PostgreSQL Gotchas for App Developers

Day of the Programmer 2018
Jönköping, Sweden

Magnus Hagander
magnus@hagander.net

Magnus Hagander

- Redpill Linpro
  - Infrastructure services
  - Principal database consultant
- PostgreSQL
  - Core Team member
  - Committer
  - PostgreSQL Europe

I'm not an app dev

I'm not a full stack dev

But I work with them

Not an appdev

- Also dabble in some web dev
- Not particularly good at it
- Mainly *python+django*
- So there will be examples...

So what's a gotcha?

- Seemed like a good idea
  - Or at least a simple one
- Unintended or unknown consequences
- Can be done better

So, let's get started?

Some of these you already know

Connection pooler

Connection pooler

- OK, you know to use one, right?
- Extra important on PostgreSQL
- Designed to work with poolers

Pooler issues

- Too many pools
  - One per user/db combination!
- Too large pools
  - Almost nobody has too small!

Speaking of users

Superuser

- Don't *ever* use from app
- Not even for migrations
  - Use schema owner!
- Not just permissions override!

Trust authentication

- Just don't used it.

JSON

JSON

Please don't:

```
CREATE TABLE all_my_data (
 id int GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
 j jsonb
)
```

JSON

Instead do:

```
CREATE TABLE all_my_data (
 id int GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
 something int NOT NULL,
 somethingelse text,
 anotherone timestamptz,
 ...,
 ...
 actual_unstructured_data jsonb
)
```

Migrations

Migrations

- *Avoid* doing in application
  - At almost any cost
- Raw SQL implementation much more performant
  - Easier and safer too!

Migrations

```
for o in MyModel.objects.filter(
      created__lt=datetime.datetime(2018,1,1)):

   o.something = True
   o.modified = datetime.now()
   o.save()
```

Migrations

```
UPDATE mymodel
 SET
  something=true,
  modified=CURRENT_TIMESTAMP
 WHERE created < '2018-01-01'
```

Migrations

- Even for complex transformations
- Worthwhile to create temporary stored proc
  - Maybe even in app language?
- But usually, use a *DO-block*

SQL

SQL

Never use it!

SQL

You didn't buy that, eh?

CTEs

- Common Table Expressions
- "WITH" queries
- Non-recursive or recursive
- Neat way to structure code

CTEs

```
WITH w AS (
  SELECT a,b FROM t1 WHERE x
)
SELECT a,b,c
FROM w INNER JOIN t2 ON t2.x=w.a
```

CTEs

Optimization barrier!

- Optimizer doesn't see through CTE boundaries!
- Avoid using unless intentional
- Or recursive

Speaking of optimization

Representative data

- Test with representative data!
- Certainly not empty dataset
- But also avoid fabricated data
  - Unless you can reproduce patterns
- Copy of production is best
  - But may be bad for other reasons

Representative data

- Query optimizer uses statistics
  - Common values
  - Distributions
  - NULL fractions
  - etc...
- Same size can give different plan
- Especially for non-linear data

Generating data

```
SELECT * FROM
 generate_series(1,10000);

1
2
3
4
5
...
```

Generating data

```
SELECT * FROM
 generate_series(NOW(), NOW()+'1 week', '8 hours')

 2018-03-02 15:28:22.394444+01
 2018-03-02 23:28:22.394444+01
 2018-03-03 07:28:22.394444+01
 2018-03-03 15:28:22.394444+01
 2018-03-03 23:28:22.394444+01
...
```

Avoid ORM

ORM overselect

- ORM version of

```
SELECT * FROM
```

- Usually expanded across joins
- Fields unknown at query definition
  - So easier to get everything
- Disables some optimisations

