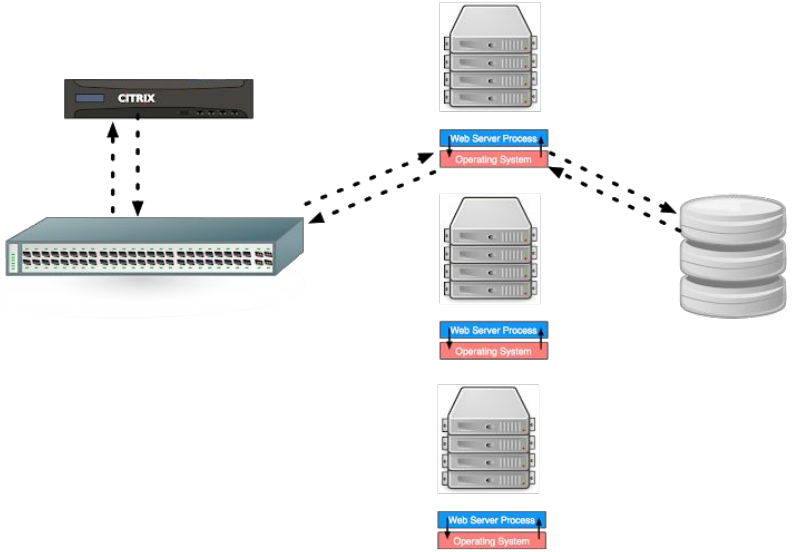
⁉️ **Why do we separate the concerns of managing state and running our application?**



Scaling databases

Managing state is hard.

- If a request writes data and we return success to the user, we never want a subsequent machine failure to lose that data.
- If requests A and B want to read and write the same state at the same time, how do we handle it?
- Scaling stateless servers independently from state is much easier!



Scaling relational databases

The **ACID** properties of relational databases make these DBs particularly difficult to scale.

- **A**tomicity: transactions are all or nothing
- **C**onsistency: DB is consistent at the beginning and end of all transactions (e.g. no dangling foreign key constraint, invalid NULL column, etc.)
- **I**solation: another transaction doesn't see the effects of uncommitted transactions
- **Durability**: transaction effects don't disappear

Scaling relational databases

The **ACID** properties of relational databases make these DBs particularly difficult to scale. They have overlapping concerns:

- Atomicity and Durability are related and generally provided by **journaling**.
- Consistency and Isolation are provided by concurrency control (usually implemented via **locking**).

⚠️ ACID does not help with external side-effects (e.g. actions taken outside the system, transferring money, communicating with a web service, etc.)

Scaling relational databases: transactions

A **schedule** (or “**history**”) is an abstract model used to describe the transactions running on a system.

T1	R(X), W(X), Com.		
T2		R(Y), W(Y), Com.	
T3			R(Z), W(Z), Com.

Scaling relational databases: transactions

Two actions are said to be in **conflict** if:

- The actions belong to different transactions
- At least one of the actions is a write operation
- The actions access the same object (read or write)

Example conflict: T1: R(X), T2: W(X), T3: W(X)

Non-conflict: T1: R(X), T2: R(X), T3: R(X)

⚠️ A conflict means we can’t blindly execute the transactions in parallel!

Scaling relational databases: transactions

Why can’t we blindly execute conflicts in parallel?

T1		R(X)	W(Y), Com.
T2	W(X)		Abort

The **dirty read problem** is where transactions read a value that is later aborted and removed from the database. Reading transactions will have incorrect results.

Scaling relational databases: transactions

Why can’t we blindly execute conflicts in parallel?

T1	R(All X), AVG	W(Y)	Com.
T2	W(Some X), Com.		

The **incorrect summary problem** is where the 1st transaction takes a summary over the values of all the instances of a repeated data item. While a 2nd transaction updates some instances of the data item. Resulting summary will not reflect a correct result for any deterministic order of the transactions. Result will be random depending on the timing of the updates.

Scaling relational databases: transactions

A schedule is **serial** if:

- The transactions are executed non-interleaved

Two schedules are **conflict equivalent** if:

- They involve the same actions of the same transactions
- Every pair of conflicting actions is ordered in the same way

Schedule S is **conflict serializable** if:

- S is conflict equivalent to some serial schedule

A schedule is **recoverable** if txns commit only after all txns whose changes they read, commit.

Scaling relational databases: transactions

Schedule that is **not conflict serializable**:

T1	R(A), W(A)		R(B), W(B)
T2		R(A), W(A), R(B), W(B)	

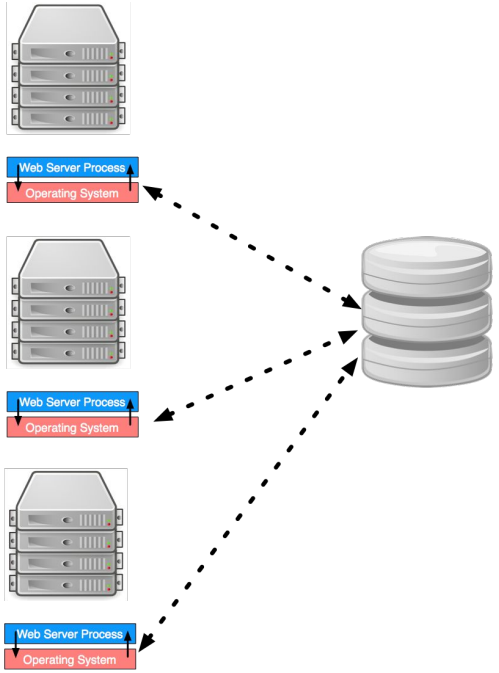
Because it is **not conflict equivalent to either of these**:

T1	R(A), W(A), R(B), W(B)	
T2		R(A), W(A), R(B), W(B)

T1		R(A), W(A), R(B), W(B)
T2	R(A), W(A), R(B), W(B)	

Scaling relational databases: transactions

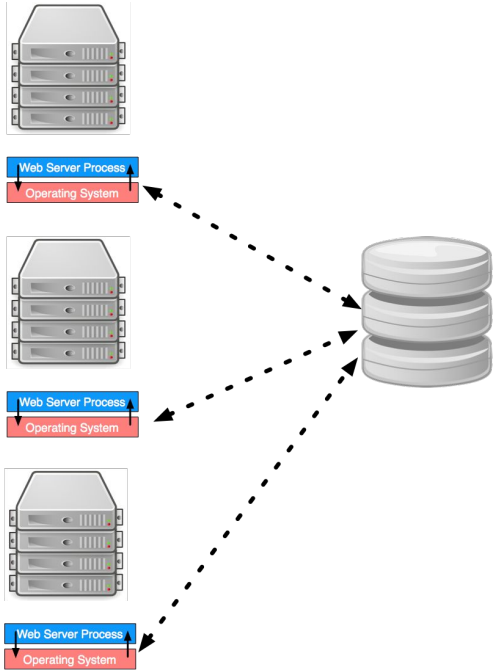
⁉️ **Why is it important that we get a serializable schedule? Why not just execute serially?**



Scaling relational databases: transactions

⁉ Why is it important that we get a serializable schedule? Why not just execute serially?

- A serial schedule is important for consistent results (good when you are keeping track of your bank balance in the database).
- A serial execution of transactions is safe but slow
- Most general purpose relational databases default to employing conflict-serializable and recoverable schedules.
- If you don't want to do a serial execution, what else can you do?



Scaling relational databases: locks

⁉️ **How do we implement a database with schedules that are conflict serializable and recoverable?**

Locks

- a system object associated with a shared resource such as a data item, a row, or a page in memory
- prevent undesired, incorrect, or inconsistent operations on shared resources by concurrent transactions
- a database lock may need to be acquired by a transaction before accessing the object

Scaling relational databases: locks

Two types of database locks:

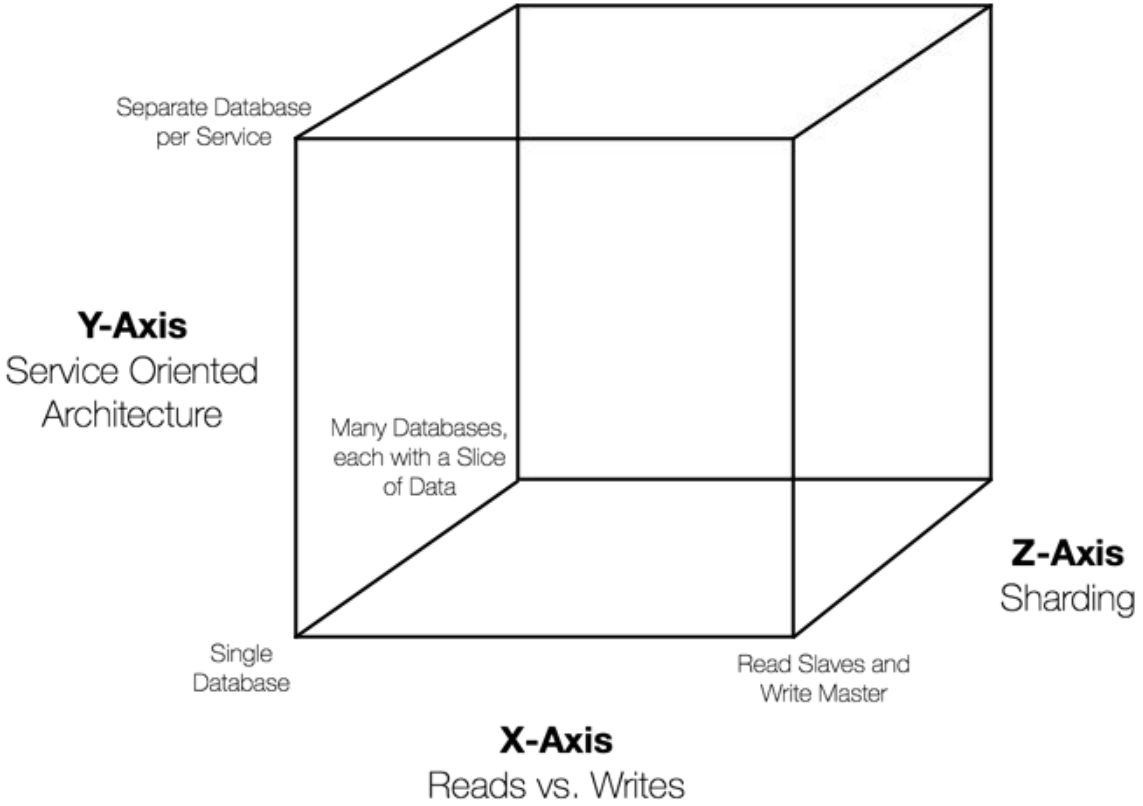
- **Write-lock**
  - Blocks writes and reads
  - Also called “exclusive lock”
- **Read-lock**
  - Blocks writes
  - Also called “shared lock”

