Quando Django incontra PostgreSQL!

Tamara Nocentini
2ndQuadrant

Leonardo Cecchi
2ndQuadrant

2ndQuadrant PostgreSQL

Cosa si aspettano gli sviluppatori web?

- Affidabilità
- Velocità
- Facilità di manutenzione

Secondo noi PostgreSQL è perfetto per tutto questo.

Ma allora perché i progetti Django vengono creati con SQLite?

- SQLite non ha bisogno di essere installato
- Python incorpora SQLite al suo interno
- È molto veloce cancellare un database SQLite e rifarlo

SQLite è pratico per sviluppare, e lascia spazio agli altri quando le esigenze cambiano.

PostgreSQL: accessi concorrenti

- Implementa il modello MVCC (Multi-Version Concurrency Control)
- Le letture non bloccano mai le scritture e viceversa

SQLite: accessi concorrenti

SQLite non ha un server, ma è semplicemente una libreria che lavora all'interno del server.

- Implementa il modello MVCC
- Ogni scrittura blocca completamente il database (**letture comprese**)
- Ogni transazione non ancora finita potrebbe generare problemi .

Concorrenza...

```
> create table persone
(nome text);
> insert into persone
values ('leonardo');
> insert into persone
values ('tamara');
> select * from persone;
leonardo
tamara

> begin;
> update persone
set nome='mario'
where nome='leonardo';
> [...]
```

```
> begin;
> select * from persone;
leonardo
tamara
> update persone
set nome='marco'
where nome='andrea';
Error: database is locked
>
```

Affidabilità

PostgreSQL, essendo un vero server, ha funzioni che SQLite non può avere:

- Archiviazione continua per il journal
- Replica fisica e logica
- Backup fisico e logico, possibile il Point In Time Recovery

SQLite: affidabilità

- Gestione del journal
- Utilizzato solo per la crash recovery
- Backup fisico e logico, no PiTR

PostgreSQL: coerenza dei dati trattati

- Modello strong-typing: una colonna intera non conterrà mai qualcos'altro
- Gestione integrità referenziale

SQLite: coerenza

- Modello dynamic-typing: i tipi vengono dichiarati ma non usati
- Gestione integrità referenziale

Hic sunt leones

```
n = models.NumeroPrimo(numero=1); n.save()
n = models.NumeroPrimo(numero=2); n.save()
n = models.NumeroPrimo(numero=3); n.save()

self.assertEqual(3, models.NumeroPrimo.objects.count())

# Un'altra applicazione potrebbe introdurre un errore...
with connection.cursor() as cursor:
    cursor.execute(
        "INSERT INTO prove_numeroprimo (numero) VALUES ('test')")

risultato = models.NumeroPrimo.objects.all().aggregate(
    Sum('numero'))['numero__sum']
self.assertEqual(6, risultato)
```

PostgreSQL per sviluppare

Un DBMS deve essere buono non solo per la produzione, ma anche per noi programmatori.

Test di unità

Dovrebbero essere veloci e non intrusivi, in modo da non distrarre il programmatore *(Kent Beck)*

Tempi di esecuzione

SQLite

```
$ time python manage.py test
Creating test database
System check identified no issues
.
Ran 1 test in 0.001s
OK
Destroying test database

real    0m0.455s
user    0m0.422s
sys     0m0.031s
```

PostgreSQL

```
$ time python manage.py test
Creating test database
System check identified no issues
.
Ran 1 test in 0.002s
OK
Destroying test database

real    0m6.280s
user    0m0.797s
sys     0m0.048s
```

Perché?

