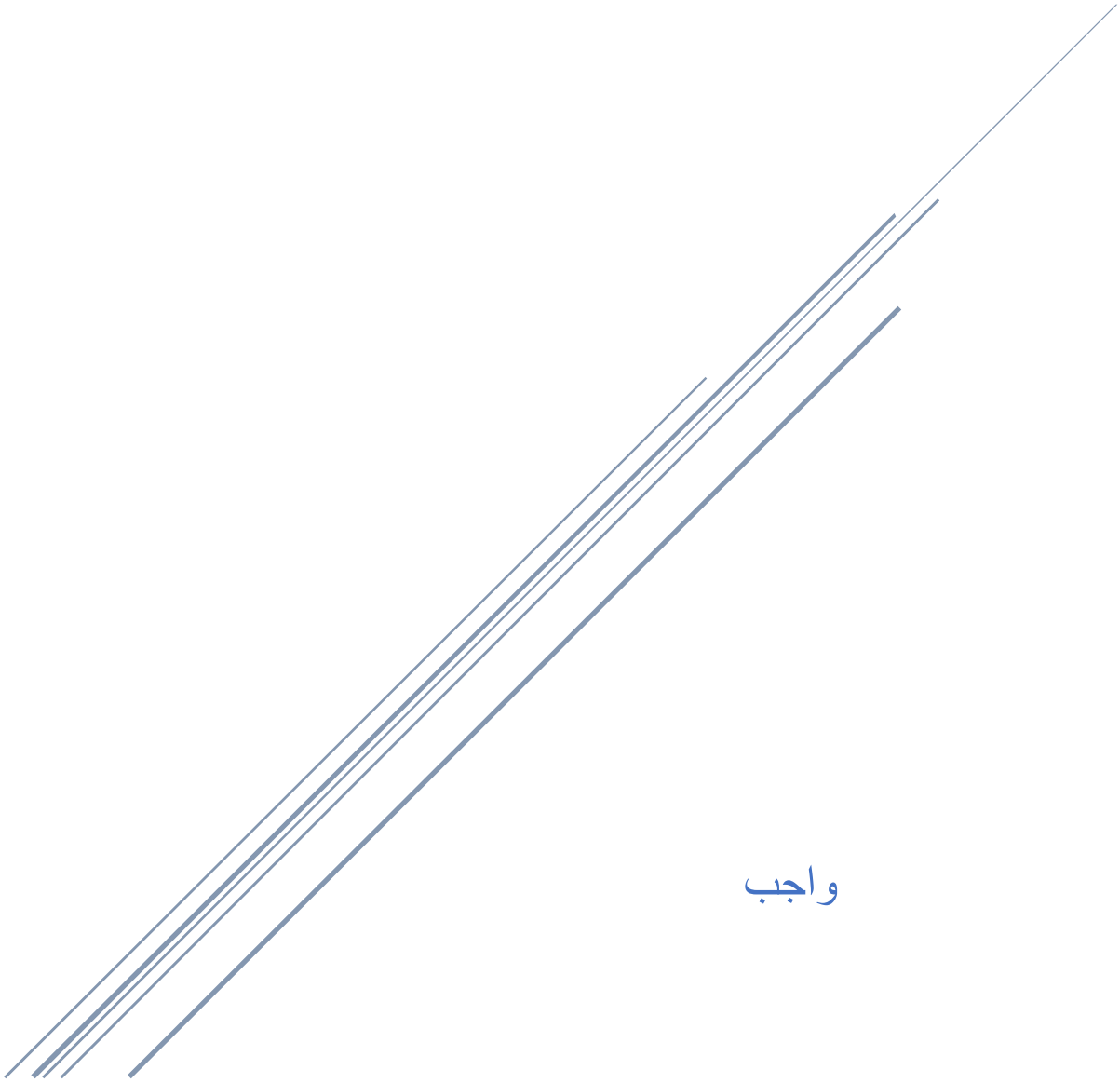
بسم الله الرحمن الرحيم



واجب

اسم الطالب : ريان امين سيف عبد القوي
اسم المقرر :هندسة برمجيات
اسم الأستاذ : مالك المصنف
ايميل : riancomp.ye@gmail.com

بحث شامل عن أنظمة إدارة قواعد البيانات المدعومة في Django

جدول المحتويات

1. المقدمة
2. دور ORM في Django وربطه بقواعد البيانات
3. أنظمة قواعد البيانات المدعومة
    - 3.1 PostgreSQL
    - 3.2 MySQL
    - 3.3 SQLite
    - 3.4 Oracle
4. مقارنة عملية بين أنظمة قواعد البيانات
5. إعداد الاتصال وقواعد التهيئة
6. أفضل الممارسات لتحسين الأداء وإدارة المعاملات
7. التوصيات العملية لاختيار قاعدة البيانات
8. الخاتمة

1.المقدمة

يعد اختيار نظام إدارة قواعد البيانات (DBMS) المناسب أحد القرارات الحاسمة في تطوير التطبيقات باستخدام Django. يدعم Django أربعة أنظمة رئيسية لقواعد البياناتPostgreSQL :، MySQL، SQLite، و Oracle. يوفر إطار العمل طبقة تجريدية قوية (ORM) تسمح للمطورين بالتفاعل مع قواعد البيانات المختلفة باستخدام نفس واجهة البرمجة في بايثون.

