# Django et PostgreSQL sous la charge

Rodolphe Quiédeville

Pourquoi couper la queue du poulet ?
RMLL - Beauvais

8 juillet 2015

# #mylife

- Admin/Sys tendance DevOps depuis 20 ans
- 84000 heures de connections au web
- Nourri au logiciel libre exclusivement
- Contributeur à Tsung
- Lead Architect chez PeopleDoc
- Consultant sur les performances des SI(G)

# 2 mots sur PeopleDoc

- Métiers, dématérialisation des docuemnts RH
- Conservation des documents **50 ans**
- 3 projets Django distincts
- 7 plateformes
- 3 clusters PostgreSQL par plateforme (9.4 et 9.1)
- une centaine de bases de données
- 50 Millions de documents, +100% tous les ans

# Assistant pour cette présentation

photo by InAweofGod'sCreation flickr http://bit.ly/1EBQPrQ

Un dimanche matin dans la cuisine le petit Yohann demande à sa mère

> *Maman pourquoi tu coupes la queue du poulet avant de le cuire dans le four ?*

La maman de répondre

> *Je ne sais pas mais j'ai toujours vu ma mère faire comme ça, vient on va demander à ta grand-mère.*

# Créer des données

L'import est le parent pauvre des optimisations, sous prétexte qu'il se produit rarement, ou pas ...
Prenons un cas simple d'import de 300K objets dans une base.

# Créer des données

Ce qui vient en premier à l'esprit, la boucle classique sur chaque élément

## create()

```
def insert(values):
    """Insert values in DB
    """
    for val in values:
        res = Item.objects.create(method="sequential",
                                  datetms=val['datetms'],
                                  name=val['name'],
                                  email=val['email'],
                                  value=val['value'])
```

# Créer des données

Il existe des méthodes pour l'insertion en batch dans la doc de Django.

## bulk_create()

```
def dobulk(part_values, method):
    """Do bulk insert on DB
    """
    poms = []
    for val in part_values:
        poms.append(Item(datetms=val['datetms'],
                         method=method,
                         name=val['name'],
                         email=val['email'],
                         value=val['value']))

    Item.objects.bulk_create(poms)
```

# Créer des données

Il existe des méthodes pour l'insertion en batch dans la doc de Django.

## bulk_create()

```
def dobulk(part_values, method):
    """Do bulk insert on DB
    """
    poms = []
    for val in part_values:
        poms.append(Item(datetms=val['datetms'],
                         method=method,
                         name=val['name'],
                         email=val['email'],
                         value=val['value']))

    Item.objects.bulk_create(poms)
```

Attention tout de même aux effets de bords possibles.

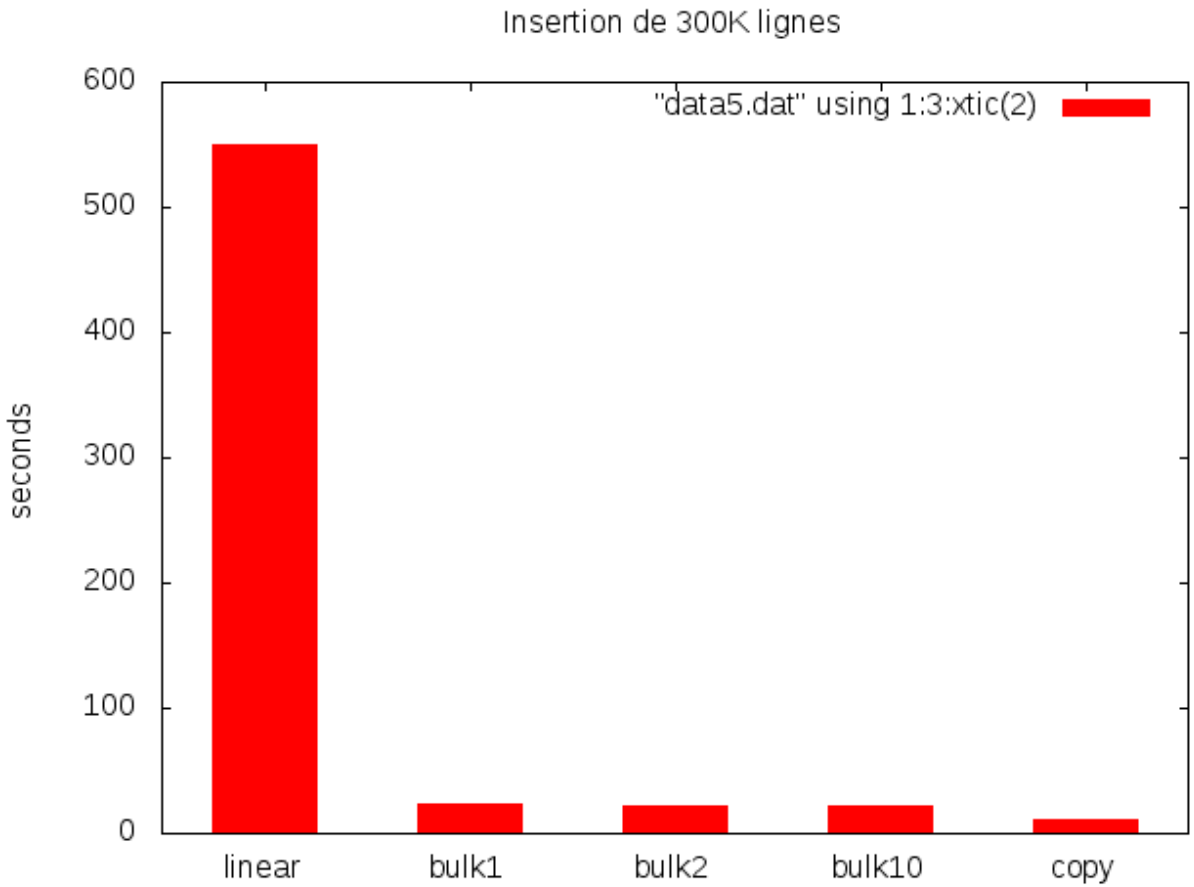
# Créer des données

En utilisant l'instruction COPY de PostgreSQL et un fichier temporaire (parfois inutile)

