Efficient Django QuerySet Use

Jayson Falkner, PhD
jayson@counsyl.com
@jaysonfalkner

Overview

- 30 min talk
- How to go from Django QuerySet to SQL
- Straight-forward way to profile
- Fast CRUD (Create, Read, Update, Delete)

Assumptions

- You are using Python + Django
    - Avoiding direct SQL use
    - Postgres (aka PostgreSQL)
- Performance matters
- Slides and code are on GitHub

github.com/jfalkner/Efficient-Django-QuerySet-Use

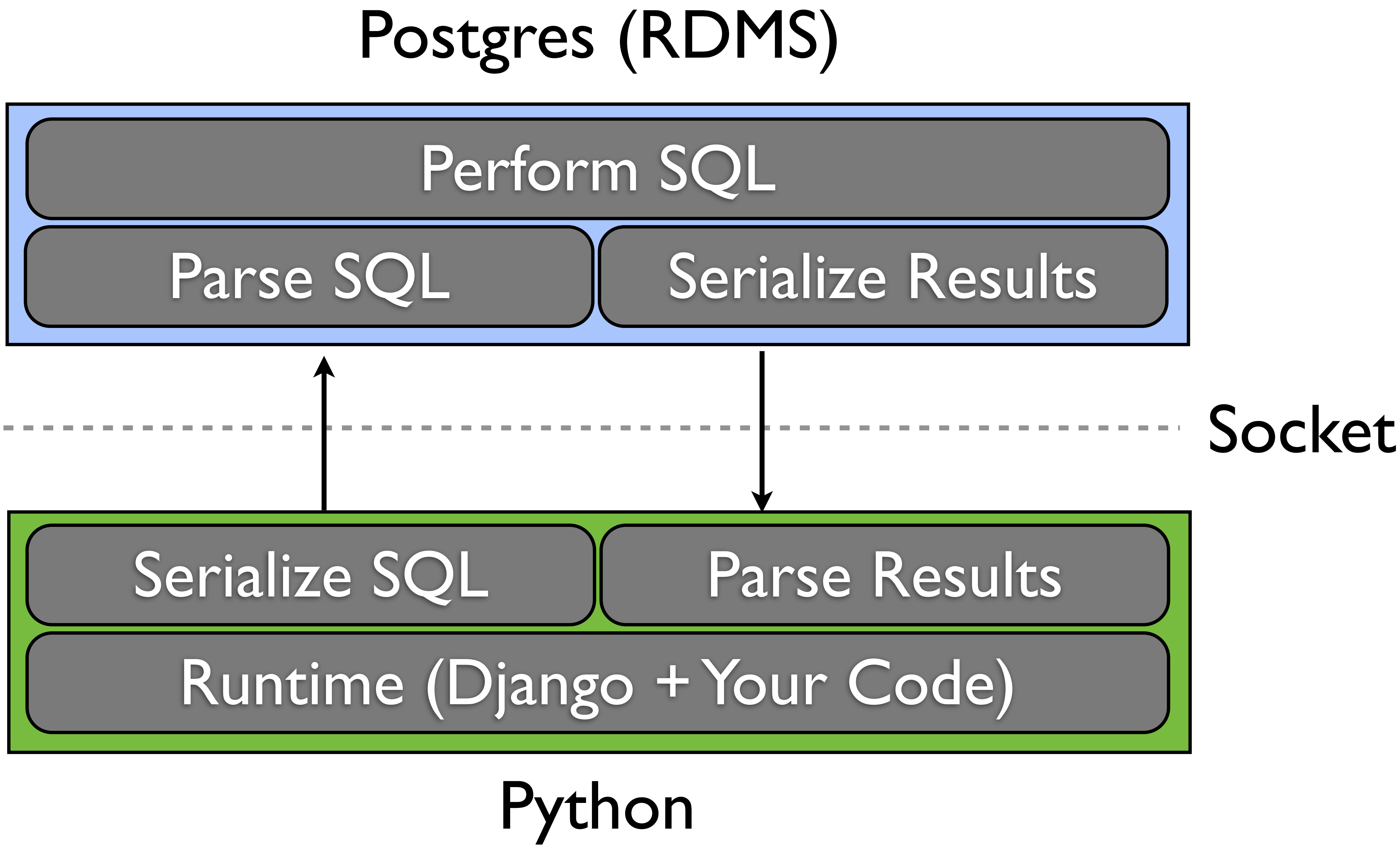
A simple data model

```
class Sample(models.Model):
    barcode = models.CharField(max_length=10, unique=True)
    production = models.BooleanField()
    created = models.DateTimeField()

    def status(self):
        return self.statuses.all()[0]

class SampleStatus(models.Model):
    sample = models.ForeignKey(Sample,
                               related_name='statuses')
    status_code = models.PositiveSmallIntegerField()
    created = models.DateTimeField()

    RECEIVED = 1; LAB = 2; COMPLETE = 3
```

Don’t do this

```
# What samples are in the lab?
samples = Sample.objects.filter(
    production=True,
    statuses__status_code=SampleStatus.LAB)

# Loop through. It’s business time!
for s in samples:
    do_something(s.barcode,
                 s.status().created)
```

What takes time?



