Scaling database with Django and HAProxy

07 Oct 2013

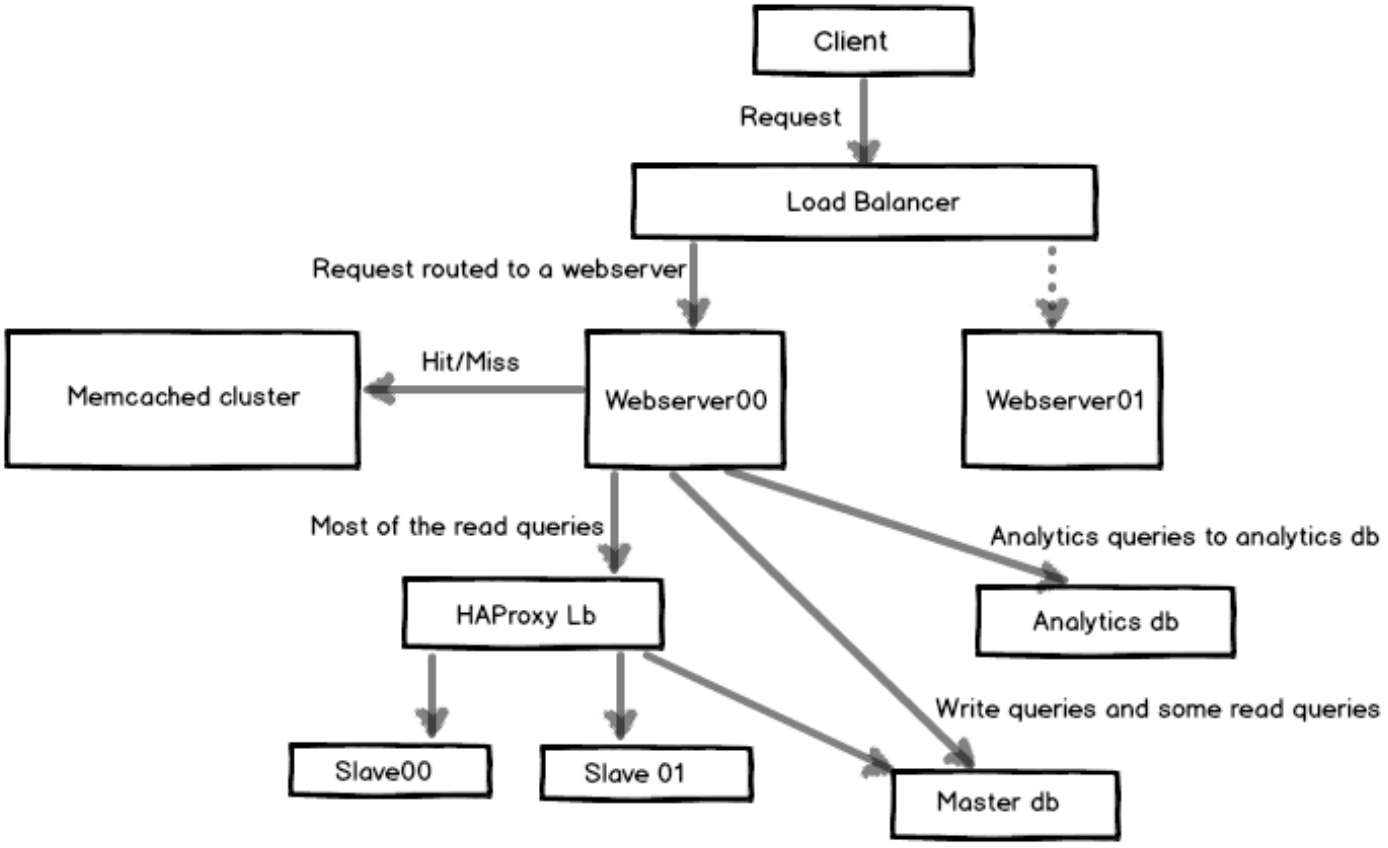
MySQL - Primary data store

At HackerEarth, we use MySQL database as the primary data store. We have experimented with a few NoSQL databases on the way, but the results have been largely unsatisfactory. The distributed databases like MongoDB or CouchDB aren't very scalable or stable. Right now, our status monitoring services use RethinkDB for storing the data in JSON format and that's all for the NoSQL database usage right now.

