Very, Very ***Fast*** Django

Christophe Pettus
PostgreSQL Experts, Inc.

thebuild.com
pgexperts.com

Who?

- Christophe Pettus. Hi!
- pgexperts.com
- thebuild.com
- @xof
- christophe.pettus@pgexperts.com

What is this talk?

- PostgreSQL Experts, Inc. is a database consultancy.
  - You probably guessed that.
- We also have an applications development practice.
  - We mostly do Django development.

PGX
POSTGRESQL
EXPERTS, INC.

Go Faster Button.

Tales from the battlefield.

- We have clients who have very, very large Django sites.
- We've collected a lot of wisdom on how they managed to keep their sites up.
- This talk is a distillation of their wisdom.
- Others (especially us) have made all these mistakes, so you don't have to.

Structure.

- Tips and tricks.
- Things not to do.
- Please ask questions!
- Please disagree!
- And now, let’s start with…

How fast is Django, anyway?

You hear things.

- “The ORM is incredibly slow.”
- “Django’s template engine isn’t as fast as Jinja2 / PHP / JSP / this hand-coded C-language thing from 1998.”
- “You can’t scale a Django site because the only language I’ve ever learned is Ruby.”

When in doubt, measure.

- Basic timing tests on this very laptop.
- Using the development server.
- Very simple view functions and model.
- Django 1.6.4, out of the box.

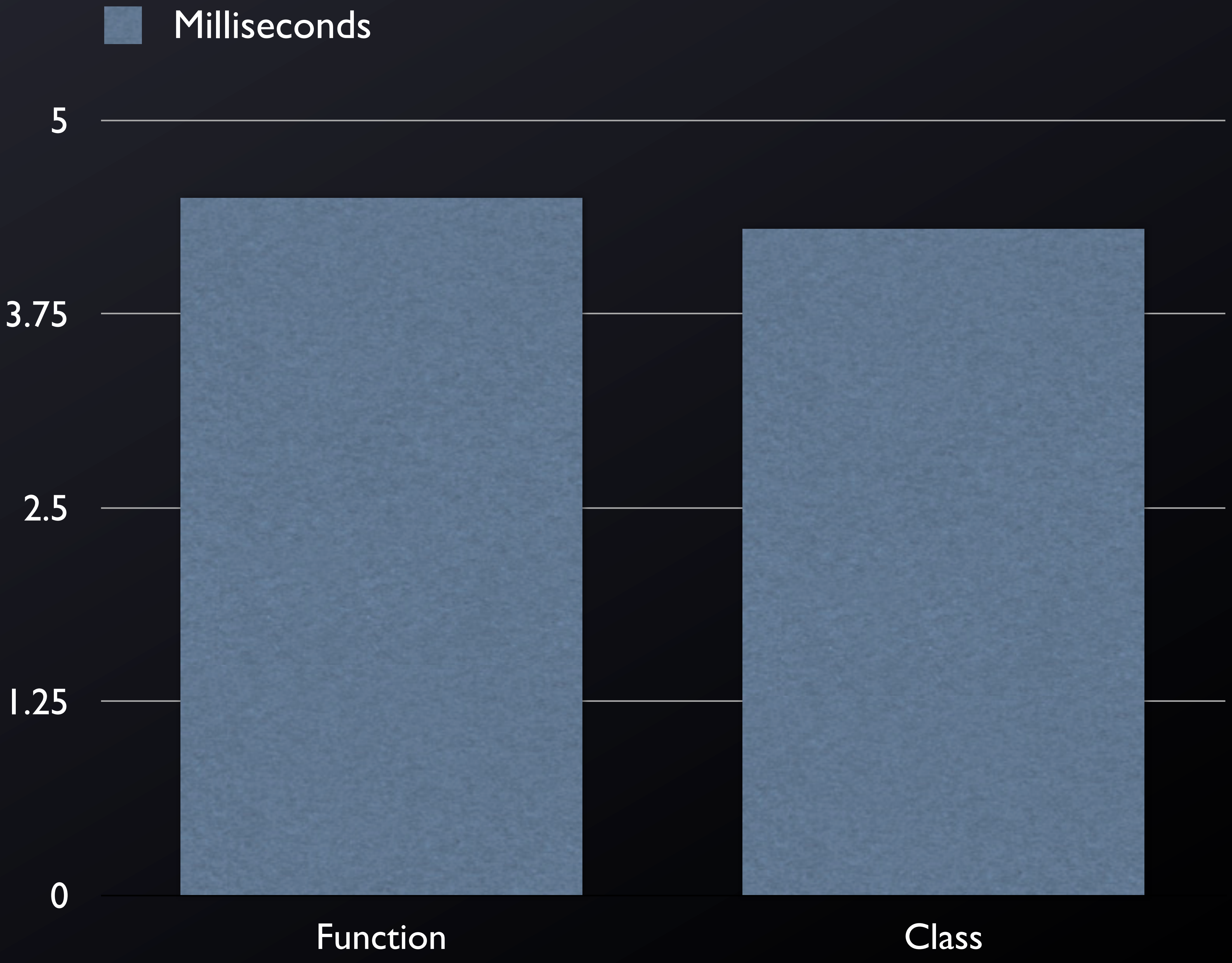
The goal.