ORM overselect

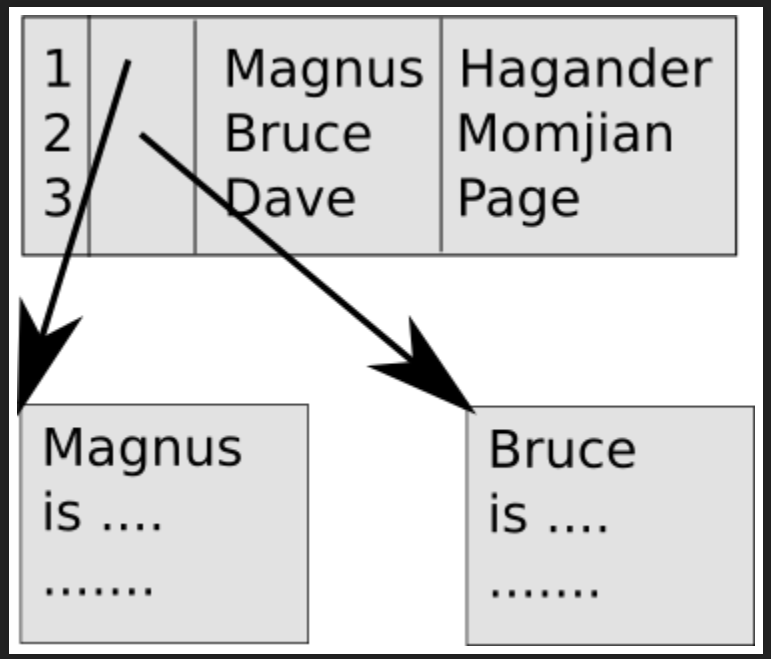
Simple table example

1	Magnus	Hagander
2	Bruce	Momjian
3	Dave	Page

Simple table example

1	Magnus is...	Magnus	Hagander
2	Bruce is	Bruce	Momjian
3	Dave is....	Dave	Page

Simple table example



ORM overselect

Including

```
Message.objects \
       .only('date', 'sender') \
       .filter(datefilter, hiddenstatus__isnull=True)
```

Excluding

```
Message.objects \
       .defer('bodytxt', 'cc', 'to') \
       .filter(datefilter, hiddenstatus__isnull=True)
```

ORM loops

- Looping in app causes many queries
- Can easily become exponential
- Queries have to be restarted
- Latency starts to matter!

Avoiding loops

- Force joins to happen early
- Often much cheaper
- Even if it results in more data

Forced joins

```
ConferenceRegistration.objects \
   .select_related('regtype') \
   .select_related('registrationwaitlistentry') \
   .filter(conference=conference) \
   .order_by('-created')
```

Avoiding loops

GROUP BY queries

- Often handled badly in ORMs
- In particular hierarchical data
- Collecting aggregates to the rescue!
- Native conversion
  - Good support in most drivers

Collecting aggregates

- List of conference session
- Each session has zero or more speakers
- Get the list to draw schedule

Collecting aggregates

```
SELECT s.id, s.title, s.starttime
      ,array_agg(spk.name ORDER BY spk.name) AS speakers
FROM sessions s
LEFT JOIN session_speakers ss ON ss.session=s.id
LEFT JOIN speakers spk ON spk.id=ss.speaker
WHERE s.conference_id=17
GROUP BY s.id
ORDER BY starttime
```

Collecting aggregates

```
>>> curs = connection.cursor()
>>> curs.execute("...")
>>> r = curs.fetchone()
>>> pprint(r)
(957,
 u'PostgreSQL Replication & Upgrades',
 datetime.datetime(2015, 10, 27, 9, 0),
 [u'Petr Jelinek', u'Simon Riggs'])
>>> type(r[3])
<type 'list'>
>>> type(r[3][0])
<type 'unicode'>
```

What about structure

- More advanced patterns than arrays
- Multiple keys
- Array-of-array is a PITA to deal with
  - Template languages unhappy
- JSON to the rescue!
  - Driver support is decent

Collecting aggregates

```
SELECT s.id, s.title, s.starttime,json_agg(
  json_build_object('name', fullname, 'company', company)
ORDER BY spk.name) AS speakers
FROM sessions s
LEFT JOIN session_speakers ss ON ss.session=s.id
LEFT JOIN speakers spk ON spk.id=ss.speaker
WHERE s.conference_id=17
GROUP BY s.id
ORDER BY starttime
```

Collecting aggregates

```
>>> r = curs.fetchone()
>>> pprint(r)
(957,
 u'PostgreSQL Replication & Upgrades',
 datetime.datetime(2015, 10, 27, 9, 0),
 [{u'company': u'ACME Global', u'name': u'Petr Jelinek'},
  {u'company': u'ACME Rockets', u'name': u'Simon Riggs'}])
>>> type(r[3])
<type 'list'>
>>> type(r[3][0])
<type 'dict'>
```

Avoiding ORM loops

Acting on multiple objects

- We always use bound parameters
  - Right?
- Such as:

```
curs.execute("... WHERE x=%(id)s", {
  'id': id,
})
```

- But what about multiple objects?

Multiple objects

```
curs.execute("... WHERE x IN (%(id1)s, %(id2)s)", {
 'id1': idlist[1],
 'id2': idlist[2],
}
```

- Does not scale
- Too many unique queries
- Annoys monitoring

Multiple objects

```
curs.execute("... WHERE x IN ({0})".format(
   ",".join(idlist),
))
```

- Did we not learn yet?!

Binding arrays

```
curs.execute("... WHERE x=ANY(%(idlist)s)", {
   'idlist': idlist,
})
```

- Also works for functions

```
curs.execute("SELECT myfunction(%(idlist)s)", {
   'idlist': idlist,
})
```

One final thing

Deadlocks

- A blocking lock is **not** a deadlock
- Much smaller problem than most people think
- But look for the right thing..

Thank you!

Magnus Hagander
magnus@hagander.net
@magnushagander
https://www.hagander.net/talks/