Three Years of Worst Practice

or

Unbreaking Your Django Application

PyCon Argentina 2012

Christophe Pettus <cpettus@pgexperts.com> **@xof**
PostgreSQL Experts, Inc.

Welcome!

- Time for the quick skills check:
  - Time with Django?
  - Database expertise?

On the menu.

- Model Design.
- Efficient use of the ORM.
- Transactions and Locking.
- External Tools and Techniques.
- Database-Level Debugging.

Batteries included.

- Nothing to buy, nothing to download.
- Stop me to ask questions.
- I'll get some things wrong…
  - So call me on it!

Welcome to my world.

- Lots of clients using Django.
- Lots of clients with minimal database experience.
    - But very skilled Python programmers.
- Lots of emergency calls about failing applications…
    - … applications that worked great in test.

The good.

- Django can get a plausible application going very fast.
  - The Django admin is amazingly useful for quick prototyping.
- Development generally happens on laptops or virtual servers with small datasets.
  - … and then production happens.

The bad.

- The Django ORM does not encourage scalable programming techniques.
  - It is not unique in this regard, of course.
  - Many Django programmers have never built very large systems before.
  - Naïve programming techniques can result in some really, really bad code.

The ugly.

- Major public-facing site.
  - Old site in proprietary everything, replaced by Django and Postgres.
  - Worked great in test.
  - On launch day…
    - 300,000 users…
    - … simultaneously.

What happened?

- Horribly slow performance (> 2 minutes to load a single page).
- 25% of requests died with deadlocks.
- Nightly batch update process required 26 hours to run.
- Customers were… somewhat dissatisfied with the transition to the new site.

The stages.

- Denial: "It's just launch jitters."
- Anger: "Why is POSTGRES SO FSCKING SLOW?"
- Bargaining: "All we have to do is make this little tweak here, and here, and here…"
- Depression: "We're all going to die. And lose our jobs. And our customers will feast upon our corpses. Worse, we need a consultant."

Acceptance.

- Models created locking problems.
- Code made terrible use of the ORM.
- Design of the templates defeated caching.
- Transaction model was not thought through.
- The database was a black box that no one there could really diagnose.

The goal here is to…

- Understand how we get into these bad situations.
- Avoid the problems at the start of the process, when they're easy to fix.
- Laugh painfully in recognition of these problems in our own code. (Mine too!)
- And talk myself out of a job.

Model Design

The model and you!

- The classes which Django maps into database tables.
- Includes basic index specifications.
- Includes foreign-key relationships.
- Includes default values and constraints…
  - … none of which passed on to the database for enforcement. Doh.

Let's talk about…

- Good model design.
- Foreign key relationships.
  - Many-to-many.
- Indexing strategy.
- Migrations and how to avoid crushing your application right in the middle of the highest traffic period.
  - Not that you'd do that, of course.

Advice from the field.

Don’t use NULL.

- NULL does not do what you expect.
  - No matter what you expect.
  - SUM(1+NULL) is not 1 + NULL
  - It’s tempting to use as a flag value. Resist.
  - It’s not as consistent as None in Python.

Separate derived data.

- Data that can be from other sources should be in its own model class.
- Allows recalculation without locking the main object.
- Allows implementation as a view or a materialized view.
- At minimum, wrap in a getter to hide the underlying implementation.

Normalize.

- Don’t duplicate fields in derived or related objects.
- That’s what foreign keys are for.
- Get the data model right, and then get the templates to work with the model.

Primary Keys

- Django will create a primary key for you if you don't specify one.
- Sometimes that's the right idea.
- Sometimes the data has a natural primary key already.
  - Use that if it does.
  - Admittedly, Django's primary key support is not everything it could be.

Separate Application Zones

- Data frequently is coming from different applications.
- Don’t smash them all into one object.
    - #1 cause of deadlocks.
- In the extreme case, you can reduce the main object to just an identity table.
    - Or at least just the most static data.

