1,000,000 daily users and no cache
Who is that guy?

Jesper Richter-Reichhelm / @jrierei
Berlin, Germany
Head of Engineering @ wooga
Wooga does social games

Diamond Dash
60 seconds, gem rush

wooga is now no. 4 worldwide
over 25 m active users every month

Happy Hospital
Cure cute pets from funny diseases

Jens Begemann at Quo Vadis
May 3, 2011

On May 2nd, Jens Begemann will be speaking at the Deutschen Gamestage from 10 am until 11:15 am, in Room A03. The topic of his lecture will be: "Social Games – Making Publishers Obsolete". Included below is a description of his talk. Read more...

365 Days and 20 Million Users – Monster World's Birthday

Based in Berlin, wooga is the leading European social games developer.

Follow us on Facebook Twitter

Like this
Jesper Richter-Reichhelm, Jenny Schulz and 102,795 others like this.
Wooga has dedicated game teams

- PHP Cloud
- Ruby Cloud
- Ruby Bare metal
- Cooming soon, Erlang Cloud
Flash client sends state changes to backend
Social games need to scale quite a bit

1 million daily users
200 million daily HTTP requests
Social games need to scale quite a bit

1 million daily users

200 million daily HTTP requests

100,000 DB operations per second

40,000 DB updates per second
Social games need to scale quite a bit

1 million daily users
200 million daily HTTP requests
100,000 DB operations per second
40,000 DB updates per second

... but no cache
A journey to 1,000,000 daily users

Start of the journey

6 weeks of pain

Paradise

Looking back
We used two MySQL shards right from the start

data_fabric gem

Easy to use

Sharding by Facebook’s user id

Sharding on controller level
Scaling application servers was very easy

Running at Amazon EC2
Scalarium for automation and deployment
  Role based
  Chef recipes for customization
Scaling app servers up and out was simple
Master-slave replication for DBs worked fine
We added a few application servers over time.

Diagram showing a load balancer (lb) followed by a series of application servers (app) and database servers (db) with slaves.
Basic setup worked well for 3 months

Life was good until 260K daily users
A journey to 1,000,000 daily users

Start of the journey

6 weeks of pain

Paradise

Looking back
SQL queries generated by Rubyamf gem

Rubyamf serializes returned objects
Wrong config => it also loads associations
SQL queries generated by Rubyamf gem

Rubyamf serializes returned objects
Wrong config => it also loads associations

Really bad but very easy to fix
New Relic shows what happens during an API call.

### Performance Breakdown

#### TilesController#harvest
- **% Time:** 20%
- **Avg Calls (per Txn):** 1.0
- **Avg Exclusive Time (ms):** 6.8

#### Redis - HGET
- **% Time:** 16%
- **Avg Calls (per Txn):** 3.4
- **Avg Exclusive Time (ms):** 5.5

#### mission/process_task!
- **% Time:** 13%
- **Avg Calls (per Txn):** 0.39
- **Avg Exclusive Time (ms):** 4.4

#### Redis - HINCRBY
- **% Time:** 9.8%
- **Avg Calls (per Txn):** 2.0
- **Avg Exclusive Time (ms):** 3.3

#### Redis - ZADD
- **% Time:** 7.6%
- **Avg Calls (per Txn):** 1.0
- **Avg Exclusive Time (ms):** 2.6

#### Redis - HSET
- **% Time:** 7.5%
- **Avg Calls (per Txn):** 2.0
- **Avg Exclusive Time (ms):** 2.6

#### GC
- **% Time:** 7.2%
- **Avg Calls (per Txn):** 0.01
- **Avg Exclusive Time (ms):** 2.5

#### SQL - OTHER
- **% Time:** 6.3%
- **Avg Calls (per Txn):** 0.81
- **Avg Exclusive Time (ms):** 2.1

#### ActiveRecord
- **% Time:** 4.5%
- **Avg Calls (per Txn):** 0.39
- **Avg Exclusive Time (ms):** 1.5
More traffic using the same cluster

lb

app app app app app app app app app

db db

db slave db slave
Our DB problems began

MySQL hiccups at 320K users!
MySQL hiccups were becoming more frequent

Everything was running fine ...