- Django crea e distrugge DB Sqlite in memoria (**no IO**)
- PostgreSQL deve creare e distruggere database (**che è molto costoso**)

Si può far meglio (--keep)

SQLite

```
$ time python manage.py test
Creating test database
System check identified no issues
.
Ran 1 test in 0.001s
OK
Destroying test database

real    0m0.455s
user    0m0.422s
sys     0m0.031s
```

PostgreSQL

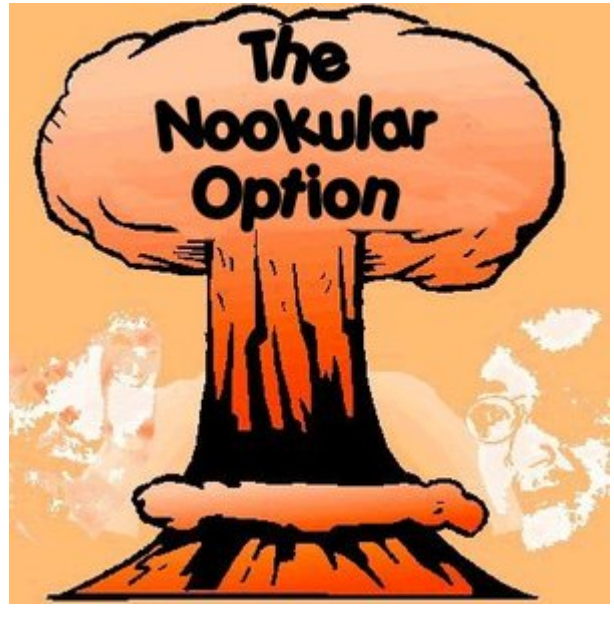
```
$ time python manage.py test --keep
Creating test database
System check identified no issues
.
Ran 1 test in 0.002s
OK
Destroying test database

real    0m0.349s
user    0m0.303s
sys     0m0.032s
```

Si possono ancora accorciare i tempi

PostgreSQL si può configurare in modo da rinunciare alla **durabilità**.

NON LO FATE DA SOLI A CASA!

- fsync=off
- synchronous_commit=off
- full_page_writes=off



django.contrib.postgres - Tipi Vettore

Creare un campo vettore

```python
from django.db import models

from django.contrib.postgres
    import fields


class Modello(models.Model):
    uno = models.CharField(
        max_length=200)
    due = models.TextField()

    tags = fields.ArrayField(
        base_field=
            models.CharField(
                max_length=10)
    )
```

```
Column |          Type             |
-------+---------------------------+
id     | integer                   |
uno    | character varying(200)    |
due    | text                      |
tags   | character varying(10)[] |
```

Come si esegue la ricerca...

contains == @>

```
>>> Modello.create(tags=['uno', 'due', 'tre'])
>>> Modello.create(tags=['due', 'tre', 'quattro'])

>>> Modello.objects.all()
<QuerySet [<Modello: Modello object (1)>,
           <Modello: Modello object (2)>]>

>>> Modello.objects.filter(tags__contains=['uno'])
<QuerySet [<Modello: Modello object (1)>]>


>> str(Modello.objects.filter(tags__contains=['uno']).query)
    SELECT "modello"."id",  "modello"."uno",
           "modello"."due", "modello"."tags"
    FROM "modello"
    WHERE "modello"."tags" @> ['uno']::varchar(10)[]
```

Come si esegue la ricerca...

contained_by == <@

```
>>> Modello.create(tags=['uno', 'due', 'tre'])
>>> Modello.create(tags=['due', 'tre', 'quattro'])

>>> Modello.objects.filter(tags__contained_by=['uno'])
<QuerySet []>


>> str(Modello.objects.filter(tags__contained_by=['uno']).query)
    SELECT "modello"."id",  "modello"."uno",
           "modello"."due", "modello"."tags"
    FROM "modello"
    WHERE "modello"."tags" <@ ['uno']::varchar(10)[]
```

Creazione indici GIN

```
class TestModel(models.Model):
    tags = fields.ArrayField(
      base_field=
          models.CharField(
              max_length=10))

    class Meta:
        indexes = [indexes.GinIndex(fields=['tags'])]
```

```
CREATE INDEX
    "[name]_gin"
ON "[tblname]"
USING gin ("tags");
```

Campi JSON

Permettono di memorizzati dati JSON in PostgreSQL, ottenendo i benefici di un database schema-less in un database relazionale.

