Moving tables across clusters

Scaling a high traffic database
Nice to meet you!

- Developer on database-services team @ GitHub
- @bryanaknight on Twitter
- Tech lead for Women Who Code Boston
- Spoken at Railsconf and Syntaxcon (2017) about db performance
The state of (db) affairs.

• Several availability incidents in a short time caused by “pressure” on our db, specifically our master server of our main cluster

• Anything that caused a higher than usual load posed a risk

• Needed to act FAST, but make enough of an impact to give us medium-term breathing room
What did this look like?

• Increase in load that’s “out of the norm”
What did this look like?

- Increase in load == increase in WRITEs
- Caused replication lag and a spike in MySQL threads running
What did this look like?

- Replication lag can lead to a decrease in Job throughput
- throttled WRITEs in jobs to tables in the same cluster —> jobs take longer to complete
“High load on the master is less immediately obvious, but also user visible. When it's bad, it means GitHub is slow. When it's really bad, it means GitHub is entirely unresponsive.”
Solution: move ALL the tables!
Actually, just 4 of them.
What makes a table a good candidate to move?

1. Relatively high rate of reads and **writes** so impact of moving it off the main cluster is significant (our biggest concern was read and write IO)

2. Few or no JOINs with other tables

3. WRITEs can be backgrounded or already are
GitHub the Swag Shop

- The world’s largest Octocat T-shirt and stickers store
- And water bottles
- And hoodies
- We also do stuff related to things
Tables we moved using Strategy #1

1. **coffee_mugs** —> ‘beverage’ cluster
   - one of our largest tables
   - only a few JOINs
   - writes all came from background jobs
   - “easy” first table to move
“Give me gift bags with small coffee mugs that have the Octocat logo”

def bags_with_octocat_mugs
end

==

SELECT gift_bags.* FROM gift_bags
    JOIN coffee_mugs ON coffee_mugs.gift_bag_id = gift_bags.id
WHERE coffee_mugs.logo = 'Octocat'
AND coffee_mugs.size = 'small'

Original JOIN
Grab the gift_bag id’s from the JOIN’d table (coffee_mugs) from records that satisfy the conditions, and feed them into a separate query to the table with the data you need (gift_bags)

```ruby
def split_bags_with_octocat_mugs
  gift_bag_ids = CoffeeMug.where(logo: 'Octocat',
                                 size: 'small').uniq.pluck(:gift_bag_id)

  GiftBag.where(id: gift_bag_ids)
end
```

```
SELECT DISTINCT gift_bag_id
FROM coffee_mugs
WHERE logo = 'Octocat'
AND size = 'small'
```

```
SELECT * FROM gift_bags
WHERE id IN [...]
```

Split it!
Strategy #1 - application + db changes

1. Split JOINs and UNIONs with table to move and other tables (can’t join tables across clusters)

2. Background writes, put jobs in separate queue

3. Provision new cluster and replicate from mysql1 - most efficient way to get the table from mysql1 into new cluster
Strategy #1 - application + db changes

4. Make our model establish connection to new cluster (but actually still point ENV var for ‘new’ connection at mysql1)

class CoffeeMug < ApplicationRecord
  establish_connection_primary "#{Rails.env}_beverage_cluster_primary"
  establish_connection_readonly "#{Rails.env}_beverage_cluster_readonly"
end
Strategy #1 - application + db changes

5. Create SQL client for coffee_mugs queries that establishes connection to new cluster

```ruby
module Beverage
  class SQL < ::GitHub::SQL

    # Overrides GitHub::SQL's connection method to
    # force the use of the beverage_cluster connection
    def connection
      CoffeeMug.connection
    end
  end
end
```

6. Update GitHub::SQL callers to coffee_mugs table to use Beverage::SQL
Strategy #1 - reads cutover

1. Switch read-only connection to point to new cluster (via ENV var)
2. Deploy to a staging environment
3. Test in staging: make sure pages render doing readonly queries to the new cluster
4. Deploy master so that env var change is picked up everywhere
5. Observe reads going to the new cluster via Datadog and VividCortex
Readonly cutover

Before:

MYSQL_BEVERAGE_READER_HOST=db-main-cluster-readonly.net
MYSQL_BEVERAGE_WRITER_HOST=db-main-cluster-read-write.net

After:

MYSQL_BEVERAGE_READER_HOST=db-beverage-cluster-readonly.net
MYSQL_BEVERAGE_WRITER_HOST=db-main-cluster-read-write.net
Strategy #1 - writes cutover

1. Pause queues with backgrounded writes
2. Cluster/table configuration
   1. Detach replica (new cluster)
   2. set readonly=OFF
   3. dummy drop (rename) coffee_mugs table in mysql1
3. Switch primary (writes + some reads) connection to point to new cluster (via ENV var)
4. Deploy master
5. Resume queues
6. Change your shirt because you sweated through it since this is SO SCARY.
Writes cutover
From the MySQL side