Scaling relational databases

Techniques to scale *relational databases*:

- (be suspicious of ORM 🧐)
  - Look at the actual queries your ORM is creating (turn on **`logging: true`** in **`initORM()`**)
- Modify database schema, add indices
  - Use EXPLAIN on your expensive queries to diagnose query problems
    - Won't actually execute the query
    - Helps us understand how and when MySQL will use indices.
  - **Different database schemas have different scaling characteristics!**
- Break up the database
  - Sharding
  - Service oriented architecture
  - Read replicas

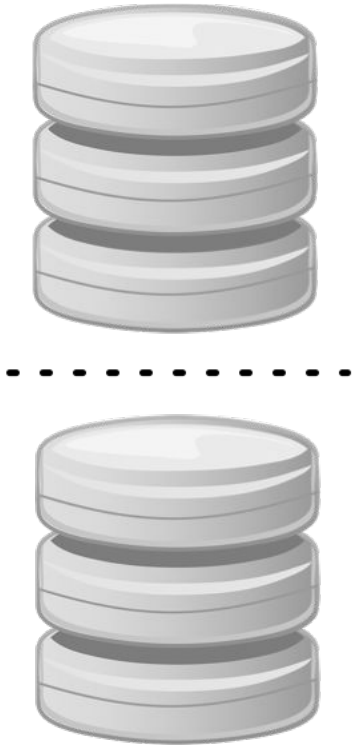
Scaling relational databases



Scaling relational databases: 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.

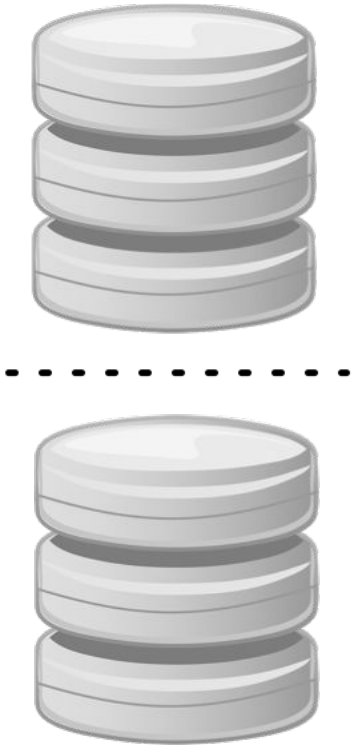


Scaling relational databases: sharding

To shard a database is to find some partition that can divide your data into groups that are not related.

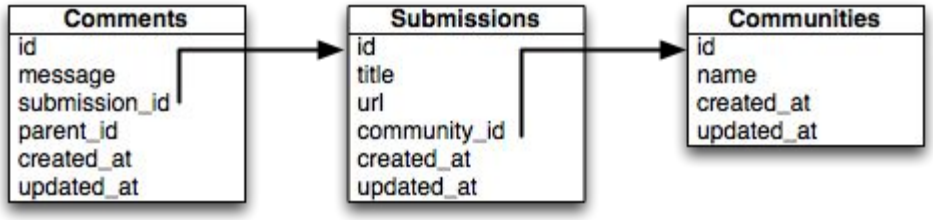
When they have been separated into these shards, you will never JOIN across them.

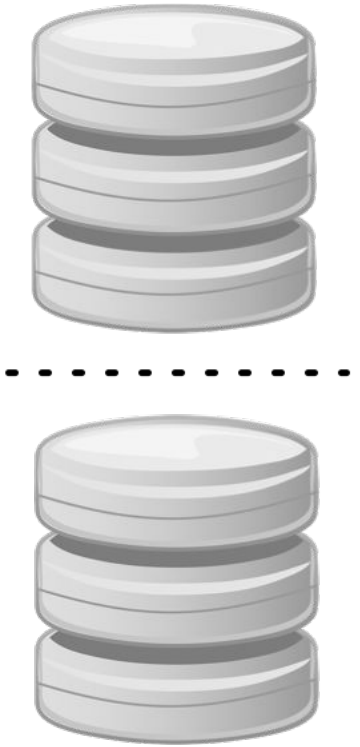
If you ever need to perform operations that cross shards, you must do it at the application level (in code).



Scaling relational databases: sharding

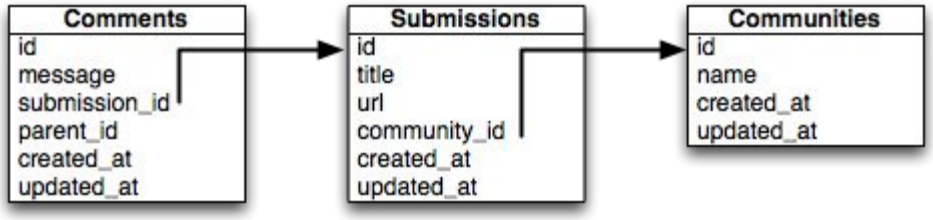
Partitioning your database in a way you will never JOIN across is easier said than done...





Scaling relational databases: sharding

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



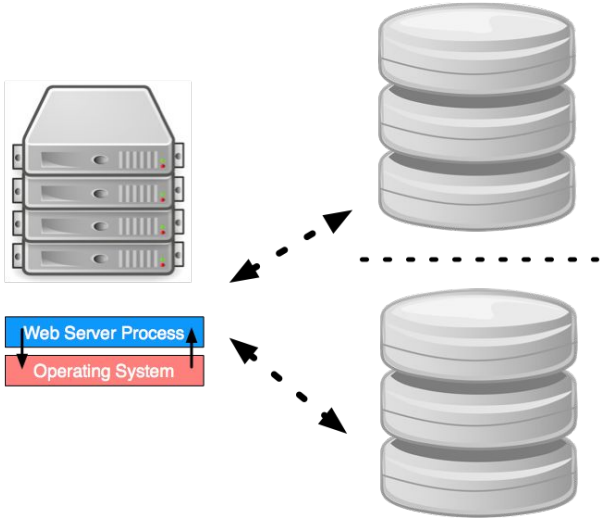
- comment is only related to parent, children, and submission
- submissions can relate to each other through communities.

Transitively, someone could create joins across all these relations.

Scaling relational databases: sharding

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

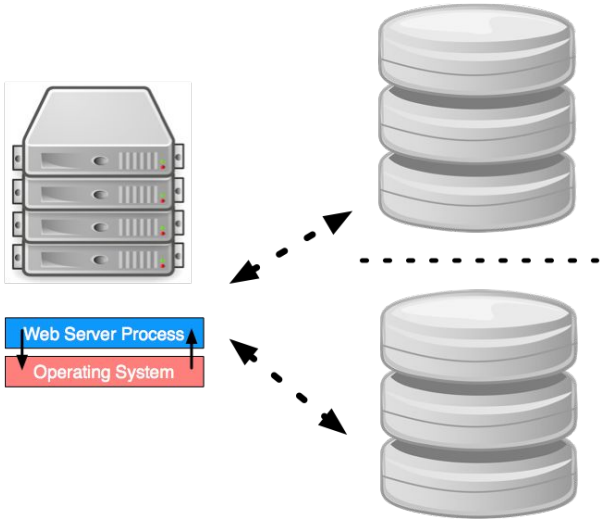
- ✔️ **Sharding**: splitting comments between multiple DBs
- 🚫 **Not sharding**: keeping comments and submissions in separate DBs



Scaling relational databases: 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.

⁉️ **How should we do this?**

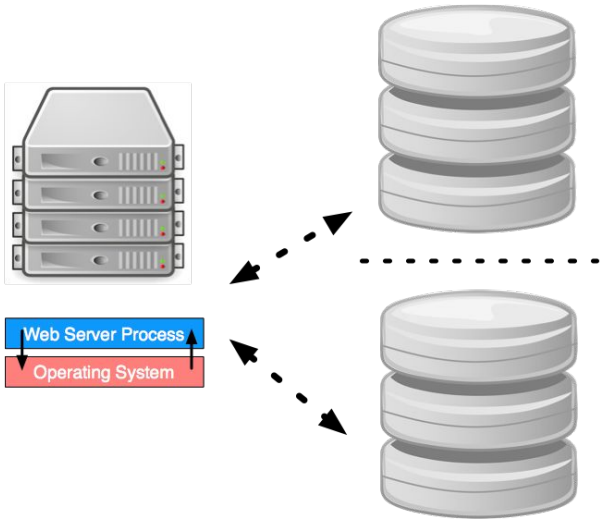


Scaling relational databases: sharding

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

⁉️ **How should we do this?**

- at the appserver layer?
- at the load balancer?
- across multiple load balancers?