JSONB

È il tipo di campo in PostgreSQL che memorizza dati JSON in formato binario, compresso ed indicizzabile!

`django.contrib.postgres.fields.JSONField` utilizza campi **JSONB** di default

Esempio

```
class TestJson(models.Model):
    analytics = fields.JSONField()
```

crea:

```
CREATE TABLE prova_testjson (
    id integer NOT NULL,
    analytics jsonb NOT NULL
);
```

Come si interroga?

```
TestJson.objects.filter(analytics__population__sex='F')
```

diventa:

```
SELECT
  "prova_testjson"."id",
  "prova_testjson"."analytics"
FROM "prova_testjson"
WHERE ("prova_testjson"."analytics"
      #> ['population', 'sex']) = '"F"'
```

Altro esempio

```
TestJson.objects.filter(
  analytics__contains={'sex': 'M', 'age': 37})
```

diventa:

```
SELECT
  "prova_testjson"."id",
  "prova_testjson"."analytics"
FROM "prova_testjson"
WHERE
  "prova_testjson"."analytics" @> '{"age": 37, "sex": "M"}'
```

Ordinamenti

```
TestJson.objects.order_by('analytics__age')
```

Da Django 2.1 si potrà fare!

Ordinamenti

```
TestJson.objects.annotate(
  c=KeyTransform('age', 'analytics')).order_by('c')
```

diventa:

```
SELECT
  "prova_testjson"."id",
  "prova_testjson"."analytics",
  ("prova_testjson"."analytics" -> 'age') AS "c"
FROM "prova_testjson"
ORDER BY "c" ASC
```

GIN e Json - OpClasses disponibili

- **jsonb_ops** per gli operatori ? ?& ?| @>
- **jsonb_path_ops** per gli operatori @>

Non ancora supportato in Django, vedi ticket #28077

PostgreSQL come gestore di code

Un semplice gestione di code

- Creiamo una tabella, dove ogni record rappresenterà un task. Abbiamo anche i campi JSONB per memorizzare i parametri del task, se serve
- Per inserire un nuovo task basta creare un nuovo record
- I job verranno eseguiti da un cron job implementato con un custom command

Prelevare i task da fare

- SELECT * FROM tasks WHERE NOT is_done **funzionerebbe?** No
- SELECT * FROM tasks FOR UPDATE WHERE NOT is_done **funzionerebbe?** No
- SELECT * FROM tasks FOR UPDATE SKIP LOCKED WHERE NOT is_done **funzionerebbe?** Sì!

E con Django?

```
Tasks.objects. \
    select_for_update(skip_locked=True). \
    filter(is_done=False)
```

Vantaggi:

- se abbiamo PostgreSQL in produzione, abbiamo tutto quel che serve;
- i task sono tenuti in un sistema altamente consistente e durevole.

Svantaggi:

- i sistemi dedicati sono più veloci, ma la loro manutenzione ha un costo da non sottovalutare;
- SKIP LOCKED non è standard SQL.

Cose da produzione

pg_stat_statements

pg_stat_statements

pg_stat_statements è una estensione di PostgreSQL che tiene traccia delle query nel sistema e dei loro tempi di esecuzione.

È sviluppata insieme al core di PostgreSQL, e si trova nel package **postgres-contrib** delle distribuzioni Linux.

Per installare **pg_stat_statements** basta caricarla all'interno della configurazione di PostgreSQL fra le librerie da usare all'avvio del server:

```
shared_preload_libraries='pg_stat_statements'
pg_stat_statements.max = 10000
pg_stat_statements.track = all
```

pg_stat_statements monitorerà la versione normalizzata delle query che poniamo al server dicendoci quanto tempo hanno occupato.

È consigliata anche tenerla in produzione, perché non rallenta il lavoro del server PostgreSQL ed è molto utile.

