Sharding @ Instagram

SFPUG April 2012
Mike Krieger
Instagram
me

- Co-founder, Instagram
- Previously: UX & Front-end @ Meebo
- Stanford HCI BS/MS
- @mikeyk on everything
pug!
communicating and sharing in the real world
Union Station. All mine.

view all 51 comments
braynelson liked 7 photos.

7 seconds ago

edroste left a comment on ernandaputra's photo:
@ernandaputra wow!
25 seconds ago

zachbulick and brenton_clarke liked wahldesign's photo.
29 seconds ago
30+ million users in less than 2 years
at its heart, Postgres-driven
a glimpse at how a startup with a small eng team scaled with PG
a brief tangent
the beginning
2 product guys
no real back-end experience
(you should have seen my first time finding my way around psql)
analytics & python @ meebo
CouchDB
early mix: PG, Redis, Memcached
...but were hosted on a single machine somewhere in LA
less powerful than my MacBook Pro
okay, we launched.  
now what?
25k signups in the first day
everything is on fire!
best & worst day of our lives so far
load was through the roof
friday rolls around
not slowing down
let's move to EC2.
PG upgrade to 9.0
scaling = replacing all components of a car while driving it at 100mph
this is the story of how our usage of PG has evolved
Phase 1: All ORM, all the time
why pg? at first, postgis.
./manage.py syncdb
ORM made it too easy to not really think through primary keys
pretty good for getting off the ground
Media.objects.get(pk = 4)
first version of our feed
(pre-launch)
friends = Relationships.objects.filter(source_user = user)

recent_photos = Media.objects.filter(user_id__in = friends).order_by(‘-pk’) [0:20]
main feed at launch
Redis:

// user 33 posts
friends = SMEMBERS followers:33
for user in friends:
    LPush feed:<user_id> <media_id>

// for reading
LRANGE feed:4 0 20
canonical data: PG
feeds/lists/sets: Redis
object cache: memcache
post-launch
moved db to its own machine
at time, largest table: photo metadata
ran master-slave from the beginning, with streaming replication
backups: stop the replica, xfs_freeze drives, and take EBS snapshot
AWS tradeoff
3 early problems we hit with PG
1 oh, *that* setting was the problem?
work_mem
shared_buffers
cost_delay
2 Django-specific:

<idle in transaction>
3 connection pooling
we use PGBouncer
somewhere in this crazy couple of months, Christophe to the rescue!
photos kept growing and growing...
...and only 68GB of RAM on biggest machine in EC2
so what now?
Phase 2: Vertical Partitioning
django db routers make it pretty easy
def db_for_read(self, model):
    if app_label == 'photos':
        return 'photoddb'
...once you untangle all your foreign key relationships
(all of those user/user_id interchangeable calls bite you now)
plenty of time spent in PGFouine
read slaves (using streaming replicas) where we need to reduce contention
a few months later...
photosdb > 60GB
precipitated by being on cloud hardware, but likely to have hit limit eventually either way
what now?
horizontal partitioning!
Phase 3: sharding
“surely we’ll have hired someone experienced before we actually need to shard”
never true about scaling
1 choosing a method
2 adapting the application
3 expanding capacity
evaluated solutions
at the time, none were up to task of being our primary DB
NoSQL alternatives
Skype’s sharding proxy
Range/date-based partitioning
did in Postgres itself
requirements
1 low operational & code complexity
2 easy expanding of capacity
3 low performance impact on application
schema-based logical sharding
many many many
(thousands) of logical shards
that map to fewer physical ones
// 8 logical shards on 2 machines

user_id % 8 = logical shard

logical shards -> physical shard map

{  
    0: A, 1: A,
    2: A, 3: A,
    4: B, 5: B,
    6: B, 7: B

}
// 8 logical shards on 2 4 machines

user_id % 8 = logical shard

logical shards → physical shard map

{ 
  0: A, 1: A, 
  2: C, 3: C, 
  4: B, 5: B, 
  6: D, 7: D
}
¡schemas!
all that ‘public’ stuff I’d been glossing over for 2 years
- database:
  - schema:
    - table:
      - columns
spun up set of machines
using fabric, created thousands of schemas
machineA:
  shard0
    photos_by_user
  shard1
    photos_by_user
  shard2
    photos_by_user
  shard3
    photos_by_user

machineB:
  shard4
    photos_by_user
  shard5
    photos_by_user
  shard6
    photos_by_user
  shard7
    photos_by_user
(fabric or similar parallel task executor is essential)
application-side logic
SHARD_TO_DB = {}