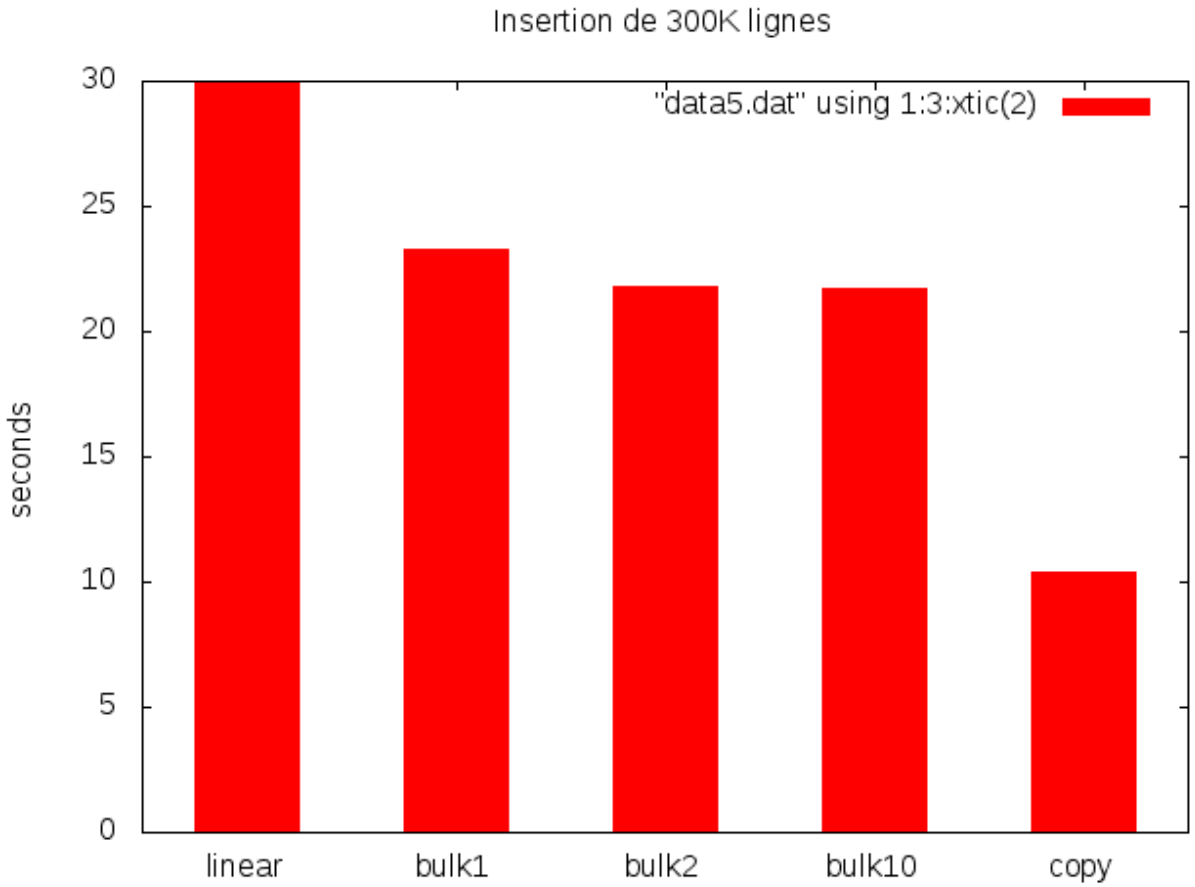
## cursor.copy_from

```
def copyinsert(values, method):
    """Use COPY SQL FUNCTION to insert datas
    """
    fields = ['datetms', 'name', 'email', 'value', 'method']

    fpath = os.path.join('/tmp/', str(uuid4()))
    f = open(fpath, 'w')
    for val in values:
        f.write('{},"{}","{}",{},{}\n'.format(val['datetms'],
                                              val['name'],
                                              val['email'],
                                              val['value'],
                                              method))
    f.close()
    cursor = connection.cursor()
    cursor.copy_from(open(fpath, 'r'), 'loader_item', columns=tuple(fields), sep=',')
    unlink(fpath)
```

# Les résultats



# Les résultats




Un peu plus prêt des étoiles

# Pourquoi bulk_create est plus rapide ?

## Syntaxe courante

```
INSERT INTO bar (value, ratio) VALUES (1, 3.4);
INSERT INTO bar (value, ratio) VALUES (2, 5.4);
INSERT INTO bar (value, ratio) VALUES (3, 3.14);
```

# Pourquoi bulk_create est plus rapide ?

## Syntaxe connue

```
INSERT INTO bar (value, ratio) VALUES (1, 3.4);
INSERT INTO bar (value, ratio) VALUES (2, 5.4);
INSERT INTO bar (value, ratio) VALUES (3, 3.14);
```

## Syntaxe moins connue

```
INSERT INTO bar (value, ratio) VALUES (1, 3.4), (2, 5.4), (3, 3.14);
```

Yohann et sa maman demande à mère grand pourquoi elle coupait la queue du poulet avant de le cuire

> *Je ne sais pas mais j'ai toujours vu ma mère faire comme ça, on va demander à ta grand-mère.*

Yohann et sa maman demande à mère grand pourquoi elle coupait la queue du poulet avant de le cuire

> *Je ne sais pas mais j'ai toujours vu ma mère faire comme ça, on va demander à ta grand-mère.*

Par chance l'arrière grand-mère de Yohann vit toujours, le voilà en route avec sa mère pour aller lui poser la question.

# Mettre à jour un lot de données

Pendant que Yohann est sur la route nous allons augmenter les ratios de 5% des pommes qui ont un **indice = 4**

## Le modèle

```
class Apple(models.Model):
    """An apple
    """
    name = models.CharField(max_length=300)
    indice = IntegerField(default=0)
    ratio = FloatField(default=0)
```

# Premier réflexe

## On lit et on boucle

```
objs = Apple.objects.filter(indice=4)

for obj in objs:
   obj.ratio = obj.ratio * 1.05
   obj.save()
```

# Premier réflexe

## On lit et on boucle

```
objs = Apple.objects.filter(indice=4)

for obj in objs:
   obj.ratio = obj.ratio * 1.05
   obj.save()
```

*1000 tuples dans la base ==> 1001 requêtes (ou pire)*

# F comme Facile

Au sens SQL cela revient à un UPDATE des plus classiques.

## SQL

```
UPDATE apple_apple SET ratio=ratio * 1.5 WHERE indice=4;
```

# F comme Facile

Au sens SQL cela revient à un UPDATE des plus classiques.

## SQL

```
UPDATE apple_apple SET ratio=ratio * 1.5 WHERE indice=4;
```

## update et F

```
objs = Apple.objects.filter(indice=indice).update(ratio=F('ratio')*1.05)
```

# F comme Facile

Au sens SQL cela revient à un UPDATE des plus classiques.

## SQL

```
UPDATE apple_apple SET ratio=ratio * 1.5 WHERE indice=4;
```

## update et F

```
objs = Apple.objects.filter(indice=indice).update(ratio=F('ratio')*1.05)
```

*1 requête SQL exécutée au lieu de 1001, succès garantit*

# Pagination

Et si pour paginer on utilisait **Paginator** ?