1. Pause all jobs doing writes

   bryanaknight 4:15 PM
   .resque pause coffee_mugs

   hubot APP 4:15 PM
   Paused work on coffee_mugs

2. Detach Replica

   orchestrator -c detach-replica -i db-beverage-b3hce56 (beverage cluster)

3. Dummy Drop (rename) coffee_mugs table in the main cluster

   jonahberquist 9:25 PM
   .mysql dummy-drop coffee_mugs. .github_production magic_word=apple

   hubot APP 9:25 PM
   This will dummy drop the github_production coffee_mugs. by renaming it.
4. Set readonly-only off on the beverage cluster

   `mysql -h db-beverage-b3hce56 -e "set global read_only=0;"`

5. Change the value of the MYSQL_BEVERAGE_WRITER_HOST env var to point to the beverage cluster

   `MYSQL_BEVERAGE_WRITER_HOST=db-beverage-cluster-read-write.net`

6. Set the role as master

   `.instance_attribute instance=db-beverage-cluster-b3hce56.net mysql_role=master`
Writes cutover
From the MySQL side

7. Since you've updated the role, do a puppet run (does some reconfiguration because the role was changed)

   .puppet run db-mysql-b3hce56.net

8. Resume jobs (writes)

   .resque resume coffee_mugs.

   Resumed work on coffee_mugs.

9. Reset beverage master replication so it won't connect to main cluster again (Forget you were ever a replica!)

   mysql -h db-beverage-cluster-c4fba34 -e "reset slave all;"
Cutover coffee_mugs from mysql1 to beverages cluster

GitHub application

```
SELECT * FROM coffee_mugs WHERE id = 1
INSERT INTO coffee_mugs ...
```

MySQL replication

Readonly queries

Backgrounded Writes (resque)

Mysql1

Beverages

Backgrounded Writes (resque)

Replica

Replica
Pitfalls of Strategy #1

- “Update callers to use Beverage::SQL client”
  - There was a SQL write query to the coffee_mugs table that was not using the correct SQL client

- “Pause queues with backgrounded writes”
  - Several additional jobs were triggering writes to the coffee_mugs table that we hadn’t planned on pausing
Pitfalls of Strategy #1

• “Deploy to staging”
• “Deploy master”
• “Deploy master”
• …deploy are slow.
- coffee_mugs
- all other tables
Let’s iterate.

• Need more confidence that we are using the correct connection to the right cluster EVERYWHERE

• Get multiple clusters in test/dev

• Add a “lint test” for SQL queries using correct connection
Let’s iterate.

- Rely on feature flag to change connections from one cluster to another
  - Can gradually enable feature (in this case, new connection) by changing a setting in the UI or chat-ops instead of waiting on deploy
  - https://github.com/jnunemaker/flipper
Tables we moved using Strategy 1.1

1. **t_shirts** —> best_sellers cluster
   - Top 5 in write rate
   - Top 10 in write total time
   - All queries in one place

2. **Hoodies** —> best_sellers cluster
   - 4% of master writes
   - 3.5% of master reads
Strategy #1.1 - application + db changes

1. Split JOINs

2. Background writes, put jobs in separate queue

3. Provision new cluster and replicate from mysql1 - this means all the data is in the new table in the new cluster ready for us

4. Make new configurable SQL client and AR model and establish connection to new cluster (but actually still point ENV var for ‘new’ connection at mysql1) set connection toggle based on a feature flag

5. Update callers to use new SQL client/AR model
class TransitionalHoodieModel < ApplicationRecord
  self.abstract_class = true

  def self.ro_connection
    if read_feature_flag_enabled?
      BestSellersModel.connection
    else
      ActiveRecord::Base.connection
    end
  end
end

def self rw_connection
  if read_feature_flag_enabled? && write_feature_flag_enabled?
    BestSellersModel.connection
  else
    ActiveRecord::Base.connection
  end
end

def self.read_feature_flag_enabled?
  GitHub.flipper[:hoodies_best_sellers_writes].enabled?
end

def self.write_feature_flag_enabled?
  GitHub.flipper[:hoodies_best_sellers_writes].enabled?
end
end

class Hoodie < TransitionalHoodieModel
  ...
end

module Hoodies
  class SQL < ::GitHub::SQL
    def connection
      TransitionalHoodieModel.connection
    end
  end
end
Strategy #1.1 - reads cutover

1. Switch read-only connection to point to new cluster (via ENV var) Slowly enable hoodies_best_sellers_reads feature flag to move reads to the best sellers cluster

2. Deploy to staging

3. Test in lab: make sure pages render doing read-only queries to the new cluster

4. Deploy master

5. Observe reads going to the new cluster via Datadog and VividCortex
Strategy #1.1 - writes cutover

1. Pause queues with backgrounded writes (so replication catches up and new cluster has up-to-date data)

2. Detach replica (new cluster) and set readonly=OFF, dummy drop table in mysql1

3. Watch for needles

4. Switch master (writes + some reads) connection to point to new cluster (via ENV var change) Fully enable hoodies_best_sellers_writes feature flag to move writes to best_sellers

5. Deploy master

6. Resume queues

7. Change your shirt because you sweated through it since this is STILL SO SCARY.
Pitfalls of Strategy #1.1

 readonly != ALL reads

- We dummy dropped the mysql1 table before enabling the WRITES feature flag (as writes were paused), but a non-readonly read was still trying to go to the mysql1 table (primary connection)

- We needed to enable the WRITES feature flag before dummy dropping the table so those few reads (non-readonly connection reads) were routed to the new cluster

Even with all our improvements, this approach is imperfect.
Should we keep using this strategy?

