DISQUS

Building Scalable Web Apps

David Cramer
@zeeg

Agenda

- Terminology
- Common bottlenecks

- Building a scalable app
  - Architecting your database
  - Utilizing a Queue
  - The importance of an API

Performance vs. Scalability

“Performance measures the speed with which a single request can be executed, while scalability measures the ability of a request to maintain its performance under increasing load.”

(but we’re not just going to scale your code)

Sharding

“Database sharding is a method of horizontally partitioning data by common properties”

Denormalization

"Denormalization is the process of attempting to optimize the performance of a database by adding redundant data or by grouping data."

Common Bottlenecks

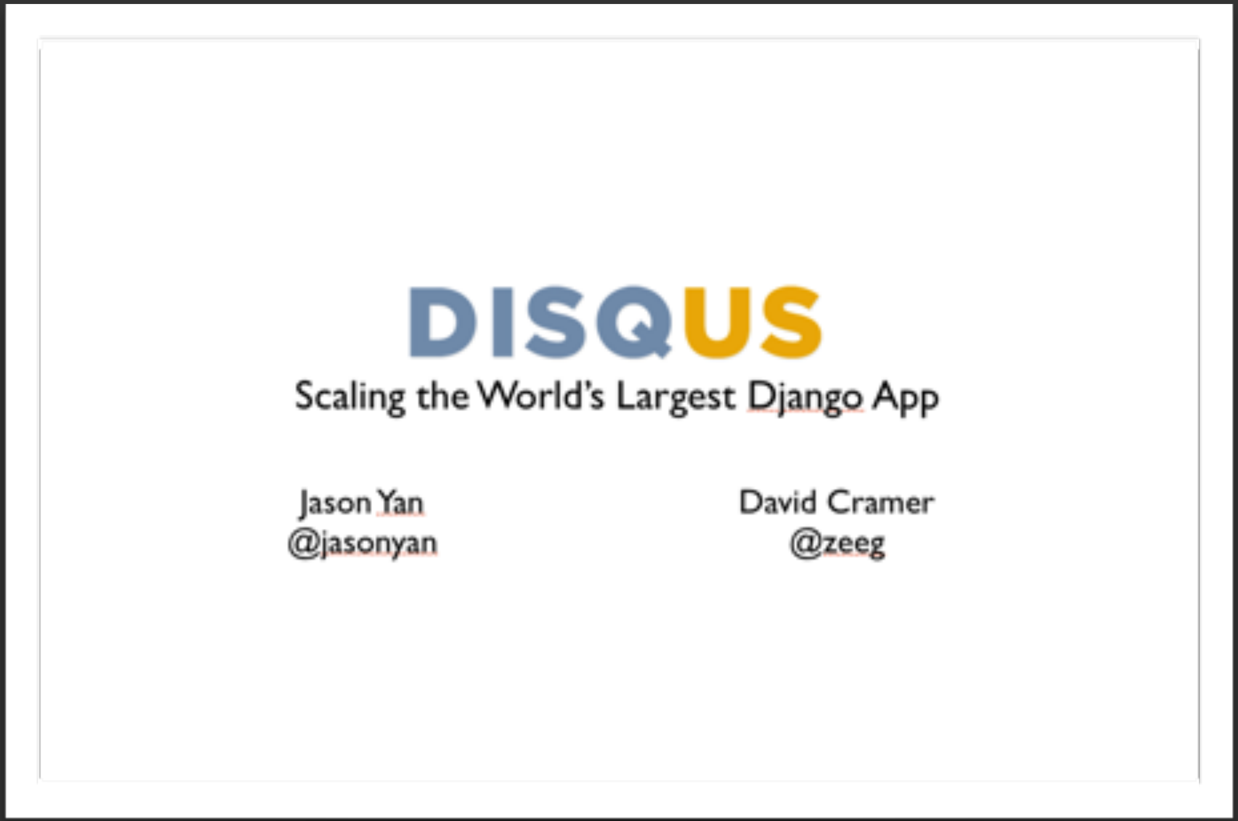
- **Database** (almost always)
- Caching, Invalidation
- Lack of metrics, lack of tests

Building Tweeter

Getting Started

- Pick a framework: Django, Flask, Pyramid
- Package your app; Repeatability
- Solve **problems**
- Invest in architecture