Foreign Keys

- Good news, bad news.
- Keep rows small and focused.
  - Better locking characteristics.
- Django has select_related() … perfect for this situation.
  - Ever seen it used?
  - I didn't even know it existed!

The Best Feature You'll Never Use.

- Django 1.3 introduced foreign key deletion options.
- Do not use them.
- on_delete=DO_NOTHING
- Use a real database foreign key constraint.
    - Use South migrations to set them.

So, what do to?

- If you almost always return the parent along with the related object, consider object inheritance instead.
  - That does do a nice join operation.
  - Don't create an insane class graph, though.
- If you frequently use the related object without reference to the parent object, use a foreign key.

Indexing.

- Django makes it really easy to create indexes.
- Really, really easy.
- `db_index=True` and away you go!
- What could possibly go wrong?

This could.

- Database size on disk: 157GB.
- Database as a pg_dump (excludes bloat and indexes): 9GB.
- Total bloat: 200MB.
- Percentage of indexes which had ever been used since the server was brought up three months prior: 2%.

What is a good index?

- Highly selective.
    - Greatly reduces the number of candidate rows when used.
- Used very regularly by the application code.
- Or required to enforce constraints.

What is a bad index?

- Every other index.
- Bad selectivity.
- Rarely used.
- Expense to maintain compared to the acceleration of queries that use it.
- An unselective index is more expensive than a sequential scan.

Indexing Strategy

- Start with none (except those required by keys or constraints).
  - No, none. Zero. Zip.
- Analyze the queries and look for ones which:
  - Query a constant column or set of columns AND
  - Select <10%…25% of the rows in the target table.

Exceptions to the rules.

- There are always some.
- SSDs have different seek characteristics from spinning disks.
- So do highly virtualized storage mechanisms like Amazon EBS.
- 50%-75% selectivity may be in a win in those cases.
- Test, test, test. Don’t assume anything.

Multi-column indexes.

- Two choices:
  - A single index that includes both. (You'll need to do this outside of Django in 1.3, but that's OK: we're all grown-ups, right?)
  - Two indexes, one for each column.
- The size will be roughly equivalent between the two.
- So, how do choose?

A single composite index…

- … will be somewhat smaller than two indexes.
- … will definitely be faster to update.
- … will accelerate a more restricted range of queries.
- An index on (A, B) does nothing for queries just using B.
- But on the queries it does accelerate, it will be faster.

Two indexes…

- … will be somewhat larger.
- … will be definitely be slower to update.
- … will help on queries on both columns, but not as much as a single composite index.
- … will help accelerate queries on either of the columns.
- … can be expressed directly in Django.
- So, test already.

About that testing thing.

- Your database has much to tell you. Let it do so.
- `pg_stat_activity` is a wealth of useful information…
  - … like which indexes are actually being used.
- If an index is not being used (or not being used very often), drop it like a hot potato…
  - … or, if you are flabbergasted that it is not being used, find out why.

Migrations

- Django doesn't have any migration tools out of the box.
  - South is pretty nice.
- So, we have a table that needs a new column.
- Either manually, or using South, or whatever, we issue the fatefully command:

```
ALTER TABLE applabel_ginormoustable
    ADD COLUMN hot_new_flag BOOLEAN
        NOT NULL DEFAULT FALSE;
```

What could go wrong?

- Well, nothing, but…
- That table had 65,000,000 rows in it…
- And was a critical, constantly-used table…
- On very slow I/O.
- And that ALTER had to rewrite the entire table.
    - Which meant taking an exclusive lock.

Eight hours later, the site was back up.

Amazingly, no one lost their jobs.

- Adding a column with a DEFAULT rewrites the table in Postgres.
- Adding a NULLable column with no default *generally* does not rewrite the table.
- No migration tool (that I know of) does anything particularly helpful in this regard.
    - Open-source street cred opportunity!

