Flex schemas with PostgreSQL and Django

Advanced PostgreSQL: Models halfway between Relational and NoSQL

José Antonio Perdiguero López
https://github.com/PeRDy
https://www.linkedin.com/in/josé-antonio-perdiguero-lópez-732a9768/
@ perdy.hh@gmail.com

May 5, 2017

Málaga Python MeetUp
OpenSouthCode 2017

Index

Introduction

Relational Databases

Relational databases are those based on *Relational Model*, invented by *E.F. Codd*.

The relational model define that all the data is represented in terms of tuples, grouped in relations. These relations consists of a set of attributes and a set of *n* tuples, and are called tables.

Relational Databases: Pros vs Cons

👍

- Structured data
- Strong typing
- Integrity and constraints
- Indexes
- Support
- Fast querying
- Maturity

👎

- Admin
- Poor horizontal scaling
- Schema and data migrations

NoSQL Databases

Key-Value

Key-Value stores are the simplest NoSQL data stores. The value is a blob, and due to primary-key access they have a great performance and scalability.

Document

Documents are the main concept in document databases. The database stores and retrieves documents, which can be XML, JSON, BSON, and so on.

Column family

Column-family databases store data in column families as rows that have many columns associated with a row key.

Graph

Graph databases allow you to store entities and relationships between these entities. Entities are also known as nodes, which have properties. Relations are known as edges that can have properties.

NoSQL Databases: Pros vs Cons

👍

- Fast writing
- Horizontal scaling
- Reduced admin tasks
- Flexible
- Mostly open-source

👎

- Slow querying
- Support
- Maturity

Should I use a Relational or NoSQL database?

Fields

ArrayField

A field for storing lists of data that can be nested to store multi-dimensional arrays.

Example

```
from django.db import models
from django.contrib.postgres.fields import ArrayField

class Post(models.Model):
    name = models.CharField(max_length=200)
    bar = ArrayField(models.CharField(max_length=200), blank=True)

    def __str__(self):
        return self.name

>>> Post.objects.create(name='First post', tags=['thoughts', 'django'])
>>> Post.objects.create(name='Second post', tags=['thoughts'])
>>> Post.objects.create(name='Third post', tags=['tutorial', 'django'])
```

ArrayField: Queries

Contains (@>)

```
>>> Post.objects.filter(tags__contains=['thoughts'])
<QuerySet [<Post: First post>, <Post: Second post>]>

>>> Post.objects.filter(tags__contains=['django'])
<QuerySet [<Post: First post>, <Post: Third post>]>

>>> Post.objects.filter(tags__contains=['django', 'thoughts'])
<QuerySet [<Post: First post>]>
```

Contained By (<@)

```
>>> Post.objects.filter(tags__contained_by=['thoughts', 'django'])
<QuerySet [<Post: First post>, <Post: Second post>]>

>>> Post.objects.filter(tags__contained_by=['thoughts', 'django', 'tutorial'])
<QuerySet [<Post: First post>, <Post: Second post>, <Post: Third post>]>
```

ArrayField: Queries

Overlap (&&)

```
>>> Post.objects.filter(tags__overlap=['thoughts'])
<QuerySet [<Post: First post>, <Post: Second post>]>

>>> Post.objects.filter(tags__overlap=['thoughts', 'tutorial'])
<QuerySet [<Post: First post>, <Post: Second post>, <Post: Third post>]>
```

Len

```
>>> Post.objects.filter(tags__len=1)
<QuerySet [<Post: Second post>]>
```

ArrayField: Queries

Index Transforms

```
>>> Post.objects.filter(tags__0='thoughts')
<QuerySet [<Post: First post>, <Post: Second post>]>

>>> Post.objects.filter(tags__1__iexact='Django')
<QuerySet [<Post: First post>]>

>>> Post.objects.filter(tags__276='javascript')
<QuerySet []>
```

Slice Transforms

```
>>> Post.objects.filter(tags__0_1=['thoughts'])
<QuerySet [<Post: First post>, <Post: Second post>]>

>>> Post.objects.filter(tags__0_2__contains=['thoughts'])
<QuerySet [<Post: First post>, <Post: Second post>]>
```

ArrayField: Forms

SimpleArrayField
A simple field which maps to an array. It is represented by an HTML `<input>`.

SplitArrayField
This field handles arrays by reproducing the underlying field a fixed number of times.

HStoreField

A field for storing key-value pairs. The Python data type used is a dict that maps *strings* into *nullable strings*.