Let’s use django

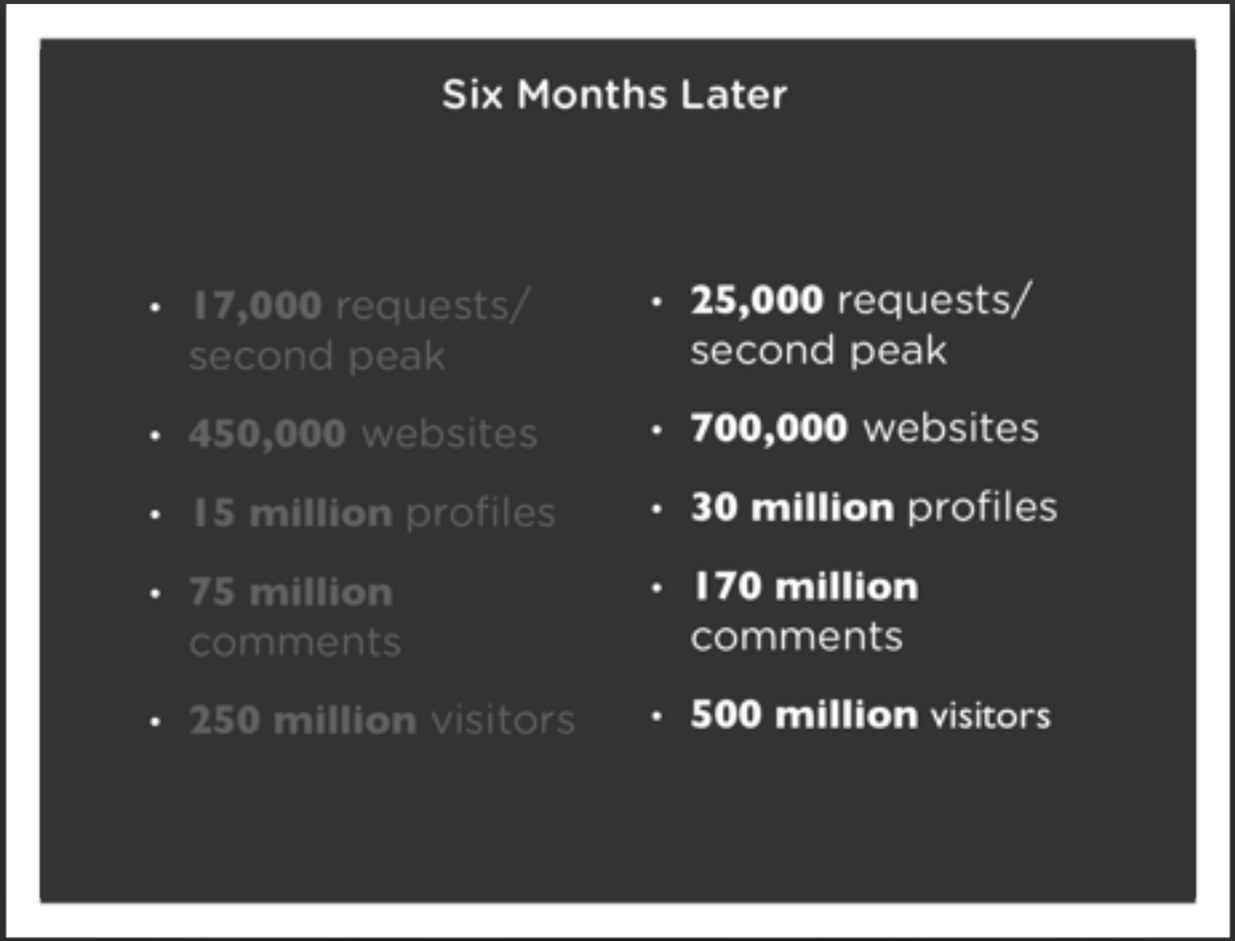


DISQUS

Scaling the World's Largest Django App

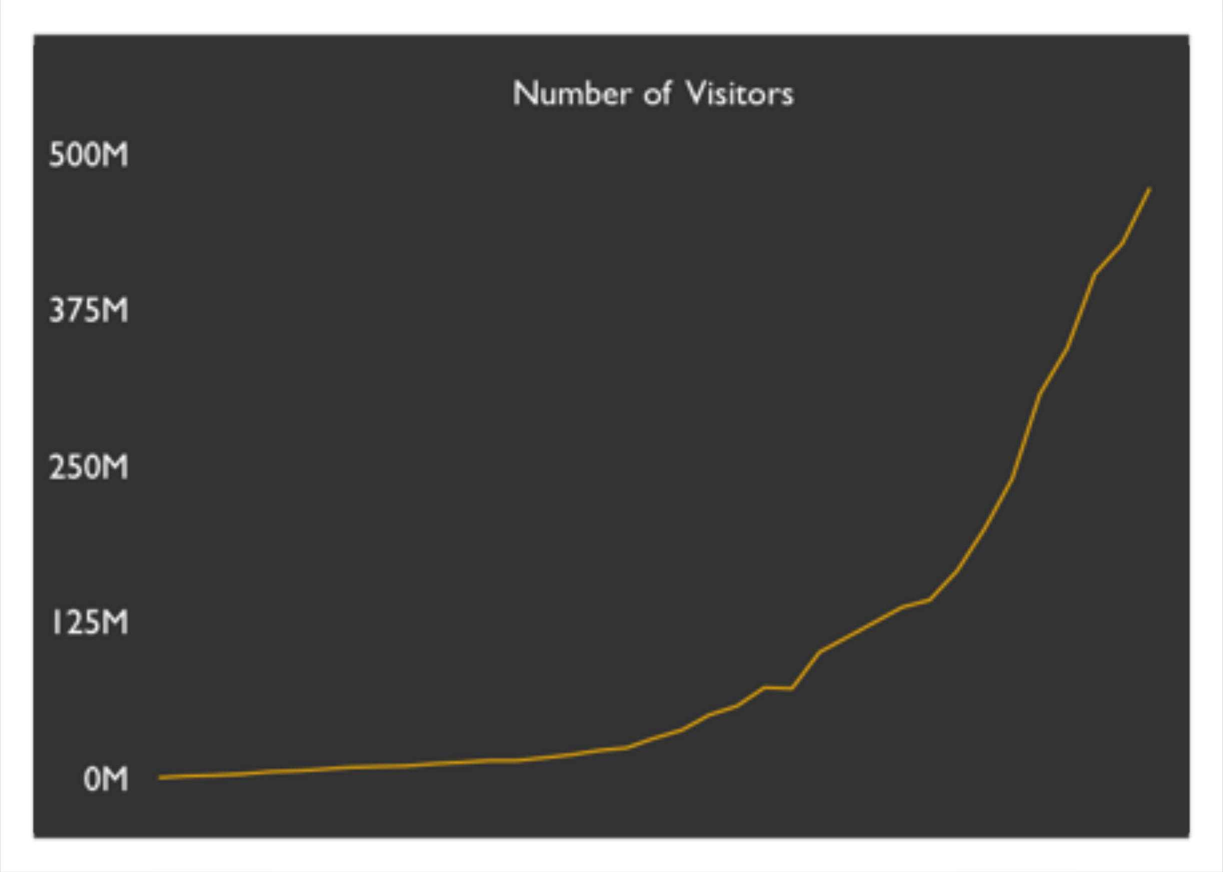
Jason Yan
@jasonyan

David Cramer
@zeeg



Six Months Later

17,000 requests/ second peak	**25,000** requests/ second peak
450,000 websites	**700,000** websites
15 million profiles	**30 million** profiles
75 million comments	**170 million** comments
250 million visitors	**500 million** visitors



Number of Visitors

Django is..

- Fast (enough)
- Loaded with goodies
- Maintained
- Tested
- Used

Packaging Matters

setup.py

```
#!/usr/bin/env python
from setuptools import setup, find_packages

setup(
    name='tweeter',
    version='0.1',
    packages=find_packages(),
    install_requires=[
        'Django==1.3',
    ],
    package_data={
        'tweeter': [
            'static/*.*',
            'templates/*.*',
        ],
    },
)
```

setup.py (cont.)

```
$ mkvirtualenv tweeter
$ git clone git.example.com:tweeter.git
$ cd tweeter
$ python setup.py develop
```

setup.py (cont.)

```
## fabfile.py
def setup():
    run('git clone git.example.com:tweeter.git')
    run('cd tweeter')
    run('./bootstrap.sh')
```

```
## bootstrap.sh
#!/usr/bin/env bash
virtualenv env
env/bin/python setup.py develop
```

setup.py (cont.)

```
$ fab web setup

setup executed on web1
setup executed on web2
setup executed on web3
setup executed on web4
setup executed on web5
setup executed on web6
setup executed on web7
setup executed on web8
setup executed on web9
setup executed on web10
```