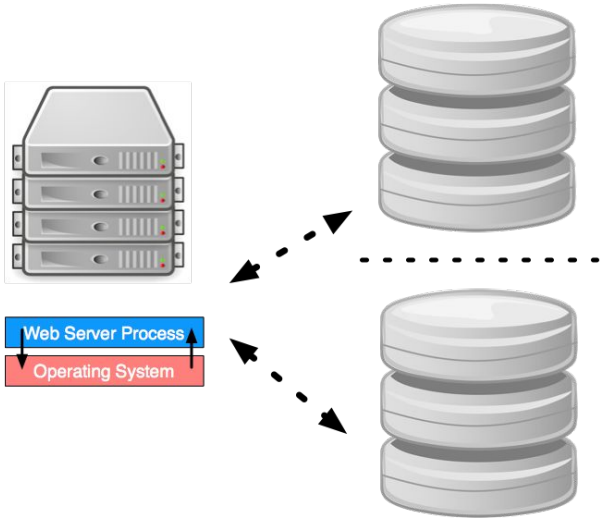


Scaling relational databases: sharding

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

⁉️ **How should we do this?**

- **at the appserver layer?**
  - 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!



Scaling relational databases: 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.

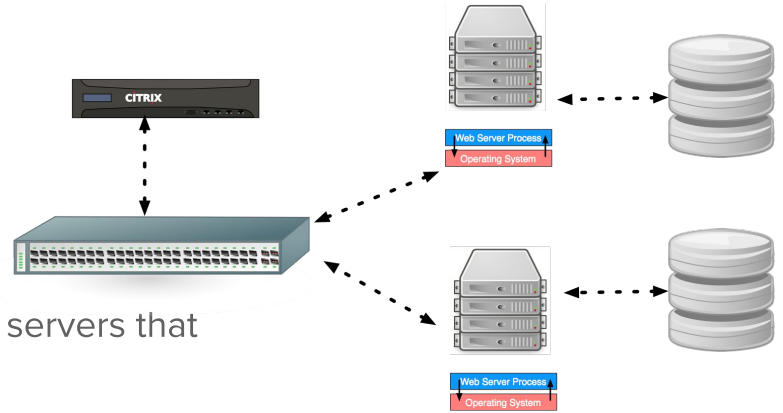
⁉️ **How should we do this?**

- **at the appserver layer?**

```
if (user.name.startsWith('j')) {
  database.connect('shard1')
} else if (user.name.startsWith('k')) {
  database.connect('shard2')
} else {
  // ...
}
```

Scaling relational databases: 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.



⁉ **How should we do this?**

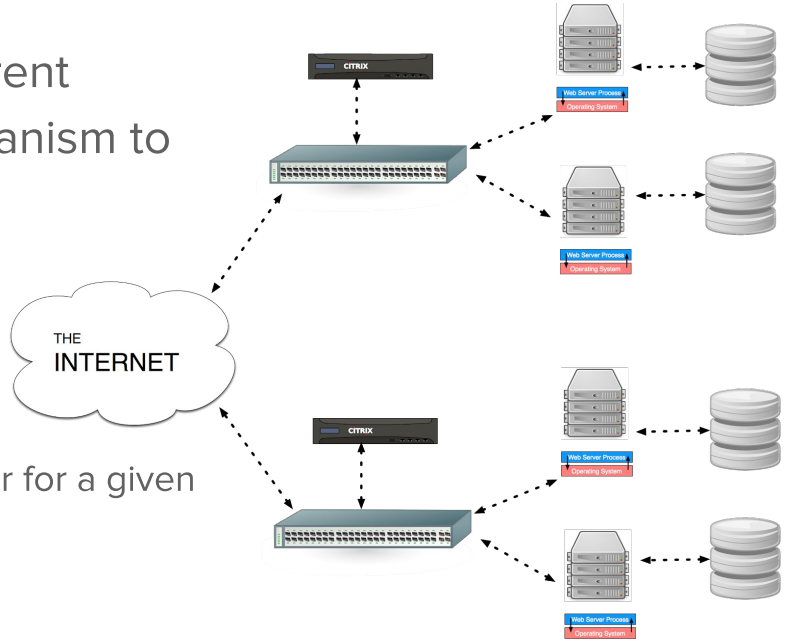
- **at the load balancer?**
  - load balancer is configured to route requests to app servers that are talking to the right database
  - mapping decision can only be based on information visible to LB
    - resource
    - cookie
    - host header
    - ...

Scaling relational databases: 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.

⁉️ **How should we do this?**

- **across load balancers?**
  - DNS is configured to point to the right load balancer for a given request
    - na6.salesforce.com
    - user.github.io

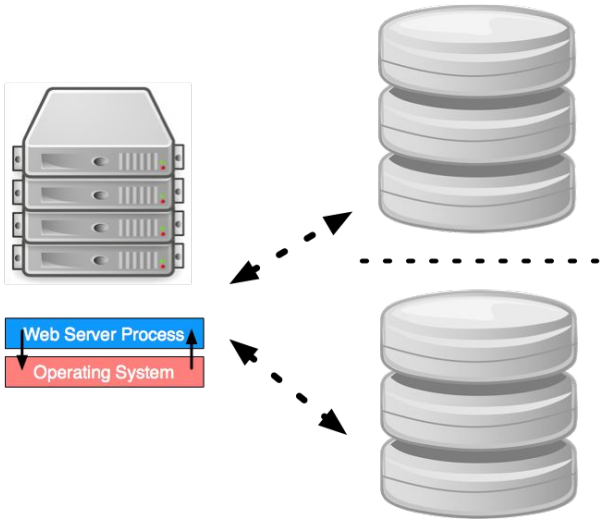


Scaling relational databases: sharding

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

⁉ **How should we do this?**

- at the appserver layer?
- at the load balancer?
- across multiple load balancers?



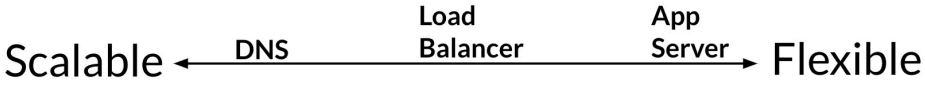
⁉ **What are the tradeoffs of each?**

Scaling relational databases: 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.

⁉️ **How should we do this?**

- at the appserver layer?
- at the load balancer?
- across multiple load balancers?

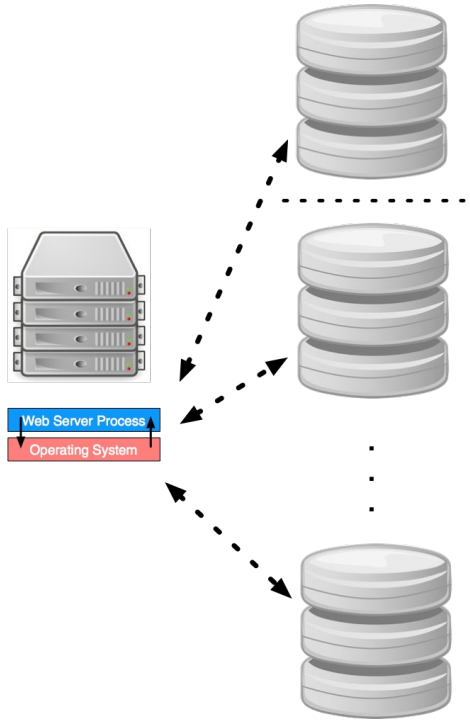


⁉️ **What are the tradeoffs of each?**

Scaling relational databases: sharding

Ideally, the number of your partitions increase as usage of your application increases.

💡 If each customer's data can be partitioned from the rest, then doubling the number of customers doubles the number of shards.



Scaling relational databases: sharding

⁉️ **How would we shard a database for Gmail?**

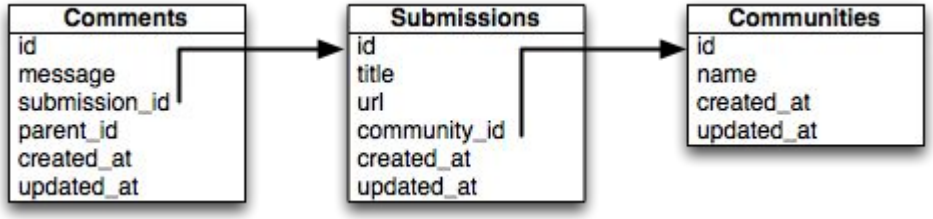
Scaling relational databases: sharding

⁉️ **How would we shard a database for 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