How to handle this?

```
-- First alternative: Works during table writes.

ALTER TABLE x ADD COLUMN b BOOLEAN NULL;
  -- Very fast.
UPDATE x SET b = FALSE;
  -- Rewrites table but does not take AccessExclusive.
ALTER TABLE x ALTER COLUMN b SET NOT NULL;
  -- Takes AccessExclusive, but just scans the table.
```

How to handle this?

```
-- Second alternative: Read-only table.

BEGIN;
CREATE TABLE x_new AS SELECT x.*, FALSE AS b FROM x;
  -- Duplicate table, adding new column.
ALTER TABLE x_new ALTER COLUMN b SET NOT NULL;
   -- Scans column.
DROP TABLE x;
ALTER TABLE x_new RENAME TO x;
COMMIT;
```

How to handle this?

- Or just wait until 3am.
- None of these solutions are without their flaws.
- In the particular case, compounded by the terrible I/O speed of their hosting environment.
- Make sure you test everything in a realistic QA environment before deployment.

The ORM and Its Discontents

Let us now praise famous ORMs.

- The Django ORM:
  - Allows for very Python-esque code.
  - Handles a wide range of very common database interactions elegantly.
  - Can generate a remarkable range of queries.

But.

- There are some things at which it does not excel.
- The farther one moves from the load-modify-store paradigm, the wilder the scenery gets.
- Knowing when to step outside the ORM is important.
- The good news is that Django has great raw SQL facilities that compliment the ORM.

The basic rules.

1. Do not iterate over QuerySets.
2. If you think you have to iterate over a QuerySet, see rule #1.
3. If you are absolutely, certainly, 100% positive that the only possible solution to your problem is iterating over a QuerySet, see rule #2.

Real code.

```
qs = Orders.objects.all()
   # There are about 2,500,000 rows in “orders”.

for order in qs:
   order.age_in_days += 1
   order.save()

# Astonishingly, this code snippet did not perform
# to the standards the client expected.  They wanted
# to know why Postgres was so slow.
```

Iteration: the awful truth

- Defeats Django's lazy-evaluation mechanism by dragging everything into memory.
- Hauls data across the wire from the database just to process it locally.
- Does filtration or summarization in the application that is much more efficiently done in the database.
- Databases have decades of experience in this kind of munging… respect your elders!

Alternatives

- QuerySet.update()
- cursor.execute("UPDATE app_orders ...")
- Stored procedures.
- The list goes on.

But, what about…

- Complex filtration that cannot be done in the database (especially on small result sets).
- Pushing data to another application in an ELT-style operation.
- Legitimately huge result sets (mass emailing).
  - All fair, but examine if there is a way to keep the data out of the Django application.
  - Because…

How much many objects are in memory at point A?

```
qs = Orders.objects.all()
   # There are about 2,500,000 rows in “orders”.

for order in qs:
   order.age_in_days += 1 # POINT A
   order.save()
```

All 2,500,000.

Wait, what?

- Django does lazy evaluation… everyone tells me so!
- The Django code carefully asks for a slice of 100 objects…
- which trickles down through lots of really convoluted Python to psycopg2…
- which dutifully asks for 100 rows from Postgres…
- which sends all 2,500,000 over the wire.

For the want of a named cursor…

- The protocol between the Postgres client and server only does partial sends when using named cursors.
- psycopg2 fully supports named cursors.
- Django doesn't use them.
- So, the first time you ask for any object in a QuerySet, you get all of them.
- This is a very good reason not to ask for large result sets.

OK, what about LIMIT?

- In Django, qs[:x] adds a LIMIT clause to the SQL.
- Remember that LIMIT isn't really useful without a sort.
- And that the database has to sort the entire result set before applying the LIMIT.
- An index on the sort key is a superb idea.

The perils of OFFSET.

- qs[x:y] does an OFFSET x LIMIT y-x.
- OFFSET has to consider and toss every object from 1 to x-1.
- Very large OFFSETs are extremely inefficient.
- Much better to use queries on indexed columns instead.
- For pagination, consider strongly limiting how deep it can go.

IN.