- How high-overhead are Django’s standard components?
  - Are they really slow, or are people using them in slow ways?
- What are good and bad ways to use them?

Test 1: Empty HTTPResponse

- Just return HTTPReponse(“”).
- Both class-based and function based views.
- Utterly meaningless number…
- … but provides a baseline for the others.

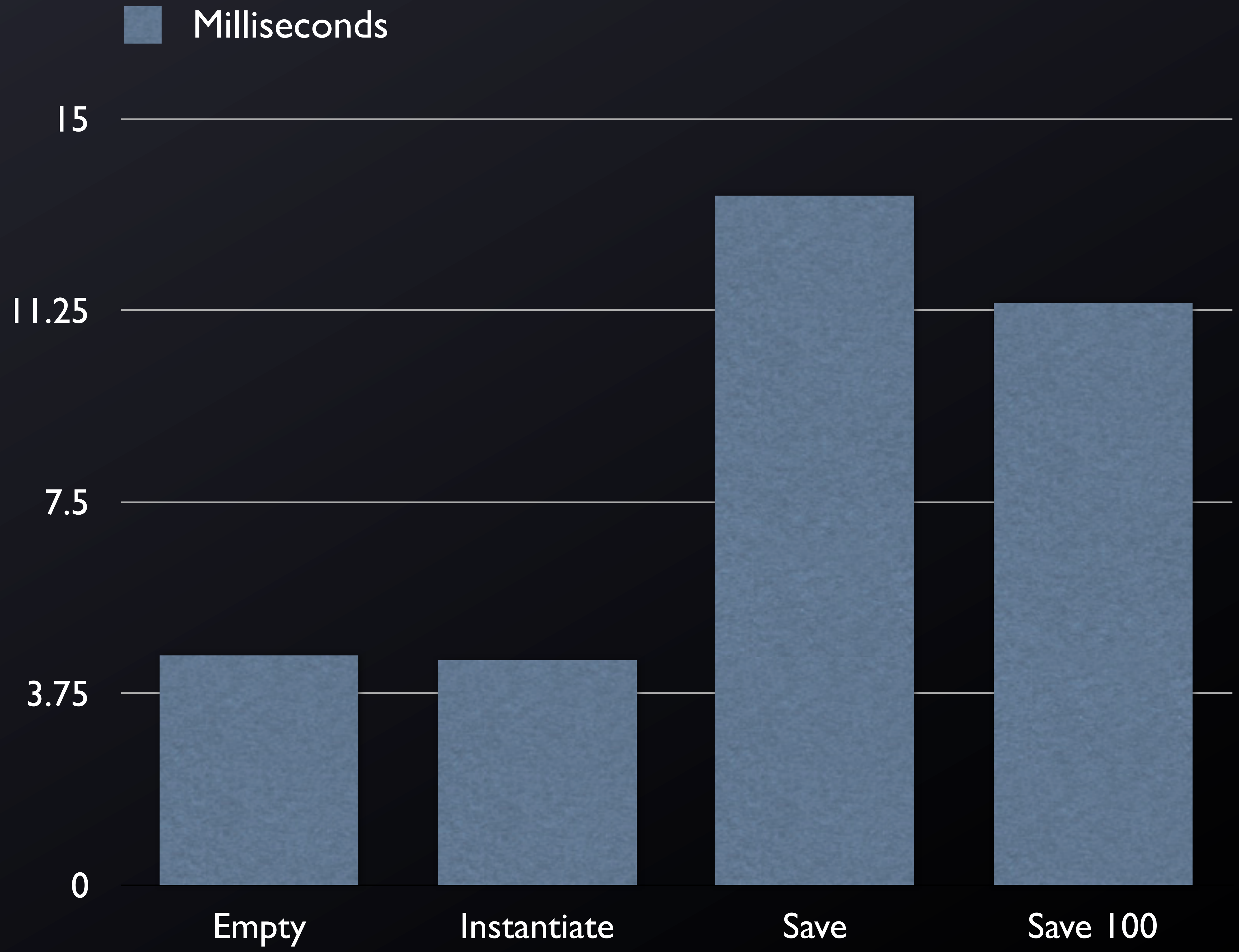
Test 1: Empty HTTPResponse



Test 2: Create, save model objects

- Model object has nine fields.
  - Most ORM operations are O(N) on the number of fields.
- Create, do not save.
- Create, save.

