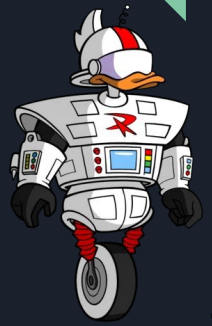
Lazy Defenses: Using Scaled TTLs to Keep Your Cache Correct

@brindelle, for Strange Loop

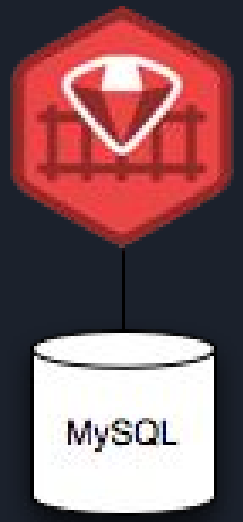
Hello!



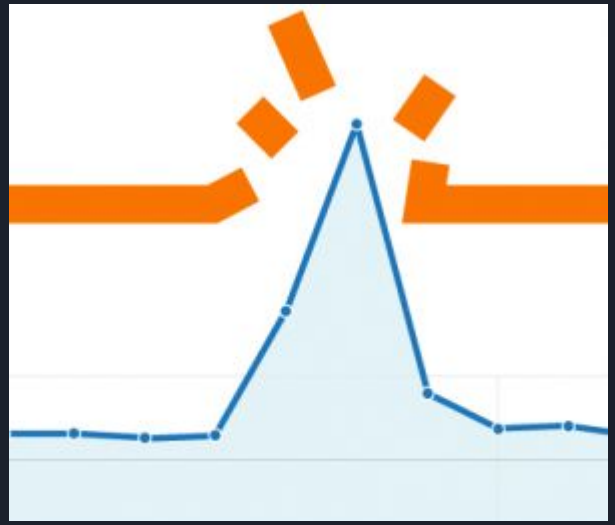
- TL on Core Services at Twitter
- @brindelle
- blog.bonnieeisenman.com
- http://bit.ly/lrnamaff

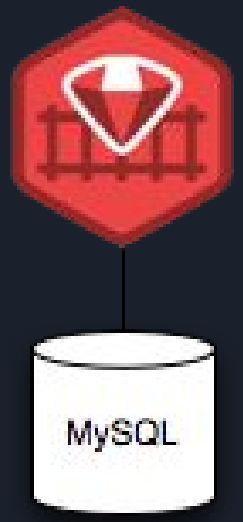
Image source: http://disney.wikia.com/wiki/File:DuckTales_Remastered_-Gizmo_Duck.png

A Common Story

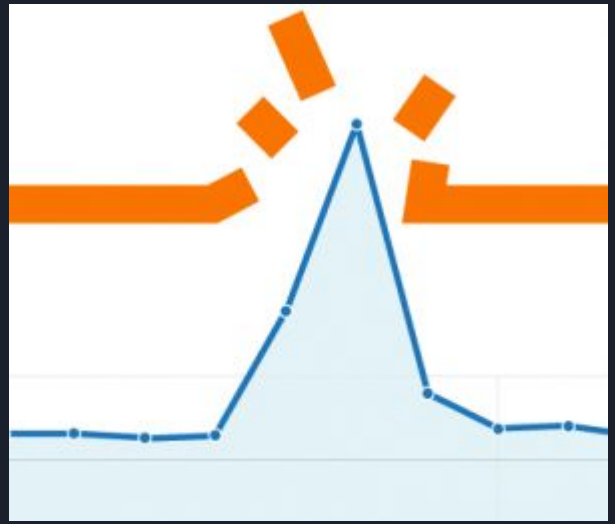


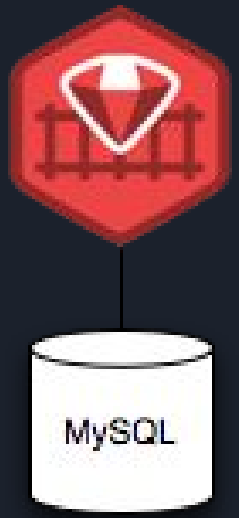
A Common Story

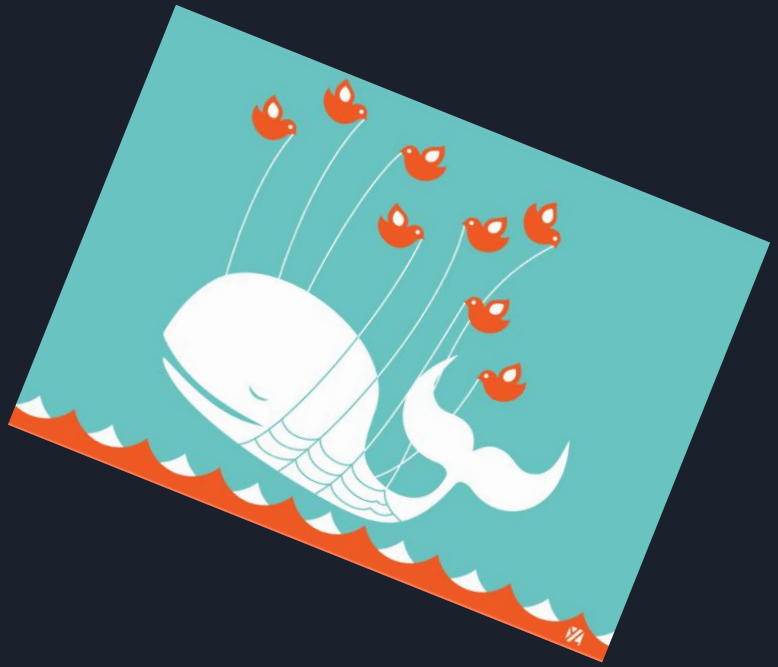




A Common Story







So what do you do?

Spoiler alert: there are lots of right answers

"Premature optimization is the root of all evil."

Today, let's talk about **caching**.

caching, verb

storing and retrieving data from a high-performance store

Image from https://flic.kr/p/qaxbGk

"Numbers every programmer should know"

Main memory reference = **100 ns**

SSD random read = **16,000 ns = 16µs**

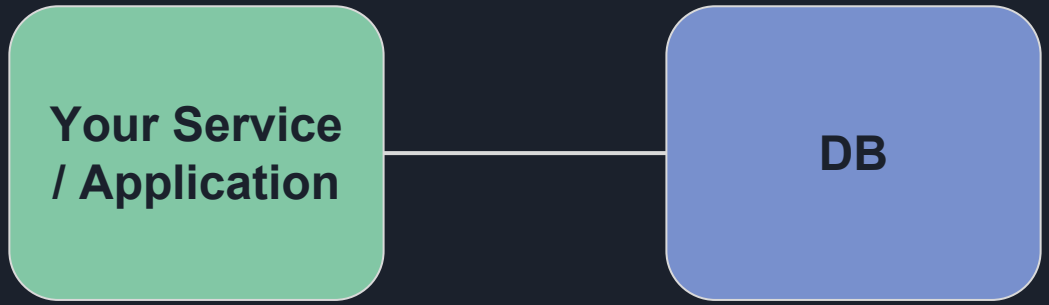
Same-DC roundtrip = **500,000ns = 500µs**