- The most abused query operation in Django.
- It looks so innocent, just sitting there, doesn't it?
- `Thing.objects.filter(id__in=my_little_list)`

Not so innocent now.

```
SELECT ""stuff_thing"".""id"", ""stuff_thing"".""thing1"", ""stuff_thing"".""thing2"" FROM ""stuff_thing"" WHERE
""stuff_thing"".""id"" IN (3702, 3705, 3708, 3711, 3714, 3717, 3720, 3723, 3726, 3729, 3732, 3735, 3738, 3741, 3744, 3747, 3750,
3753, 3756, 3759, 3762, 3765, 3768, 3771, 3774, 3777, 3780, 3783, 3786, 3789, 3792, 3795, 3798, 3801, 3804, 3807, 3810, 3813, 3816,
3819, 3822, 3825, 3828, 3831, 3834, 3837, 3840, 3843, 3846, 3849, 3852, 3855, 3858, 3861, 3864, 3867, 3870, 3873, 3876, 3879, 3882,
3885, 3888, 3891, 3894, 3897, 3900, 3903, 3906, 3909, 3912, 3915, 3918, 3921, 3924, 3927, 3930, 3933, 3936, 3939, 3942, 3945, 3948,
3951, 3954, 3957, 3960, 3963, 3966, 3969, 3972, 3975, 3978, 3981, 3984, 3987, 3990, 3993, 3996, 3999, 4002, 4005, 4008, 4011, 4014,
4017, 4020, 4023, 4026, 4029, 4032, 4035, 4038, 4041, 4044, 4047, 4050, 4053, 4056, 4059, 4062, 4065, 4068, 4071, 4074, 4077, 4080,
4083, 4086, 4089, 4092, 4095, 4098, 4101, 4104, 4107, 4110, 4113, 4116, 4119, 4122, 4125, 4128, 4131, 4134, 4137, 4140, 4143, 4146,
4149, 4152, 4155, 4158, 4161, 4164, 4167, 4170, 4173, 4176, 4179, 4182, 4185, 4188, 4191, 4194, 4197, 4200, 4203, 4206, 4209, 4212,
4215, 4218, 4221, 4224, 4227, 4230, 4233, 4236, 4239, 4242, 4245, 4248, 4251, 4254, 4257, 4260, 4263, 4266, 4269, 4272, 4275, 4278,
4281, 4284, 4287, 4290, 4293, 4296, 4299, 4302, 4305, 4308, 4311, 4314, 4317, 4320, 4323, 4326, 4329, 4332, 4335, 4338, 4341, 4344,
4347, 4350, 4353, 4356, 4359, 4362, 4365, 4368, 4371, 4374, 4377, 4380, 4383, 4386, 4389, 4392, 4395, 4398, 4401, 4404, 4407, 4410,
4413, 4416, 4419, 4422, 4425, 4428, 4431, 4434, 4437, 4440, 4443, 4446, 4449, 4452, 4455, 4458, 4461, 4464, 4467, 4470, 4473, 4476,
4479, 4482, 4485, 4488, 4491, 4494, 4497, 4500, 4503, 4506, 4509, 4512, 4515, 4518, 4521, 4524, 4527, 4530, 4533, 4536, 4539, 4542,
4545, 4548, 4551, 4554, 4557, 4560, 4563, 4566, 4569, 4572, 4575, 4578, 4581, 4584, 4587, 4590, 4593, 4596, 4599, 4602, 4605, 4608,
4611, 4614, 4617, 4620, 4623, 4626, 4629, 4632, 4635, 4638, 4641, 4644, 4647, 4650, 4653, 4656, 4659, 4662, 4665, 4668, 4671, 4674,
4677, 4680, 4683, 4686, 4689, 4692, 4695, 4698, 4701, 4704, 4707, 4710, 4713, 4716, 4719, 4722, 4725, 4728, 4731, 4734, 4737, 4740,