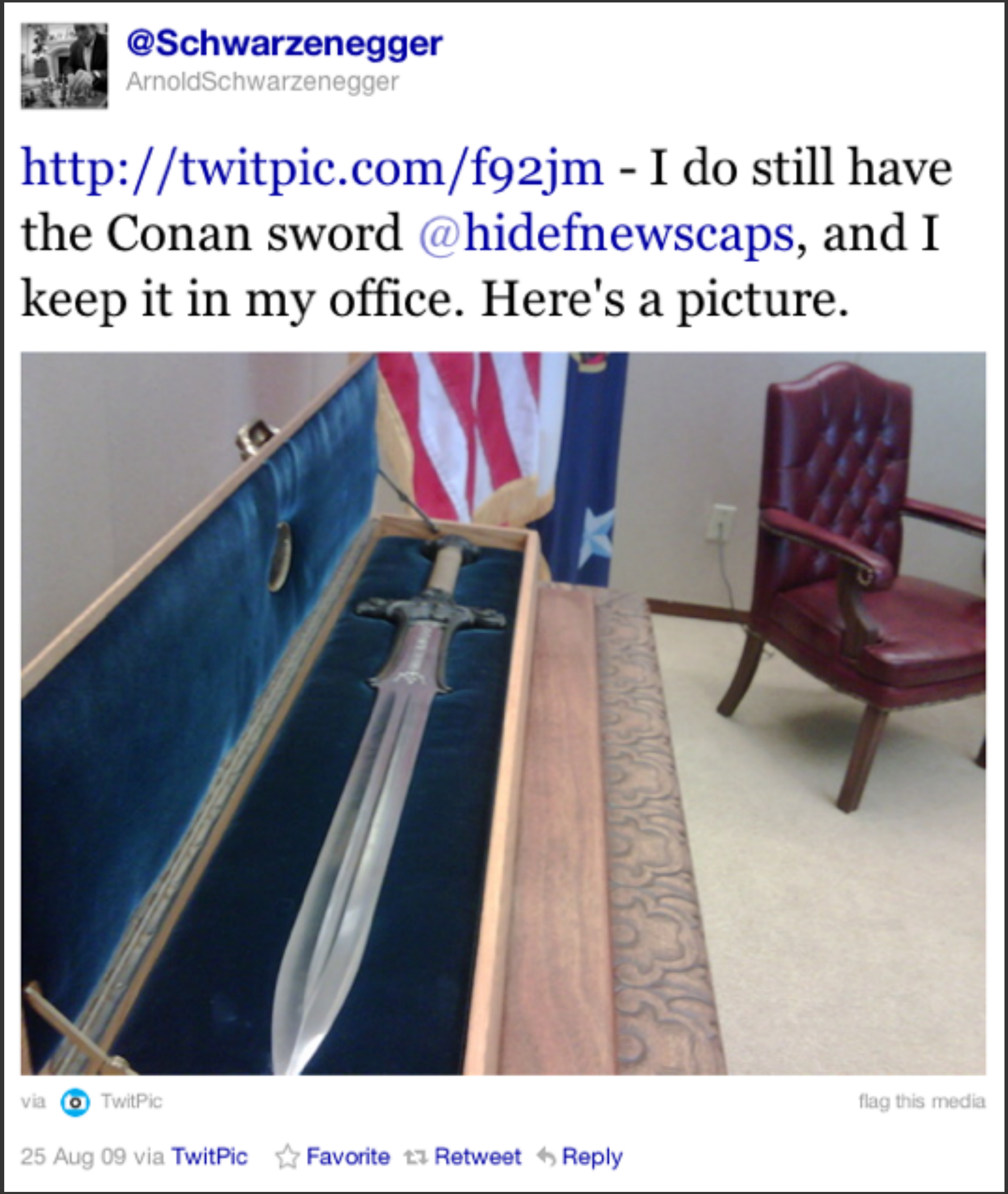
Database(s) First

Databases

- **Usually** core
- Common bottleneck
- Hard to change
- Tedious to scale

http://www.flickr.com/photos/adesigna/3237575990/

What a tweet “looks” like



@Schwarzenegger
ArnoldSchwarzenegger

http://twitpic.com/f92jm - I do still have the Conan sword @hidefnewscaps, and I keep it in my office. Here's a picture.

25 Aug 09 via TwitPic

Modeling the data

```
from django.db import models

class Tweet(models.Model):
    user    = models.ForeignKey(User)
    message = models.CharField(max_length=140)
    date    = models.DateTimeField(auto_now_add=True)
    parent  = models.ForeignKey('self', null=True)



class Relationship(models.Model):
    from_user = models.ForeignKey(User)
    to_user   = models.ForeignKey(User)
```

(Remember, **bare bones**!)