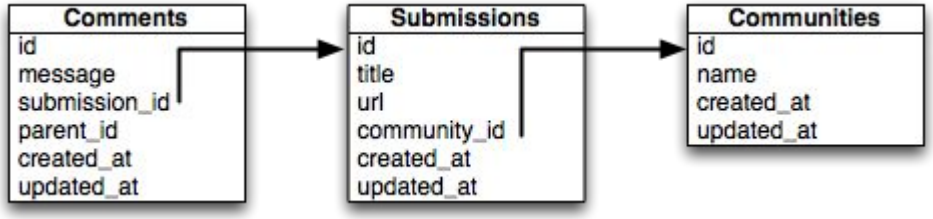
Scaling relational databases: sharding

⁉️ **How would we shard a database for this schema?**



Scaling relational databases: sharding

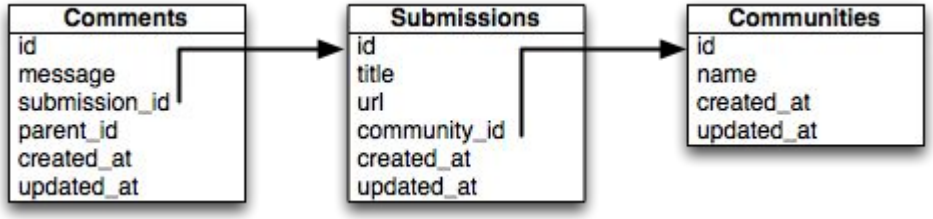
⁉️ **How would we shard a database for this schema?**



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

Scaling relational databases: sharding

⁉️ **How would we shard a database for this schema?**

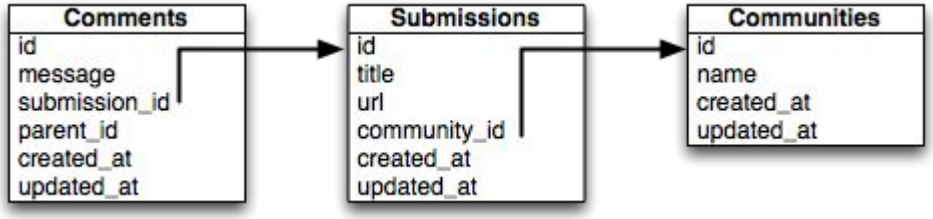


...by User?

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

Scaling relational databases: sharding

⁉️ **How would we shard a database for this schema?**

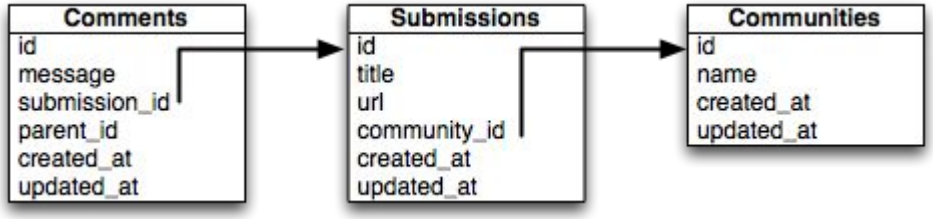


...by Submission?

- Users should be able to view many submissions when looking at a community. We might have to look up submissions in multiple databases.

Scaling relational databases: sharding

⁉️ **How would we shard a database for this schema?**

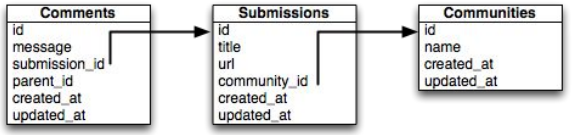


...by Community?

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

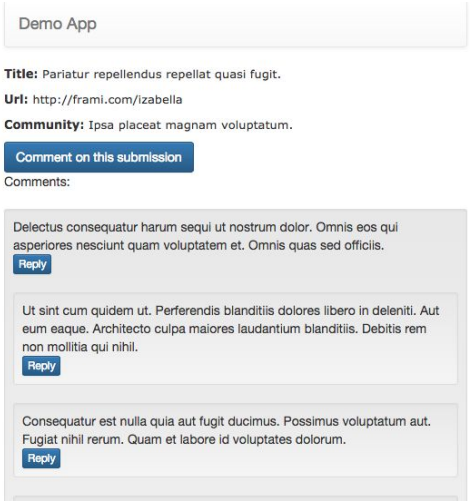
Scaling relational databases: sharding

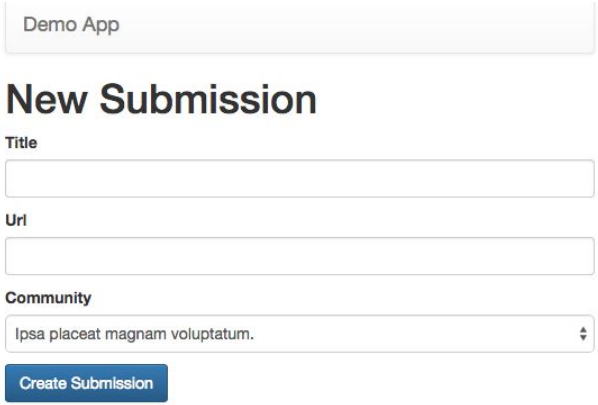
⁉ How would we shard a database for this schema?



...by Community?

Some parts of the user interface would work well with community-based sharding.



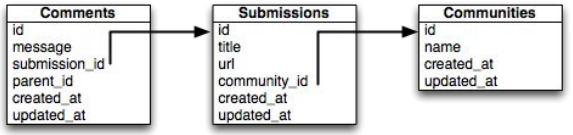


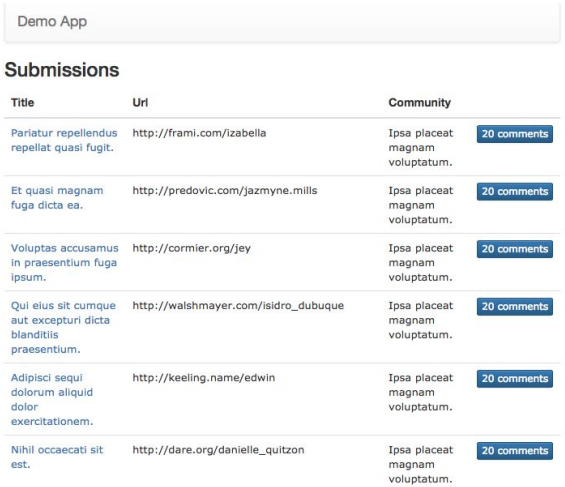
Scaling relational databases: sharding

⁉ **How would we shard a database for this schema?**

...by Community?

Other views would be more difficult (e.g. global views of submissions across communities).





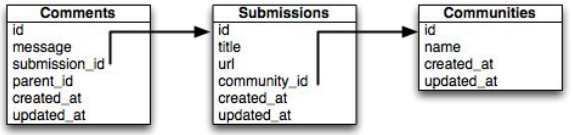
Scaling relational databases: sharding

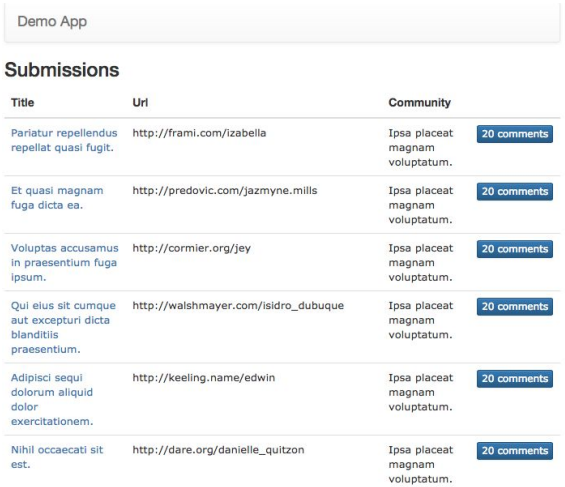
⁉ **How would we shard a database for this schema?**

...by Community?

Other views would be more difficult (e.g. global views of submissions across communities).

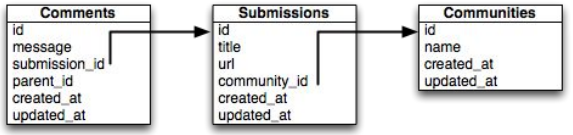
⁉ **Solutions?**





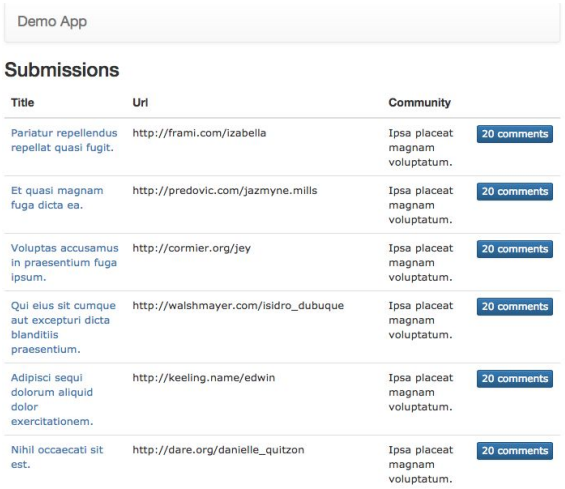
Scaling relational databases: sharding

⁉ How would we shard a database for this schema?



...by Community?

Other views would be more difficult (e.g. global views of submissions across communities).



⁉ Solutions?

- Modify the user interface so the difficult to shard page doesn't need to exist. 😉
  - A semi-static list of communities and you need to dig down into each to see what is submitted?
- Construct that page using aggregator service?

Scaling relational databases: sharding

Pros:

- If you genuinely have zero relations across shards, this scaling path is very powerful.
- Works best when partitions grow with usage.