4743, 4746, 4749, 4752, 4755, 4758, 4761, 4764, 4767, 4770, 4773, 4776, 4779, 4782, 4785, 4788, 4791, 4794, 4797, 4800, 4803, 4806,
4809, 4812, 4815, 4818, 4821, 4824, 4827, 4830, 4833, 4836, 4839, 4842, 4845, 4848, 4851, 4854, 4857, 4860, 4863, 4866, 4869, 4872,
4875, 4878, 4881, 4884, 4887, 4890, 4893, 4896, 4899, 4902, 4905, 4908, 4911, 4914, 4917, 4920, 4923, 4926, 4929, 4932, 4935, 4938,
4941, 4944, 4947, 4950, 4953, 4956, 4959, 4962, 4965, 4968, 4971, 4974, 4977, 4980, 4983, 4986, 4989, 4992, 4995, 4998, 5001, 5004,
5007, 5010, 5013, 5016, 5019, 5022, 5025, 5028, 5031, 5034, 5037, 5040, 5043, 5046, 5049, 5052, 5055, 5058, 5061, 5064, 5067, 5070,
5073, 5076, 5079, 5082, 5085, 5088, 5091, 5094, 5097, 5100, 5103, 5106, 5109, 5112, 5115, 5118, 5121, 5124, 5127, 5130, 5133, 5136,
5139, 5142, 5145, 5148, 5151, 5154, 5157, 5160, 5163, 5166, 5169, 5172, 5175, 5178, 5181, 5184, 5187, 5190, 5193, 5196, 5199, 5202,
5205, 5208, 5211, 5214, 5217, 5220, 5223, 5226, 5229, 5232, 5235, 5238, 5241, 5244, 5247, 5250, 5253, 5256, 5259, 5262, 5265, 5268,
5271, 5274, 5277, 5280, 5283, 5286, 5289, 5292, 5295, 5298, 5301, 5304, 5307, 5310, 5313, 5316, 5319, 5322, 5325, 5328, 5331, 5334,
5337, 5340, 5343, 5346, 5349, 5352, 5355, 5358, 5361, 5364, 5367, 5370, 5373, 5376, 5379, 5382, 5385, 5388, 5391, 5394, 5397, 5400,
5403, 5406, 5409, 5412, 5415, 5418, 5421, 5424, 5427, 5430, 5433, 5436, 5439, 5442, 5445, 5448, 5451, 5454, 5457, 5460, 5463, 5466,
5469, 5472, 5475, 5478, 5481, 5484, 5487, 5490, 5493, 5496, 5499, 5502, 5505, 5508, 5511, 5514, 5517, 5520, 5523, 5526, 5529, 5532,
5535, 5538, 5541, 5544, 5547, 5550, 5553, 5556, 5559, 5562, 5565, 5568, 5571, 5574, 5577, 5580, 5583, 5586, 5589, 5592, 5595, 5598,
5601, 5604, 5607, 5610, 5613, 5616, 5619, 5622, 5625, 5628, 5631, 5634, 5637, 5640, 5643, 5646, 5649, 5652, 5655, 5658, 5661, 5664,
5667, 5670, 5673, 5676, 5679, 5682, 5685, 5688, 5691, 5694, 5697, 5700, 5703, 5706, 5709, 5712, 5715, 5718, 5721, 5724, 5727, 5730,
5733, 5736, 5739, 5742, 5745, 5748, 5751, 5754, 5757, 5760, 5763, 5766, 5769, 5772, 5775, 5778, 5781, 5784, 5787, 5790, 5793, 5796,
5799, 5802, 5805, 5808, 5811, 5814, 5817, 5820, 5823, 5826, 5829, 5832, 5835, 5838, 5841, 5844, 5847, 5850, 5853, 5856, 5859, 5862,
5865, 5868, 5871, 5874, 5877, 5880, 5883, 5886, 5889, 5892, 5895, 5898, 5901, 5904, 5907, 5910, 5913, 5916, 5919, 5922, 5925, 5928,
```