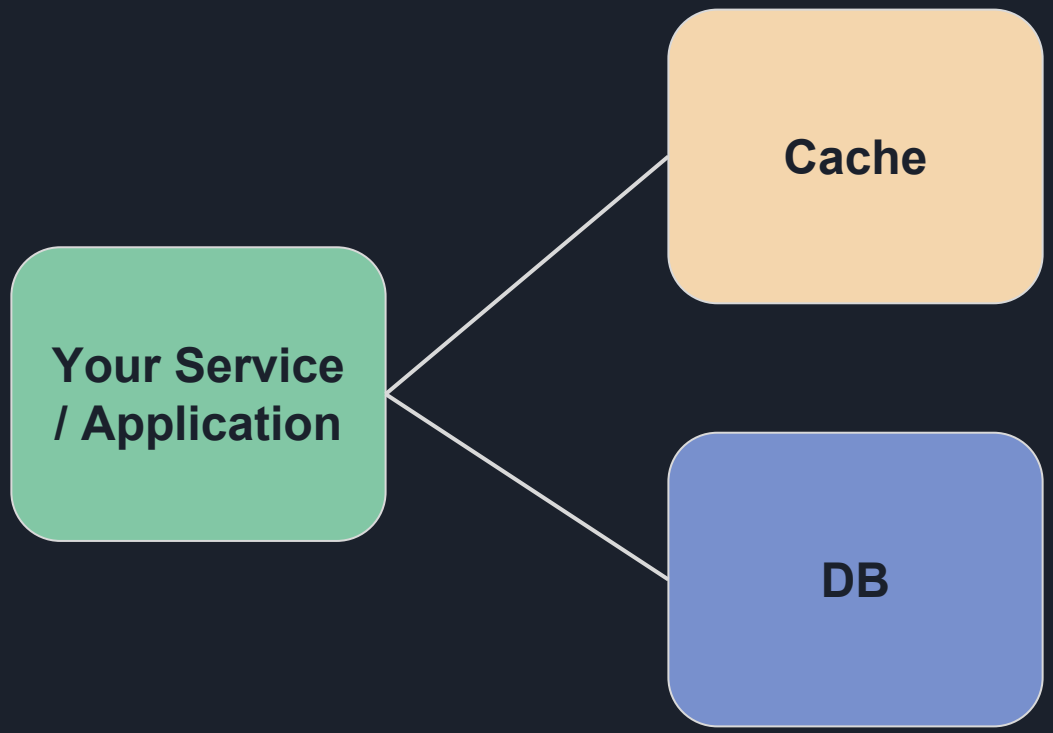
Disk seek = **3,000,000ns = 3,000µs**

Sources: https://github.com/colin-scott/interactive_latencies


redis

https://github.com/twitter/twemcache


Your Service
/ Application
DB


Your Service
/ Application
Cache
DB

Some questions...

- What do you store in cache?
- For how long?
- What happens when cache is full?
- What do you do when the data changes?

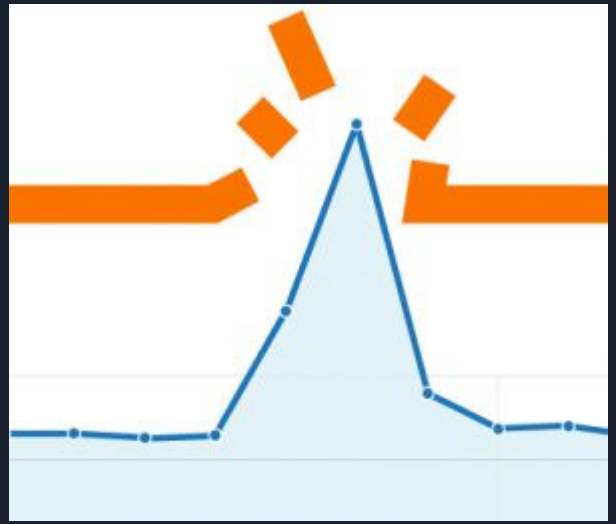
...and some terms...

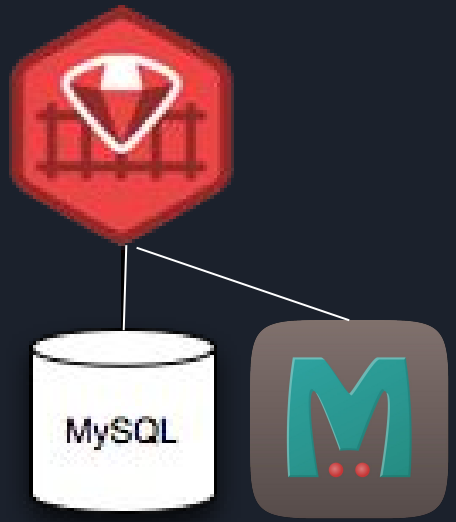
- TTL (time to live)
- "hit" / "miss"
- LRU (least recently used)

A naive solution: read-through LRU

- All queries check cache first
- On miss, query DB, put data in cache w/ a TTL
- On write, invalidate the key
- Treat expired keys as misses
- When the cache is full, evict based on LRU

Does that work?





Well....sometimes.

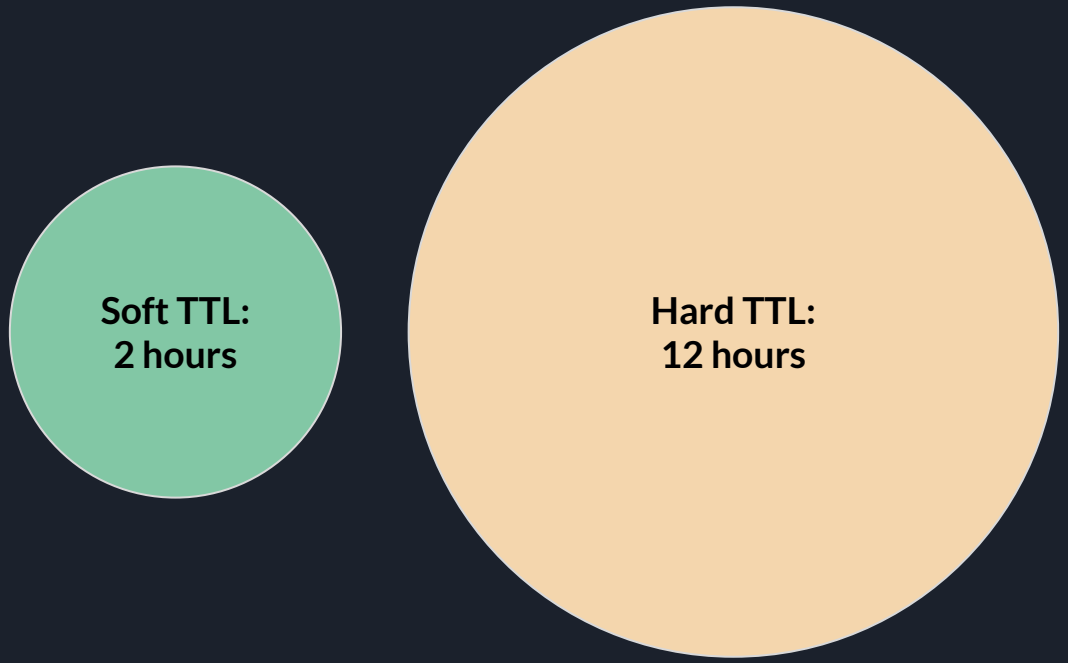
Naive solution: problems

- End-users pay latency penalty of a cache miss
- Misses are common, and slow
- Backing stores see inconsistent QPS

Good news:
this is solvable.

Soft vs hard TTLs

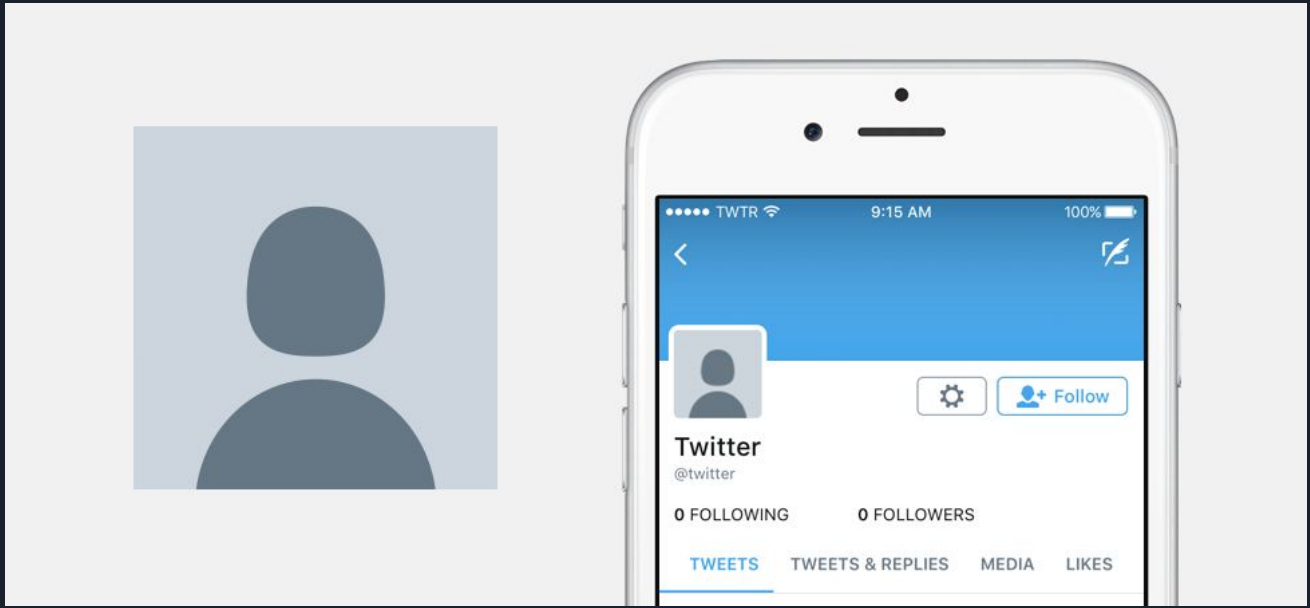
- Soft TTL < hard TTL
- On reads, if soft TTL is expired, treat as "cache hit" but refresh the cache in the background
- If hard TTL expired, treat as cache miss


Soft TTL:
2 hours
Hard TTL:
12 hours

* this is not supported by memcached native ttl

OK, let's stop talking hypotheticals.