## Avant

```
apples = Apple.objects.all().order_by('pk')
paginator = Paginator(apples, 250)
for p in paginator.page_range:
    for apple in paginator.page(p).object_list:
        apple.ratio = apple.ratio * 1.05
        apple.save()
```

# Pagination

Et si pour paginer on utilisait **Paginator** ?

## Avant

```
apples = Apple.objects.all().order_by('pk')
paginator = Paginator(apples, 250)
for p in paginator.page_range:
    for apple in paginator.page(p).object_list:
        apple.ratio = apple.ratio * 1.05
        apple.save()
```

*Ceci est une situation tirée de faits réels*

# Pagination

## Key pagination

```
keyid = 0
while True:
    apples = Apple.objects.filter(id__gt=keyid).order_by('id')[:250]
    for apple in apples:
        keyid = apple.id
        # do want you want here
        apple.ratio = apple.ratio * 1.05
        apple.save()

    if len(apples) < 250:
        break
```

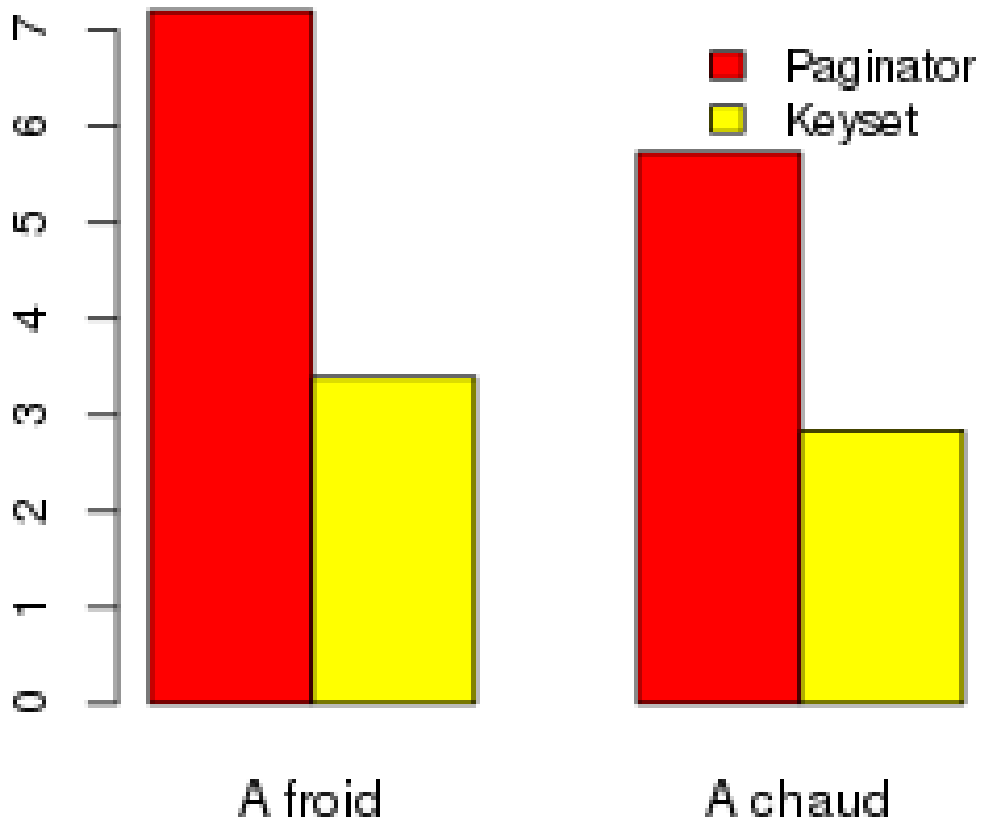
# Pagination

## Key pagination

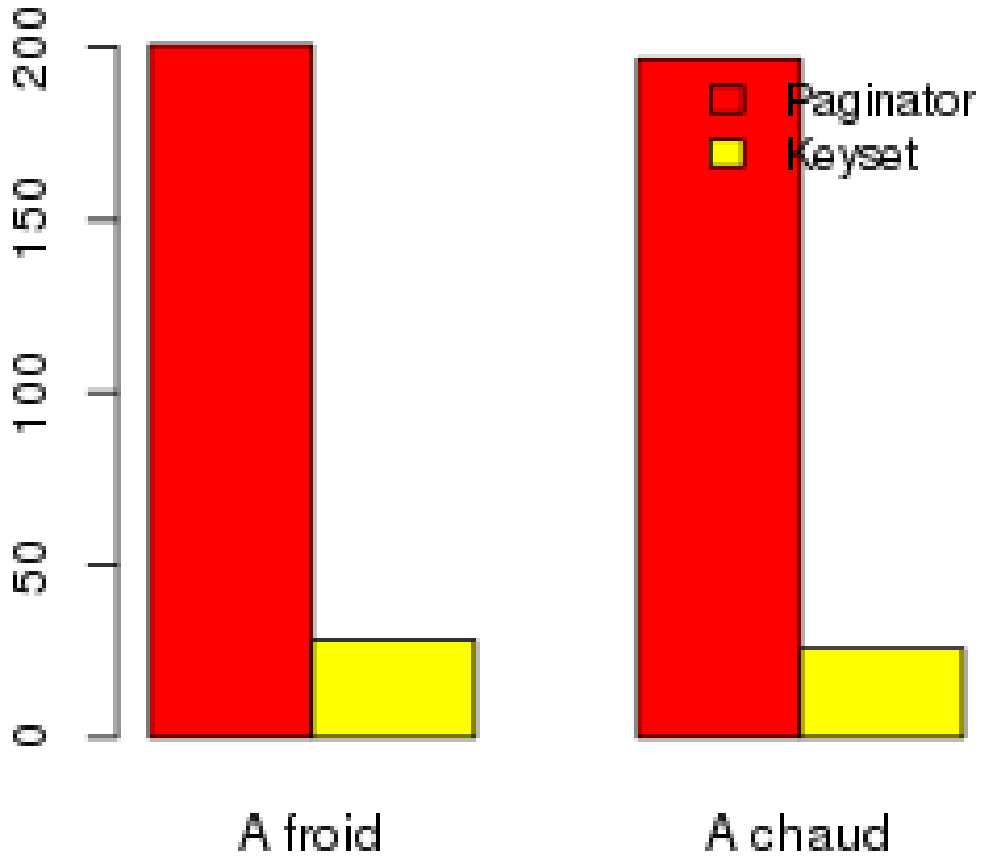
```
keyid = 0
while True:
    apples = Apple.objects.filter(id__gt=keyid).order_by('id')[:250]
    for apple in apples:
        keyid = apple.id
        # do want you want here
        apple.ratio = apple.ratio * 1.05
        apple.save()

    if len(apples) < 250:
        break
```

*Django crée par défaut des clés primaires, ça tombe bien on en a besoin ici*

# La table tient en cache



# La table dépasse la taille du cache



Si vous parcourez des tables de grandes volumétrie, vous **devez** oublier OFFSET.