Large INs are a mess.

- Very expensive for the database to parse.
- Very expensive for the database to execute.
- If there are potentially more than 10-15 items in the list, rework the IN as a JOIN against whatever the source of the keys is.

QuerySet objects.

- Once evaluated, a QuerySet holds on to its result set.
  - Reuse the same result set whenever possible!
- Remember that each filter you apply to a QuerySet creates a new query set.
  - That time can add up.
  - It can be faster to drop to raw SQL for super-complex queries.

Model Objects are Heavy.

- Model objects take a long time to create.
- If you are going to use more than six objects in one view function, switch to raw SQL.
- Why six? It could be five. It could be ten. Pick a number and stick to it.
- It is definitely not 200.

Workload separation.

- The question was, “How many of each widget has been bought to date”?
- The answer was, “Your site has ground to a halt each time the vendor logs in and tries to find out this information.”

What it looked like.

```
SELECT COUNT(*) FROM orders WHERE widget_id=3001;
SELECT COUNT(*) FORM orders WHERE widget_id=3013;
SELECT COUNT(*) FROM orders WHERE widget_id=3017;
SELECT COUNT(*) FORM orders WHERE widget_id=3022;
SELECT COUNT(*) FROM orders WHERE widget_id=3045;
SELECT COUNT(*) FORM orders WHERE widget_id=3056;
SELECT COUNT(*) FROM orders WHERE widget_id=3098;
SELECT COUNT(*) FORM orders WHERE widget_id=3104;
SELECT COUNT(*) FROM orders WHERE widget_id=3109;
SELECT COUNT(*) FORM orders WHERE widget_id=3117;
```

The code was, of course…

```
wgt = Widget.orders.filter(vendor=cur_vend)
total_sold = 0
for widget in wgt:
    total_sold +=
        Orders.objects.filter(widget=wgt.id).count()
```

Well, at least it wasn't an IN.

- Each vendor could run this query as much as they wanted…
- … on the same system that was taking orders.
- Problem 1: Not the best written code, ever.
- Problem 2: Doing reporting on the transactional system.

Separate reporting and transactions.

- First option: Daily dump/restore of the transactional system.
  - Cheap and cheerful.
  - One day lag.
  - Impractical as the database grows.
- Second option: A replication solution.

Learn to love the ORM.

- Eschew iteration: let the database do the data reduction part.
- Limit the size of query sets.
- Avoid problematic constructs, like IN and OFFSET.
- Don’t be afraid of using raw SQL.

Interlude: Celery.

Celery: negative calories!

- Very popular Python job queuing system.
- Integrates very nicely with Django
  - Used to require Django, in fact.
- It's very nicely done, but…
- No matter what…

Don’t use the database as the job queue!

- Celery polls the job queue.
- All the time.
- From every worker process.
- Generating hundreds to thousands of queries a minute.
- Can easily take 100% of the capacity of the database.
- So, don’t do that.

Transactions and Locking

Let's be honest with ourselves.

- Django's transaction management is a mess.
- It clearly "just grew" from humble origins.
- Even the terminology is utterly unclear.
- But there is hope!
- Let us don pith helmet and pick up our machete to hack our way through the underbrush.

Transactions out of the box.

```
address = Address(street_address="1112 E Broad St",
   city="Westfield", state="NJ", zip="07090")

address.save()
```

```
order = Order(customer_name="Gomez Addams",
   shipping_address=address)

order.save()
```

Wrong, wrong, wrong.

- BEGIN;
- INSERT INTO Address VALUES (...);
- COMMIT;

- BEGIN;
- INSERT INTO Order VALUES (...);
- COMMIT;

Default behavior.

- If you do nothing, the default behavior is to wrap each .save() in its own transaction.
- This is probably not what you want.
- If you are defining multi-object graphs, it is definitely not what you want.

What *do* we want?

- Read-only view functions don't bother creating a transaction at all.
- Read/write view functions run in their own transaction.
- COMMIT on success, ROLLBACK on failure.