With the growing data and number of requests/sec, it turns out that the database becomes the major bottlneck to scale the application dynamically. At this point if you are thinking that there are mythical (cloud) providers who can handle the growing need of your application, you can't be more wrong. To make the problem even harder, you can't spin a new database whenever you want to just like your frontend servers. To achieve a horizontal scalability at all levels, it requires massive rearchitecture of the system while being completely transparent to the end user. This is what a part of our team has focussed on in last few months, resulting in very high uptime and availability.

The master (and only) MySQL database had started being under heavy load recently. We thought we will delay any scalability at this level till the single database could handle the load, and we will work on other high priority things. But that was not supposed to go as planned and we experienced a few downtimes. After that we did a rearchitecture of our application, sharded the database, wrote database routers and wrappers on top of django ORM, put HAProxy load balancer infront of the MySQL databases, and refactored our codebase to optimize it significantly.

The image below shows a part of the architecture we have at HackerEarth. Many other components have been omitted for simplicity.



Database slaves and router

The idea was to create read replicas and route the write queries to master database and read queries to slave (read replica) databases. But that was not so simple again. We couldn't and wouldn't want to route all the read queries to slaves. There were some read queries which couldn't afford stale data, which comes as a part of database replication. Though stale data might be the order of just a few seconds, these small number of read queries couldn't even afford that.

The first database router was simple:

```
class MasterSlaveRouter(object):
    """
    Represents the router for database lookup.
    """
    def __init__(self):
        if settings.LOCAL:
            self._SLAVES = []
        else:
            self._SLAVES = SLAVES

    def db_for_read(self, model, **hints):
        """
        Reads go to default for now.
        """
        return 'default'
```

```
    def db_for_write(self, model, **hints):
        """
        Writes always go to default.
        """
        return 'default'

    def allow_relation(self, obj1, obj2, **hints):
        """
        Relations between objects are allowed if both objects are
        in the default/slave pool.
        """
        db_list = ('default',)
        for slave in zip(self._SLAVES):
            db_list += slave

        if obj1._state.db in db_list and obj2._state.db in db_list:
            return True
        return None

    def allow_migrate(self, db, model):
        return True
```

All the write and read queries go the master database, which you might think is weird here. Instead, we wrote get*from*slave(), filter*from*slave(), get*object*or*404*from*slave(), get*list*or404*from*slave(), etc. as part of django ORM in our custom managers to read from slave. So whenever we know we can read from slaves, we call one of these functions. This was a sacrifice made for those small number of read queries which couldn't afford the stale data.

Custom database manager to fetch data from slave:

```
# proxy_slave_X is the HAProxy endpoint, which does load balancing
# over all the databases.
SLAVES = ['proxy_slave_1', 'proxy_slave_2']

def get_slave():
    """
    Returns a slave randomly from the list.
    """
    if settings.LOCAL:
        db_list = []
    else:
        db_list = SLAVES

    return random.choice(db_list)
```

```
class BaseManager(models.Manager):
    # Wrappers to read from slave databases.
    def get_from_slave(self, *args, **kwargs):
        self._db = get_slave()
        return super(BaseManager, self).get_query_set().get(*args, **kwargs)

    def filter_from_slave(self, *args, **kwargs):
        self._db = get_slave()
        return super(BaseManager, self).get_query_set().filter(
                *args, **kwargs).exclude(Q(hidden=True) | Q(trashed=True))
```

HAProxy for load balancing

Now the slaves could be in any number at a time. One option was to update the database configuration in settings whenever we added/removed a slave. But that was very cumbersome and inefficient. The other better way was to put a HAProxy load balancer in front of all the databases and let it detect which one is up or down and route the read queries according to that. This would mean never editing the database configuration in our codebase, just what we wanted.

A snippet of /etc/haproxy/haproxy.cfg:

```
listen mysql *:3305
    mode tcp
    balance roundrobin
    option mysql-check user haproxyuser
    option log-health-checks
    server db00 db00.xxxxx.yyyyyyyyyy:3306 check port 3306 inter 1000
    server db01 db00.xxxxx.yyyyyyyyyy:3306 check port 3306 inter 1000
    server db02 db00.xxxxx.yyyyyyyyyy:3306 check port 3306 inter 1000
```

The configuration for slave in settings now looked like this:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'db_name',
        'USER': 'username',
        'PASSWORD': 'password',
        'HOST': 'db00.xxxxx.yyyyyyyyyy',
        'PORT': '3306',
    },
```