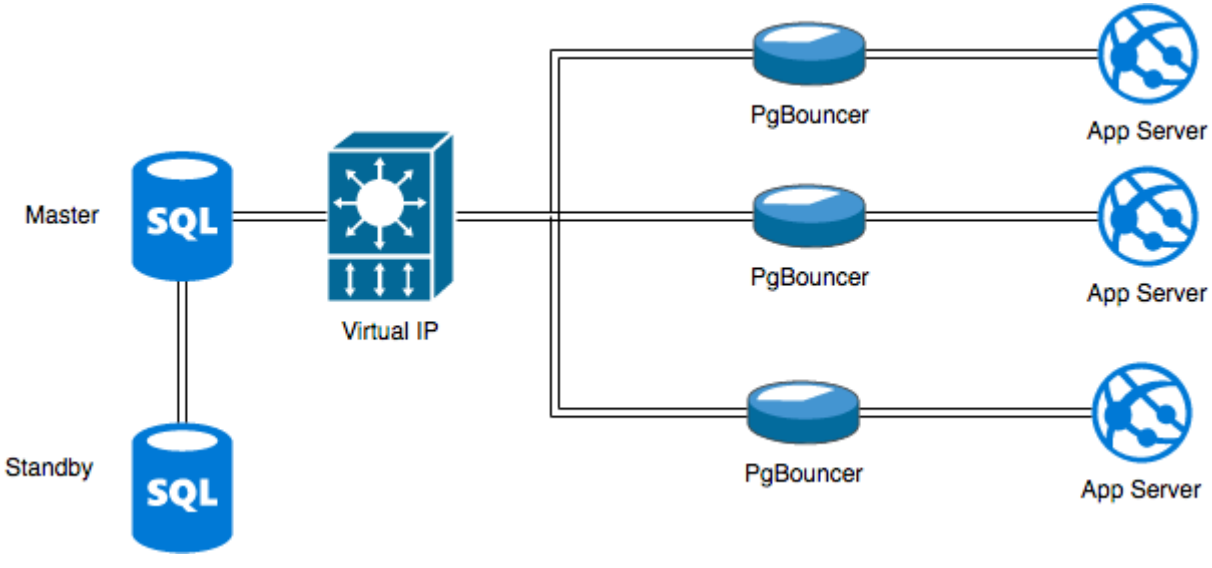
Dati raccolti

```
bench=# SELECT query, calls, total_time, rows
        FROM pg_stat_statements ORDER BY total_time DESC LIMIT 5;
-[ RECORD 1 ]----------------------------------------
query      | UPDATE pgbench_branches
           | SET bbalance = bbalance + ? WHERE bid = ?;
calls      | 3000
total_time | 9609.00100000002
rows       | 2836
-[ RECORD 2 ]----------------------------------------
query      | UPDATE pgbench_tellers
           | SET tbalance = tbalance + ? WHERE tid = ?;
calls      | 3000
total_time | 8015.156
rows       | 2990
```

PgBouncer

Lightweight connection pooler for PostgreSQL

Django ha già il suo connection pooler, ma pgbouncer ha una marcia in più :)


Master
SQL
Standby
SQL
Virtual IP
PgBouncer
PgBouncer
PgBouncer
App Server
App Server
App Server

Vantaggi

- Evita il costo di creazione delle connessioni
- **Transaction pooling**: la connessione ritorna nel pool dopo che la transazione è finita. (mai più backend in idle!)
- **PAUSE e RESUME**: abilitano/disabilitano il forwarding delle connessioni ad un certo database, e sono fondamentali per lo switchover!

Django e PostgreSQL: un'accoppiata vincente

Django is, and will continue to be, a database-agnostic web framework. [...]
However, we recognize that real world projects written using Django **need not be database-agnostic**. [...]

Cit: Django Documentation

Django provides support for a number of data types which will only work with PostgreSQL. There is no fundamental reason why (for example) a contrib.mysql module does not exist, except that PostgreSQL has the **richest feature set of the supported databases so its users have the most to gain**.

Cit: Django Documentation

Domande?

Leonardo Cecchi
@leonardo_cecchi
leonardo.cecchi@2ndquadrant.it

Tamara Nocentini
@TamaraNocentini
tamara.nocentini
@2ndquadrant.com


2ndQuadrant®
PostgreSQL