HighLoad++

Rails Scale: 1000 запросов в секунду

Макс Лапшин

max@evilmartians.com

http://evilmartians.ru/

Задача:

оптимизация приложения вконтакте

Вводные

- 30 тыс пользователей
- до 9 секунд на запрос
- 5 серверов
- надо опустить время ответа до 500 мс

Результаты

- Более 2-х млн пользователей
- 25 мс на запрос
- 14 серверов
- 40K RPM и 20 млн записей в сутки

С чем столкнулись

- Ежедневная смена требований
- Экспоненциальный рост нагрузки
- Поровну записи и чтения
- Сделать быстро, дешево и приемлемо

Что оказалось важным в нашем случае

Персонал

Грамотный менеджер

«Щасспрошу» завалит проект

Персонал

Системный администратор.

Получше, чем «aptitude-джан»

Персонал

Наша команда злых марсиан!

http://evilmartians.ru/

Волшебных гномиков нет.

Нет их даже в MongoDB и memcached

Сразу выкинули

- pgpool — master-master медленный
- memcached — нечего кешировать

Оставили

- Ruby on Rails — нужна гибкость
- PostgreSQL — часто меняется схема
- RabbitMQ — задержка записи
- внешний инструментарий

Что мы делали

Профилирование

- Без него никуда
- Догадки не работают
- newrelic.com
- Фоновые задачи очень важны

Мониторинг

- Место на дисках
- Упавшие серверы
- Длины очередей
- Ночной дежурный (?)

SQL база

- Нужны реляционные выборки
- Часто меняются критерии
- PostgreSQL быстр и удобен
- Индексы — основной дисковый IO

Шардинг

- Много данных рядом — плохо
- Нам повезло с логикой выборок
- Шардинг: user_id % 100
- Надо планировать заранее

Ruby on Rails

- Меньше всего проблем
- Zero-downtime deploy с unicorn-ом
- Плохая поддержка шардинга
- Необходимость RabbitMQ

RabbitMQ

- Самая быстрая часть проекта
- Оказался индикатором состояния
- Мучительное восстановление

Выводы

- Rails do scale
- Масштабирование — вопрос предметной области
- У вас всё будет по-другому