The User Service



millions

of queries per second

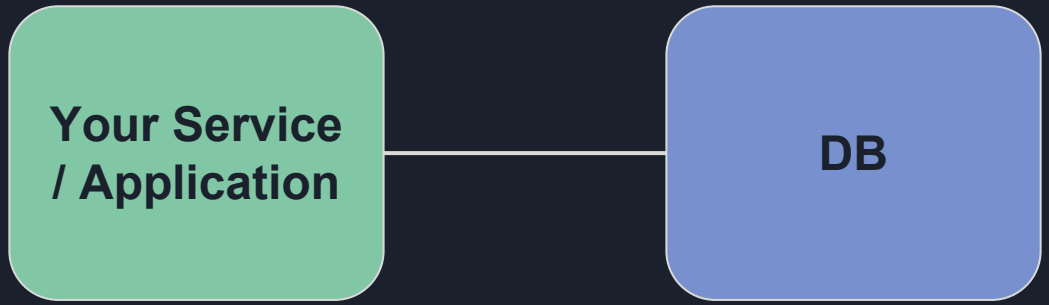
thousands

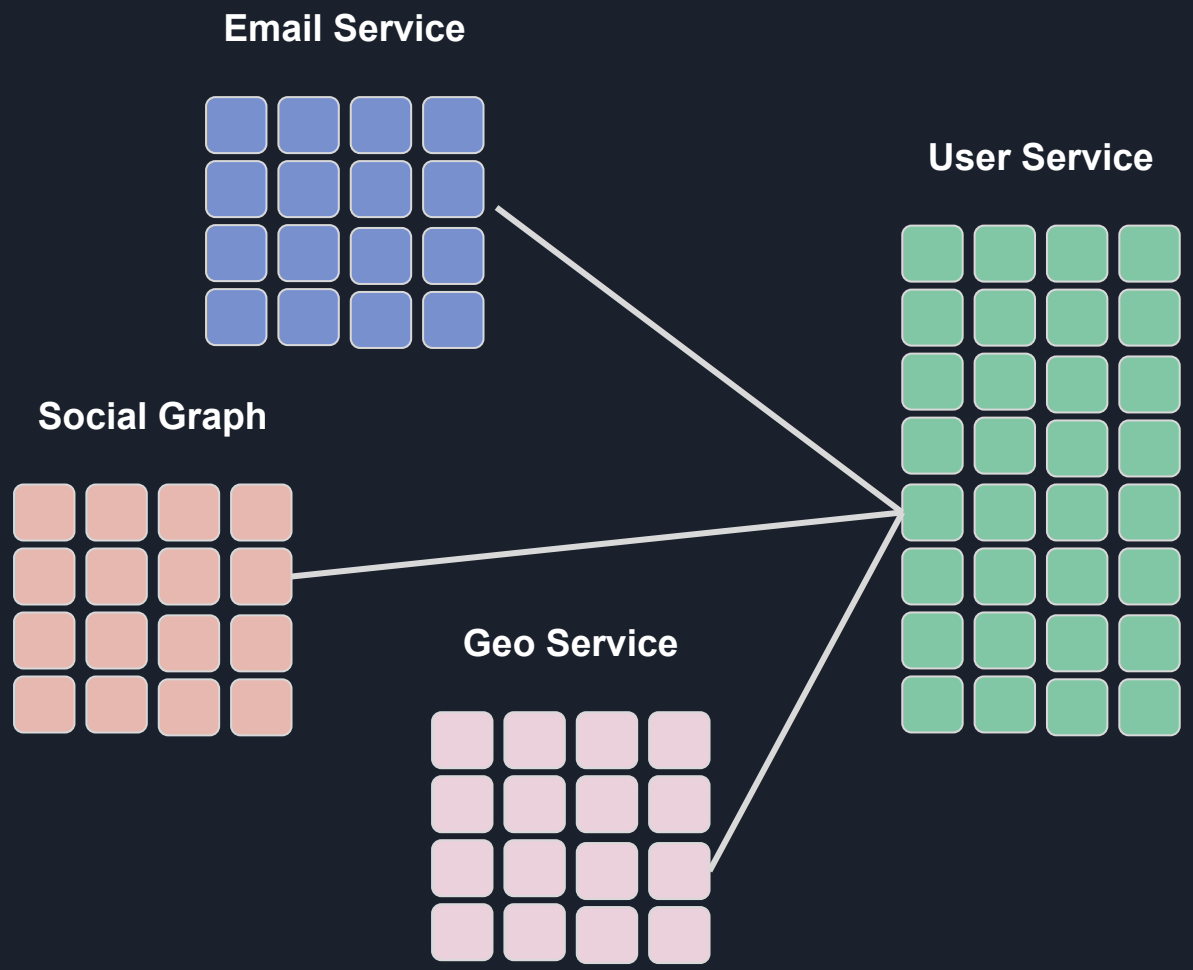
of instances

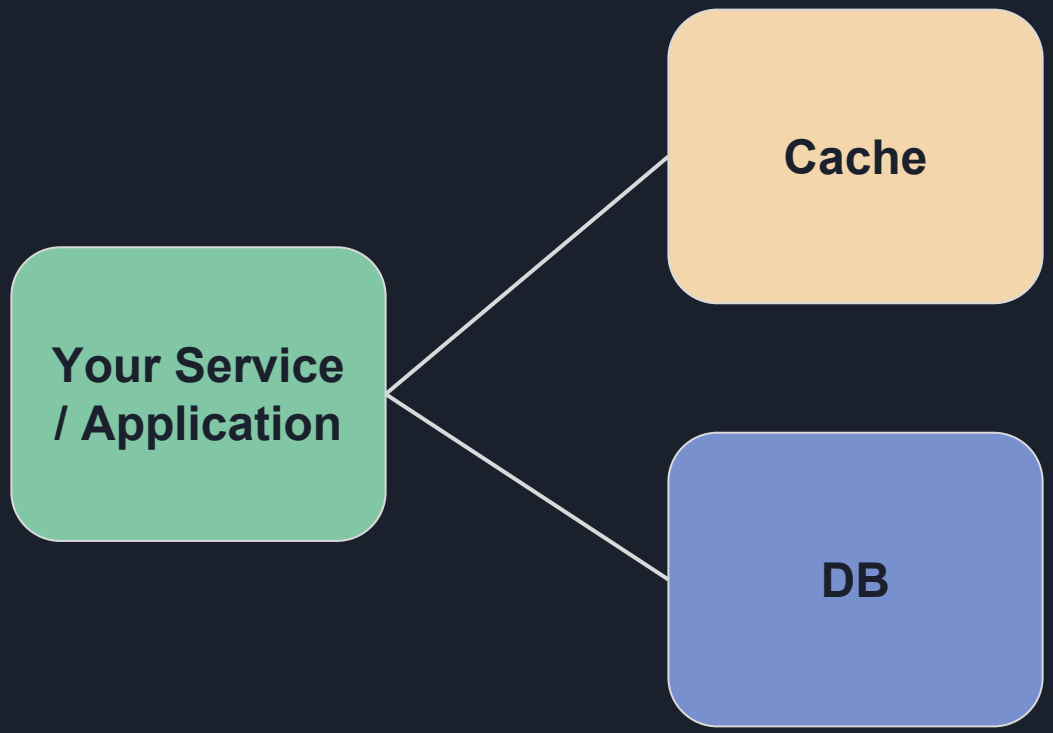
The User Service

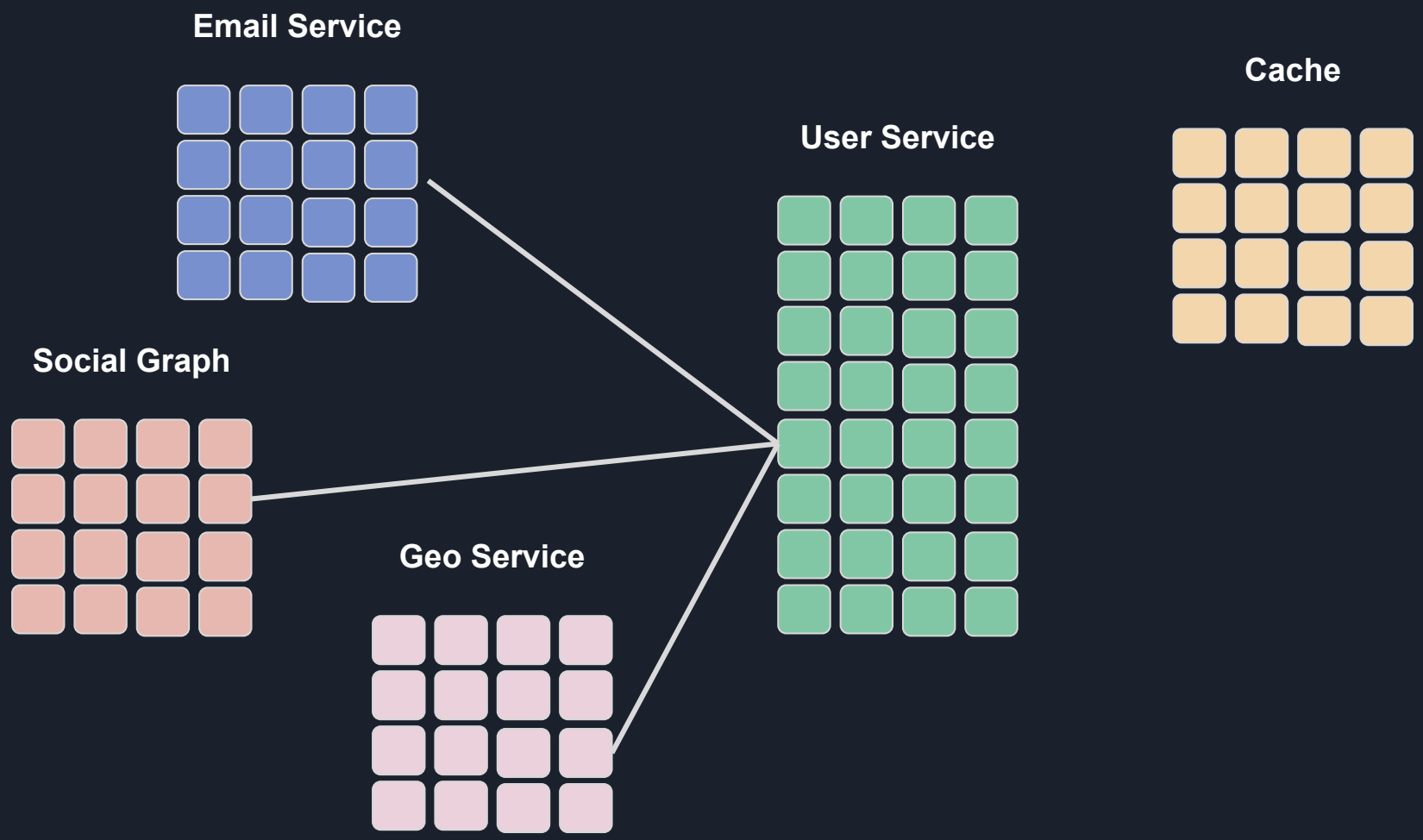
- Reads >>> writes
- Spikes driven by major events
- Uneven traffic across the ID space
- Writes are often clustered
- Huge dataset

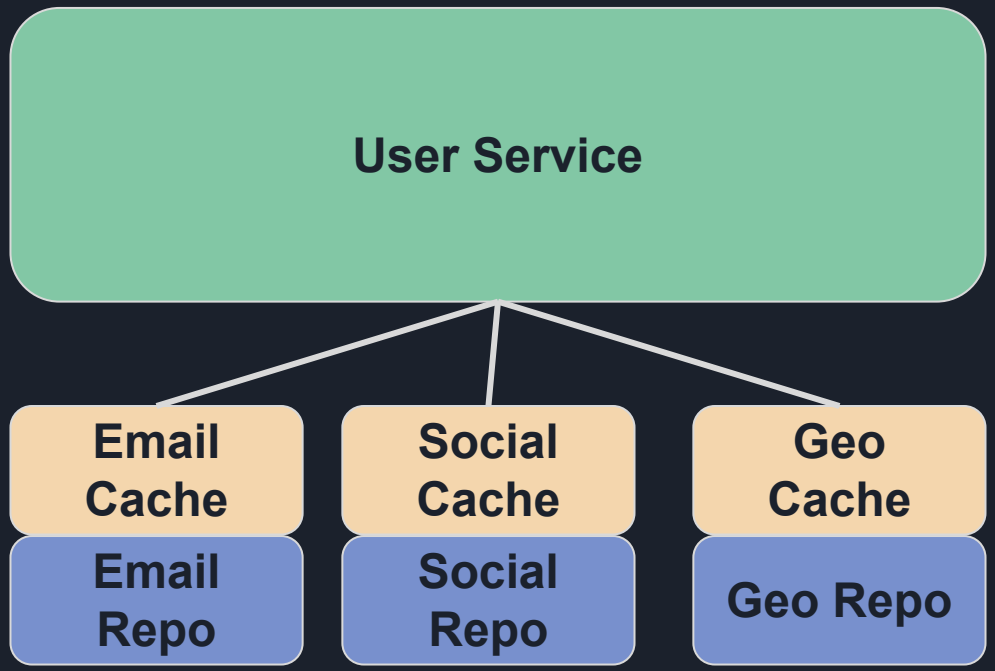
...each service has very different characteristics :)


Your Service
/ Application