Lisez l'excellent livre de Markus Winnand pour découvrir la pagination par clée et la joyeuse vie des index.

Yohann arrivé chez son arrière-grand-mère, la questionne sans attendre.

> *Dit moi grand-grand-mamie pourquoi tu coupes la queue du poulet avant de le cuire ?*

L’arrière grand-mère de répondre

*Oh mon petit t’es ben mignon, mais ça chais pus ben moé, ch’crai bi que j’a toujours vu ma mè fai’r ainsi.*

L'arrière grand-mère de répondre

> *Oh mon petit t'es ben mignon, mais ça chais pus ben moé, ch'crai bi que j'a toujours vu ma mè fai'r ainsi.*

Par chance l'arrière-arrière-grand-mère de Yohann vit toujours, le voilà une nouvelle fois sur la route avec sa mère pour aller glaner la précieuse information auprès de sa trisaïeule.

Pendant ce temps là, interessons-nous à la lecture de nos données

## modèle BigBook

```
class BigBook(models.Model):
    """The big books
    """
    keyid = models.IntegerField(unique=True)
    author = models.ForeignKey(Author)
    title = models.CharField(max_length=30)
    serie = models.IntegerField(default=0)
    nbpages = models.IntegerField(default=0)
    editors = models.ManyToManyField(Editor, blank=True)
    translators = models.ManyToManyField(Translator, blank=True)
    synopsis = models.TextField(blank=True)
    intro = models.TextField(blank=True)
```

On va faire une action sur 10% de la table environ, une opération de mise à jour conditionnelle

## SQL

```
books = BigBook.objects.filter(serie=3).order_by('keyid')
for book in books:
    keyid = book.keyid
    # do want you want here
    if book.nbpages > 500:
        book.action()
```

## SQL

```
books = BigBook.objects.filter(serie=3).order_by('keyid')
for book in books:
    keyid = book.keyid
    # do want you want here
    if book.nbpages > 500:
        book.action()
```

Dans la méthode **.action()** on a besoin uniquement du keyid et de nbpages.

## SQL

```
books = BigBook.objects.filter(serie=3).order_by('keyid')
for book in books:
    keyid = book.keyid
    # do want you want here
    if book.nbpages > 500:
        book.action()
```

Dans la méthode **.action()** on a besoin uniquement du keyid et de nbpages.

## SQL

```
>>> BigBook.objects.filter(serie=3).order_by('keyid')
SELECT "july_bigbook"."id",
       "july_bigbook"."keyid",
       "july_bigbook"."author_id",
       "july_bigbook"."title",
       "july_bigbook"."serie",
       "july_bigbook"."nbpages",
       "july_bigbook"."synopsis",
       "july_bigbook"."intro"
FROM "july_bigbook"
WHERE "july_bigbook"."serie" = 3
ORDER BY "july_bigbook"."keyid" LIMIT 21 [127.64ms]
```

Execution time : 127.64ms

# Ré-écriture de la queryset

```
books = BigBook.objects.only('keyid','nbpages').filter(serie=3).order_by('keyid')
for book in books:
    keyid = book['keyid']
    # do want you want here
    if book['nbpages'] > 500:
        book.action()
```

## Ré-écriture de la queryset

```
books = BigBook.objects.only('keyid','nbpages').filter(serie=3).order_by('keyid')
for book in books:
    keyid = book['keyid']
    # do want you want here
    if book['nbpages'] > 500:
        book.action()
```

## SQL

```
>>> BigBook.objects.only('keyid','nbpages').filter(serie=3).order_by('keyid')
SELECT "july_bigbook"."id",
       "july_bigbook"."keyid",
       "july_bigbook"."nbpages"
FROM "july_bigbook"
WHERE "july_bigbook"."serie" = 3
ORDER BY "july_bigbook"."keyid" LIMIT 21 [1.53ms]
```

Execution time : 1.53ms

# Avec effets secondaires depuis PostgreSQL 9.3

## SQL

```
>>> BigBook.objects.only('keyid','nbpages').filter(serie=3).order_by('keyid')
SELECT "july_bigbook"."id",
       "july_bigbook"."keyid",
       "july_bigbook"."nbpages"
FROM "july_bigbook"
WHERE "july_bigbook"."serie" = 3
ORDER BY "july_bigbook"."keyid" LIMIT 21 [1.53ms]
```

# Avec effets secondaires depuis PostgreSQL 9.3

## SQL

```
>>> BigBook.objects.only('keyid','nbpages').filter(serie=3).order_by('keyid')
SELECT "july_bigbook"."id",
       "july_bigbook"."keyid",
       "july_bigbook"."nbpages"
FROM "july_bigbook"
WHERE "july_bigbook"."serie" = 3
ORDER BY "july_bigbook"."keyid" LIMIT 21 [1.53ms]
```

## EXPLAIN

```
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Sort  (cost=162.31..162.31 rows=1 width=12) (actual time=0.628..0.628 rows=0 loops=1)
   Sort Key: keyid
   Sort Method: quicksort  Memory: 25kB
   ->  Seq Scan on july_bigbook  (cost=0.00..162.30 rows=1 width=12)
         Filter: (serie = 3)
         Rows Removed by Filter: 3784
 Planning time: 0.161 ms
 Execution time: 0.651 ms
(8 rows)
```

## Avec effets secondaires depuis PostgreSQL 9.3

### SQL

```
CREATE INDEX magic ON july_bigbook(id, serie, keyid, nbpages);
```

# Avec effets secondaires depuis PostgreSQL 9.3

## SQL

```
CREATE INDEX magic ON july_bigbook(id, serie, keyid, nbpages);
```

## EXPLAIN

```
QUERY PLAN
------------------------------------------------------------------------------------------
 Sort  (cost=98.43..98.44 rows=1 width=12) (actual time=0.150..0.150 rows=0 loops=1)
 Sort Key: keyid
 Sort Method: quicksort  Memory: 25kB
    ->  Index Only Scan using magic on july_bigbook  (cost=0.28..98.42 rows=1 width=12)
    Index Cond: (serie = 3)
    Heap Fetches: 0
    Planning time: 0.199 ms
    Execution time: 0.177 ms
    (8 rows)
```