Transaction Middleware

- Puts the database under "transaction management."
- Sets psycopg2 to open a transaction the first time the database is touched.
- Automatically commits the transaction on completion, rolls it back if an exception comes flying out of the view function.
- Equivalent of wrapping the view function with the `@commit_on_success` decorator.

Much closer!

- But read-only transactions still have a transaction defined around them.
- Can we get rid of it?
- Yes, we can get rid of the transaction by turning on autocommit.
  - Wait, what?

autocommit.
How does it work?

- Worst. Parameter. Name. Ever.
- autocommit turns off automatic commits.
- You want to run it on when running on Postgres…
  - … but you need to understand how it changes the standard model.

autocommit…

- … sets psycopg2 so that it does not automatically start a transaction at all.
- Gets rid of the transaction on read-only functions, but:
- Gets rid of the transaction on functions that write, too.
- So, we need to wrap writing functions in the `@commit_on_success` decorator.

IDLE IN TRANSACTION

- A Postgres session state, meaning the session has an open transaction but is not working.
- They should be extremely rare.
- This state is not good, since locks and other resources are being held open by the transaction.

Session state of MYSTERY

- Django application are notorious for leaving sessions open in Idle in Transaction.
- The path through the Django code which allows for this is extremely unclear.
- So, it is impossible, and yet it happens. ~~Summer~~ Autumn of Code project, anyone?
- Always manage your transactions: That greatly reduces the number of Idle in Transaction sessions observed.

Transaction cheat-sheet.

- Run with autocommit on.
- Wrap database-modifying code with `@commit_on_success`.
- Leave read-only code undecorated.
- Don’t forget to manually call `transaction.is_dirty()` if you modify the database outside of the ORM!

Or use xact()

- Alternative transaction decorator for PostgreSQL.
- Use it as part of a transaction recipe.
- https://github.com/Xof/xact

Locking: A Primer.

- Readers do not block readers to the same row (it would be kind of dopey if they did).
- Writers do not block readers; they create a new version of the row.
- Readers do not block writers; they read the old version of the row.
- Writers, however, do block writers.

Kinds of locks.

- There are a whole ton of different kinds of table locks.
- … which we are going to ignore.
- The only interesting lock for our purposes is the exclusive row lock.
- This is automatically acquired when a process updates or deletes a row.
- No other process can update or delete that row until the first process commits.

Transactions and locks.

- Locks are only held for the duration of a transaction.
- Once the transaction either commits or rolls back, the lock is released.
- So: Keep transactions as fast as possible.
- Long-running, database-modifying transactions are the source of most serious concurrency problems.

Avoiding lock conflicts.

- Modify tables in a fixed order.
- When updating multiple rows in a table, do so in a fixed order (by id or some other invariant).
- Avoid bulk-updates to a single table from multiple processes at the same time (UPDATE is intrinsically unordered).
- DDL changes take exclusive locks!

Foreign-key locks.

- Updating a dependent record (usually!) takes a share lock (as of 9.0) on the parent record.
  - However, it does not always do so, which can lead to deadlocks.
- Done to enforce relational integrity.
- This can be… surprising if you are not expecting it.

Interlude: Templates

Django view function style

- Collect the input data.
- Do the business logic.
- Gather the data necessary for the view template.
- Fill in the context.
- Return, rendering the template.

The problem.

- The view code really doesn't know what the template is going to draw.
- So it has to fill in a superset of the possible data.
- This can be extremely inefficient.
- There are two approaches to this.

The bad approach

```
c = RequestContext(request)
c[‘results’] = list(qs)
c[‘moresults’] = list(qs2)
c[‘ohyoumightneedthis’] = list(qs3)
c[‘ohdontforgetthat’] = list(qs4)
...
```

QuerySets are lazy!

- This is where the lazy evaluation of QuerySets is most useful.
- Don't run them unless you have to.
- Present the QuerySet itself to the template.
- For non-QuerySet data, pass in a callable that retrieves the required data.
    - This will only be called if needed, interacting nicely with Django's template caching.

Thank You!

Any Questions?