Cons:

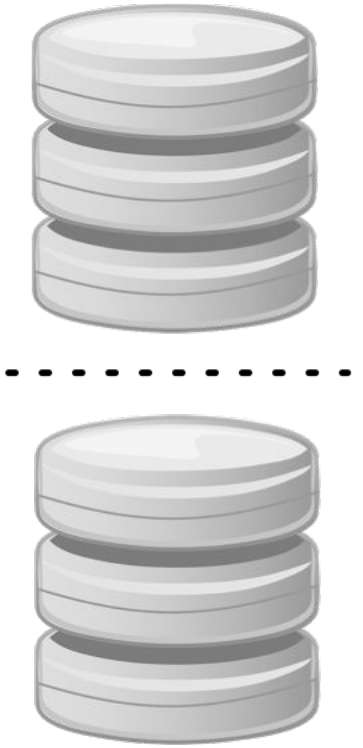
- Can inhibit feature development
  - Your app may be shardable today, but future features may change that 🤞
- Not easy to retroactively add to an application
- Transactions across shards don’t happen
- Consistent DB snapshots across shards don’t happen

Scaling relational databases: SOA

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

Service **O**riented **A**rchitectures do the opposite: partition data based on type and function.

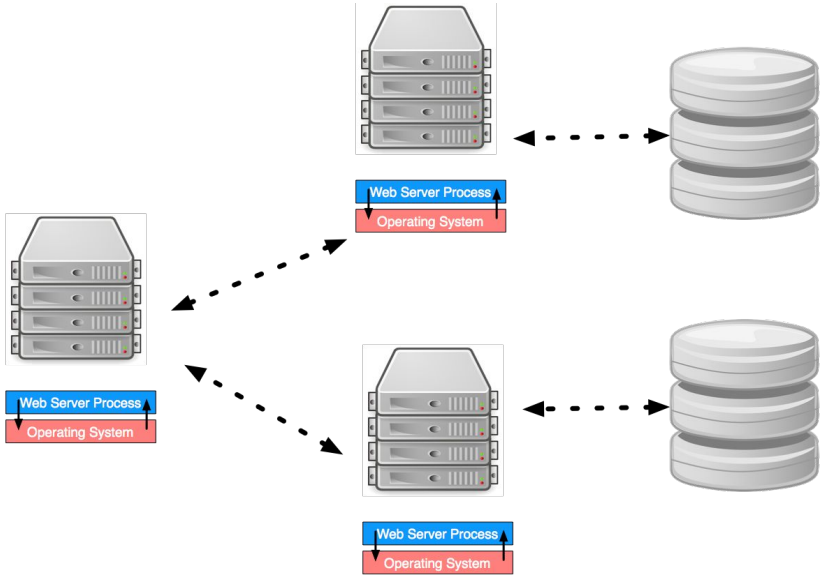
Like sharding, no relations will cross these partitions.



Scaling relational databases: SOA

In addition to separating data by function, SOAs tend to encapsulate the data within mini-applications 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.

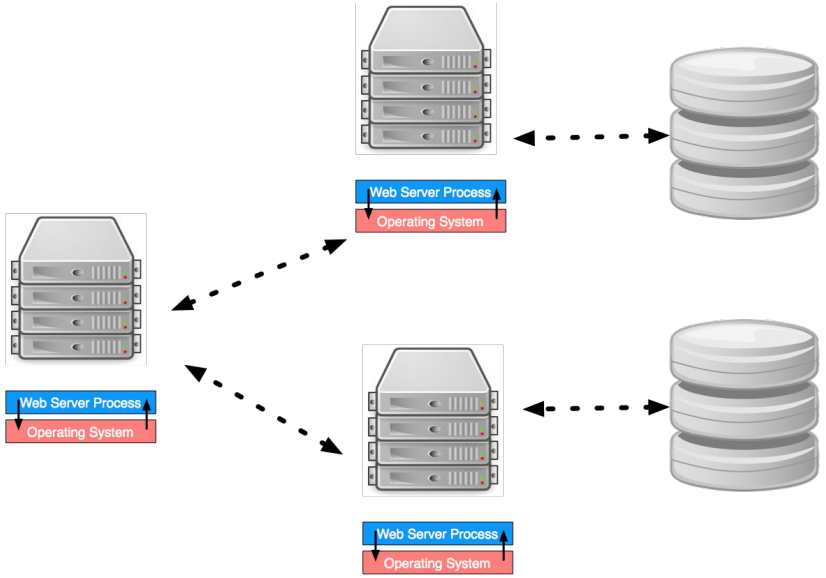


Scaling relational databases: SOA

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.

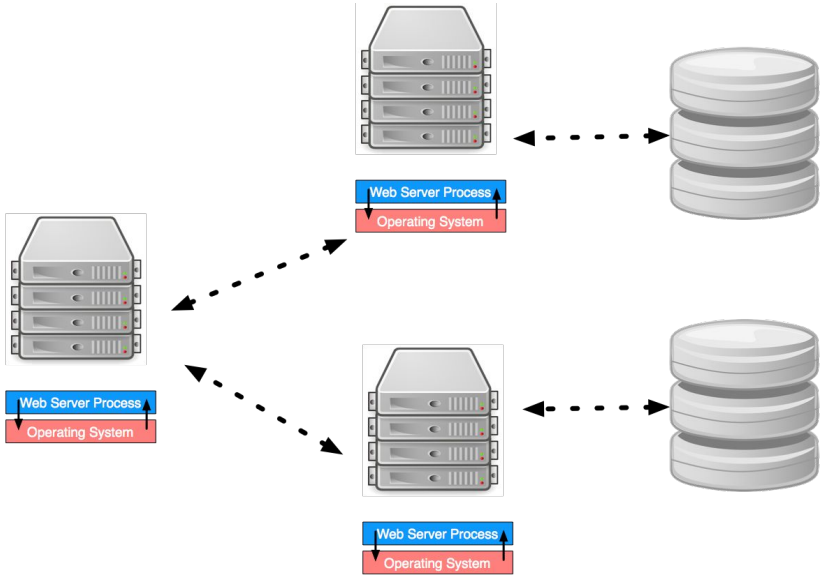


Scaling relational databases: SOA

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.

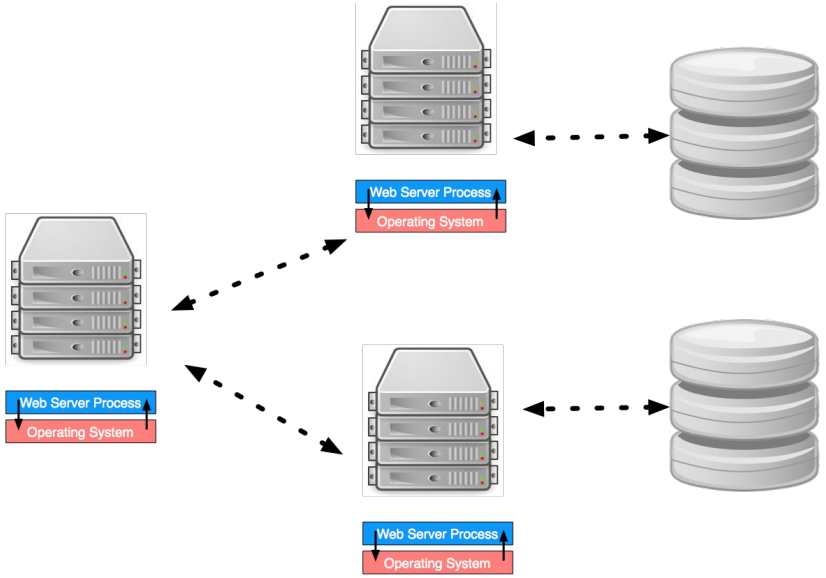
If you make your services small enough, they might be considered “**microservices**”.



Scaling relational databases: SOA

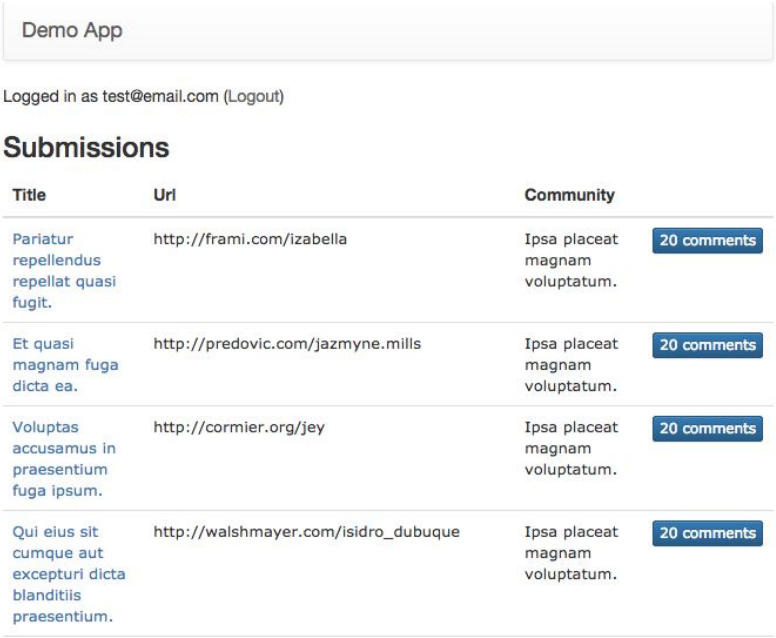
SOAs have other benefits:

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



Scaling relational databases SOA

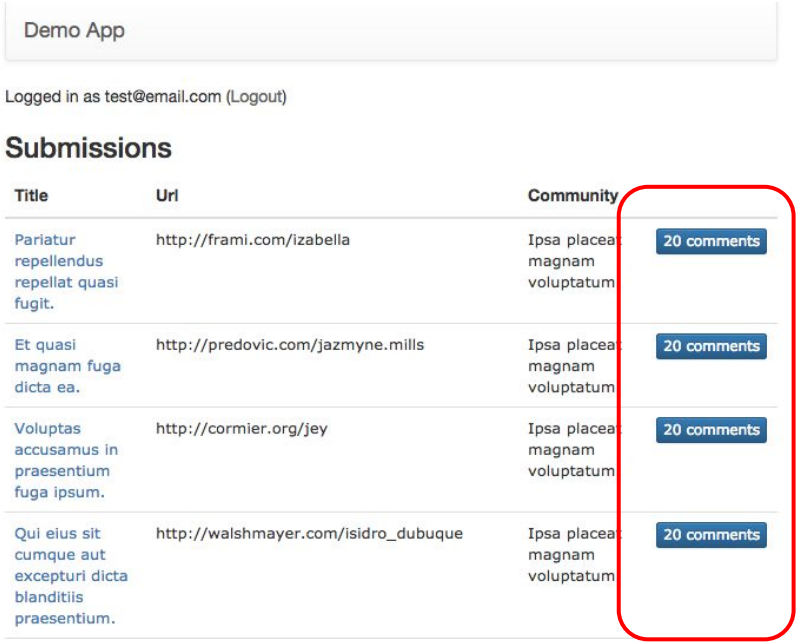
⁉ **How could you divide this app up into SOA?**



Scaling relational databases SOA

⁉ How could you divide this app up into SOA?