... and then DB throughput went down to 10%

After a few minutes everything stabilizes again ...

... just to repeat the cycle 20 minutes later!
ActiveRecord’s checks caused 20% extra DB load

Check connection state before each SQL query

MySQL process list full of ‘status’ calls
ActiveRecord’s status checks caused 20% extra DB

Check connection state before each SQL query

MySQL process list full of ‘status’ calls

```ruby
AR::ConnectionAdapters::AbstractAdapter.class_eval do
def verify!
  true
end
end
```
I/O on MySQL masters still was the bottleneck

New Relic shows DB table usage

60% of all UPDATEs were done on the ‘tiles’ table
New Relic shows most used tables

Sort by: Most time consuming

User#find: 20%
UserDetail#find: 14%
Task#find: 12%
SQL - SELECT: 6.2%
Visit#find: 5.2%
Task#save: 4.8%
Feedpost#find: 4.6%
Avatar#find: 4.6%
FeedpostClick#find: 3.8%
TileDecoration#find: 3.6%
User#save: 2.9%
TileAction#find: 2.6%
Mission#find: 2.6%
Baby#find: 2.2%
FeedpostGift#find: 2.2%
TileAction#save: 1.7%
Helper#find: 1.6%
SQL - DELETE: 1.1%
Baby#save: 0.85%
TutorialState#find: 0.77%

Top 5 database operations by wall clock time

Database throughput

1.5M cpm
1M cpm
500k cpm
0 cpm
0 19:00 19:05 19:10 19:15

User#find
UserDetail#find
Task#find
 fen
sav e
Tiles are part of the core game loop

Core game loop:
1) plant
2) wait
3) harvest

All are operations on Tiles table.
We started to shard on model, too

We put this table in 2 extra shards
We started to shard on model, too

We put this table in 2 extra shards

1) Setup new masters as slaves of old ones
We started to shard on model, too

We put this table in 2 extra shards

1) Setup new masters as slaves of old ones
We started to shard on model, too

We put this table in 2 extra shards

1) Setup new masters as slaves of old ones
2) App servers start using new masters, too
We started to shard on model, too

We put this table in 2 extra shards

1) Setup new masters as slaves of old ones
2) App servers start using new masters, too
3) Cut replication
We started to shard on model, too

We put this table in 2 extra shards

1) Setup new masters as slave of old ones
2) App servers start using new masters, too
3) Cut replication
4) Truncate not-used tables
8 DBs and a few more servers
Doubling the amount of DBs didn’t fix it

It got painful at 430K users
We improved our MySQL setup

RAID-0 of EBS volumes

Using XtraDB instead of vanilla MySQL

Tweaking my.cnf

innodb_flush_log_at_trx_commit
innodb_flush_method
Data-fabric gem circumvented AR’s internal cache

2x \texttt{Tile.find\_by\_id(id)} \Rightarrow 1x \texttt{SELECT ...}
Data-fabric gem circumvented AR’s internal cache

```ruby
2x Tile.find_by_id(id) => 1x SELECT ...

def shard(id)
  DataFabric.activate_shard(:shard => id) do
    Tile.connection.cache do
      User.connection.cache(&block)
    end
  end
end
```
2 + 2 masters and still I/O was not fast enough

We were getting desperate:

“If 2 + 2 is not enough, ... 

... perhaps 4 + 4 masters will do?”
It's no fun to handle 16 MySQL DBs
It's no fun to handle 16 MySQL DBs
We were at a dead end with MySQL

We hit a wall at 530K users
I/O remained the bottleneck for MySQL UPDATEs

Peak throughput overall: 5,000 writes/s
Peak throughput single master: 850 writes/s

We could get to ~1,000 writes/s ...