Public Timeline

```
# public timeline
SELECT * FROM tweets
  ORDER BY date DESC
  LIMIT 100;
```

- Scales to the size of one physical machine
- Heavy index, long tail
- Easy to cache, invalidate

Following Timeline

```
# tweets from people you follow
SELECT t.* FROM tweets AS t
  JOIN relationships AS r
    ON r.to_user_id = t.user_id
  WHERE r.from_user_id = '1'
  ORDER BY t.date DESC
  LIMIT 100
```

- No vertical partitions
- Heavy index, long tail
- “Necessary evil” join
- Easy to cache, expensive to invalidate

Materializing Views

```python
PUBLIC_TIMELINE = []

def on_tweet_creation(tweet):
    global PUBLIC_TIME

    PUBLIC_TIMELINE.insert(0, tweet)

def get_latest_tweets(num=100):
    return PUBLIC_TIMELINE[:num]
```

Disclaimer: don’t try this at home

Introducing Redis

```
class PublicTimeline(object):
    def __init__(self):
        self.conn = Redis()
        self.key = 'timeline:public'

    def add(self, tweet):
        score = float(tweet.date.strftime('%s.%m'))
        self.conn.zadd(self.key, tweet.id, score)

    def remove(self, tweet):
        self.conn.zrem(self.key, tweet.id)

    def list(self, offset=0, limit=-1):
        tweet_ids = self.conn.zrevrange(self.key, offset, limit)

        return tweet_ids
```

Cleaning Up

```python
from datetime import datetime, timedelta

class PublicTimeline(object):
    def truncate(self):
        # Remove entries older than 30 days
        d30 = datetime.now() - timedelta(days=30)
        score = float(d30.strftime('%s.%m'))
        self.conn.zremrangebyscore(self.key, d30, -1)
```

Scaling Redis

```
from nydus.db import create_cluster

class PublicTimeline(object):
    def __init__(self):
        # create a cluster of 9 dbs
        self.conn = create_cluster({
            'engine': 'nydus.db.backends.redis.Redis',
            'router': 'nydus.db.routers.redis.PartitionRouter',
            'hosts': dict((n, {'db': n}) for n in xrange(64)),
        })
```

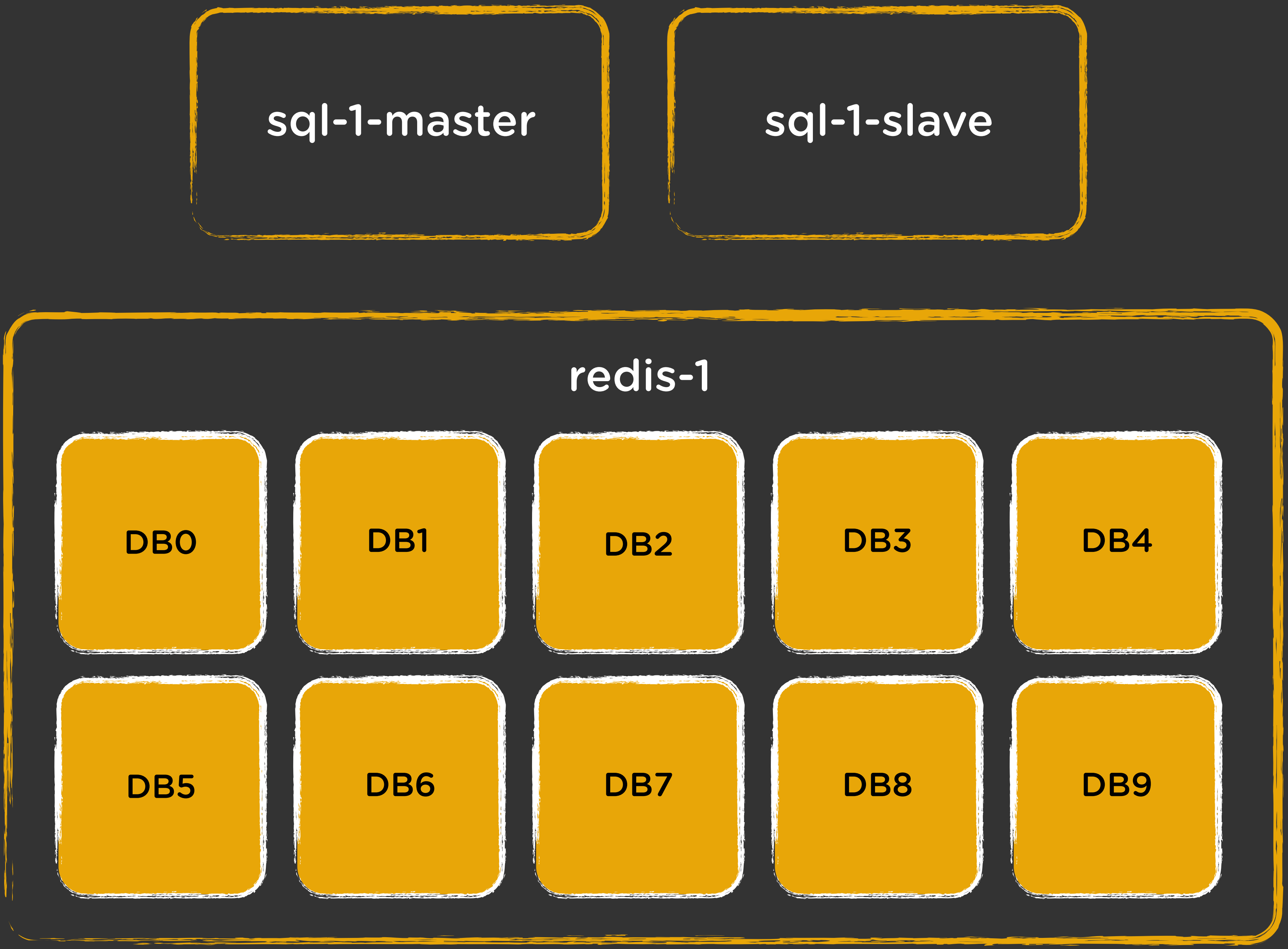
Nydus

```python
# create a cluster of Redis connections which
# partition reads/writes by key (hash(key) % size)

from nydus.db import create_cluster
redis = create_cluster({
    'engine': 'nydus.db.backends.redis.Redis',
    'router': 'nydus.db...redis.PartitionRouter',
    'hosts': {
        0: {'db': 0},
    }
})

# maps to a single node
res = conn.incr('foo')
assert res == 1

# executes on all nodes
conn.flushdb()
```