## Avec effets secondaires depuis PostgreSQL 9.3

### SQL

```
CREATE INDEX magic ON july_bigbook(id, serie, keyid, nbpages);
```

### EXPLAIN

```
QUERY PLAN
------------------------------------------------------------------------------------------
 Sort  (cost=98.43..98.44 rows=1 width=12) (actual time=0.150..0.150 rows=0 loops=1)
 Sort Key: keyid
 Sort Method: quicksort  Memory: 25kB
    ->  Index Only Scan using magic on july_bigbook  (cost=0.28..98.42 rows=1 width=12)
    Index Cond: (serie = 3)
    Heap Fetches: 0
    Planning time: 0.199 ms
    Execution time: 0.177 ms
    (8 rows)
```

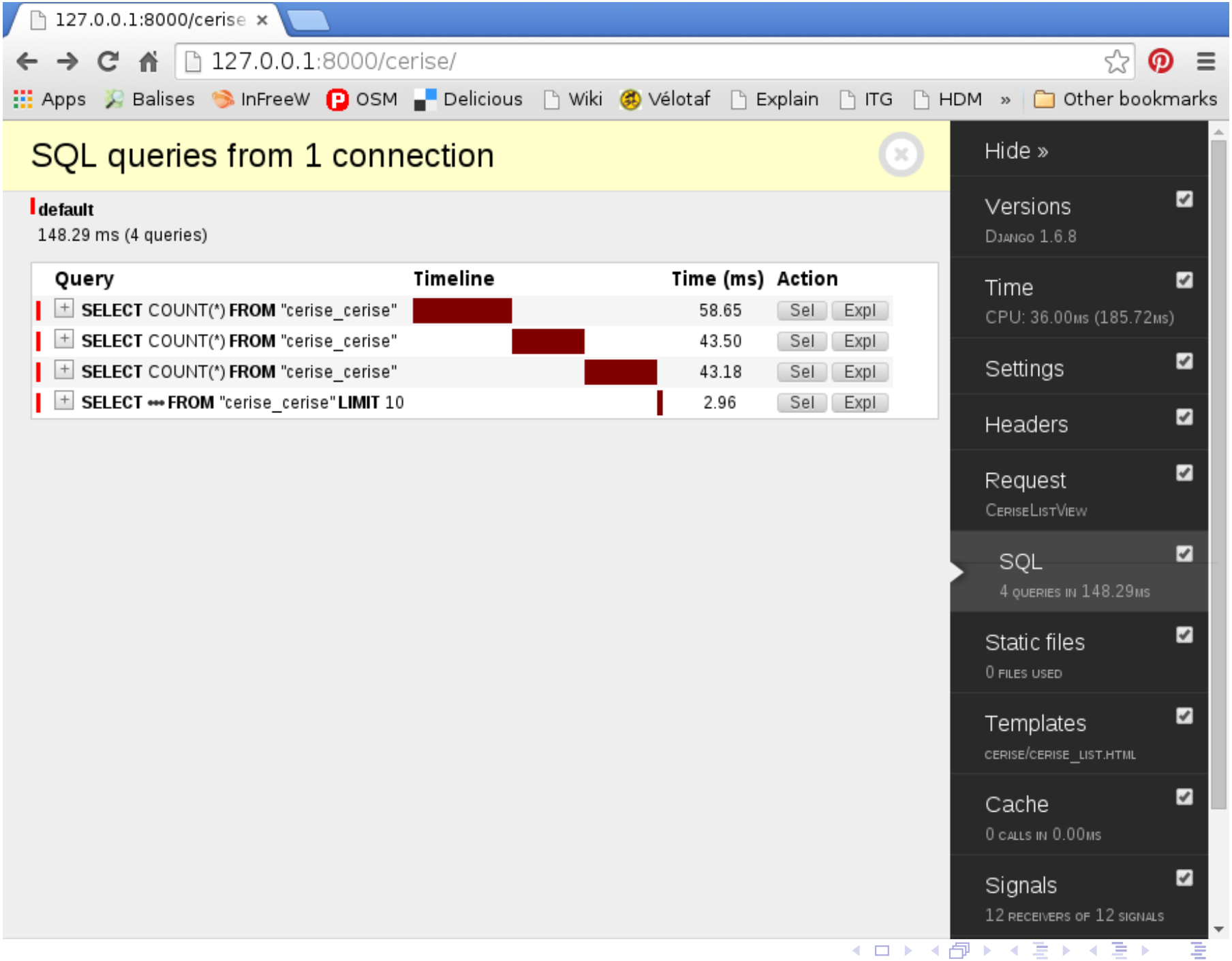
## 0.651ms à 0.177ms sur seulement 4K tuples

# Du coté des templates

# Du coté des templates

## Un template classique

```
{% if object_list.count %}
<b>Found projets : {% object_list.count %}</b>
<div>Projets :
  <ul>
    {% for projet in object_list %}
      <li>{{projet.name}}</li>
    {% endfor %}
  </ul>
</div>
{% endif %}
```


127.0.0.1:8000/cerise
127.0.0.1:8000/cerise/
Apps
Balises
InFreeW
OSM
Delicious
Wiki
Vélotaf
Explain
ITG
HDM
Other bookmarks
SQL queries from 1 connection
default
148.29 ms (4 queries)
Query
Timeline
Time (ms)
Action
SELECT COUNT(*) FROM "cerise_cerise"
58.65
Sel
Expl
SELECT COUNT(*) FROM "cerise_cerise"
43.50
Sel
Expl
SELECT COUNT(*) FROM "cerise_cerise"
43.18
Sel
Expl
SELECT ••• FROM "cerise_cerise" LIMIT 10
2.96
Sel
Expl
Hide »
Versions
Django 1.6.8
Time
CPU: 36.00ms (185.72ms)
Settings
Headers
Request
CeriseListView
SQL
4 queries in 148.29ms
Static files
0 files used
Templates
cerise/cerise_list.html
Cache
0 calls in 0.00ms
Signals
12 receivers of 12 signals