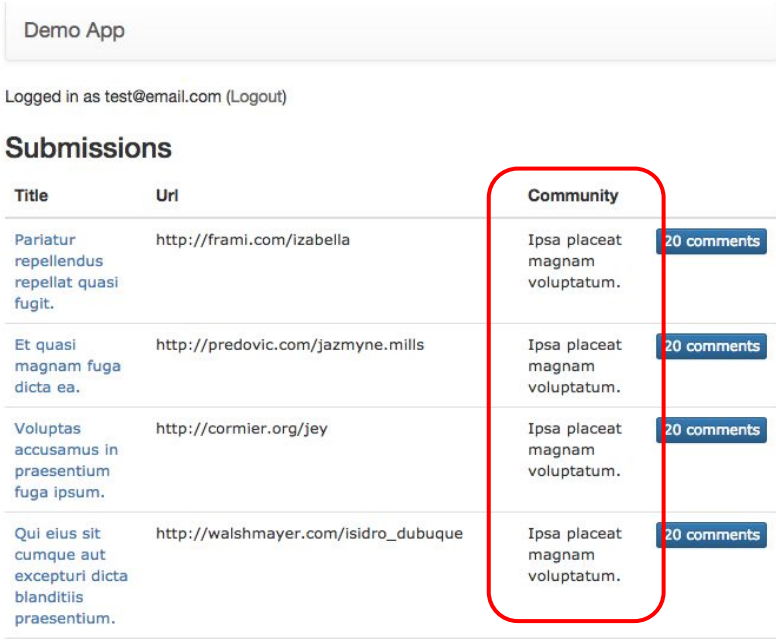
- Comments service can keep track of comments and replies for each submission.



Scaling relational databases SOA

⁉ **How could you divide this app up into SOA?**

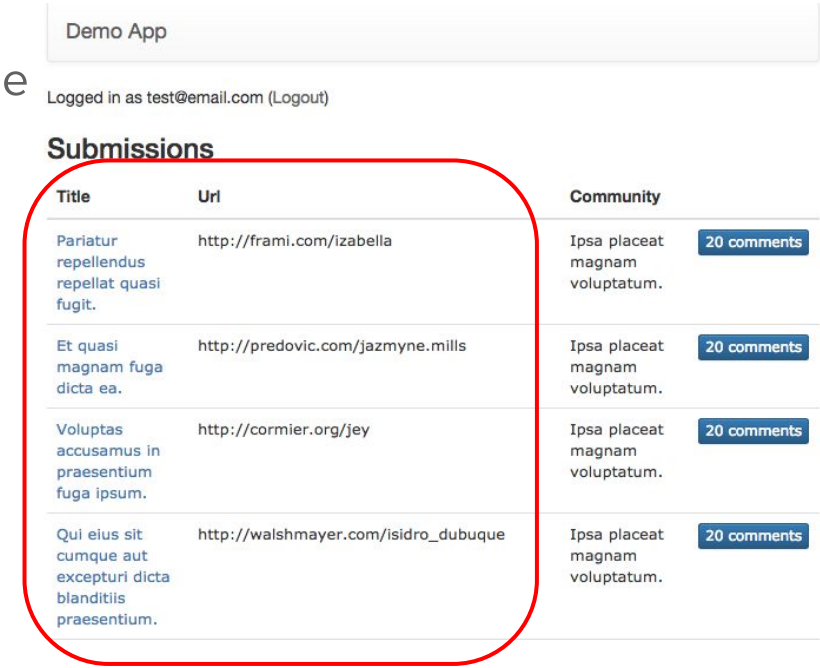
- Communities service can keep track of the list of communities, and who created them.



Scaling relational databases SOA

⁉ How could you divide this app up into SOA?

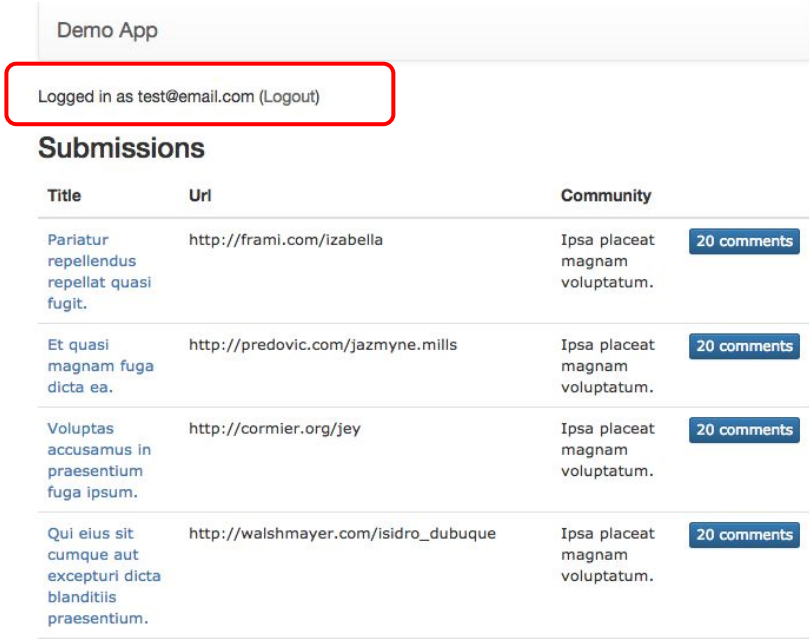
- Submissions service could keep track of the links that had been submitted.



Scaling relational databases SOA

⁉ **How could you divide this app up into SOA?**

- A users service can keep track of user management.



Scaling relational databases: SOA

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

⁉️ **Why JSON/HTTP/REST?**

Scaling relational databases: SOA

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

⁉️ Why JSON/HTTP/REST?

- Easily constructed
- Easily discovered: HTTP and JSON are both very readable. Documentation can be very light when the API is readable.
- Tech stack is shared (you're probably already building HTTP / appservers)

Scaling relational databases: SOA

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

⁉️ **Why JSON/HTTP/REST?**

- Easily constructed
- Easily discovered: HTTP and JSON are both very readable. Documentation can be very light when the API is readable.
- Tech stack is shared (you're probably already building HTTP / appservers)

⁉️ **Disadvantage?**

Scaling relational databases: SOA

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

⁉️ **Why JSON/HTTP/REST?**

- Easily constructed
- Easily discovered: HTTP and JSON are both very readable. Documentation can be very light when the API is readable.
- Tech stack is shared (you're probably already building HTTP / appservers)

⁉️ **Disadvantage?** Performance. 😢

- For high performance, use binary formats like Google Protocol Buffers.

Scaling relational databases: SOA

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!**

Scaling relational databases: SOA

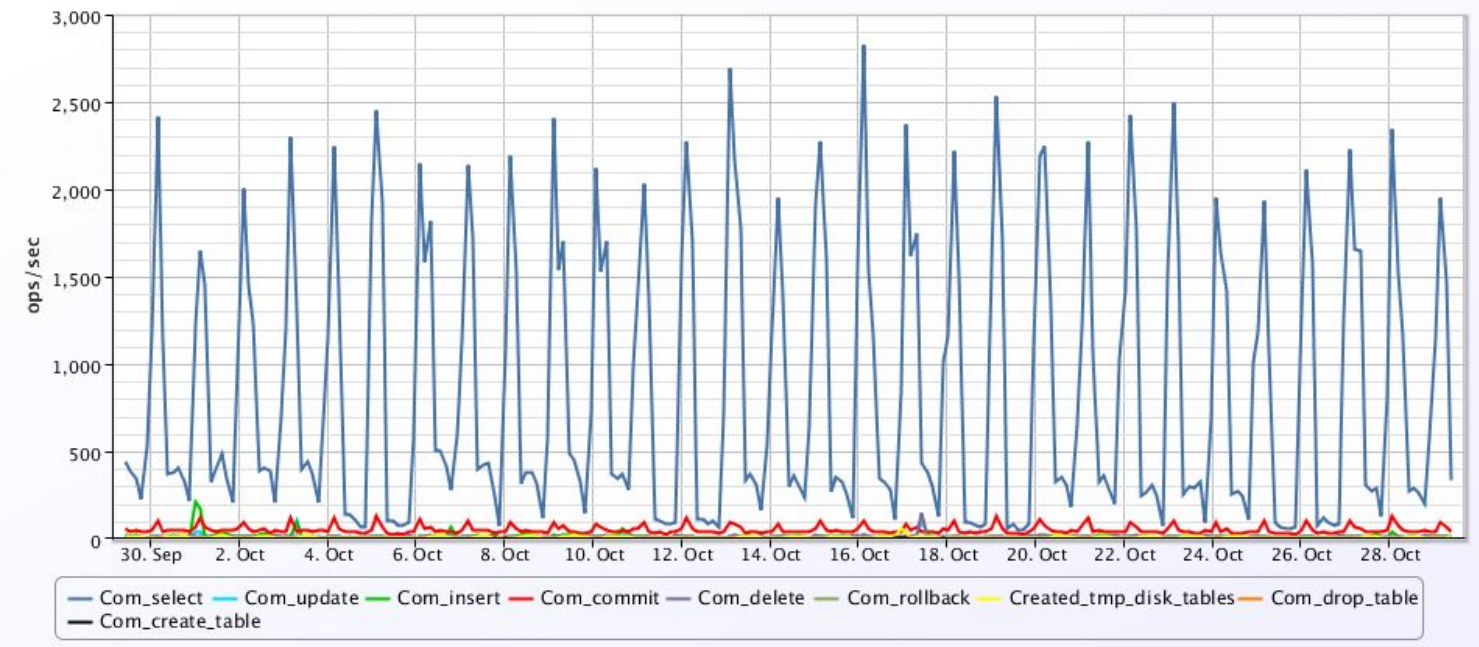
Pros:

- Small, encapsulated codebases
- Scales well as application size scales
- Scales well as team size scales

Cons:

- Doesn’t necessarily scale as number of users increases.
- Transactions across services don’t happen.
- Beware of microservices. ⚠️

Scaling relational databases: reads vs. writes

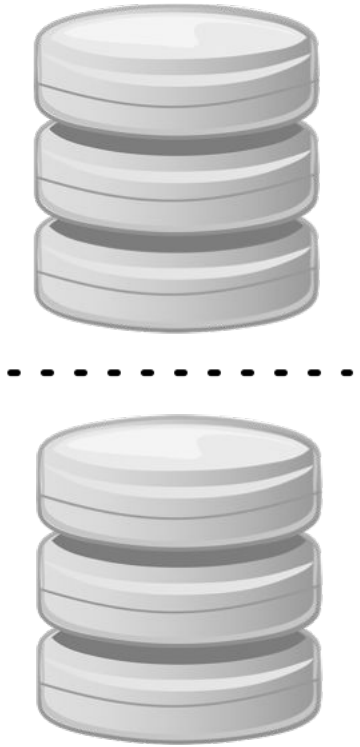


Scaling relational databases: reads vs. writes

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 replica”
- Whenever any transaction commits to the “master” database, send it over to the replica and apply it.

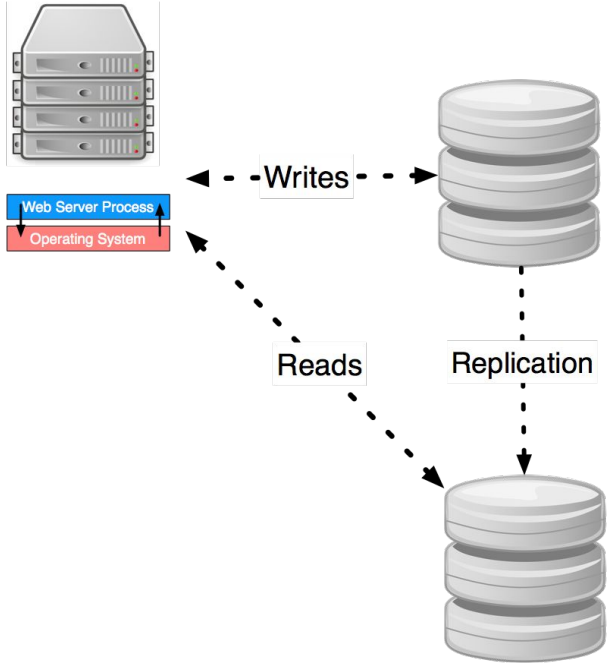


Scaling relational databases: reads vs. writes

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

If most traffic is read-only, and scaling read-only copies is easy, let's send our read traffic to a replica.

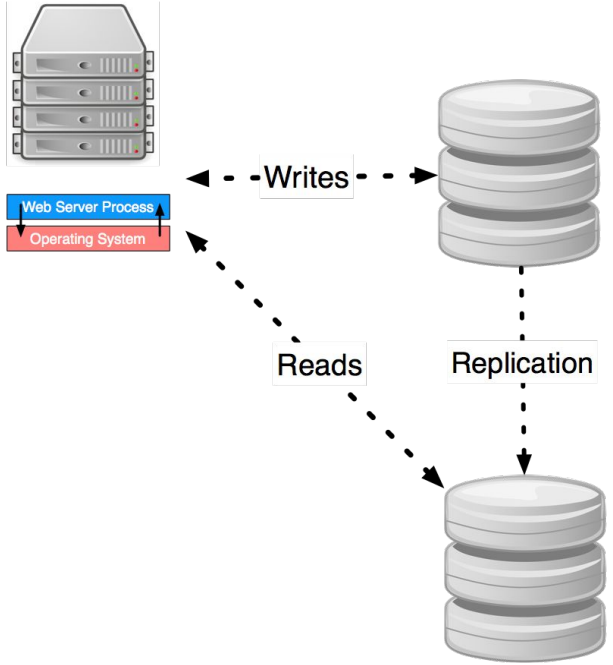
Replicas can be chained.



Scaling relational databases: 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.

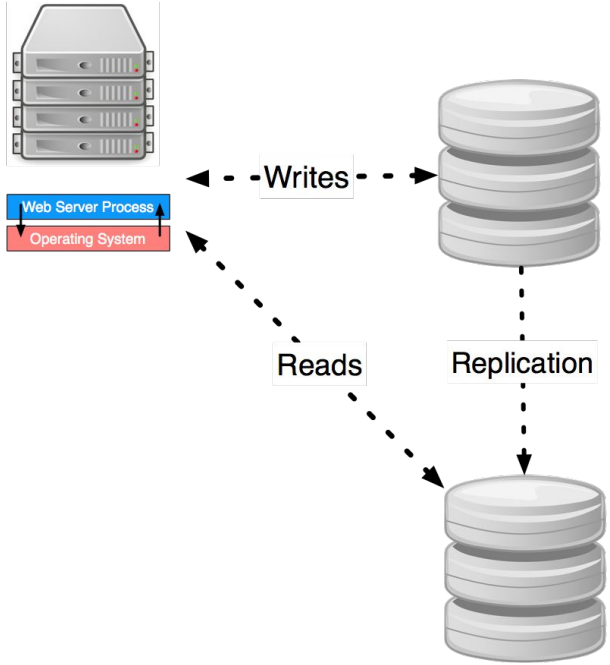


Scaling relational databases: reads vs. writes

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.

⁉ **Advantages / disadvantages?**



Scaling relational databases: reads vs. writes

What are the advantages of waiting until it a commit is applied to all replicas?

- Consistency. Subsequent read requests will see changes.

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

- Performance. There may be many replicas to apply changes to.

Scaling relational databases: reads vs. writes

How do the replicas catch up with the master?

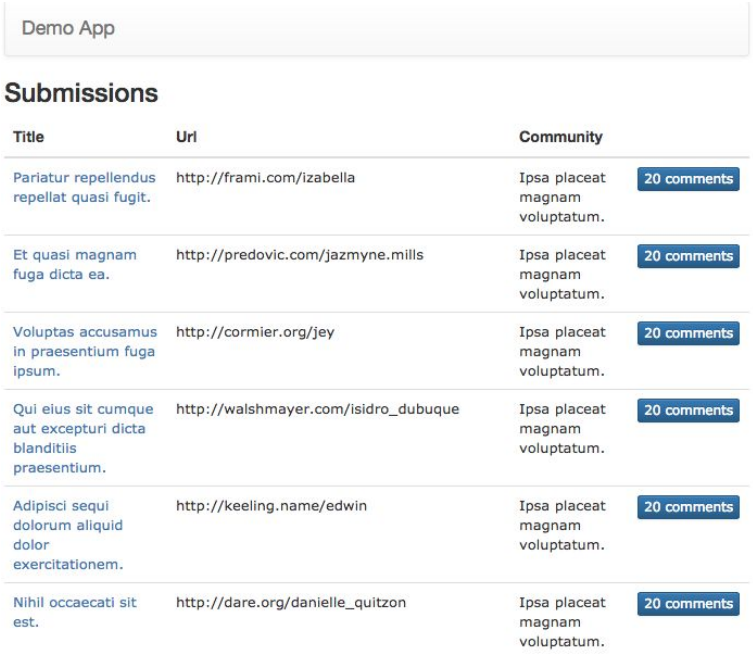
- statement-level: stream the journal from the master to the replica
- block-level: instead of sending the SQL statements to the replica, send the consequences of those statements

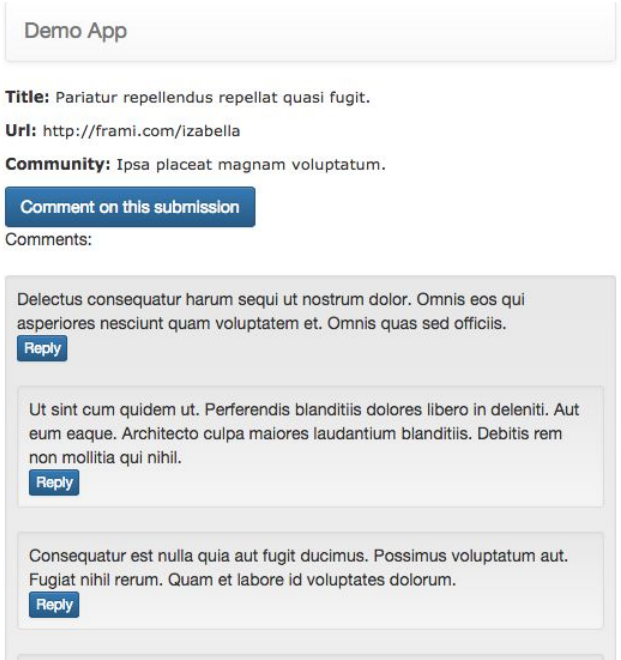
Statement-level is faster (queries are usually more compact than their consequences) with a catch! SQL statements now must be deterministic.