http://github.com/disqus/nydus

Vertical vs. Horizontal

Looking at the Cluster



sql-1-master

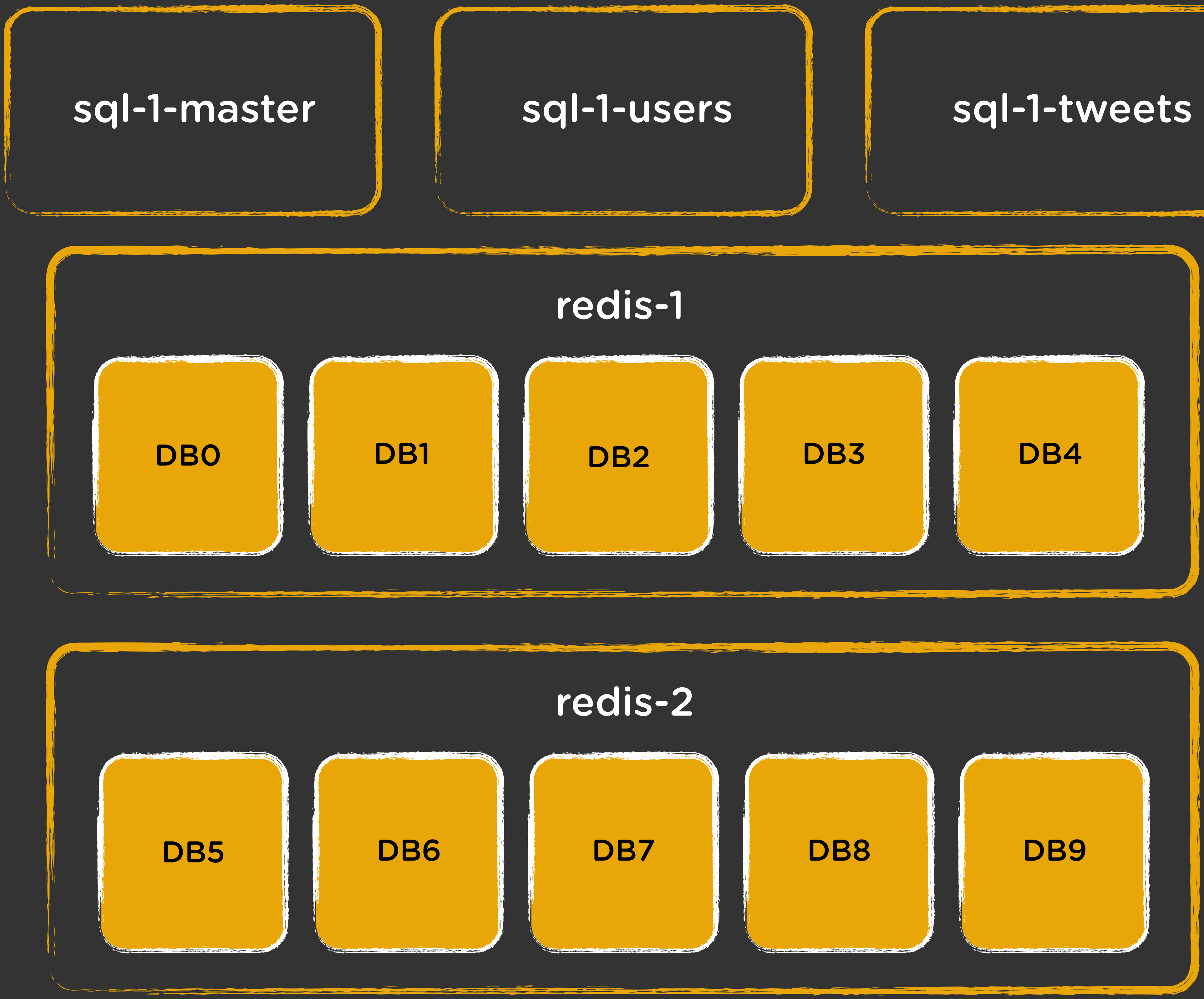
sql-1-slave

redis-1

DB0 DB1 DB2 DB3 DB4

DB5 DB6 DB7 DB8 DB9

“Tomorrow’s” Cluster



sql-1-master

sql-1-users

sql-1-tweets

redis-1

DB0 DB1 DB2 DB3 DB4

redis-2

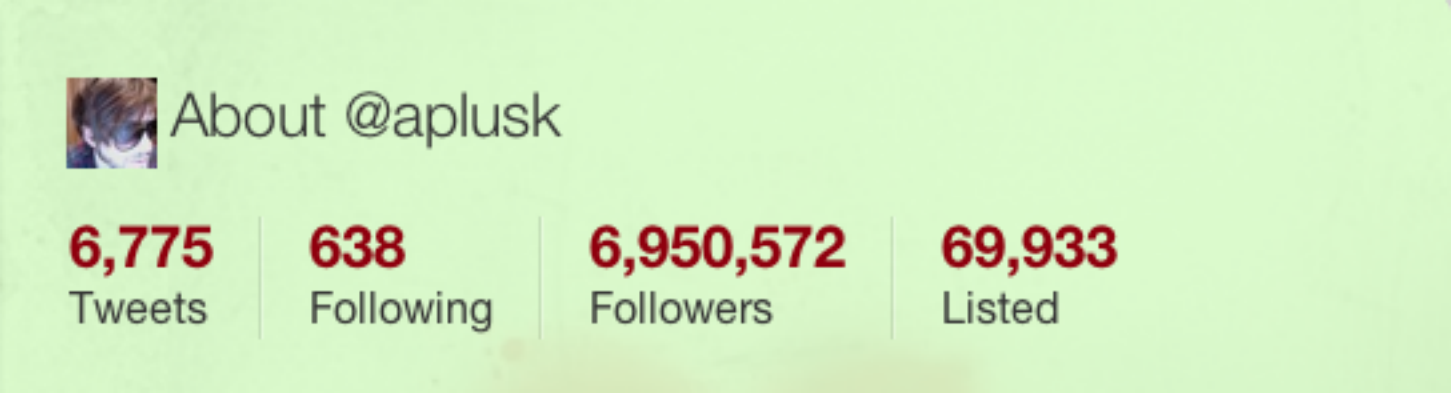
DB5 DB6 DB7 DB8 DB9

Asynchronous Tasks

In-Process Limitations

```
def on_tweet_creation(tweet):
    # O(1) for public timeline
    PublicTimeline.add(tweet)

    # O(n) for users following author
    for user_id in tweet.user.followers.all():
        FollowingTimeline.add(user_id, tweet)

    # O(1) for profile timeline (my tweets)
    ProfileTimeline.add(tweet.user_id, tweet)
```

In-Process Limitations (cont.)