# with

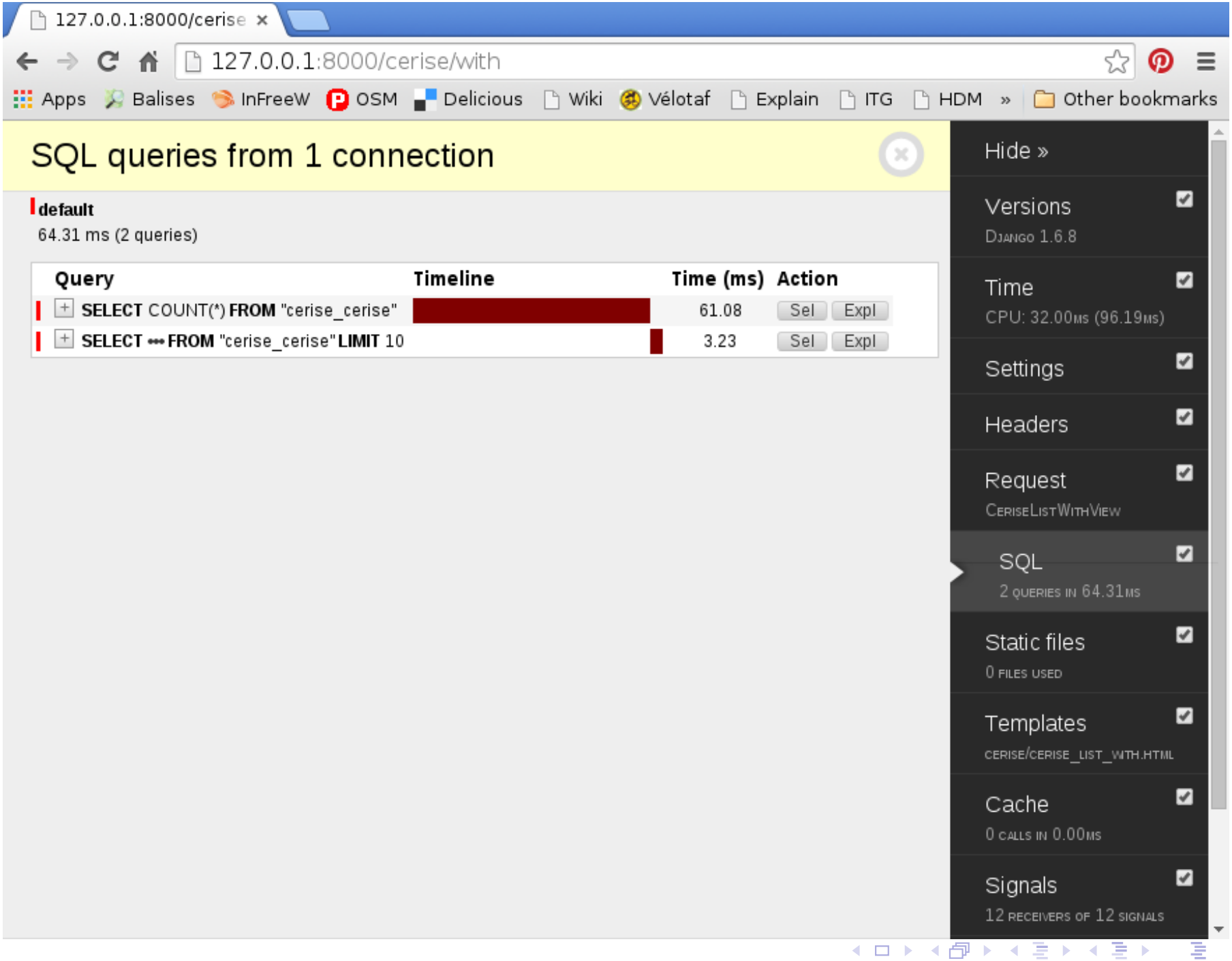
Une solution ici serait d'utiliser **with** pour éviter de refaire les mêmes requêtes plusieurs fois.

# with

Une solution ici serait d'utiliser **with** pour éviter de refaire les mêmes requêtes plusieurs fois.

## Python

```
{% with Projets=object_list.all %}

{% if Projets.count %}
<b>Found projets : <ul>{% Projets.count %}</b>
<div>Projets :
  <ul>
    {% for projet in Projets %}
      <li>{{projet.name}}</li>
    {% endfor %}
  </ul>
</div>
{% endif %}
{% endwith %}
```


127.0.0.1:8000/cerise
127.0.0.1:8000/cerise/with
Apps
Balises
InFreeW
OSM
Delicious
Wiki
Vélotaf
Explain
ITG
HDM
Other bookmarks
SQL queries from 1 connection
default
64.31 ms (2 queries)
Query
Timeline
Time (ms)
Action
SELECT COUNT(*) FROM "cerise_cerise"
61.08
Sel
Expl
SELECT ••• FROM "cerise_cerise" LIMIT 10
3.23
Sel
Expl
Hide »
Versions
Django 1.6.8
Time
CPU: 32.00ms (96.19ms)
Settings
Headers
Request
CeriseListWithView
SQL
2 queries in 64.31ms
Static files
0 files used
Templates
cerise/cerise_list_with.html
Cache
0 calls in 0.00ms
Signals
12 receivers of 12 signals

Dans la même logique nous avons les propriétés des ForeignKey

# Propriétés des ForeignKey

## Le modèle

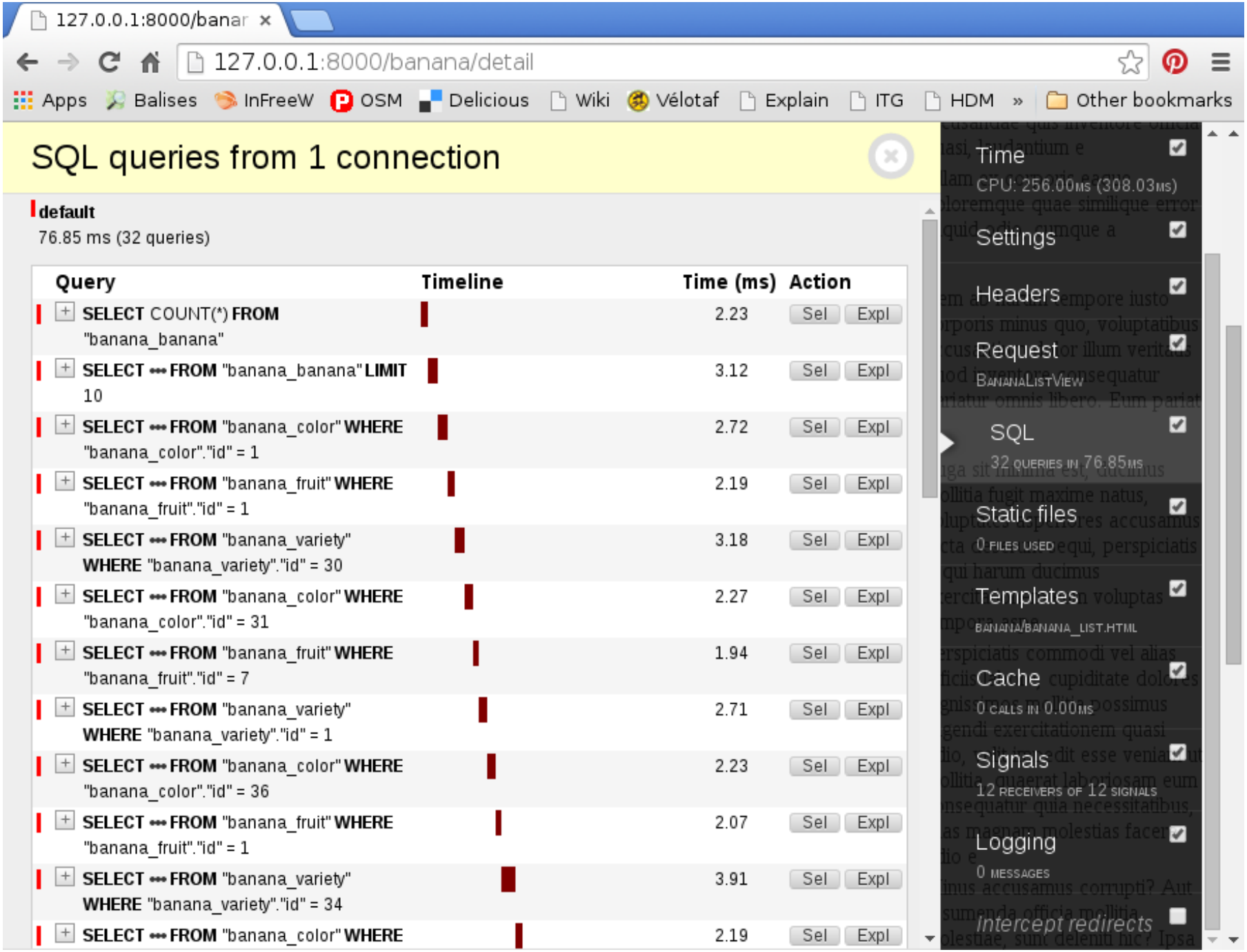
```
class Banana(models.Model):

    name = models.CharField(max_length=300)
    variety = models.ForeignKey(Variety)
```