Test 2: Create, save model objects

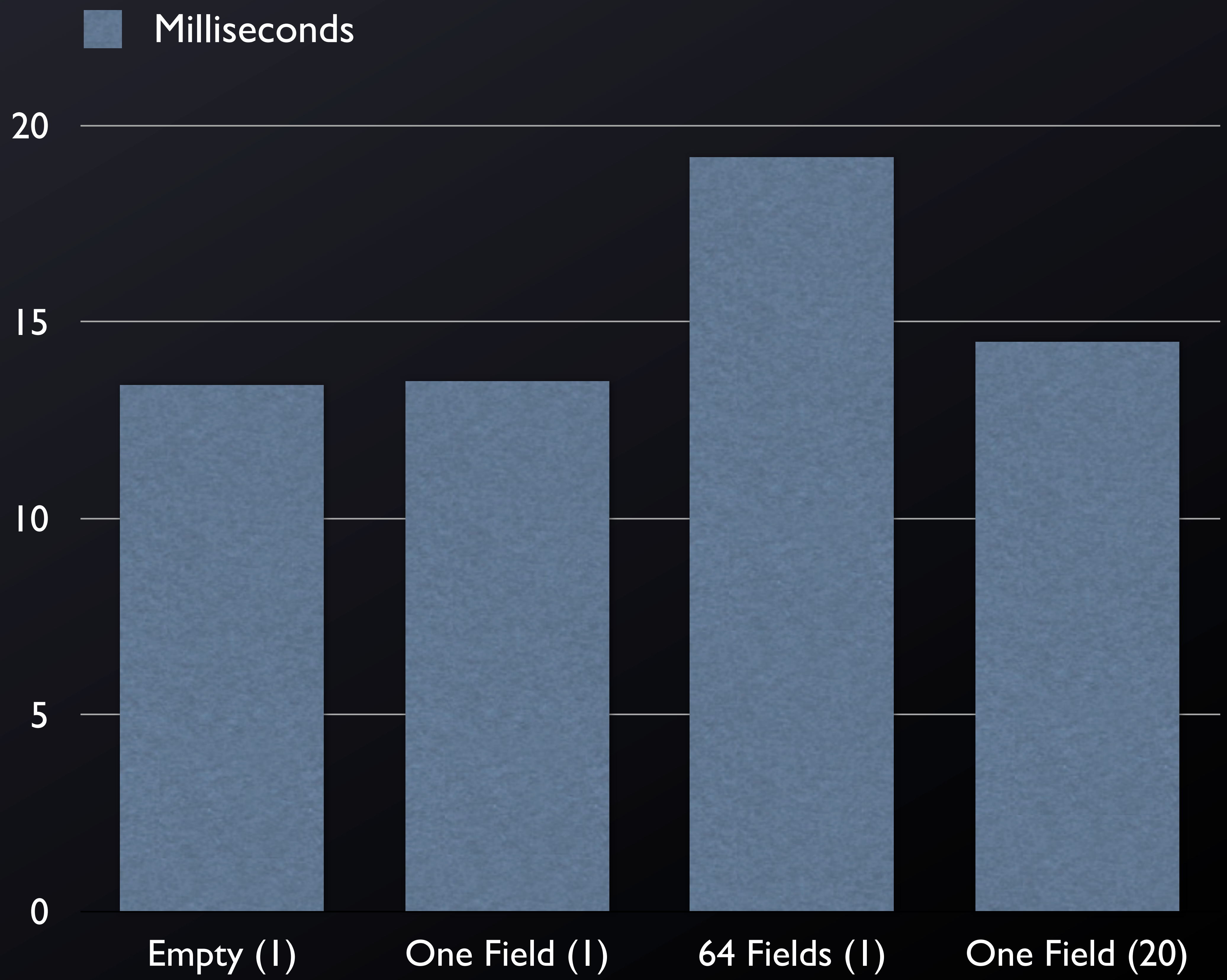


PGX POSTGRESQL EXPERTS, INC.

Test 3: Template rendering.

- Render templates of a variety of complexity.
- Includes loading 1-20 objects as the source for the render.