DB


Email Service
User Service
Social Graph
Geo Service


Your Service
/ Application
Cache
DB


Email Service
Cache
User Service
Social Graph
Geo Service


User Service
Email Cache
Email Repo
Social Cache
Social Repo
Geo Cache
Geo Repo

Why cache?

Why cache?

- Respond faster than backing stores

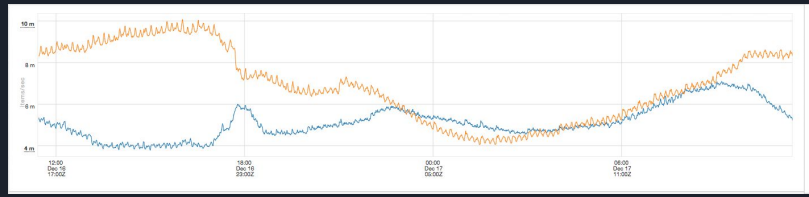
Why cache?

- Respond faster than backing stores
- Reduce pressure on backing stores
  - less traffic, more regular QPS

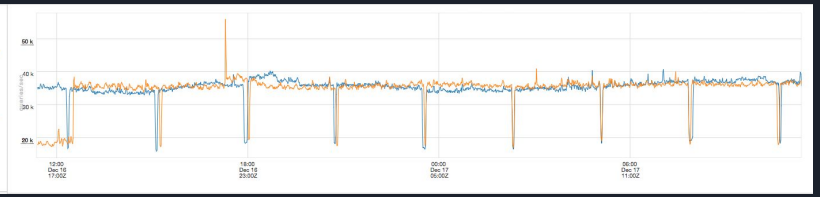
Why cache?