## La view

```
class BananaRelatedListView(ListView):
    model = Banana
    paginate_by = 10
```

## Le template

```
    {% for object in object_list %}
    <tr>
      <td>{{object.name}}</td>
      <td>{{object.variety.name}}</td>
    </tr>
    {% endfor %}
```

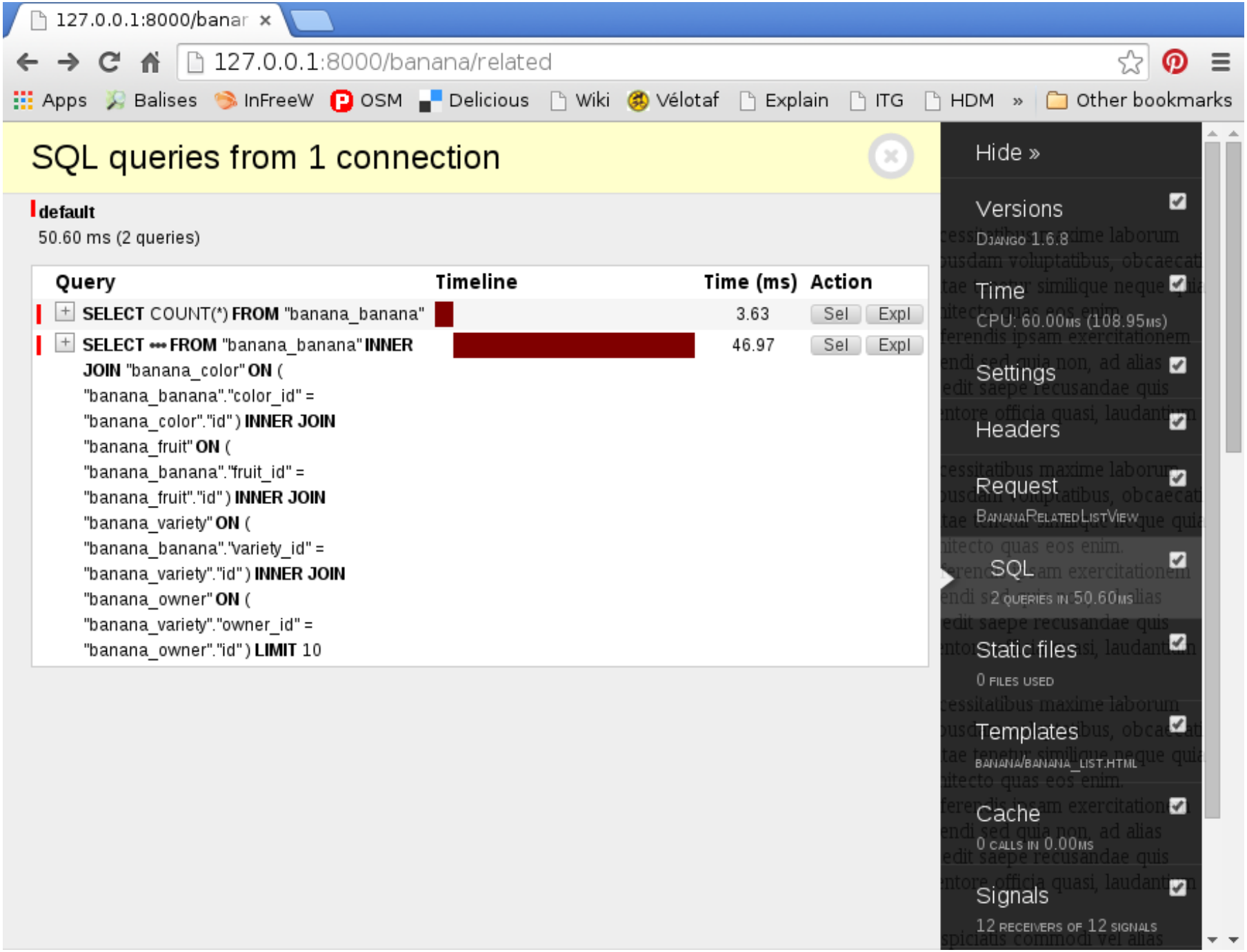

127.0.0.1:8000/banar
127.0.0.1:8000/banana/detail
Apps
Balises
InFreeW
OSM
Delicious
Wiki
Vélotaf
Explain
ITG
HDM
Other bookmarks
SQL queries from 1 connection
default
76.85 ms (32 queries)
Query
Timeline
Time (ms)
Action
SELECT COUNT(*) FROM "banana_banana" 2.23 Sel Expl
SELECT ••• FROM "banana_banana" LIMIT 10 3.12 Sel Expl
SELECT ••• FROM "banana_color" WHERE "banana_color"."id" = 1 2.72 Sel Expl
SELECT ••• FROM "banana_fruit" WHERE "banana_fruit"."id" = 1 2.19 Sel Expl
SELECT ••• FROM "banana_variety" WHERE "banana_variety"."id" = 30 3.18 Sel Expl
SELECT ••• FROM "banana_color" WHERE "banana_color"."id" = 31 2.27 Sel Expl
SELECT ••• FROM "banana_fruit" WHERE "banana_fruit"."id" = 7 1.94 Sel Expl
SELECT ••• FROM "banana_variety" WHERE "banana_variety"."id" = 1 2.71 Sel Expl
SELECT ••• FROM "banana_color" WHERE "banana_color"."id" = 36 2.23 Sel Expl
SELECT ••• FROM "banana_fruit" WHERE "banana_fruit"."id" = 1 2.07 Sel Expl
SELECT ••• FROM "banana_variety" WHERE "banana_variety"."id" = 34 3.91 Sel Expl
SELECT ••• FROM "banana_color" WHERE 2.19 Sel Expl
Time
CPU: 256.00ms (308.03ms)
Settings
Headers
Request
BananaListView
SQL
32 queries in 76.85ms
Static files
0 files used
Templates
banana/banana_list.html
Cache
0 calls in 0.00ms
Signals
12 receivers of 12 signals
Logging
0 messages
Intercept redirects

On affiche des informations liées au modèle principal, pourquoi ne pas utiliser **select_related**

On affiche des informations liées au modèle principal, pourquoi ne pas utiliser **select_related**

## Python

```
class BananaRelatedListView(ListView):
    model = Banana
    paginate_by = 10

    def get_queryset(self):
        bananas = Banana.objects.all().select_related()
        return bananas
```


127.0.0.1:8000/banar
127.0.0.1:8000/banana/related
Apps
Balises
InFreeW
OSM
Delicious
Wiki
Vélotaf
Explain
ITG
HDM
Other bookmarks
SQL queries from 1 connection
default
50.60 ms (2 queries)
Query
Timeline
Time (ms)
Action
SELECT COUNT(*) FROM "banana_banana"
3.63
Sel
Expl
SELECT ••• FROM "banana_banana" INNER JOIN "banana_color" ON ( "banana_banana"."color_id" = "banana_color"."id" ) INNER JOIN "banana_fruit" ON ( "banana_banana"."fruit_id" = "banana_fruit"."id" ) INNER JOIN "banana_variety" ON ( "banana_banana"."variety_id" = "banana_variety"."id" ) INNER JOIN "banana_owner" ON ( "banana_variety"."owner_id" = "banana_owner"."id" ) LIMIT 10
46.97
Sel
Expl
Hide »
Versions
Django 1.6.8
Time
CPU: 60.00ms (108.95ms)
Settings
Headers
Request
BananaRelatedListView
SQL
2 queries in 50.60ms
Static files
0 files used
Templates
banana/banana_list.html
Cache
0 calls in 0.00ms
Signals
12 receivers of 12 signals

- cas simple, petite volumétrie passe de 76 à 50msec

- cas simple, petite volumétrie passe de 76 à 50msec
- forte volumétrie, facteur 10x voir 50x

- cas simple, petite volumétrie passe de 76 à 50msec
- forte volumétrie, facteur 10x voir 50x