• This approach is risky, still

• Backgrounding and pausing writes isn’t ideal or possible for most other big tables

• We can’t keep provisioning new clusters. We should use the clusters we have now.

• There has to be another way!
Goliath Stickers table

- >10% of reads to the master
- 7% of writes to the master
- ~1.5 billion rows
- No JOINs
- No UPDATEs (stickers are immutable)
Strategy #2 - temporary sharding
Moving stickers across clusters

1. Utilize a lookup table that stores info about which sticker packs’ stickers are stored in which cluster
2. Do a lookup against that table before any stickers query
3. Gradually copy sticker records into best_sellers and update the info in the lookup table
4. Copy until we’re not doing any reads or writes to the stickers table in mysql1 (all traffic going to best_sellers)
5. Drop mysql1 stickers
Advantages of Strategy #2

• Can move stickers in the context of sticker packs — this is our shard key
• No pausing writes
• No env var changes to do connection switching
• Fully testable
• Gradual increase of traffic onto the new cluster and off of mysql1
• **We kind of had to do it because we couldn’t use MySQL replication.**
How the f*** did we copy 1.5 billion records?

And preserve data quality, and not bring our site down ?????????
def move_stickers(sticker Packs)
    return if sticker Packs.empty?
    iterator = GitHub::SQL::Batched.new limit: sticker_batch_size
    iterator.add #<SQL, columns: COLUMNS_SQL, sticker_pack_ids: sticker Packs
        SELECT :columns
        FROM stickers
        WHERE sticker_pack_id IN :sticker_pack_ids
        AND id > :last
        ORDER BY id ASC
        LIMIT :limit
    SQL

    GitHub::SQL::Readonly.new(iterator.batches).each do |stickers|
        sql = ::BestSellers::SQL.new
        sql.add #<~SQL, columns: COLUMNS_SQL, statuses: prepare_rows_for_insertion(stickers)
            INSERT IGNORE INTO stickers
            (:columns)
            VALUES :stickers
        SQL

        unless dry_run
            throttle_to_best_sellers { sql.run }
        end
    end
end

- **Batch!**
- **Go to readonly for reads!**
- **Throttle!**
Moving stickers into the best_sellers cluster via temporary sharding

Where should I route this query with sticker_pack 1?

We haven’t touched that sticker_pack 1’s stickers yet, so go to mysql!

Ok! The sticker I need is in mysql!

StickersService
lib/github/stickers_service.rb

SELECT * FROM stickers WHERE sticker_pack_id = 1

INSERT INTO stickers ...

Transition copying stickers
Moving statuses into the best_sellers cluster via temporary sharding

Where should I route this query with sticker_pack 1?

Ok! The sticker I need is in best_sellers!

sticker_routing table

sticker_pack_id 1 cluster "best_sellers"

lib/github/stickers_service.rb

```
SELECT * FROM stickers WHERE sticker_pack_id = 1
INSERT INTO stickers ...
```

Go to best_sellers!

Done copying! Update sticker_pack 1’s cluster to “best_sellers”

Transition copying stickers

Moving statuses into the best_sellers cluster via temporary sharding
Where should I route this query with sticker_pack 2?

Ok! I’ll read from mysql1 for this sticker pack’s stickers

Write new stickers for sticker_pack 2

We’re in the process of copying, read from mysql1 and write to both!

Moving sticker_pack 2’s stickers now!

Transition copying stickers

Moving statuses into the best_sellers cluster via temporary sharding

sticker_routing table
sticker_pack_id 1
cluster “best_sellers”
sticker_pack_id 2
cluster “moving”

SELECT * FROM stickers WHERE sticker_pack = 1

INSERT INTO stickers...

Write new stickers for sticker_pack 2

lib/github/stickers_service.rb

StickersService

Where should I route this query with sticker_pack 2?
Goliath Stickers table

• >10% of reads to OFF the master
• 7% of writes to OFF the master

They now represent 0%
Dude, where’s my tables?

- Moved 4 busy tables out of our main database cluster into two new clusters
  - All the traffic to those tables now doesn’t affect our master db cluster
- Used two very different but successful strategies
  - MySQL replication + pause writes + switch connections
  - Temporary sharding and gradual data copying
Conclusions.

• We haven’t had any availability incidents from load on our db that brought us down since we did this.

• We did it iteratively; quick and dirty the first few times, learned from that, and iterated. Circumstances were different every time.

• We didn’t do anything dramatic. We did the simplest and most effective thing we could.
It gave us the time and room to think long term.
Thanks!