- Respond faster than backing stores
- Reduce pressure on backing stores
  - less traffic, more regular QPS

Before



After



99.9%+

typical cache hit rate

Problem #1: Spikes

https://blog.twitter.com/official/en_us/a/2014/looking-back-at-the-2014-oscars-on-twitter.html

"Numbers every programmer should know"

Main memory reference = 100 ns

SSD random read = 16,000 ns = 16µs

Same-DC roundtrip = 500,000ns = 500µs

Disk seek = 3,000,000ns = 3,000µs

Sources: https://github.com/colin-scott/interactive_latencies

Solution:
in-process hotkey cache

In-process hotkey cache

- "Hot" keys
  - > n lookups per minute for a given instance
- 30 second TTL
  - no hard/soft distinction

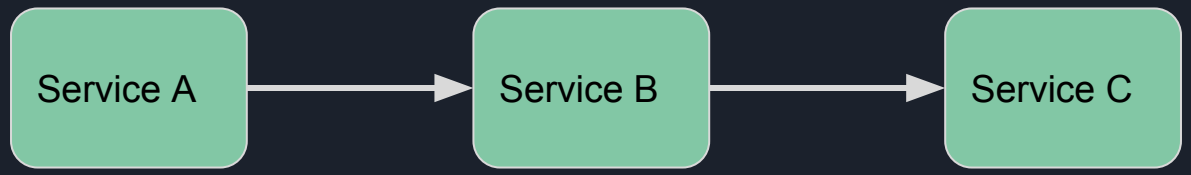
Other events that we no longer notice...

- the world cup
- the Oscars
- the Olympics
- presidential inaugurations
- ...etc

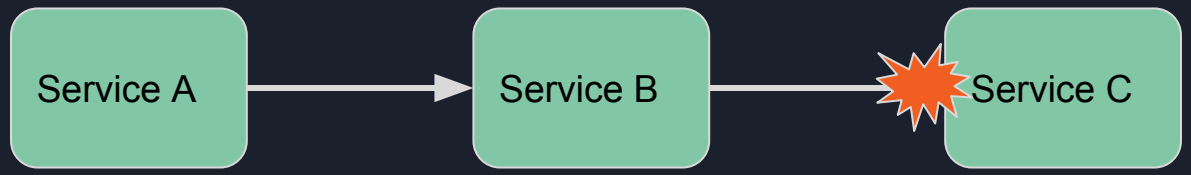
Problem #2: Outages

Source: https://www.flickr.com/photos/computerhotline/7796986926;
License: https://creativecommons.org/licenses/by/2.0/

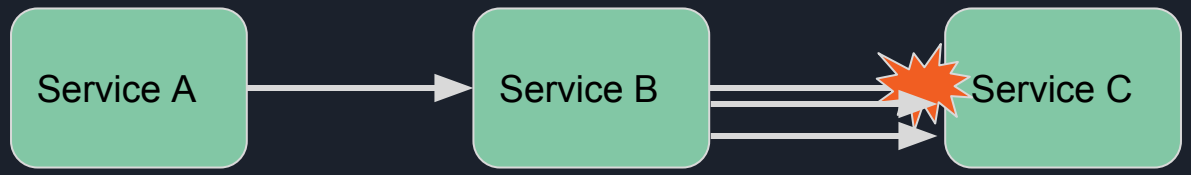
Problem: Retry Storms



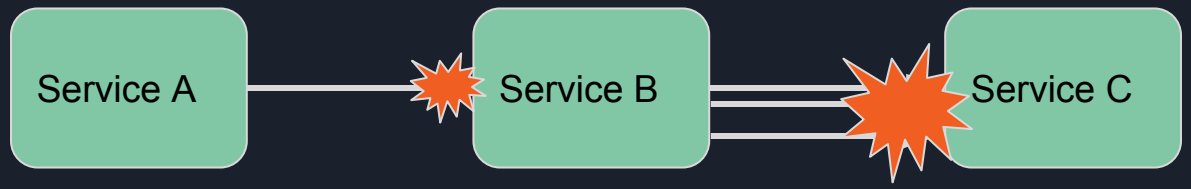
Problem: Retry Storms



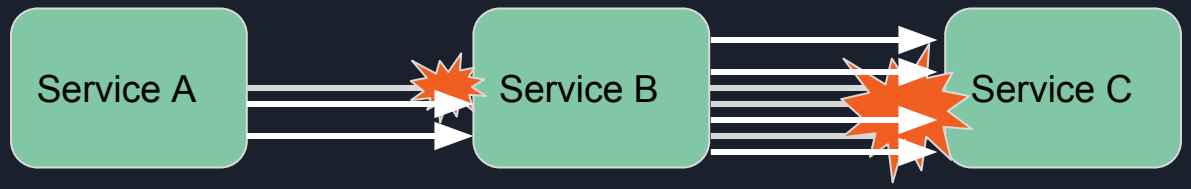
Problem: Retry Storms



Problem: Retry Storms



Problem: Retry Storms



Solution: darkmoding

- Pre-emptively short-circuit and serve stale data from cache
- Continue sending some requests to backends and monitor their SR
- As backends recover, gradually restore traffic

TLDR; darkmoding is great

The Case of the Missing Account

Twitter, Inc. [US] | https://twitter.com/lakers

Sorry, that page doesn't exist!

You can **search Twitter** using the search box below or **return to the homepage**.

Etsy Code as Craft

Blameless PostMortems and a Just Culture

Posted by **John Allspaw** on May 22, 2012

Twitter, Inc. [US] | https://twitter.com/lakers

Sorry, that page doesn't exist!

You can **search Twitter** using the search box below or **return to the homepage**.

What do you do when you don't trust your datastore?

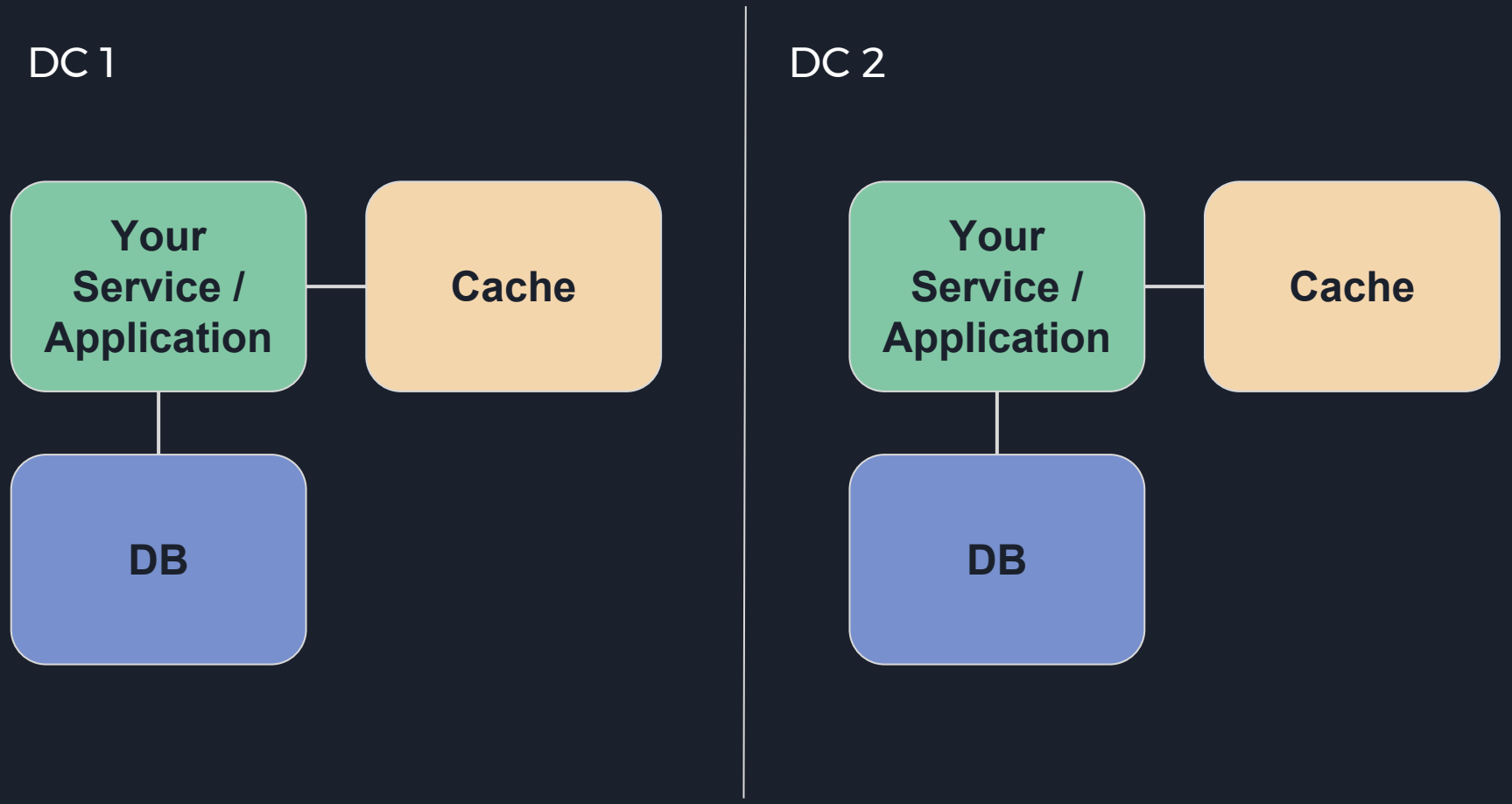
This problem isn't limited to not founds.