About @aplusk

6,775 Tweets | 638 Following | 6,950,572 Followers | 69,933 Listed

```
# O(n) for users following author
# 7 MILLION writes for Ashton Kutcher
for user_id in tweet.user.followers.all():
    FollowingTimeline.add(user_id, tweet)
```

Introducing Celery

```
#!/usr/bin/env python
from setuptools import setup, find_packages

setup(
    install_requires=[
        'Django==1.3',
        'django-celery==2.2.4',
    ],
    # ...
)
```

Introducing Celery (cont.)

```
@task(exchange='tweet_creation')
def on_tweet_creation(tweet_dict):
    # HACK: not the best idea
    tweet = Tweet()
    tweet.__dict__ = tweet_dict

    # O(n) for users following author
    for user_id in tweet.user.followers.all():
        FollowingTimeline.add(user_id, tweet)

on_tweet_creation.delay(tweet.__dict__)
```

Bringing It Together

```
def home(request):
    "Shows the latest 100 tweets from your follow stream"

    if random.randint(0, 9) == 0:
        return render('fail_whale.html')

    ids = FollowingTimeline.list(
        user_id=request.user.id,
        limit=100,
    )

    res = dict((str(t.id), t) for t in \
                  Tweet.objects.filter(id__in=ids))

    tweets = []
    for tweet_id in ids:
        if tweet_id not in res:
            continue
        tweets.append(res[tweet_id])

    return render('home.html', {'tweets': tweets})
```

Build an API

APIs

- `PublicTimeline.list`
- `redis.zrange`
- `Tweet.objects.all()`
- example.com/api/tweets/

Refactoring

```
def home(request):
    "Shows the latest 100 tweets from your follow stream"

    tweet_ids = FollowingTimeline.list(
        user_id=request.user.id,
        limit=100,
    )
```

```
def home(request):
    "Shows the latest 100 tweets from your follow stream"

    tweets = FollowingTimeline.list(
        user_id=request.user.id,
        limit=100,
    )
```

Refactoring (cont.)

```
from datetime import datetime, timedelta

class PublicTimeline(object):
   def list(self, offset=0, limit=-1):
       ids = self.conn.zrevrange(self.key, offset, limit)

       cache = dict((t.id, t) for t in \
                    Tweet.objects.filter(id__in=ids))

       return filter(None, (cache.get(i) for i in ids))
```

Optimization in the API

```
class PublicTimeline(object):
   def list(self, offset=0, limit=-1):
       ids = self.conn.zrevrange(self.list_key, offset, limit)

       # pull objects from a hash map (cache) in Redis
       cache = dict((i, self.conn.get(self.hash_key(i)))
                    for i in ids)

       if not all(cache.itervalues()):
           # fetch missing from database
           missing = [i for i, c in cache.iteritems() if not c]
           m_cache = dict((str(t.id), t) for t in \
                          Tweet.objects.filter(id__in=missing))

           # push missing back into cache
           cache.update(m_cache)
           for i, c in m_cache.iteritems():
               self.conn.set(hash_key(i), c)

       # return only results that still exist
       return filter(None, (cache.get(i) for i in ids))
```

Optimization in the API (cont.)

```
def list(self, offset=0, limit=-1):
       ids = self.conn.zrevrange(self.list_key, offset, limit)

       # pull objects from a hash map (cache) in Redis
       cache = dict((i, self.conn.get(self.hash_key(i)))
                    for i in ids)
```

Store each object in it’s own key

Optimization in the API (cont.)

```
if not all(cache.itervalues()):
    # fetch missing from database
    missing = [i for i, c in cache.iteritems() if not c]
    m_cache = dict((str(t.id), t) for t in \
                   Tweet.objects.filter(id__in=missing))
```

Hit the database for misses

Optimization in the API (cont.)

Store misses back in the cache

```
    # push missing back into cache
    cache.update(m_cache)
    for i, c in m_cache.iteritems():
        self.conn.set(hash_key(i), c)

# return only results that still exist
return filter(None, (cache.get(i) for i in ids))
```

Ignore database misses

(In)validate the Cache

```python
class PublicTimeline(object):
   def add(self, tweet):
        score = float(tweet.date.strftime('%s.%m'))

        # add the tweet into the object cache
        self.conn.set(self.make_key(tweet.id), tweet)

        # add the tweet to the materialized view
        self.conn.zadd(self.list_key, tweet.id, score)
```

(In)validate the Cache

```
class PublicTimeline(object):
   def remove(self, tweet):
        # remove the tweet from the materialized view
        self.conn.zrem(self.key, tweet.id)

        # we COULD remove the tweet from the object cache
        self.conn.del(self.make_key(tweet.id))
```

Wrap Up

Reflection

- **Use a framework!**
- Start simple; grow naturally
- Scale can lead to performance
  - Not the other way around
- Consolidate entry points

Reflection (cont.)

- 100 shards > 10; Rebalancing sucks
  - Use VMs
- Push to caches, don’t pull
- “Denormalize” counters, views
- Queue everything

Food for Thought

- Normalize object cache keys
- Application triggers directly to queue
- Rethink pagination
- Build with future-sharding in mind

DISQUS

Questions?

psst, we’re hiring
jobs@disqus.com