2.دور ORM في Django وربطه بقواعد البيانات

2.1ما هوORM ؟

Object-Relational Mapping (ORM)هي تقنية تسمح بتحويل البيانات بين نظام الأنواع غير المتوافقة باستخدام لغات البرمجة كائنية التوجه. فيDjango ، يقوم الـ ORM بتحويل كائنات Pythonإلى جداول وقواعد بيانات علائقية والعكس.

2.2مزايا استخدام ORM فيDjango

- **الاستقلالية عن قاعدة البيانات** :الكتابة مرة واحدة والتشغيل على أنظمة متعددة
- **الأمان** :الحماية من هجمات حقن(SQL Injection) SQL
- **الإنتاجية** :تقليل كود SQL المكتوب يدوياً
- **الصيانة** :كود أنظف وأسهل في الفهم والتعديل

2.3كيفية عمل الربط مع قواعد البيانات

يعتمد Django على محولات قاعدة البيانات (database adapters) للاتصال بالأنظمة المختلفة:

python

```python
# مثال على هيكل اتصال ORM في Django
from django.db import models

class User(models.Model):
    name = models.CharField(max_length=100)
    email = models.EmailField(unique=True)
    created_at = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return self.name
```

.3أنظمة قواعد البيانات المدعومة

3.1 PostgreSQL

الأداء :ممتاز للتحميل الثقيل والبيانات المعقدة
الأمان :متقدم مع دعم التشفير على múltiples مستويات
سهولة الإعداد :متوسطة، يتطلب تثبيت منفصل
التكامل مع :**Django** ممتاز، يدعم جميع ميزات Django بما فيها أنواع البيانات المتقدمة

نقاط القوة:

- دعم أنواع البيانات المتقدمة(JSONB, Array, Hstore)
- أداء عالي في التطبيقات المعقدة
- دعم كامل للمعاملات والتهيئة المتقدمة
- المجتمع النشط والتحديثات المستمرة

نقاط الضعف:

- استهلاك أعلى للذاكرة مقارنة ببعض الأنظمة
- يحتاج إلى ضبط متقدم للوصول إلى أقصى أداء

3.2 MySQL

الأداء :جيد جداً للقراءة، مقبول للكتابة
الأمان :قوي مع خيارات تصريح متقدمة
سهولة الإعداد :سهلة، توثيق ممتاز
التكامل مع :**Django** جيد جداً، مع بعض القيود على أنواع البيانات

نقاط القوة:

- سهل الإعداد والإدارة
- أداء ممتاز في عمليات القراءة
- مجتمع كبير وموارد تعليمية غنية
- متوافق مع معظم ميزاتDjango

نقاط الضعف:

- معالجة محدودة للبيانات غير المنظمة(JSON)
- أداء أقل في العمليات المعقدة مقارنة بـPostgreSQL

3.3 SQLite

الأداء :جيد للتطبيقات الصغيرة، محدود للكبيرة
الأمان :أساسي، مناسب للتطبيقات منخفضة المخاطر
سهولة الإعداد :سهل جداً، لا يتطلب إعداد منفصل
التكامل مع :**Django** كامل، مع بعض القيود في البيئاتProduction

نقاط القوة:

- لا يحتاج إلى خادم منفصل
- مثالي للتطوير والاختبار
- نسخ ونقل قاعدة البيانات سهل
- دعم كامل منDjango

نقاط الضعف:

- أداء محدود في التطبيقات الكبيرة
- قيود على الوصول المتزامن
- غير مناسب للتطبيقات عالية الأداء

3.4 Oracle

الأداء :ممتاز للتطبيقات الضخمة والمؤسسية
الأمان :متقدم جداً بمستويات أمان متعددة
سهولة الإعداد :معقد، يتطلب خبرة متخصصة
التكامل مع Django: جيد، مع بعض المتطلبات الإضافية

نقاط القوة:

- أداء عالي جداً للحمل الثقيل
- أدوات إدارة متقدمة
- موثوقية عالية واستقرار
- دعم فني محترف

نقاط الضعف:

- تكلفة عالية للتراخيص
- إعداد معقد يتطلب خبرة
- ثقيل على الموارد

.4مقارنة عملية بين أنظمة قواعد البيانات

جدول المقارنة الشاملة

المعيار	PostgreSQL	MySQL	SQLite	Oracle
التكلفة	مجاني مفتوح المصدر	مجاني مفتوح المصدر	مجاني مفتوح المصدر	مدفوع (مكلف)
الأداء العام	ممتاز	جيد جداً	جيد (صغير)	ممتاز
القابلية للتوسع	عالية	متوسطة إلى عالية	محدودة	عالية جداً
الأمان	متقدم	قوي	أساسي	متقدم جداً
سهولة الإعداد	متوسطة	سهلة	سهلة جداً	معقدة
دعمJSON	ممتاز(JSONB)	جيد (من الإصدار 5.7)	محدود	جيد
الدعم المجتمعي	ممتاز	ممتاز	جيد	محترف (مدفوع)
الملاءمة للتطوير	جيد جداً	جيد	ممتاز	متوسط
الملاءمة للإنتاج	ممتاز	جيد جداً	محدود	ممتاز

مقارنة حسب سياق الاستخدام

للتطبيقات الصغيرة (مشاريع تعليمية، تطبيقات بسيطة)

- **SQLite**: الخيار الأمثل لسهولة الاستخدام وعدم الحاجة لإعداد منفصل
- **MySQL**: بديل جيد إذا كان هناك احتمال للتوسع مستقبلاً

للمشاريع المتوسطة (تطبيقات ويب، متاجر إلكترونية متوسطة)

- **PostgreSQL**: الخيار الأفضل للأداء والميزات المتقدمة
- **MySQL**: مناسب إذا كان الفريق لديه خبرة سابقة معه

للأنظمة الضخمة) منصاتenterprise ، أنظمة مالية(

- **Oracle**: للحلول المؤسسية ذات المتطلبات العالية
- **PostgreSQL**: البديل المفتوح المصدر القوي للمشاريع الكبيرة

5.إعداد الاتصال وقواعد التهيئة

5.1تهيئةPostgreSQL

python

```python
# settings.py
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'mydatabase',
        'USER': 'mydatabaseuser',
        'PASSWORD': 'mypassword',
        'HOST': 'localhost',
        'PORT': '5432',
        'OPTIONS': {
            'sslmode': 'require',
            'connect_timeout': 30,
        },
        'CONN_MAX_AGE': 600,  # إعادة استخدام الاتصالات لتحسين الأداء
    }
}
```

5.2تهيئةMySQL

python

```python
# settings.py
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'mydatabase',
        'USER': 'mydatabaseuser',
        'PASSWORD': 'mypassword',
        'HOST': 'localhost',
        'PORT': '3306',
        'OPTIONS': {
            'init_command': "SET sql_mode='STRICT_TRANS_TABLES'",
            'charset': 'utf8mb4',
            'connect_timeout': 30,
        },
        'CONN_MAX_AGE': 3600,
    }
}
```

5.3تهيئةSQLite

python

```
# settings.py
DATABASES = {
 'default': {
'ENGINE': 'django.db.backends.sqlite3',
 'NAME': BASE_DIR / 'db.sqlite3',
 'OPTIONS': {
'timeout': 20,
 }
 }
 }
```

5.4تهيئةOracle

python

```
# settings.py
DATABASES = {
 'default': {
 'ENGINE': 'django.db.backends.oracle',
 'NAME': 'myoracleservice',  #格式: host:port/service_name
 'USER': 'myuser',
 'PASSWORD': 'mypassword',
 'HOST': 'localhost',
 'PORT': '1521',
 'OPTIONS': {
 'threaded': True,
 'encoding': 'UTF-8',
 'nencoding': 'UTF-8'
 },
 }
 }
```

.6أفضل الممارسات لتحسين الأداء وإدارة المعاملات

6.1تحسين أداء الاستعلامات

python

```
#استخدم select_related للعلاقات ForeignKey وOneToOne
users = User.objects.select_related('profile').all()

#استخدم prefetch_related للعلاقات ManyToMany وreverse ForeignKey
articles = Article.objects.prefetch_related('tags').all()

#استخدم only و defer للحد من الحقول المسترجعة
```

```
users = User.objects.only('id', 'name', 'email')

#تجنب N+1 query problem
#سيء ) يسبب N+1(
for article in Article.objects.all():
    print(article.author.name)  # استعلام منفصل لكل مؤلف

#جيد (استعلام واحد)
articles = Article.objects.select_related('author')
for article in articles:
    print(article.author.name)  # لا استعلامات إضافية
```