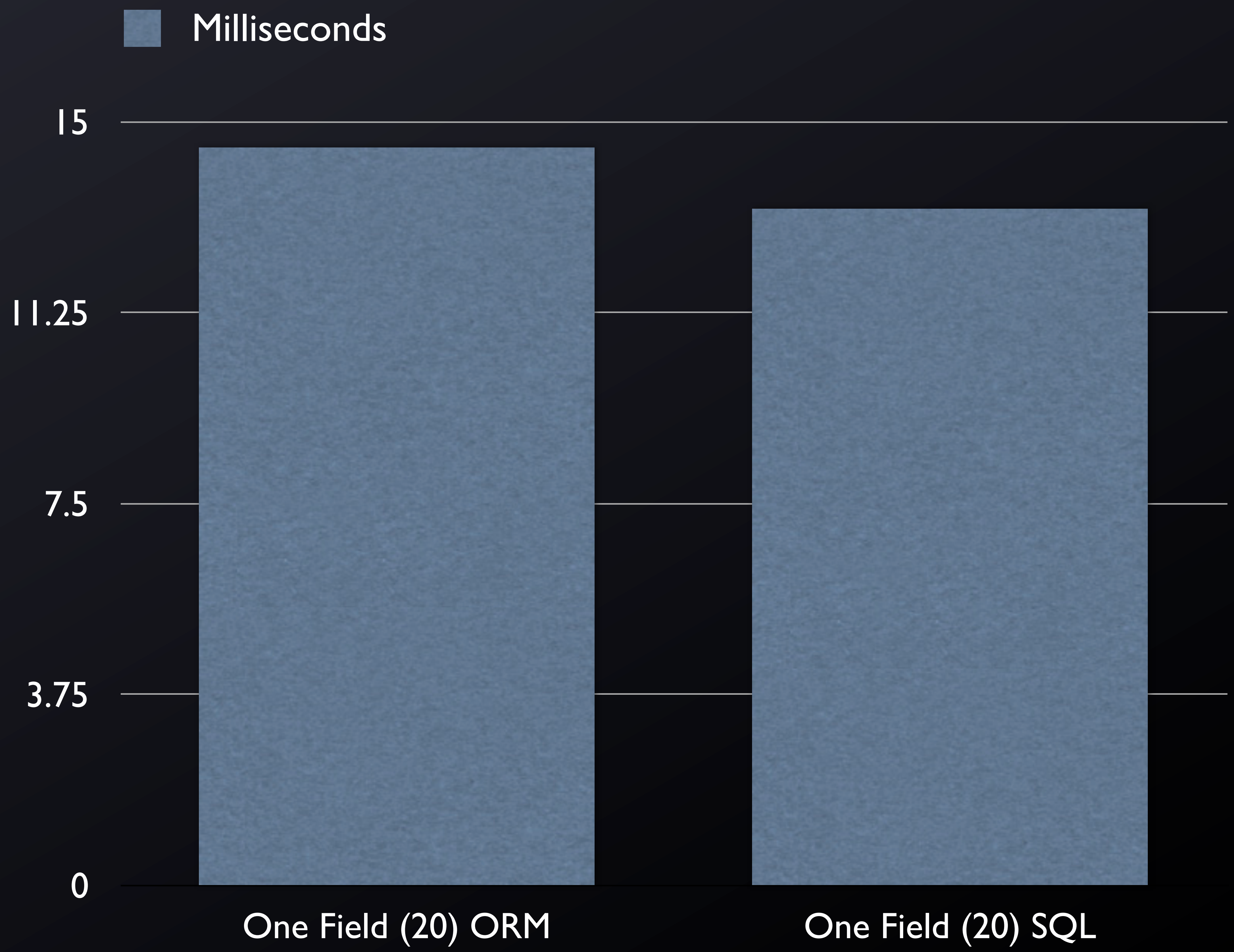
Test 3: Template rendering.



Test 4: Raw SQL vs ORM

- Use raw SQL (cursor.execute) to retrieve data instead of the ORM.
- 20 rows, one field.