Example

```
from django.contrib.postgres.fields import HStoreField
from django.db import models

class Dog(models.Model):
    name = models.CharField(max_length=200)
    data = HStoreField()

    def __str__(self):
        return self.name

>>> Dog.objects.create(name='Rufus',
                       data={'breed': 'labrador', 'owner': 'Bob', 'friend': 'Bobby'})
>>> Dog.objects.create(name='Meg',
                       data={'breed': 'collie', 'owner': 'Bob', 'toy': 'yellow ball'})
>>> Dog.objects.create(name='Fred', data={})
```

HStoreField

A field for storing key-value pairs. The Python data type used is a dict that maps *strings* into *nullable strings*.

PostgreSQL extension

```
from django.contrib.postgres.operations import HStoreExtension

class Migration(migrations.Migration):
    ...

    operations = [
        HStoreExtension(),
        ...
    ]
```

HStoreField: Queries

Key Lookups

```
>>> Dog.objects.filter(data__breed='collie')
<QuerySet [<Dog: Meg>]>

>>> Dog.objects.filter(data__breed__contains='l')
<QuerySet [<Dog: Rufus>, <Dog: Meg>]>
```

Contains (@>)

```
>>> Dog.objects.filter(data__contains={'owner': 'Bob'})
<QuerySet [<Dog: Rufus>, <Dog: Meg>]>

>>> Dog.objects.filter(data__contains={'breed': 'collie'})
<QuerySet [<Dog: Meg>]>
```

HStoreField: Queries

Contained By (<@)

```
>>> Dog.objects.filter(data__contained_by={'breed': 'collie', 'owner': 'Bob'})
<QuerySet [<Dog: Meg>, <Dog: Fred>]>

>>> Dog.objects.filter(data__contained_by={'breed': 'collie'})
<QuerySet [<Dog: Fred>]>
```

Has Key (?)

```
>>> Dog.objects.filter(data__has_key='toy')
<QuerySet [<Dog: Meg>]>
```

Has Any Keys (?|)

```
>>> Dog.objects.filter(data__has_any_keys=['toy', 'friend'])
<QuerySet [<Dog: Rufus>, <Dog: Meg>]>
```

HStoreField: Queries

Has Keys (?&)

```
>>> Dog.objects.filter(data__has_keys=['breed', 'owner'])
<QuerySet [<Dog: Rufus>, <Dog: Meg>]>
```

Keys

```
>>> Dog.objects.filter(data__keys__overlap=['breed', 'toy'])
<QuerySet [<Dog: Rufus>, <Dog: Meg>]>
```

Values

```
>>> Dog.objects.filter(data__values__contains=['collie'])
<QuerySet [<Dog: Meg>]>
```

HStoreField: Forms

HStoreField

A field which accepts JSON encoded data for an HStoreField, casting all values (except nulls) to strings. It is represented by an HTML `<textarea>`.

JSONField

A field for storing JSON encoded data. In Python the data is represented in its Python native format: dictionaries, lists, strings, numbers, booleans and None.

Example

```
from django.contrib.postgres.fields import JSONField
from django.db import models

class Dog(models.Model):
    name = models.CharField(max_length=200)
    data = JSONField()

    def __str__(self):
        return self.name

>>> Dog.objects.create(name='Rufus', data={
...     'breed': 'labrador',
...     'owner': {
...         'name': 'Bob',
...         'other_pets': [{'name': 'Fishy'}],
...     },
... })
>>> Dog.objects.create(name='Meg', data={'breed': 'collie'})
```

JSONField: Queries

JSONField shares lookups with HStoreField:

- Contains (@>)
- Contained By (<@)
- Has Key (?)
- Has Any Keys (?|)
- Has Keys (?&)

Key, Index and Path Lookups

```
>>> Dog.objects.filter(data__breed='collie')
<QuerySet [<Dog: Meg>]>

>>> Dog.objects.filter(data__owner__name='Bob')
<QuerySet [<QuerySet <Dog: Rufus>]>

>>> Dog.objects.filter(data__owner__other_pets__0__name='Fishy')
<QuerySet [<Dog: Rufus>]>
```

JSONField: Forms

JSONField

A field which accepts JSON encoded data for a JSONField. It is represented by an HTML `<textarea>`.

Indexes

Hash

Hash indexes can only handle simple equality comparisons.

Operations

=	Equal to

Creation

```
from django.db import migrations

class Migration(migrations.Migration):
    operations = [
        migrations.RunSQL('CREATE INDEX IF NOT EXISTS {index_name} ON '
                          '{table_name} USING HASH ("{field_name}");'),
    ]
```

Binary Tree (B-Tree)

B-trees can handle equality and range queries on data that can be sorted into some ordering.

B-tree indexes can also be used to retrieve data in sorted order.