6.2إدارة المعاملات(Transactions)

python

```
from django.db import transaction

#استخدام transaction.atomic لإدارة المعاملات
@transaction.atomic
def transfer_funds(sender_id, receiver_id, amount):
    sender = Account.objects.select_for_update().get(id=sender_id)
    receiver = Account.objects.select_for_update().get(id=receiver_id)

    if sender.balance < amount:
        raise ValueError("رصيد غير كافي")

    sender.balance -= amount
    receiver.balance += amount

    sender.save()
    receiver.save()

#المعاملات اليدوية
def complex_operation():
    try:
        with transaction.atomic():
            # عمليات database هنا
            operation1()
            operation2()
            # إذا وصلنا إلى هنا، يتم commit التغييرات
    except Exception as e:
        # يتم rollback تلقائياً
        handle_error(e)
```

6.3الفهرسة لتحسين الأداء

python
```
class Article(models.Model):
    title = models.CharField(max_length=200, db_index=True)
    content = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)
    author = models.ForeignKey(User, on_delete=models.CASCADE)

    class Meta:
        indexes = [
            models.Index(fields=['created_at', 'author']),
            models.Index(fields=['title'], name='title_idx'),
        ]
```

6.4استخدام قاعدة البيانات للتحقق من الصحة

python
```
class Product(models.Model):
    name = models.CharField(max_length=100)
    price = models.DecimalField(
        max_digits=10,
        decimal_places=2,
        validators=[MinValueValidator(0)]  # التحقق من أن السعر غير سالب
    )
    stock = models.IntegerField(
        validators=[MinValueValidator(0)]  # التحقق من أن المخزون غير سالب
    )

    class Meta:
        constraints = [
            models.CheckConstraint(
                check=models.Q(price__gte=0),
                name='price_non_negative'
            ),
            models.CheckConstraint(
                check=models.Q(stock__gte=0),
                name='stock_non_negative'
            )
        ]
```

7.التوصيات العملية لاختيار قاعدة البيانات

7.1معايير الاختيار

1. **حجم المشروع والتعقيد**:
   - الصغيرSQLite :
   - المتوسط PostgreSQL :أوMySQL
   - الكبير PostgreSQL :أوOracle
2. **متطلبات الأداء**:
   - عمليات قراءة كثيفةMySQL :
   - عمليات معقدة وكتابة كثيفةPostgreSQL :
   - أنظمةenterprise: Oracle
3. **الخبرة الفنية للفريق**:
   - اختر النظام الذي يجيده فريقك
4. **الميزانية**:
   - مفتوحة المصدرPostgreSQL :، MySQL، SQLite
   - enterprise: Oracle
5. **متطلبات التوسع المستقبلية**:
   - اختر نظاماً يمكنه النمو مع مشروعك

7.2سيناريوهات الاختيار

سيناريو 1: تطبيق تعليمي أو نموذج أولي

- **الاختيار الموصى به**SQLite :
- **السبب** :لا يحتاج إعداداً، سهل النقل والتطوير

سيناريو 2: متجر إلكتروني متوسط

- **الاختيار الموصى به**PostgreSQL :
- **السبب** :دعم أنواع البيانات المتقدمة، أداء عالي في العمليات المعقدة

سيناريو 3: منصة محتوى (مدونة، موقع أخبار)

- **الاختيار الموصى به**MySQL :

- **السبب** :أداء ممتاز في القراءة، انتشار واسع

سيناريو 4: نظام مالي أو مؤسسي

- **الاختيار الموصى به** Oracle :أوPostgreSQL
- **السبب** :موثوقية عالية، دعم للمعاملات المعقدة

7.3نصائح للهجرة بين أنظمة قواعد البيانات

1. **استخدم django-db-backup** لعمل نسخ احتياطية
2. **اختبر الاستعلامات** على النظام الجديد قبل الهجرة
3. **استخدم أدوات Django** مثل `inspectdb`لفهم الاختلافات
4. **نفذ الهجرة في بيئة 试测أولاً**
5. **خطط لفترة توقف** محتملة للخدمة

8.الخاتمة

يقدم Django دعمًا قويًا لمجموعة متنوعة من أنظمة قواعد البيانات، كل منها له ميزاته وعيوبه. يعتمد الاختيار المناسب على متطلبات المشروع المحددة، والخبرة الفنية للفريق، والميزانية، وخطط التوسع المستقبلية.

من خلال الاستفادة من قوة ORM فيDjango ، يمكن للمطورين بناء تطبيقات قوية وقابلة للصيانة مع المرونة للتبديل بين أنظمة قواعد البيانات إذا لزم الأمر. يبقى فهم خصائص كل نظام ومتى يجب استخدامه عاملاً حاسماً في نجاح المشاريع البرمجية.

التوصية العامة :يعد PostgreSQL الخيار الأفضل لمعظم مشاريع Django بسبب دعمه الشامل لميزاتDjango ، والأداء العالي، والمجتمع النشط، دون تكاليف ترخيص.