... but then what?
Pick the right tool for the job!
Redis is fast but goes beyond simple key/value

Redis is a key-value store
- Hashes, Sets, Sorted Sets, Lists
- Atomic operations like set, get, increment

50,000 transactions/s on EC2
- Writes are as fast as reads
Shelf tiles: An ideal candidate for using Redis

Shelf tiles:
{ plant1 => 184,
  plant2 => 141,
  plant3 => 130,
  plant4 => 112,
  ... }

Shelf tiles: An ideal candidate for using Redis

Redis Hash

- HGETALL: load whole hash
- HGET: load a single key
- HSET: set a single key
- HINCRBY: increase value of single key

...
Migrate on the fly when accessing new model

def self.find(uid)
    if not_migrated?(uid)
        OldSQL.all(:conditions=>{:uid=>uid}).each { |old|
            NewRedis.create_from_old!(old)
        }
    end
    # continue as normal
end
Migrate on the fly - but only once

```ruby
def not_migrated?(id)
  redis.hsetnx('migrated_ids', id)
end
```

true if id could be added
else false
Migrate on the fly - and clean up later

1. Let this running everything cools down
2. Then migrate the rest manually
3. Remove the migration code
4. Wait until no fallback necessary
5. Remove the SQL table
Migrations on the fly all look the same

Typical migration throughput over 3 days
Initial peak at 100 migrations / second
A journey to 1,000,000 daily users

Start of the journey

6 weeks of pain

Paredise (or not?)

Looking back
Again: Tiles are part of the core game loop

Core game loop:
1) plant
2) wait
3) harvest

All are operations on Tiles table.
Size matters for migrations

Migration check overloaded Redis
   Migration only on startup

We overlooked an edge case
   Next time only migrate 1% of users
   Migrate the rest when everything worked out
In-memory DBs don’t like to saving to disk

You still need to write data to disk eventually
  SAVE is blocking
  BGSAVE needs free RAM

Dumping on master increased latency by 100%
In-memory DBs don’t like to dump to disk

You still need to write data to disk eventually
  SAVE is blocking
  BGSAVE needs free RAM

Dumping on master increased latency by 100%

Running BGSAVE slaves every 15 minutes
Replication puts Redis master under load

Replication on master starts with a BGSAVE
Not enough free RAM => big problem

Master queue requests that slave cannot process
Restoring dump on slaves is blocking
Redis has a memory fragmentation problem

-in 8 days

-memory grew from 24 GB to 44 GB
Redis has a memory fragmentation problem

38 GB in 3 days

24 GB
If MySQL is a truck...

Fast enough for reads
Can store on disk
Robust replication
If MySQL is a truck, Redis is a Ferrari

Fast enough for reads
Can store on disk
Robust replication

Super fast reads/writes
Out of memory => dead
Fragile replication
Big and static data in MySQL, rest goes to Redis

Fast enough for reads
Can store on disk
Robust replication

Super fast reads/writes
Out of memory => dead
Fragile replication

256 GB data
10% writes

60 GB data
50% writes

http://www.flickr.com/photos/erix/245657047/
95% of operation effort is handling DBs!
We fixed all our problem - we were in Paradise!
A journey to 1,000,000 daily users

Start of the journey
6 weeks of pain
Paradise (or not?)

Looking back
Data handling is most important

How much data will you have?
How often will you read/update that data?
What data is in the client, what in the server?

It’s hard to “refactor” data later on
Monitor and measure your application closely

On application level / on OS level

Learn your app’s normal behavior

Store historical data so you can compare later

React if necessary!
Listen closely and answer truthfully

Q & A

Jesper Richter-Reichhelm

@jrirei

slideshare.net/wooga
If you are in Berlin just drop by

Thank you

Jesper Richter-Reichhelm
@jrirei

slideshare.net/wooga