Postgres (RDMS)

Perform SQL

Parse SQL

Serialize Results

Socket

Serialize SQL

Parse Results

Runtime (Django + Your Code)

Python

Poor Man's Django/QuerySet Profiling

```
samples = Sample.objects.filter(
    production=True,
    statuses__status_code =SampleStatus.LAB)
for s in samples:
    do_something(s.barcode,
                 s.status().created)
```

Poor Man’s Django/QuerySet Profiling

```
from datetime import datetime
from django.db import connection as con
from example.models import Sample, SampleStatus

start = datetime.now()
query_count = len(con.queries)

samples = Sample.objects.filter(
    production=True,
    statuses__status_code = SampleStatus.LAB)
for s in samples:
    do_something(s.barcode,
                 s.status().created)

print “Time: %s”%(datetime.now()-start)
print “Queries: %s”%(len(con.queries)-query_count)
for query in connection.queries[query_count:]:
    print query
```

Poor Man’s Django/QuerySet Profiling

```
from example.models import Sample, SampleStatus
import django_pg_utils

django_pg_utils.track_sql()

samples = Sample.objects.filter(
    production=True,
    statuses__status_code = SampleStatus.LAB)
for s in samples:
    do_something(s.barcode,
                 s.status().created)

django_pg_utils.print_sql()
```

github.com/jfalkner/Efficient-Django-QuerySet-Use

SQL Read in QuerySet (1,000 samples)

```
# Read all matching samples.
samples = Sample.objects.filter(
    production=True,
    statuses__status_code=SampleStatus.LAB)
# Loop with query for `created` value.
for s in samples:
    do_something(s.barcode, s.status().created)
```

Python: 1.112s, Postgres: 0.009s, Queries: 1,001

```
SELECT "id", "barcode", "production", "created"
FROM "db_sample"
WHERE "db_sample"."production" = TRUE
```

```
SELECT "id", "sample_id", "status", "created"
FROM "db_samplestatus"
WHERE "sample_id" = 10
ORDER BY "created" DESC LIMIT 1
```

```
(And one more for each other sample...)
```

Why is it slow?

- O(n) queries when it could be O(1)
  - One query for all samples
  - One query per status of sample
- Serializes/deserializes unused values
  - Sample: id, created
  - SampleStatus: id, status, sample_id

SQL Read in QuerySet (1,000 samples)

```
samples = Sample.objects.filter(
    production=True,
    statuses__status_code=SampleStatus.LAB)
# Cache all lookups once in Python dict.
samples = samples.prefetch_related(‘statuses’)
for s in samples:
    do_something(s.barcode, s.status().created)
```

Python: 0.460s, Postgres: 0.018s, Queries: 2

```
SELECT "id", "barcode", "production", "created"
FROM "db_sample"
INNER JOIN "db_samplestatus" ON
     ("id" = "db_samplestatus"."sample_id")
WHERE ("production" = true AND
      "db_samplestatus"."status_code" = 2 )
```

```
SELECT "id", "sample_id", "status_code", "created"
FROM "db_samplestatus"
WHERE "sample_id" IN (1220001, 1220002, 1220003 ...
```

Improving even more

- O(1) query = Good
- Python dict for lookups is relatively slow
  - Let Postgres do the work
- Return only params of interest
  - Minimizes serialize/deserialize time
  - Smaller memory footprint

SQL Read in QuerySet (10,000 samples)

```
Sample.objects
    # JOIN and calc max status code from SampleStatus.
    .annotate(latest_status_code=Max(
              'statuses__status_code'))
    .filter(production=True,
            latest_status_code__eq=SampleStatus.LAB)
    # Return exactly the fields to use.
    .values_list('barcode', 'statuses__created')
```

Python: 0.23s, Postgres: 0.18s, Queries: 1

```
SELECT "barcode", "created"
FROM "db_sample"
LEFT OUTER JOIN "db_samplestatus" ON
      ("id" = "db_samplestatus"."sample_id")
WHERE ("production" = TRUE)
GROUP BY "id", "barcode", "production", "created",
         "status_code", "db_samplestatus"."status_code"
HAVING MAX("db_samplestatus"."status_code") = 2
```