Operations

<	Less than
<=	Less than or equal to
=	Equal to
>=	Greater than or equal to
>	Greater than

Creation

```python
from django.db import models

class Foo(models.Model):
    bar = models.IntegerField(db_index=True)
```

Generalized Search Tree (GiST)

GiST indexes are not a single kind of index, it is a balanced, tree-structured access method, that acts as a base template in which to implement arbitrary indexing schemes.

Operations

`<<`	Is strictly left of?
`&<`	Does not extend to the right of?
`&>`	Does not extend to the left of?
`>>`	Is strictly right of?
`<<|`	Is strictly below?
`&<|`	Does not extend above?
`|&>`	Does not extend below?
`|>>`	Is strictly above?
`@>`	Contains?
`<@`	Contained in or on?
`~=`	Same as?
`&&`	Overlaps?

Generalized Search Tree (GiST)

GiST indexes are not a single kind of index, it is a balanced, tree-structured access method, that acts as a base template in which to implement arbitrary indexing schemes.

Creation

```
from django.db import migrations

class Migration(migrations.Migration):
    operations = [
        migrations.RunSQL('CREATE INDEX IF NOT EXISTS {index_name} ON '
                          '{table_name} USING GIST (”{field_name}”);'),
    ]
```

Space-partitioned GiST (SP-GiST)

SP-GiST supports partitioned search trees, which facilitate development of a wide range of different non-balanced data structures.

Operations

`<<`	Is strictly left of?
`>>`	Is strictly right of?
`~=`	Same as?
`<@`	Contained in or on?
`<^`	Is below (allows touching)?
`>^`	Is above (allows touching)?

Creation

```
from django.db import migrations

class Migration(migrations.Migration):
    operations = [
        migrations.RunSQL('CREATE INDEX IF NOT EXISTS {index_name} ON '
                          '{table_name} USING SPGIST ("{field_name}");'),
    ]
```

Generalized Inverted Index (GIN)

GIN is designed for handling cases where the items to be indexed are composite values, and the queries to be handled by the index need to search for element values that appear within the composite items.

Operations

<@	Contained in or on?
@>	Contains?
=	Equal to
&&	Overlaps?

Creation

```
from django.db import migrations
from django.contrib.postgres.indexes import GinIndex

class Migration(migrations.Migration):
    operations = [
        migrations.AddIndex(”Foo”, GinIndex(fields=[], name=None)),
    ]
```

Block Range Indexes (BRIN)

BRIN is designed for handling very large tables in which certain columns have some natural correlation with their physical location within the table.

Operations

<	Less than
<=	Less than or equal to
=	Equal to
>=	Greater than or equal to
>	Greater than

Creation

```python
from django.db import migrations
from django.contrib.postgres.indexes import GinIndex

class Migration(migrations.Migration):
    operations = [
        migrations.AddIndex(”Foo”, BrinIndex(fields=[], name=None, pages_per_range=None)),
    ]
```

Conclusion

PostgreSQL as Relational and NoSQL?

Use case: Scraper schema

Problem

Wants to scrape some websites crawling over search results and store all these results. These information must be accesible querying by website, date, and fuzzy matching over item name.

Use case: Scraper schema

Problem

Wants to scrape some websites crawling over search results and store all these results. These information must be accesible querying by website, date, and fuzzy matching over item name.

Schema

```python
from django.contrib.postgres.fields import JSONField
from django.db import models

class Item(models.Model):
    website = models.URLField()
    name = models.CharField(max_length=200)
    timestamp = models.DateTimeField(auto_now_add=True)
    data = JSONField()

>>> Item.objects.annotate(
...     similarity=TrigramSimilarity('name', 'Similar item name'),
... ).filter(
...     similarity__gt=0.5,
...     timestamp__year=2017,
...     website='http://www.example.com'
... ).order_by('-similarity')
```

Use case: Audit schema

Problem

Integrate an audit or logging system into an application. This system generates an entry for each request-response done by users and associates it to they. Entries must contains request and response data and metadata.

Use case: Audit schema

Problem

Integrate an audit or logging system into an application. This system generates an entry for each request-response done by users and associates it to they. Entries must contains request and response data and metadata.

Schema

```python
import datetime

from django.contrib.auth.models import User
from django.contrib.postgres.fields import JSONField
from django.db import models

class Entry(models.Model):
    user = models.ForeignKey(User)
    timestamp = models.DateTimeField(auto_now_add=True)
    request = JSONField()
    response = JSONField()

>>> Entry.objects.filter(user__id=1, timestamp=datetime.datetime.today())
```

Use the right Tool for the Job