Оптимизация производительности

Теоретическая часть

- Мониторинг потребления ресурсов;
- Профилирование backend;
- Приемы оптимизации: кеширование, денормализация, изменение архитектуры.

Мониторинг потребления ресурсов

top

Консольная команда, которая выводит список работающих в системе процессов и информацию о них.

- `PID` — идентификатор процесса
- `USERNAME` — пользователь, от которого запущен процесс
- `SIZE` — размер процесса (данные, стек и т. д.) в килобайтах
- `RES` — текущее использование оперативной памяти
- `VIRT` — полный объем виртуальной памяти, которую занимает процесс

iotop

iotop -- утилита, выводящая данные по использованию жесткого диска.

```
# Выведим наиболее активные процессы.
iotop -o
# Собрать статистику за определённое время
iotop -o -a
```

iostat

Утилита, предназначенная для мониторинга использования дисковых разделов.

```
iostat -d -t -p sda -x
```

- `-c` — вывести отчёт по CPU;
- `-d` — вывести отчёт по использования диска;
- `-t` — интервал, за который усредняются значения и вычисляются "средние" значения в секундах;

Профилирование backend

Для чего нужно профилирование?

- Позволяет найти “узкие места” в вашем коде;
- Чем быстрее код, тем больше работы за единицу времени;

Для чего нужно профилирование?

- Основные способы - замеры
  - CPU
  - Память
  - Частота/продолжительность вызовов функций
- Методы
  - Статистический метод (сэмплирование)
  - Инструментирование

Python profiler

- cProfile — относительно новый (с версии 2.5) модуль, написанный на C и оттого быстрый;
- profile — нативная реализация профайлера (написан на чистом питоне), медленный, и поэтому не рекомендуется к использованию;
- hotshot — экспериментальный модуль на си, очень быстрый, но больше не поддерживается и в любой момент может быть удалён из стандартных библиотек;

Flame graph

- Метод визуализации собранных фреймов стека;
- Введены в обиход Бренданом Греггом (Brendan Gregg);
- Помогают понять общую картину выполнения приложения;
- Работает с разными формата результатов (perf, DTrace и т.д.).

Flame graph. Установка

```
# Установить библиотеку Python flamegraph
pip3 install flamegraph
# Получить логи профилирования для .
python3 -m flamegraph -o perf.log your_script.py <args>

# Получить svg.
./flamegraph.pl --title "MyScript CPU" perf.log > perf.svg
```

Оптимизация ORM

- select_related();
- prefetch_related();
- values;

select_related()

Возвращает QuerySet, который автоматически включает в выборку данные связанных объектов при выполнении запроса.

- \+ Повышает производительность;
- − Увеличивает(иногда значительно) объем получаемых данных;

- Можно указывать глубину через параметр depth;
- Можно указывать название полей;

select_related(). Примерчик

```
class City(models.Model):
    # ...
    pass

class Person(models.Model):
    # ...
    hometown = models.ForeignKey(City)

class Book(models.Model):
    # ...
    author = models.ForeignKey(Person)
```

select_related(). Примерчик

```
b = Book.objects.select_related().get(id=4)
p = b.author          # Нет обращения к БД.
c = p.hometown        # Нет обращения к БД.

b = Book.objects.get(id=4)
p = b.author          # Обращение к БД.
c = p.hometown        # Обращение к БД.
```

prefetch_related()

Возвращает QuerySet, который получает “за один подход” связанные объекты для каждого из указанных параметра поиска.

- \+ Повышает производительность;
- − Увеличивает(иногда значительно) объем получаемых данных;

- Можно указывать глубину через параметр depth;
- Можно указывать название полей;

prefetch_related(). Примерчик

```
class Topping(models.Model):
    name = models.CharField(max_length=30)

class Pizza(models.Model):
    name = models.CharField(max_length=50)
    toppings = models.ManyToManyField(Topping)

    def __unicode__(self):
        return u"%s (%s)" % (self.name, u", ".join([topping.name
                                                    for topping in self.toppings.all()]))
```

prefetch_related(). Примерчик

```
# Будет выполнен запрос к таблице
# Toppings для каждого объекта Pizza.
Pizza.objects.all()

# Все соответствующие начинки(toppings)
# будут получены одним запросом.
Pizza.objects.all().prefetch_related('toppings').
```

values()

Возвращает ValuesQuerySet — подкласс QuerySet, который возвращает словари с результатом вместо объектов моделей.

- Помогает разграничивать получаемые данные (не грузить тяжёлые объекты без необходимости);
- Каждый словарь представляет объект, ключи которого соответствуют полям модели;
- Принимает дополнительные позиционные аргументы, *fields, которые определяют какие поля будут получены через SELECT;

values(). Примерчик

```
>>> Blog.objects.values()
[{'id': 1, 'name': 'Beatles Blog', 'tagline':
             'All the latest Beatles news.'}],
>>> Blog.objects.values('id', 'name')
[{'id': 1, 'name': 'Beatles Blog'}]
```

Кеширование

Виды кеширования

Кэширование означает сохранение результатов дорогостоящего вычисления, чтобы избежать его повторного вычисления в следующий раз.

- Memcached;
- Кэширование в базу данных;
- Кэширование на файловую систему;
- Кэширование в оперативной памяти;

Memcached

- Все данные хранятся прямо в оперативной памяти;
- Работает как демон и захватывает определённый объём оперативной памяти;
- Нет никакой дополнительной нагрузки на базу данных или файловую систему.

Установка Memcached

```
# Установить библиотеку Python
# для работы с memcached
pip3 install python-memcached
# Установить пакет memcached.
sudo apt-get install memcached

# Запустить демона memcached
# По умолчанию порт 11211
# /etc/memcached.conf
systemctl start memcached
```

Настройка Memcached в Django

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': [
            '172.19.26.240:11211',
            '172.19.26.242:11211',
        ]
    }
}
```

Кэширование в базу данных

```
# Создать таблицу в БД для кэша
python manage.py createcachetable [cache_table_name]
# Заполнить секцию CACHES в settings.py
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.db.DatabaseCache',
        'LOCATION': 'my_cache_table',
    }
}
```

Кэширование на файловую систему

```
# Заполнить секцию CACHES в settings.py
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',
        'LOCATION': '/var/tmp/django_cache',
    }
}
```

- Путь до каталога должен быть абсолютным;
- удостовериться, что указанный каталог существует и доступен для чтения и записи для пользователя, от которого работает ваш веб сервер.

Кэширование в оперативной памяти

```
# Заполнить секцию CACHES в settings.py
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',
        'LOCATION': 'unique-snowflake'
    }
}
```

- Каждый процесс будет работать со своим собственным экземпляром кэша.

Денормализация

Когда нужна денормализация?

В запросах к полностью нормализованной базе нередко приходится соединять до десятка, а то и больше, таблиц. А каждое соединение — операция весьма ресурсоемкая.

- Денормализация путем сокращения количества таблиц.;
- Денормализация путём ввода дополнительного поля в одну из таблиц.

Когда нужна денормализация?

В запросах к полностью нормализованной базе нередко приходится соединять до десятка, а то и больше, таблиц. А каждое соединение — операция весьма ресурсоемкая.

- Денормализация путем сокращения количества таблиц.;
- Денормализация путём ввода дополнительного поля в одну из таблиц.

Домашнее задание

- Использовав профайлер, сделать замеры и определить самые ресурсоёмкие методы (3 балла)
- Настроить memcached (3 балла)
- Добиться загрузки первой страницы за 500мс (3 балла);
- Добиться времени отклика всех API вызовов в 200мс (2 балла)

Срок сдачи: следующее занятие.