Using select_related()

JOIN via denormalized foreign key on Sample

```
class Sample(models.Model):
    barcode = models.CharField(max_length=10, unique=True)
    production = models.BooleanField()
    created = models.DateTimeField()
    latest_status = models.ForeignKey(
        'SampleStatus', related_name='+', null=True)

    def status(self):
        return self.statuses.all()[0]

class SampleStatus(models.Model):
    ...
```

SQL Read in QuerySet (10,000 samples)

```
samples = Sample.objects.filter(
    production=True,
    statuses__status_code=SampleStatus.LAB)
samples = samples.select_related('latest_status')
for s in samples:
    do_something((s.barcode, s.latest_status.created))
```

Python: 0.71s, Postgres: 0.13s, Queries: 1

```
SELECT "id", "barcode", "production", "created",
       "latest_status_id",
       T3."id", T3."sample_id", T3."status",
       T3."status_code", T3."created"
FROM "example_sample"
INNER JOIN "example_samplestatus" ON
           ("id" = "example_samplestatus"."sample_id")
LEFT OUTER JOIN "example_samplestatus" T3 ON
                ("latest_status_id" = T3."id")
WHERE ("production" = TRUE
       AND "example_samplestatus"."status_code" = 2)
```

Denormalize for speed

Change the data model to avoid a JOIN

```
class Sample(models.Model):
    barcode = models.CharField(max_length=10, unique=True)
    production = models.BooleanField()
    created = models.DateTimeField()
    status_code = models.PositiveSmallIntegerField()
    status_changed = models.DateTimeField()

    def status(self):
        return self.statuses.all()[0]

class SampleStatus(models.Model):
    ...
```

SQL Read in QuerySet (1,000,000 samples)

```
# Read all at once and return a list of tuples.
Sample.objects
      .filter(production=True,
              status_code = SampleStatus.LAB)
      .values_list('barcode', 'status_changed')
```

Python: 0.18s, Postgres: 0.17s, Queries: 1

```
SELECT "db_sample"."barcode",
       "db_sample"."status_code"
FROM "db_sample"
WHERE ("db_sample"."status_code" = 2
       AND "db_sample"."production" = TRUE)
```

Why is it faster?

- JOINs and table scans are *relatively* slow
- SQL’s EXPLAIN helps show this

```
EXPLAIN SELECT "db_sample"."barcode",
               "db_sample"."status_code"
FROM "db_sample"
WHERE ("db_sample"."status_code" = 2
       AND "db_sample"."production" = TRUE)
```

```
Seq Scan on db_sample  (cost=0.00..27203.00 rows=5567 width=8)
  Filter: (production AND (status_code = 2))
```

postgresql.org/docs/9.1/static/using-explain.html

SQL Read in QuerySet (1,000,000 samples)

Sample, SampleStatus JOIN = 11s

```
GroupAggregate  (cost=691622.75..787997.34 rows=2965372 width=23)
  Filter: (max(db_samplestatus.status_code) = 2)
  ->  Sort  (cost=691622.75..699036.18 rows=2965372 width=23)
        Sort Key: id, barcode, production, created, status_code,
        ->  Hash Right Join  (cost=43063.00..190400.49 rows=2965372 width=23)
              Hash Cond: (db_samplestatus.sample_id = db_sample.id)
              ->  Seq Scan on db_samplestatus  (cost=0.00..51015.90 rows=2993590 width=6)
              ->  Hash  (cost=24703.00..24703.00 rows=1000000 width=21)
                    ->  Seq Scan on db_sample  (cost=0.00..24703.00 rows=1000000 width=21
                          Filter: production
```

Denormalized = 0.175s

```
Seq Scan on db_sample  (cost=0.00..27203.00 rows=5567 width=8)
  Filter: (production AND (status_code = 2))
```

Denormalized + Multicolumn INDEX = 0.004s

```
Index Scan using db_sample_prod_lab on db_sample  (cost=0.00..9101.15 rows=5567 width=8)
  Index Cond: ((status_code = 2) AND (production = true))
  Filter: production
```

Multi-Column Indexing

Django Model ‘index’ won’t help

```
class Sample(models.Model):
    barcode = models.CharField(max_length=10, unique=True)
    production = models.BooleanField()
    created = models.DateTimeField()
    status_code = models.PositiveSmallIntegerField(index=True)
    status_changed = models.DateTimeField()
```

Postgres-specific CREATE INDEX helper method

```
from django_pg_utils import pg_multicolumn_index
pg_multicolumn_index(Sample, ['production', 'status_code'])
```

How long does it take?