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

Scaling relational databases: reads vs. writes

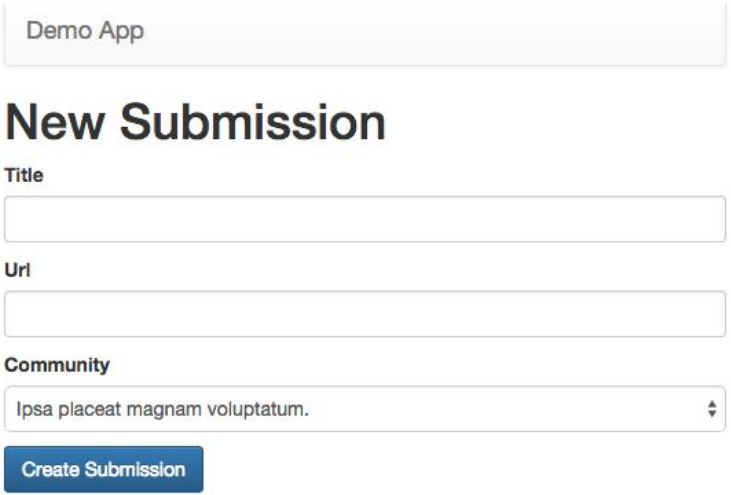
These responses could be served off read replicas.





Scaling relational databases: reads vs. writes

These would need to talk to the master database.



Scaling relational databases: reads vs. writes

Pros:

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

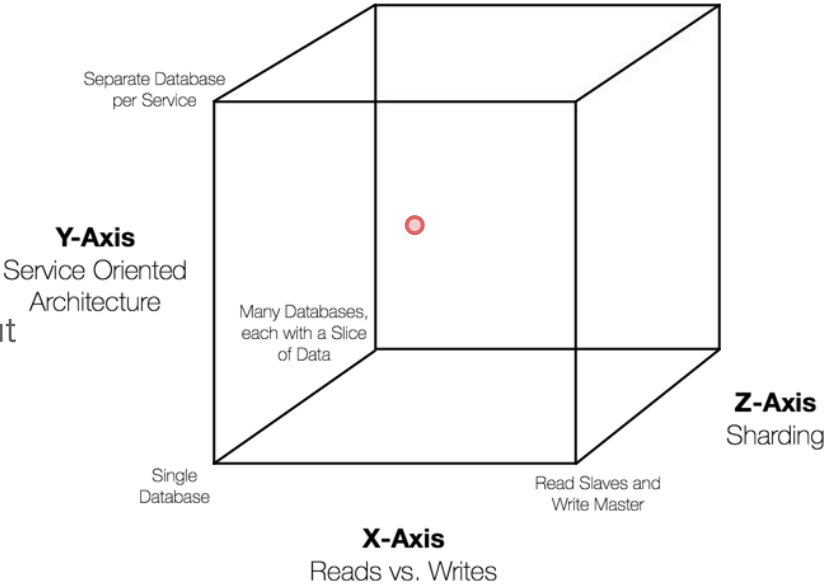
Cons:

- Application developer needs to think about where to direct queries.

Scaling relational databases at AppFolio

At Appfolio...

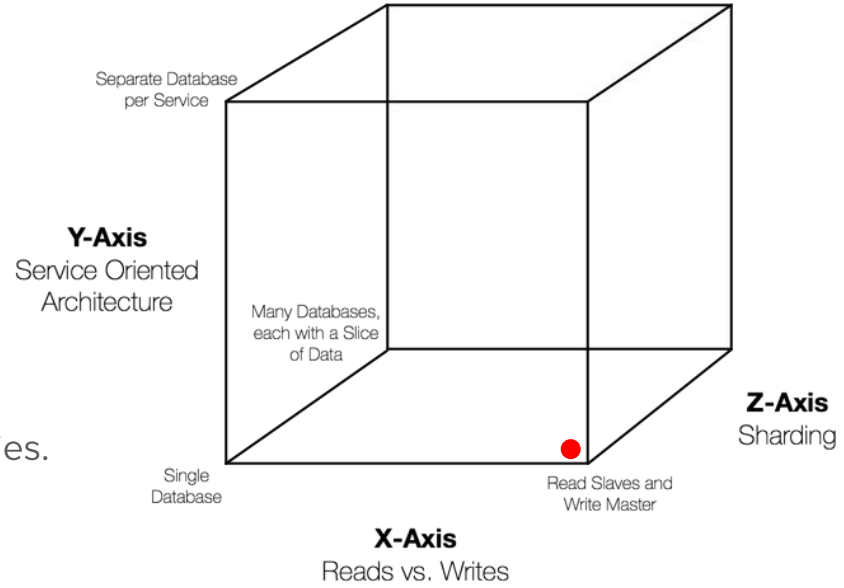
- High usage of sharding
  - Each customer in separate database.
- Medium use of SOA
  - Some functionality broken out into services, but more for scaling engineers than scaling load.
- Low use of read replicas
  - Replicas used for backup and analysis.



Scaling relational databases at Dynasty

At Dynasty...

- No usage of sharding!
  - One MySQL database for everything
- No use of SOA (at the database layer)!
  - Extensive use at the application layer
- Extensive use read replicas
  - Replicas used for backup, analysis, most queries.



Scaling relational databases: query analysis

```
mysql> EXPLAIN SELECT COUNT(DISTINCT `submissions`.`id`) FROM
`submissions`  JOIN `comments` WHERE `comments`.`submission_id` =
`submissions`.`id` AND `comments`.`message` = 'This is not a test!'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: comments
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 19188
        Extra: Using where
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: submissions
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: default_db_name.comments.submission_id
         rows: 1
        Extra: Using index
2 rows in set (0.00 sec)
```

Scaling relational databases: query analysis

How to read EXPLAIN output

select_type	The SELECT type
type	The join type
possible_keys	The indices available to be chosen
key	The index actually chosen
rows	Estimate of rows to be examined

Scaling relational databases: query analysis

`select_type`

The type of select statement being performed.

Most are fine, but two indicate potential performance problems:

- Dependent Subquery: reevaluated for every different value of the outer query
- Uncacheable Subquery: reevaluated for every value of the outer query

Scaling relational databases: query analysis

`type`

The type of JOIN being used. From best to worst:

- system - The table only has one row
- const - From uniqueness, we know only one row can match
- eq_ref, ref - Only one row at most can match from the previous table
- fulltext - mysql fulltext index
- ref_or_null - like ref, but also null values
- Index_merge
- Unique_subquery
- Index_subquery
- range - Only rows in a given range are retrieved, but can use index
- index - Full table scan, but can scan index instead of actual table
- ALL - Full table scan

Scaling relational databases: query analysis

`possible_keys & key`

Lists the indices that *could* possibly be used, and which was actually chosen.

- If you don’t like the index that MySQL is using, you can tell it to ignore indices using the IGNORE INDEX.
- If `possible_keys` is null, you have no indices that MySQL can use and should consider adding some.

Scaling relational databases: query analysis

`rows`

MySQL's estimate of how many rows need to be read. If this number is really big, that can indicate a problem.

- If this number is really big, that can indicate a problem.

Scaling relational databases: query analysis

After analyzing queries with EXPLAIN, optimizations take three forms:

- Add or modify indices
- Query optimizations
- Modify table structure (denormalization, for example)

Scaling relational databases: indices

What is an **index**?

- Fast, compact structure for identifying row locations
- Chop down your result set as quickly as possible
- MySQL will only use one index per table per query!
    - It cannot combine two separate indexes to make one more useful index

Indices work best when they can be kept in memory. Some ways to trim the fat:

- Can I reduce the characters in that VARCHAR index?
- Can I use a TINYINT instead of a BIGINT?
- Can I use an integer to describe a status instead of a text-based value?

Scaling relational databases: query optimization

```
mysql> explain select count(*) from txns where parent_id - 1600 = 16340
  select_type: SIMPLE
  table: txns
  type: index
  key: index_txns_on_reverse_txn_id
  rows: 439186
  Extra: Using where; Using index
```

Scaling relational databases: query optimization

```
mysql> explain select count(*) from txns where parent_id = 16340 + 1600
  select_type: SIMPLE
  table: txns
  type: const
  key: index_txns_on_reverse_txn_id
  rows: 1
  Extra: Using index
```

For lab tomorrow

- Only 2 more weeks dedicated to writing features!
- If your team is struggling to make progress, ask questions on Piazza and use office hours!
    - Schedule working sessions with your group outside of lab.