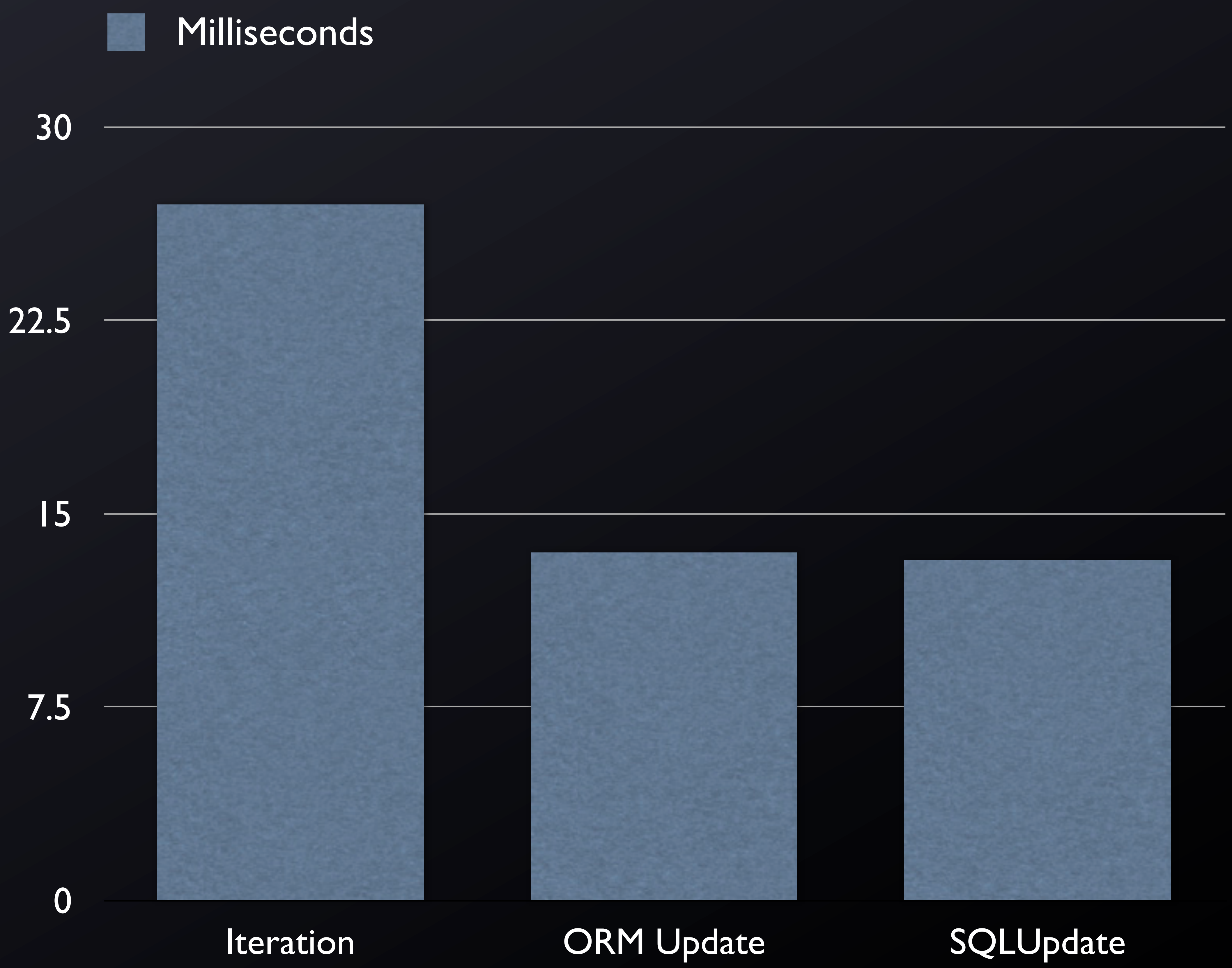
Test 4: Raw SQL vs ORM



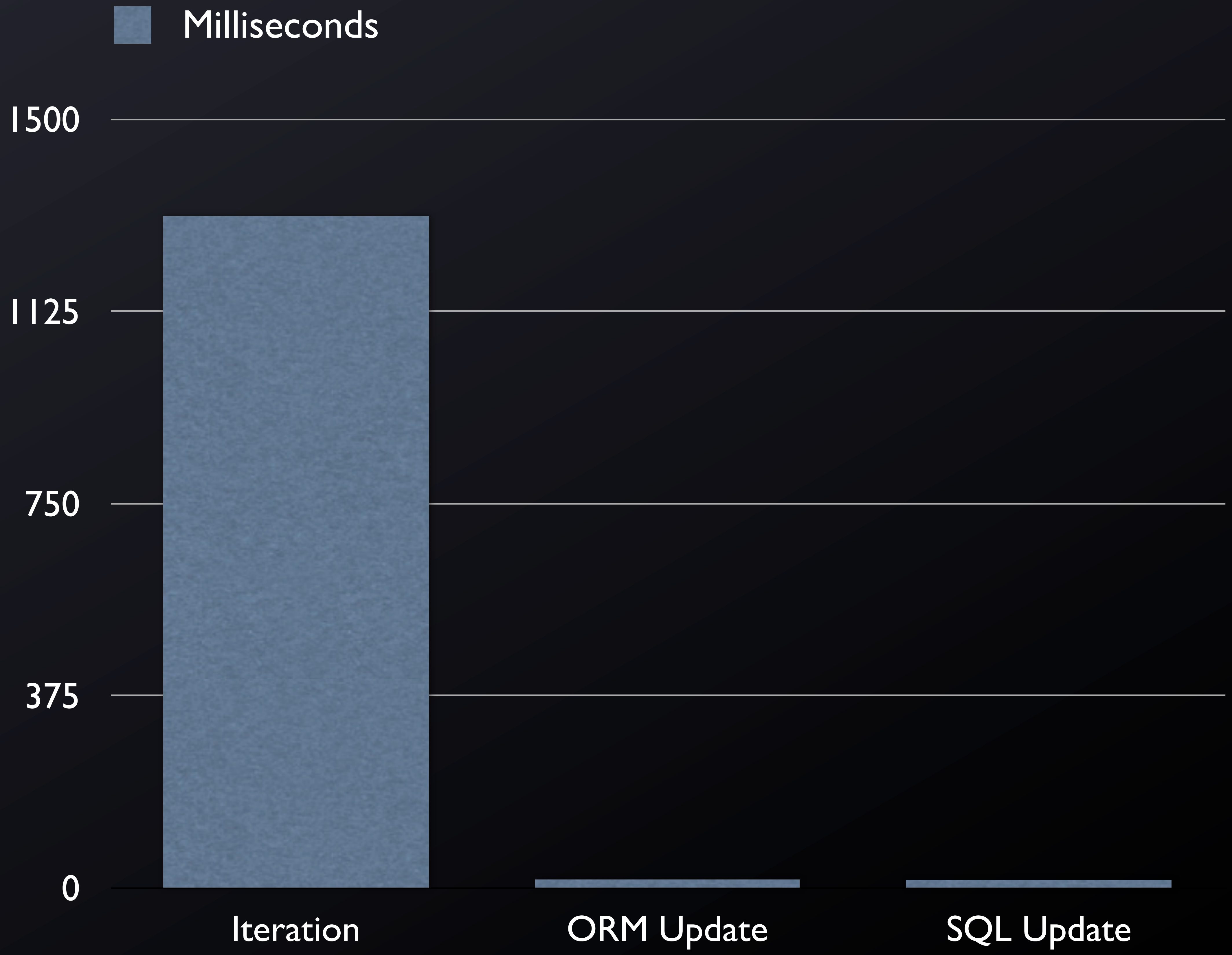
Test 5: Update 10 objects

- ORM using iteration.
  - Don’t do this.
- ORM using QuerySet.update
- Raw SQL using cursor.execute