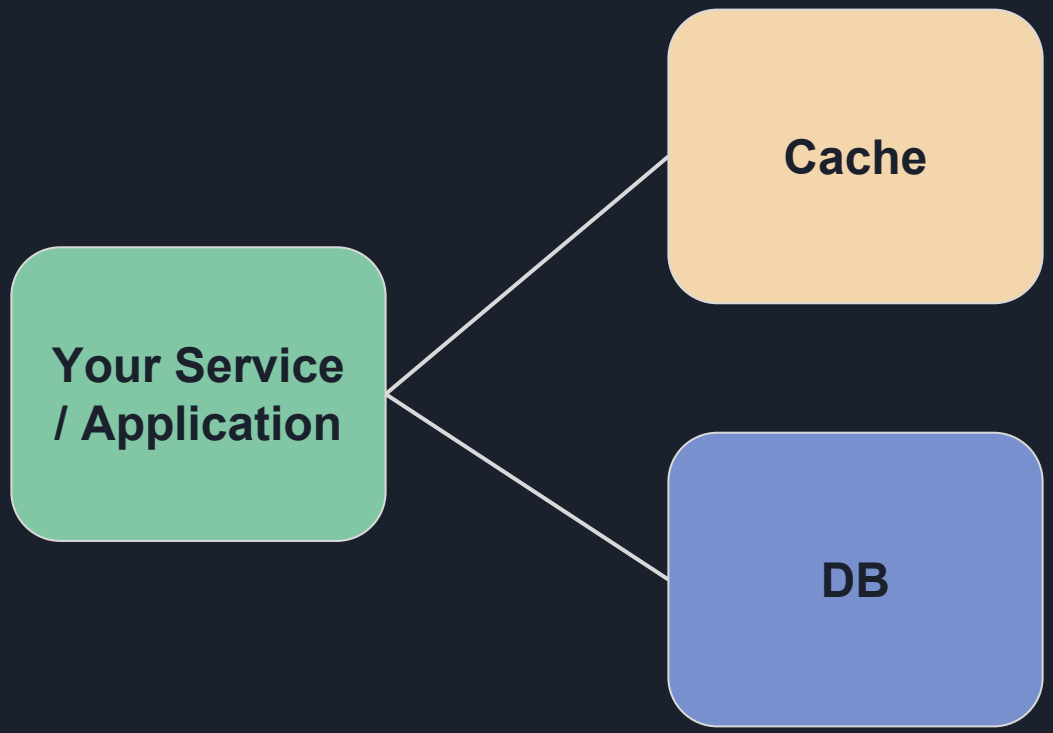
How Complex Systems Fail

(Being a Short Treatise on the Nature of Failure; How Failure is Evaluated; How Failure is Attributed to Proximate Cause; and the Resulting New Understanding of Patient Safety)

Richard I. Cook, MD
Cognitive technologies Laboratory
University of Chicago

4) **Complex systems contain changing mixtures of failures latent within them.** The complexity of these systems makes it impossible for them to run without multiple flaws being present. Because these are individually insufficient to cause failure they are regarded as minor factors during operations. Eradication of all latent failures is limited primarily by economic cost but also because it is difficult before the fact to see how such failures might contribute to an accident. The failures change constantly because of changing technology, work organization, and efforts to eradicate failures.


DC 1
Your Service / Application
Cache
DB
DC 2
Your Service / Application
Cache
DB


Your Service
/ Application
Cache
DB

Solution: Scaled TTLs

- Adjust TTL based on likelihood that the value is correct
- Start low and ramp up
- To the user, make it seem like inconsistencies "fix themselves"

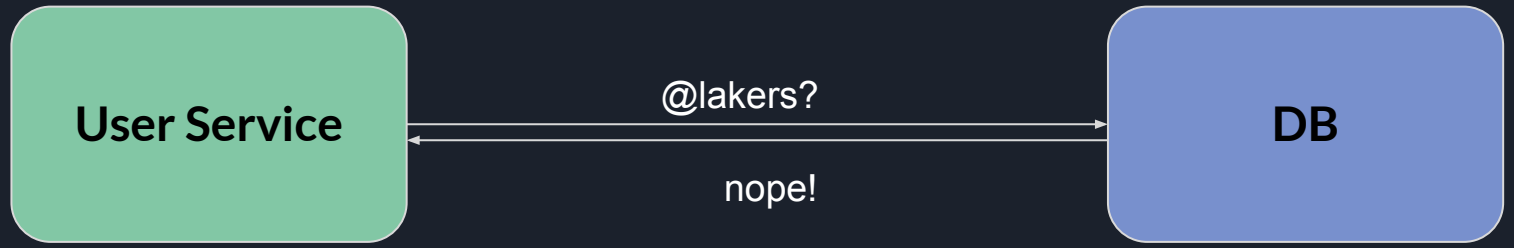
How do we build confidence?

- Repetition!
- If a value stay the same on consecutive reads, it's probably accurate