SHARD_TO_DB[0] = 0
SHARD_TO_DB[1] = 0
SHARD_TO_DB[2] = 0
SHARD_TO_DB[3] = 0
SHARD_TO_DB[4] = 1
SHARD_TO_DB[5] = 1
SHARD_TO_DB[6] = 1
SHARD_TO_DB[7] = 1
instead of Django ORM, wrote really simple db abstraction layer
select/update/insert/delete
select(fields, table_name, shard_key, where_statement, where_parameters)
select(fields, table_name, shard_key, where_statement, where_parameters)

... shard_key % num_logical_shards = shard_id
in most cases, user_id for us
custom Django test runner to create/tear-down sharded DBs
most queries involve visiting handful of shards over one or two machines
if mapping across shards on single DB, UNION ALL to aggregate
clients to library pass in:
((shard_key, id),
(shard_key, id)) etc
library maps sub-selects to each shard, and each machine
parallel execution! (per-machine, at least)
Append (cost=0.00..973.72 rows=100 width=12) (actual time=0.290..160.035 rows=30 loops=1)
  Limit (cost=0.00..806.24 rows=30 width=12) (actual time=0.288..159.913 rows=14 loops=1)
    Index Scan Backward using index on table (cost=0.00..18651.04 rows=694 width=12) (actual time=0.286..159.885 rows=14 loops=1)
    Limit (cost=0.00..71.15 rows=30 width=12) (actual time=0.015..0.018 rows=1 loops=1)
      Index Scan using index on table (cost=0.00..101.99 rows=43 width=12) (actual time=0.013..0.014 rows=1 loops=1)
      (etc)
eventually, would be nice to parallelize across machines
next challenge: unique IDs
requirements
1 should be time sortable without requiring a lookup
2 should be 64-bit
3 low operational complexity
surveyed the options
ticket servers?
UUID?
twitter snowflake?
application-level IDs ala Mongo?
hey, the db is already pretty good about incrementing sequences
[ 41 bits of time in millis ]
[ 13 bits for shard ID ]
[ 10 bits sequence ID ]
[ 41 bits of time in millis ]
[ 13 bits for shard ID ]
[ 10 bits sequence ID ]
[ 41 bits of time in millis ]
[ 13 bits for shard ID ]
[ 10 bits sequence ID ]
[ 41 bits of time in millis ]
[ 13 bits for shard ID ]
[ 10 bits sequence ID ]
CREATE OR REPLACE FUNCTION insta5.next_id(OUT result bigint) AS $$
DECLARE
  our_epoch bigint := 1314220021721;
  seq_id bigint;
  now_millis bigint;
  shard_id int := 5;
BEGIN
  SELECT nextval('insta5.table_id_seq') % 1024 INTO seq_id;
  SELECT FLOOR(EXTRACT(EPOCH FROM clock_timestamp()) * 1000) INTO now_millis;
  result := (now_millis - our_epoch) << 23;
  result := result | (shard_id << 10);
  result := result | (seq_id);
END;
$$ LANGUAGE PLPGSQL;
# pulling shard ID from ID:

```
shard_id = id ^ ((id >> 23) << 23)
timestamp = EPOCH + id >> 23
```
pros: guaranteed unique in 64-bits, not much of a CPU overhead
cons: large IDs from the get-go
hundreds of millions of IDs generated with this scheme, no issues
well, what about “re-sharding”
first recourse: pg_reorg
rewrites tables in index order
only requires brief locks for atomic table renames
20+GB savings on some of our dbs
especially useful on EC2
but sometimes you just have to reshard
streaming replication to the rescue
(btw, repmgr is awesome)
repmgr standby clone <master>
machineA:
  shard0
  photos_by_user
shard1
  photos_by_user
shard2
  photos_by_user
shard3
  photos_by_user

machineA’:
  shard0
  photos_by_user
shard1
  photos_by_user
shard2
  photos_by_user
shard3
  photos_by_user
PGBouncer abstracts moving DBs from the app logic
can do this as long as you have more logical shards than physical ones.
beauty of schemas is that they are physically different files
(no IO hit when deleting, no ‘swiss cheese’)
downside: requires ~30 seconds of maintenance to roll out new schema mapping
(could be solved by having concept of "read-only" mode for some DBs)
not great for range-scans that would span across shards
latest project: follow graph
v1: simple DB table
(source_id, target_id, status)
who do I follow?
who follows me?
do I follow X?
does X follow me?
DB was busy, so we started storing parallel version in Redis
follow_all(300 item list)
inconsistency
extra logic
so much extra logic
exposing your support team to the idea of cache invalidation
reset redis cache
redesign took a page from twitter’s book
PG can handle tens of thousands of requests, very light memcached caching
next steps
isolating services to minimize open conns
investigate physical hardware / etc to reduce need to re-shard
Wrap up
you don’t need to give up PG’s durability & features to shard
continue to let the DB do what the DB is great at
“don’t shard until you have to”
(but don’t over-estimate how hard it will be, either)
scaled within constraints
of the cloud
PG success story
(we’re really excited about 9.2)
thanks! any qs?