Test 5: Update 10 objects



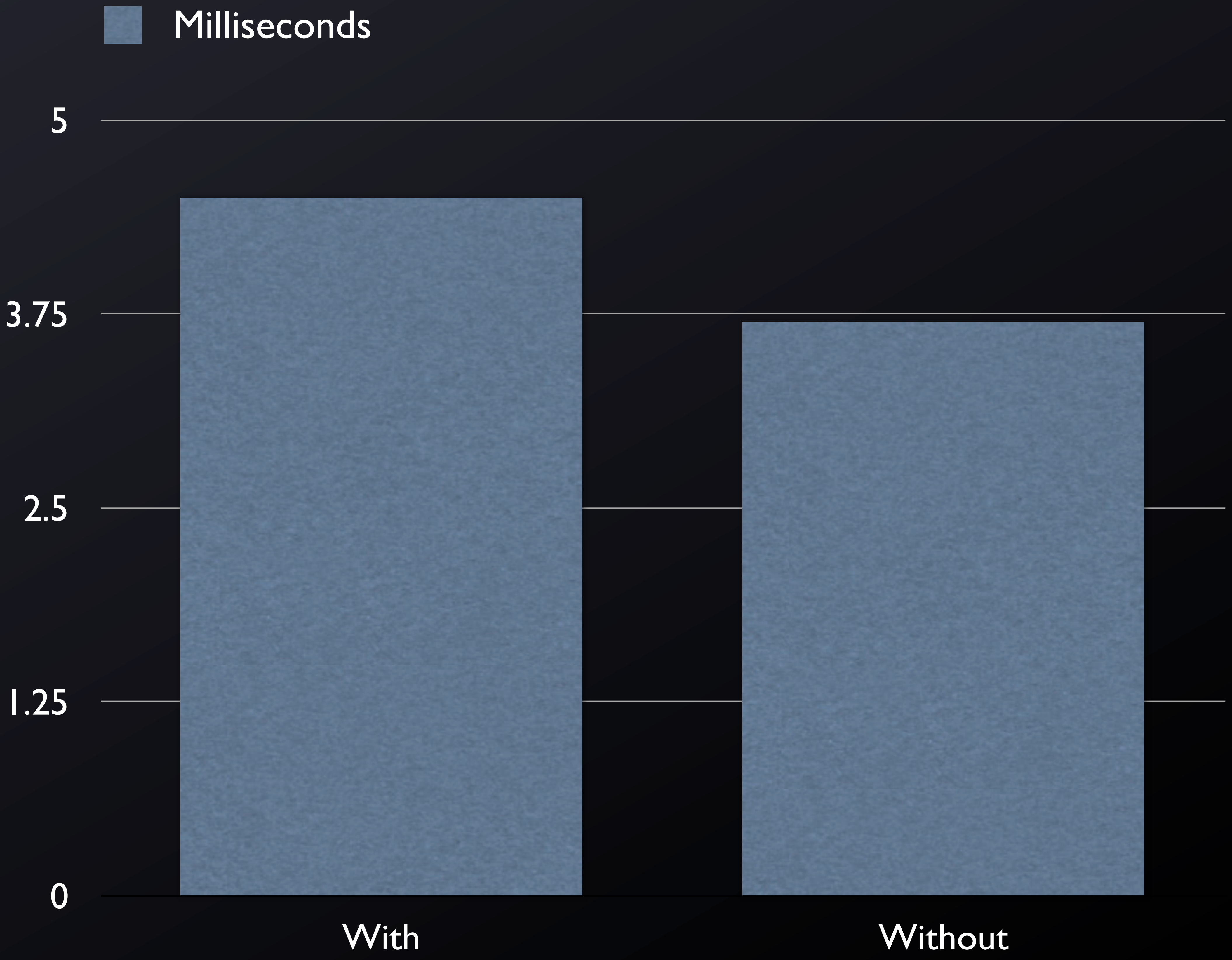
Test 6: Update 1000 objects



Test 7: Middleware Stack.

- Run empty requests with and without the standard middleware stack.

Test 7: Middleware Stack.



So, what do we know?

- Django’s basic request loop is plenty fast.
- Request/response cycles to the database generally swamp everything else.
- Always do bulk and batch operations without having to retrieve each model individually.
- The ORM’s performance isn’t that bad.

Don’t Hoard.

Don’t use components you don’t need.

- If you only need one (1) feature, just implement that one feature?
  - Do you really need an entire REST library, or just a JSON parser?
- Be aware of per-request overhead.
  - Middleware should be your last resort.

But…

- Be aware that components often have hidden benefits.
- Correct implementation of weird protocols, common security hole resistance, etc.

Caching.

There are only two hard things…

- There are only two hard things in computer science:
  - Naming things.
  - Cache invalidation.
  - Off-by-one errors.

PGX
POSTGRESQL
EXPERTS, INC.

Much caching. So complex.

- Front-end caching (nginx, Varnish).
- Template-render caching (whole page, fragments).
- Intermediate result processing (query sets, results of calculations).
- Database-level caching (materialized views, denormalized persistent tables).

First, measure.

- Don’t just throw everything at the wall and see what sticks.
- Caches *will* be inconsistent and invalid.
  - Find ways to allow for it, rather than building impossible-to-maintain invalidation architecture.

Start low, work up.

- Start with data-level caching, and work up from there.
- Easier to understand (generally), easier to come up with good invalidation models (almost always).

There's always an exception.

- Highly content-focused sites.
  - CMS-type publication sites.
- Focus on template-level rendering and full-page caching.
  - Accept a very flexible invalidation model.

Thundering herd problem.

- An invalidated cache results in every new request trying to rebuild the cache.
- **Always** separate delivery and cache rebuilding.
- Try to allow for return of stale results rather than rebuilding on the fly.

Template caching.

- Template rendering time is proportional to the number of variables and the number of files.