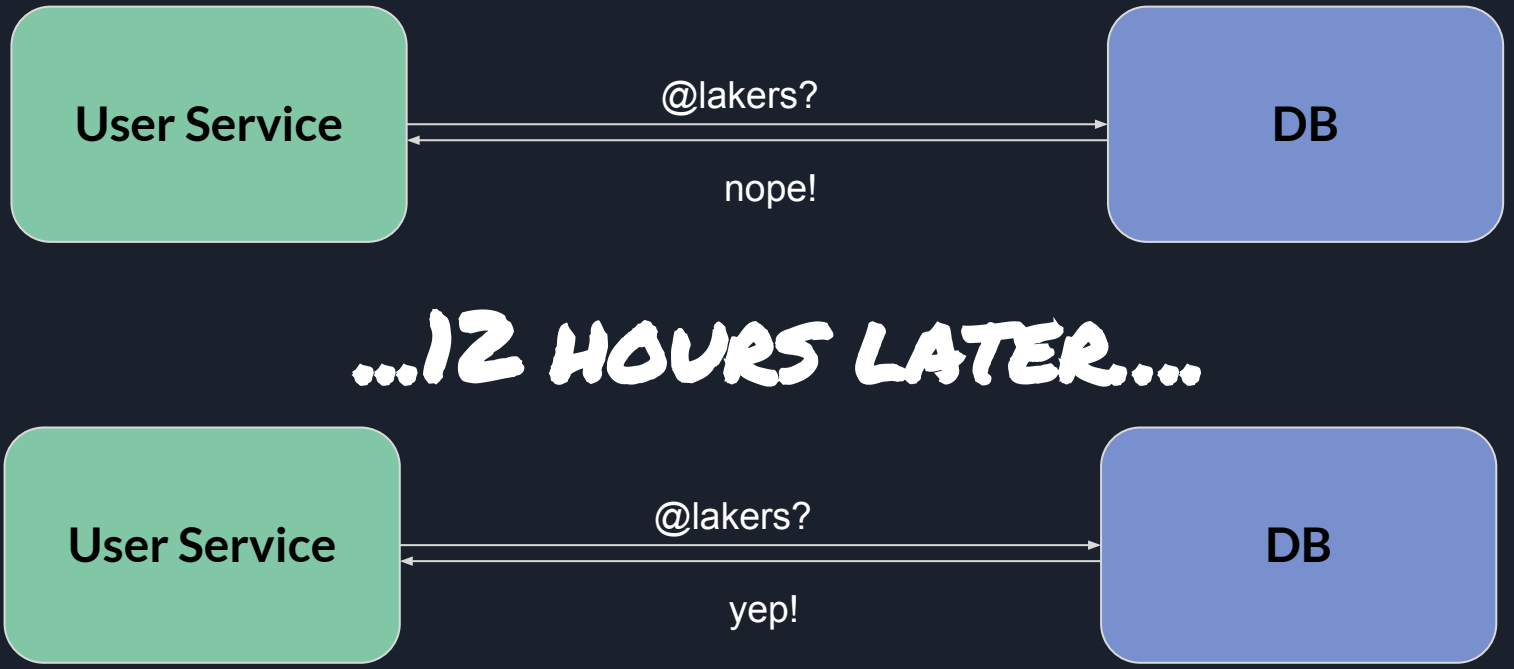
Ramping up

2 min -> 30 min -> 6 hrs
-> 12 hrs -> 24 hrs

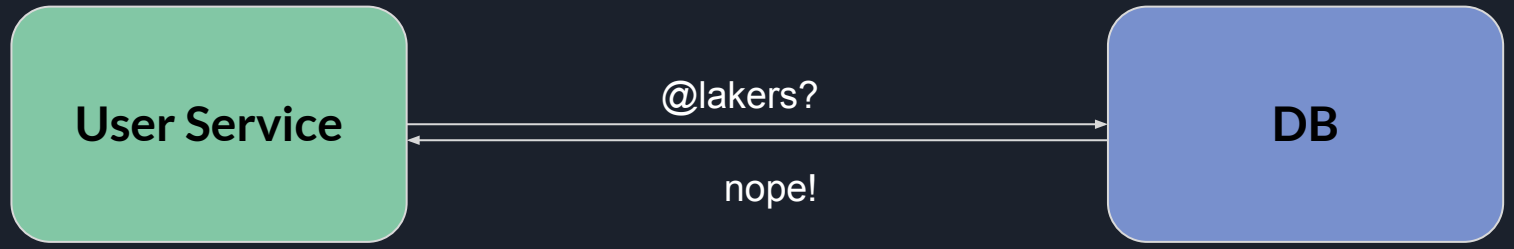
Before: unscaled TTLs



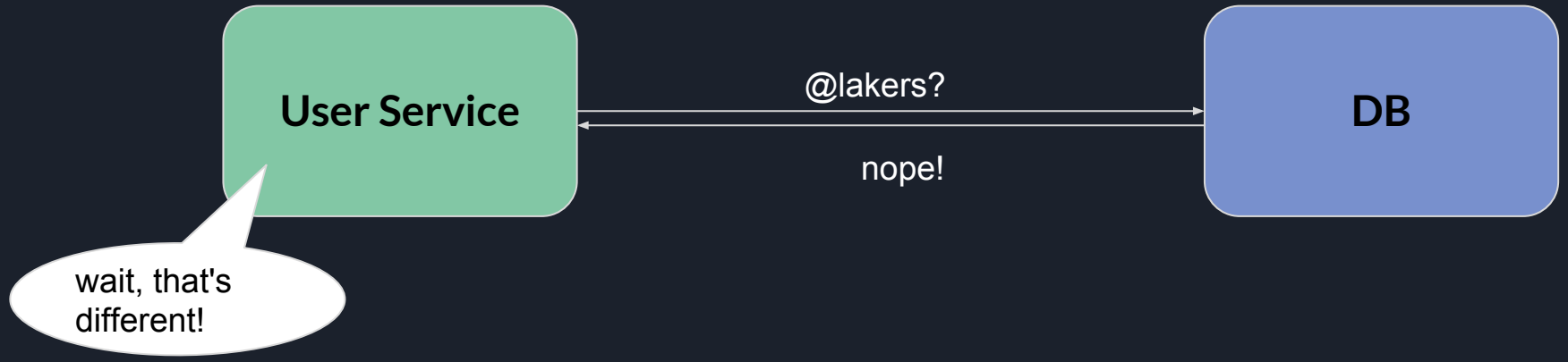
Before


User Service
@lakers?
nope!
DB
...12 HOURS LATER...
User Service
@lakers?
yep!
DB

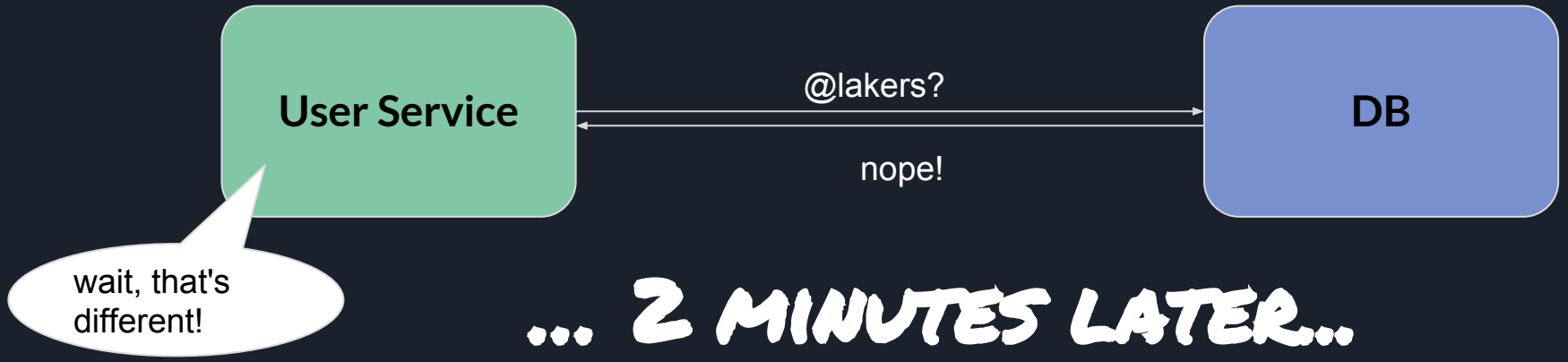
After (with scaled TTLs)



After (with scaled TTLs)

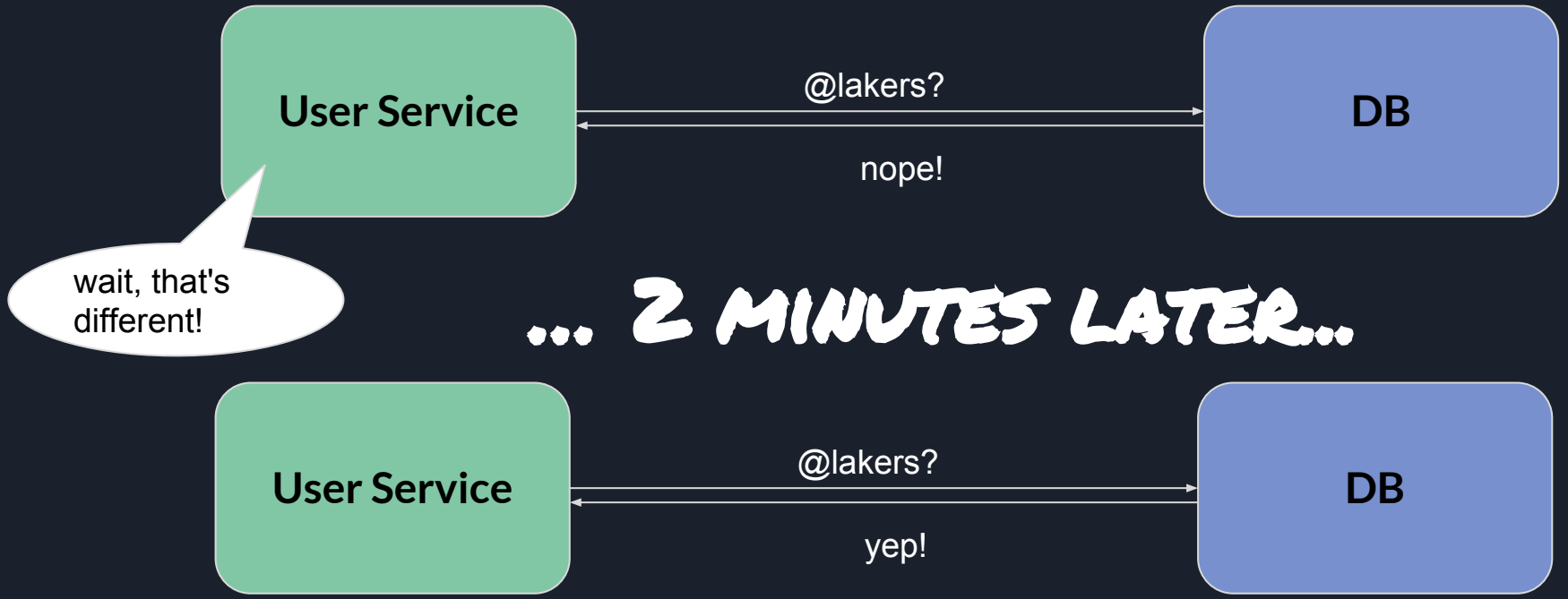


After (with scaled TTLs)