QuerySet SELECT	# Samples	Time (Postgres)
Loop	1,000	1.11s (0.01s)
Loop prefetch_selected()	1,000	0.46s (0.02s)
Loop select_related()	10,000	0.71s (0.13s)
values_list() + JOIN	10,000	0.23s (0.19s)
Denorm + INDEX	100,000,000	0.99s (0.38s)

On a MacBook Pro Retina. Postgres 9.1. No config tweaks.

In context at Counsyl

- How much is 100,000,000 samples?
  - query: samples to process this week?
- Consider Counsyl's main product
  - 4,000,000 US pregnancies per year
  - Screen 2x people (mother and father)
  - That is 12 years of samples!

Oh yeah, other stuff!

- Batching CRUD actions = speed
  - Avoid any O(n) loops on model objects
- Bulk CREATE, UPDATE, DELETE exist
  - bulk_create(), update(), delete()
  - save() won't work
- Helper code for multi-value UPDATE

docs.djangoproject.com/en/dev/topics/db/queries/

Inefficient UPDATE in QuerySet

```
samples = Sample.objects.filter(production=True)
# Loop, update, and invoke `save()` for each.
for s in samples:
    s.barcode = 'PREFIX'+s.barcode
    s.save()
```

Queries: O(n)

```
SELECT "id", "barcode", "production", "created"
FROM "db_sample"
WHERE "production" = TRUE
SELECT (1) AS "a"
FROM "db_sample"
WHERE "id" = 1 LIMIT 1
```

```
UPDATE "db_sample"
SET "barcode" = 'PREFIX0',
    "production" = TRUE, "created" = '2013-09-29'
WHERE "db_sample"."id" = 1
```

(And one more for each other sample...)

Efficient UPDATE in QuerySet

```
# Loop and make samples one at a time.
now = datetime.now()
(Sample.objects.filter(production=True)
               .update(created=now))
```

Queries: O(1)

```
UPDATE "db_sample"
SET "created" = '2013-09-29 03:40:18.925695-05:00'
WHERE "db_sample"."production" = TRUE
```

No multi-value update?

- Django’s update() is limited to one value
- SQL has no such restriction

Efficient UPDATE in QuerySet

```
values = Sample.objects.values_list('id', 'barcode')

# Loop and make samples one at a time.
filter_vals = [id for id, _ in values]
update_vals = ['PREFIX'+bar for _, bar in values]
django_pg_utils.pg_bulk_update(Sample, 'id', 'barcode',
                               filter_vals, update_vals)
```

Queries: O(1)

```
SET barcode = input.update
FROM
  (SELECT unnest(ARRAY[1, 2, 3, 4, 5, 6]),
          unnest(ARRAY[6, 6, 6, 6, 6, 6])) AS INPUT (filter,
UPDATE)
WHERE id = INPUT.filter;
```

Inefficient CREATE in QuerySet

```
# Loop and make samples one at a time.
for barcode in barcodes:
    Sample.objects.create(barcode=barcode,
                          production=True,
                          created=datetime.now())
```

Queries: O(n)

```
INSERT INTO "example_sample" ("barcode",
                              "production",
                              "created")
VALUES ('01234', TRUE, '2013-10-04 01:55:15.456163-05:00',
                       2,
                       '2013-10-04 01:55:15.456173-05:00')
RETURNING "example_sample"."id"
```

```
(And one more for each other sample...)
```

Efficient CREATE in QuerySet

```
# Buffer all Sample instances with values in a list.
samples = []
for barcode in barcodes:
    samples.append(Sample(barcode=barcode,
                          production=True,
                          created=datetime.now())
# Bulk create all samples at once.
Sample.objects.bulk_create(samples)
```

Queries: O(1)

```
INSERT INTO "example_sample" ("barcode",
                              "production",
                              "created")
VALUES ('100000', TRUE, '2013-10-04 02:00:01.556393-05:00'),
       ('100001', TRUE, '2013-10-04 02:00:01.556483-05:00'),
       ('100002', TRUE, '2013-10-04 02:00:01.556526-05:00'),
       ...
```

Summary

- Batch/Bulk Everything FTW
  - READ: values() and values_list()
    - pg_multicolumn_index()
  - CREATE: bulk_create()
  - UPDATE: update() and pg_bulk_update()
  - DELETE: delete()

github.com/jfalkner/Efficient-Django-QuerySet-Use

Efficient Django QuerySet Use

Jayson Falkner, PhD
jayson@counsyl.com
@jaysonfalkner

github.com/jfalkner/Efficient-Django-QuerySet-Use