```
    'proxy_slave_1': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'db_name',
        'USER': 'username',
        'PASSWORD': 'password',
        'HOST': '127.0.0.1',
        'PORT': '3305',
    },
    'analytics': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'db_name',
        'USER': 'username',
        'PASSWORD': 'password',
        'HOST': 'db-analytics.xxxxx.yyyyyyyyyy',
        'PORT': '3306',
    },
}
```

But there is a caveat here too. If you spin off a new server with the haproxy configuration containing some endpoints which doesn't exists, haproxy will throw an error and it won't start, making the slave useless. It turns out there is no easy solution to this, and haproxy.cfg should contain existing server endpoints while initializing. The solution then was to let the webserver update its haproxy configuration from a central location whenever it starts. We wrote a simple script in fabric to do this. Besides, the webserver already used to update its binary when spinned off from an old image.

Database sharding

Next, we sharded the database. We created another database - *analytics*. It stores all the computed data and they form a major part of read queries. All the queries to analytics database are routed using the following router:

```
class AnalyticsRouter(object):
    """
    Represents the router for analytics database lookup.
    """
    def __init__(self):
        if settings.LOCAL:
            self._SLAVES = []
            self._db = 'default'
        else:
            self._SLAVES = []
            self._db = 'analytics'

    def db_for_read(self, model, **hints):
        """
        All reads go to analytics for now.
        """
```

```
        if model._meta.app_label == 'analytics':
            return self._db
        else:
            return None

    def db_for_write(self, model, **hints):
        """
        Writes always go to analytics.
        """
        if model._meta.app_label == 'analytics':
            return self._db
        else:
            return None

    def allow_relation(self, obj1, obj2, **hints):
        """
        Relations between objects are allowed if both objects are
        in the default/slave pool.
        """

        if obj1._meta.app_label == 'analytics' or \
                obj2._meta.app_label == 'analytics':
            return True
        else:
            return None

    def allow_migrate(self, db, model):
        if db == self._db:
            return model._meta.app_label == 'analytics'
        elif model._meta.app_label == 'analytics':
            return False
        else:
            return None
```

To enable the two routers, we need to add them in our global settings:

```
DATABASE_ROUTERS = ['core.routers.AnalyticsRouter', 'core.routers.MasterSlaveRoute
```

Here the order of routers is important. All the queries for analytics are routed to the analytics database and all the other queries are routed to the master database or their slaves according the nature of queries. For now, we have not put slaves for analytics database but as the usage grows that will be fairly straightforward to do now.

At the end, we had an architecture where we could spin off new read replicas, route the queries fairly simply and had a high performance load-balancer in front of the databases. All this has resulted in much

higher uptime and stability in our application and we could focus more on what we love to do - building products for programmers. We already had an automated deployment system in place, which made the exprimentation easier and enabled us to test everything thoroughly. The refactoring and optimization that we did in codebase and architecture also helped us to reduce the servers count by more than two times. This has been a huge win for us, and we are now focussing on rolling out exciting products in next few weeks. Stay tuned!

I would love to know from others about how they have solved similar problems, give suggestions and point out potential quirks.

P.S. You might be interested in The HackerEarth Data Challenge that we are running.

Posted by Vivek Prakash. Follow me @vivekprakash. Write to me at vivek@hackerearth.com.

HackerEarth 11 | Django 8 | Database 1 | MySQL 1 | HAProxy 1 | LoadBalancing 1

"Redis is an in-memory but persistent on disk database, so it represents a different trade off where very high write and read speed is achieved with the limitation of data sets that can't be larger than memory."

I hope it explains a bit.

2 ^ | v • Reply • Share ›

pramodliv1 ➦ Vivek Prakash • 2 years ago

Yes! Thanks for sharing the thought process involved. It seems redis is popular for real time analytics. For example, displaying graphs of today's data so that the latest data is available on page refresh from a web interface very quickly and then evicting the data periodically. And then another reliable database is used for long term trends.

^ | v • Reply • Share ›

Karan Agarwal • 2 years ago

Got any benchmarking with read/write db routing and sharding? Also I am curious to know how nosql results were unsatisfactory. What instability did you notice with mongo or couch?

^ | v • Reply • Share ›

Shekhar Kadyan • 2 years ago

your link on sharded database is broken, you forgot the trailing ')' and it should be https://en.wikipedia.org/wiki/...

^ | v • Reply • Share ›