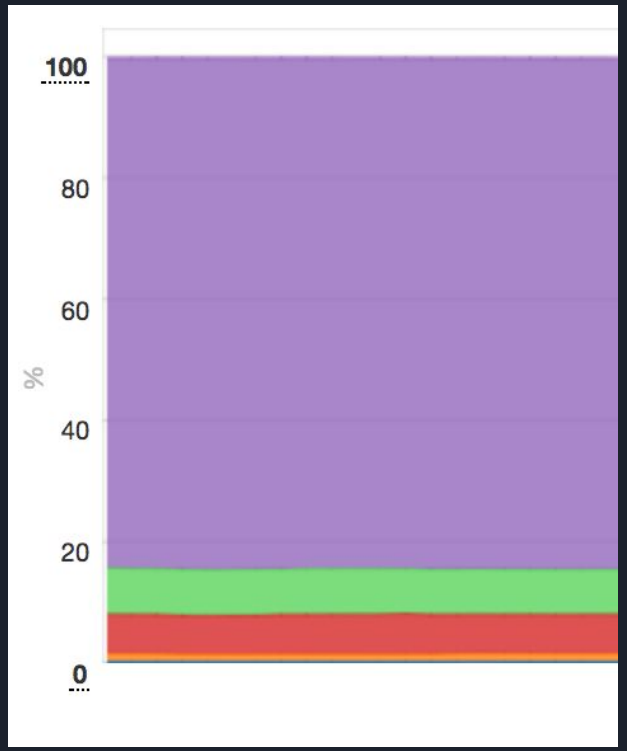


... 2 MINUTES LATER...

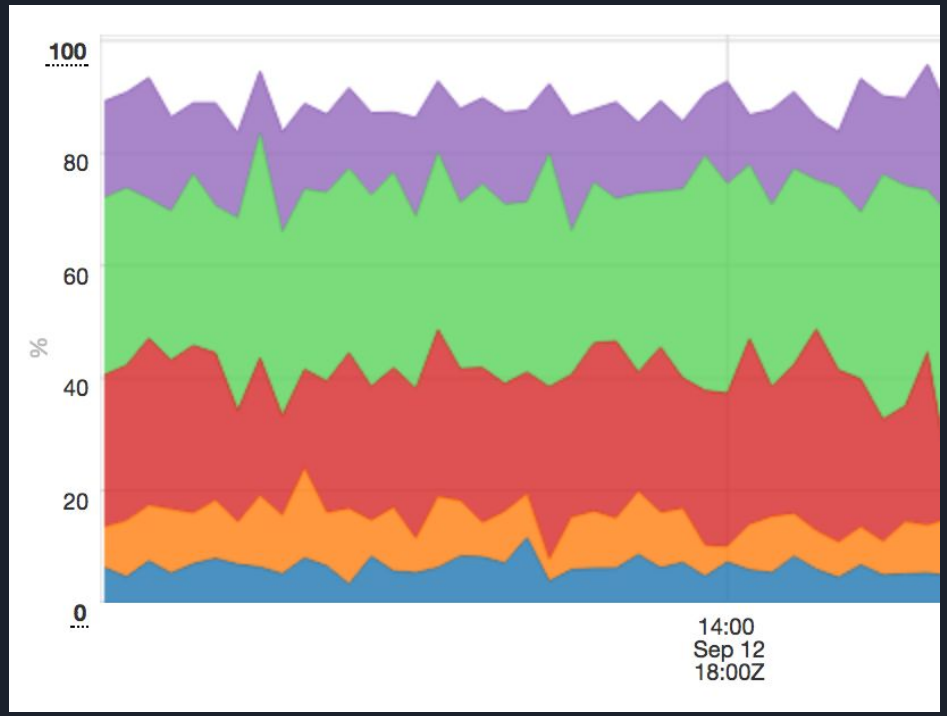
After (with scaled TTLs)



Measure!


100
80
60
%
40
20
0


Step 4
Step 3
Step 2
Step 1
Step 0


100
80
60
%
40
20
0
14:00
Sep 12
18:00Z

How to implement it

- Similar to soft TTL impl
- Store a `short` w/ number of consecutive consistent reads
- If inconsistent, reset to 0

30%

fewer inconsistencies

Benefits

- Magically mask eventual-inconsistency or flaky DB behavior
- Simple to adopt and implement
- Adapts to various query patterns
  - e.g. we have some data with many more consecutive writes; this way they get corrected

Gotchas

- Requires a gradual ramp-up
- Tuning ramp-up requires coordination with your backends
- Must be used in tandem with darkmoding, if you want cache to defend you during a DB outage
- Flaky behavior => more traffic to DB
  - see again: darkmoding!

Is this right for your service?

- Do you see these problems?
- What's your write traffic look like?
- How much staleness is acceptable?

Making it easy to Do Things Right (™)

Making it easy to do it right (™)

```
/** Maps a Query to a future Result. */
type Repository[-Q, +R] = Q => Future[R]
```

A read path entails mapping a **query** to a **future result**.

```scala
case class KeyValueResult[K, +V](
    found: Map[K, V],
    notFound: Set[K],
    failed: Map[K, Throwable])
  extends Iterable[(K, Try[Option[V]])]

type KeyValueRepository[Q, K, V] = Repository[Q, KeyValueResult[K, V]]
```

What's a Cache again?

```scala
trait Cache[K, V] {
  def get(keys: Seq[K]): Future[KeyValueResult[K, V]]
  def getWithChecksum(keys: Seq[K]):
    Future[KeyValueResult[K, (Try[V], Checksum)]]

  def add(key: K, value: V): Future[Boolean]
  def set(key: K, value: V): Future[Unit]
  def checkAndSet(k: K, v: V, checksum: Checksum): Future[Boolean]
  def delete(key: K): Future[Boolean]
}
```

Let's abstract away the cache behavior...

```
class CachingKeyValueRepository[Q <: Seq[K], K, V](
    underlying: KeyValueRepository[Q, K, V],
    cache: LockingCache[K, Cached[V]],
    ...)
  extends KeyValueRepository[Q, K, V]
```

Let's abstract away the cache behavior...

```
class DarkmodingCachingKeyValueRepository[Q <: Seq[K], K, V](
    underlying: KeyValueRepository[Q, K, V],
    cache: LockingCache[K, Cached[V]],
    ...)
  extends CachingKeyValueRepository[Q, K, V]
```

Wait, so what's a Cached[V]?

```
class CachingKeyValueRepository[Q <: Seq[K], K, V](
    underlying: KeyValueRepository[Q, K, V],
    cache: LockingCache[K, Cached[V]],
    ...)
  extends KeyValueRepository[Q, K, V]
```

Cached[V]

```
case class Cached[V](
    value: Option[V],
    status: CachedValueStatus, // e.g. Found, NotFound
    scaledTtlStep: Option[Short],
    cachedAt: Time,
    readThroughAt: Time, // the soft TTL expiration
    ...)
```

Wiring it together

```scala
def scaledDarkmodingCachingKVR[K, Q <: Seq[K]](
    repo: UserRepository[K],
    cache: LockingCache[K, Cached[User]],
    ...): CachingKeyValueRepository[Q, K, User]
```

ScaledDarkmodingCachingKeyValueRepository[Q, K, V]

Repo

Cache

- scaled TTLs
- soft / hard TTLs
- auto-darkmoding

cachingRepo(userId)

Twitter, Inc. [US] | https://twitter.com/lakers

Sorry, that page doesn’t exist!

You can **search Twitter** using the search box below or **return to the homepage**.

"Failure free operations require continuous experience with failure"

-- *How Complex Systems Fail*

TLDR;

- Cache is great
- Use it!
- But measure first :)

Thanks!

Find me after: @brindelle

Slides: bit.ly/strangeloop-cache

PS: http://careers.twitter.com