- les templates sont ma première source d'intérêt sur un projet avec des problèmes de performance


CLINT EASTWOOD
LE BON
LA BRUTE
LE TRUAND
LEE VAN CLEEF
ALDO GIUFFRE
MARIO BREGA
ELI WALLACH
DANS LE RÔLE DE TUCO
TECHNICOLOR®
TECHNISCOPE®
RÉALISÉ PAR SERGIO LEONE
UNITED ARTISTS

- le modèle


CLINT EASTWOOD
LEE VAN CLEEF
ALDO GIUFFRE
MARIO BREGA
ELI WALLACH
DANS LE RÔLE DE TUCO
LE BON
LA BRUTE
LE TRUAND
TECHNICOLOR ®
TECHNISCOPE ®
RÉALISÉ PAR SERGIO LEONE
AGE · SCARPELLI · LUCIANO VINCENZONI ET SERGIO LEONE
ENNIO MORRICONE
ALBERTO GRIMALDI
P.E.A.
UNITED ARTISTS

- le modèle
- la view

- le modèle
- la view
- le template

# Le point de vue des performances

- le modèle, les méthodes sont explicites (en limitant les niveaux d'héritages)

# Le point de vue des performances

- le modèle, les méthodes sont explicites (en limitant les niveaux d'héritages)
- la view, vous maîtrisez get_queryset() et surchargez get_context_data()

# Le point de vue des performances

- le modèle, les méthodes sont explicites (en limitant les niveaux d'héritages)
- la view, vous maîtrisez get_queryset() et surchargez get_context_data()
- le template, que va faire le **.count** ?

# Confrontations d'idéologies

- point Godwin

# Confrontations d'idéologies

- point Godwin
- point raw sql

# Cas ultime

- votre application reçoit une request. par exemple /blog/api/articles/ GET
- votre application décode la requête et va contacter votre base de données pour récupérer les lignes correspondant aux 10 premiers articles.
- Django va convertir ces lignes en objet python
- votre serializer va convertir ces objets python en JSON
- votre application va retourner un payload JSON.

*sur une idée originale de Y. Gabory, https://gitlab.com/boblefrag/LIvre_Django_Perf*

# Cas ultime

## Le template

```
from blog.models import Article
from django.views.generic import ListView

from django.core import serializers
from django import http
from django.db import connection
from psycopg2 import extras

extras.register_default_json(loads=lambda x: x)


class AJAXListMixin(object):

    def get_queryset(self):
        c = connection.cursor()
        c.execute("""
        SELECT array_to_json(array_agg(row_to_json(t)))
        FROM( SELECT * FROM blog_article LIMIT 10) t """)
        return c.fetchone()

    def render_to_response(self, context, **kwargs):
        response = http.HttpResponse(self.get_queryset())
        for t in connection.queries:
            print t
        return response

class ArticleView(AJAXListMixin, ListView):
    model = Article
    paginate_by = 10
```

Yohann est arrivé, va-t'il avoir la réponse à sa question ?

> *Grand-grand-Maman pourquoi tu coupes la queue du poulet avant de le cuire ?*

Milles merci à Bruno Bord pour cette histoire

Yohann est arrivé, va-t'il avoir la réponse à sa question ?

> *Grand-grand-Maman pourquoi tu coupes la queue du poulet avant de le cuire ?*
>
> *Parce qu'à l'époque les fours étaient trop petit, et le poulet ne pouvait pas entrer en entier.*

Milles merci à Bruno Bord pour cette histoire

# Avec le temps

Nous aurions aussi évoqué :

- django-json-dbindex
- django-aggtrigg
- django-hstore
- djorm-pgarray


Questions ?

On recrute !

Rodolphe Quiédeville

rodolphe.quiedeville@people-doc.com