- Complex, deep templates can take time to render.
- But “time” is in milliseconds, not in days.

Keep calm and do time-based rebuilds.

- Do not become obsessive about only re-rendering when absolutely required.
- If a template requires 400ms to re-render…
- … rendering it once a minute is no big deal.

Tips ‘n’ Tricks

The (Very) Front End

Front-end servers.

- Everyone obsesses about them.
- They don’t matter.
- No, really, they don’t matter.
- Once you’ve fixed everything else, worry about that.
- You’ve never fixed everything else.

OK, OK, fine.

- ngnix.
- uWSGI.
  - wsgi (rather than http) protocol.
- You now have a slide you can show your boss.
  - It’s from an expert!

Processes vs Threads

- No clear guidelines for how to configure.
- Rule of thumb:
  - Processes = CPU execution units.
  - Threads = 2-4, more for high-blocking applications.

The speed of light.

- The public Internet is far slower than your code.
  - If it’s not, well, fix that!
- The link between your application and the user’s browser is, by far, the slowest part of your application.

Party like it's 1999.

- Most of the time processing a request is after the first byte is received by the client.
- Keeping web pages small, clean and light will make more difference than almost anything else.
- Use HTML Boilerplate, Twitter Bootstrap? Trim, trim, trim to what you need.

Avoid “site pestering.”

- Avoid a large flurry of JavaScript requests back to the server from the initial page.
- Each one has the full round-trip latency of the first request.
- Reduce the amount of data you need to get, and batch the calls together.

The browser is your frienemy.

- Always set sensible cache control headers on your content.
- How often do you change that checkmark graphic, anyway?
- Modern browsers are very aggressive about caching: take advantage of it!

Use a CDN for static content.

- Serving common static content is a terrible use of your bandwidth.
- CDNs can significantly improve your overall page-load time.
- Don’t use for dynamic content: propagation rates are just too slow.
- Use a caching CDN?

Things that look good, but aren't.

- eTag
  - OK for precomputed content, bad for dynamic content.
- Template fragment caching
  - Good for large, complex segments of a template.
  - Silly for small sections.

Use a front-end cache.

- ngnix, Varnish — or both!
- Use JavaScript and HTML5 local storage for trivial customizations.
  - Cookies defeat caching!

DNS Servers.

- A surprisingly large contributor to page-load time.
- Use a specialist DNS service.
  - EasyDNS is fast and cheap.
- Especially important if you have multiple subdomains on a single page.

The View Layer

The view code.

```
c = Customer.objects.get(id=customer_id)
o = Orders.objects.filter(id=customer_id, order_id=order_id)
t = 0
for line in o.line_items:
   t += line.tax
s = o.shipping
if s > 0 then:
   # blah, blah blah.
# Load everything into context!
```

The template.

```
{% cache 500 name %}
Hi, {{ c.first_name }}!
{% endcache %}
```

Template-first design.

- Let the template drive your data acquisition.
- Don’t do ORM operations unless the particular template expansion actually needs it.
- Put QuerySets and callables, rather than evaluated data, in the template contexts.

Cache everything.

- Django has extensive template caching facilities. Use them.
  - Cache full pages if you can.
  - Cache (big, expensive) fragments if you can’t.
- Always use a memory-based cache.
  - memcached, Redis.

Cache results.

- QuerySets are serializable!
- Store them in an in-memory store.
    - Redis is great for basic queues, etc.
    - memcached if you only need a flat store.

Consider full prerendering.

- Build entire page and cache on disk.
- Let the web server serve it directly.
  - Standard ngnix config will do this for you with appropriate path settings.
- Or let ngnix or Varnish do the caching.

The “Hello, Bob” problem.

- A large static page with a very small amount of customized content.
- Prerender the entire page, then use Javascript callbacks for the customized part.
- Make one call, parse out the result.

Returning large files.

- Use X-Accel-Redirect or equivalent.
- Never hand the large file directly back through Django.
  - Never. Write it to disk if you have to.
- Especially important if using back-end worker servers like gunicorn, uWSGI.

Middleware.

- Keep the middleware stack under control.
- Do you really need this to run on every request?
- Don’t use TransactionMiddleware…
  - Use atomic(). All the cool kids are.

Defer everything.

- Do not run asynchronous tasks in your view functions.
  - Send mail, fetch other sites, etc.
- Queue those for later processing.
- Queue synchronous tasks if they are long-running.
  - Generate a “best-guess” result first.

The Model Layer

Model-building.

- Keep models simple and focused.
  - The ORM is O(N) on number of fields.
- Don’t be afraid of foreign keys.
- Do not have frequently-updated singleton rows.

Fast vs slow data.

- A single logical object can have both “fast” and “slow” sections:
  - Username vs last access time.
- Separate these into different tables.
- Avoids a large class of foreign key locking issues.

Result prefetching.

- QuerySets will fetch the *entire* database result set the first time they need a *single* row.
  - … at least using psycopg2.
- Make sure database result sets are small.
- Do not rely on QuerySet slicing.

QuerySet caching.

- QuerySets retain their iterated-over results until released.
- This can be a significant memory sink.
- Release QuerySets once you are done with them.
  - But if can you store the results for future use? Do it.

Using transactions.

- Keep transactions short and to the point.
- Like any good writing, start as late as you can, finish as early as you can.
- Never wait for an asynchronous event with an open transaction.

More friendly advice.

- Do not iterate over large QuerySets…
  - … especially while doing updates back to the database.
- Do joins in the database, not in Python.
  - Don’t be afraid of writing custom SQL if that’s what it takes.

The Database

Databases are your friend.

- The database as such is rarely the bottleneck.
- Round-trips to the database, however, are.
- Aggregate as much as possible into single operations.

Do not do this.

- Store sessions in the database.
- Store your task queue in the database.
    - Especially if your task queue runner polls the database.
    - (I’m looking at you, Celery.)
- Store high-volume data in an otherwise-transactional database (clickstream, etc.)

Django 1.6 Persistent Connections.

- Use them.
- Connection opening overhead is significant.
- Does not always obviate the need for pgbouncer.
- Remember that the database probably can’t handle every connection being active at the same time.

Database load balancing.

- If using PostgreSQL, use streaming replication.
- Ideally designed for web-type read vs write loads.
- How to route requests to the right servers?

Django database routing.

- Use Django database routing to distribute writes to the master, reads to the secondaries.
- If more than one secondary, use pgPool II or a TCP/IP-based load balancer (HAProxy).
- Remember replication lag issues.

Summary!

I thought he’d never stop.

- Django can handle massive, server-melting loads.
- There’s no one trick; it’s a collection of small things and avoiding pitfalls.
- Focus on keeping your app lean.
  - You can hardware your way out of (almost) all the rest.

Thank you!

Questions?

@xof

thebuild.com

pgexperts.com

PGX
POSTGRESQL
EXPERTS, INC.