

# HighLoad для начинающих



Конференция разработчиков  
высоконагруженных систем



# HighLoad для начинающих

Dmitry E. Oboukhov

31 октября 2014

# HighLoad, что это?

- ▶ Конференция?

# HighLoad, что это?

- ▶ Конференция?
- ▶ Высокая нагрузка?

# HighLoad, что это?

- ▶ Конференция?
- ▶ Высокая нагрузка?
- ▶ Миф?!

# Высокая нагрузка что это?

# Высокая нагрузка что это?

- ▶  $53.328 * 10^9$  запросов в секунду?

# Высокая нагрузка что это?

- ▶  $53.328 * 10^9$  запросов в секунду? - средний CPU



# Высокая нагрузка что это?

- ▶  $53.328 * 10^9$  запросов в секунду? - средний CPU
- ▶ Более реалистично!

# Высокая нагрузка что это?

- ▶  $53.328 * 10^9$  запросов в секунду? - средний CPU
- ▶ Более реалистично!
- ▶ 1 запрос в секунду?

# Высокая нагрузка что это?

- ▶  $53.328 * 10^9$  запросов в секунду? - средний CPU
- ▶ Более реалистично!
- ▶ 1 запрос в секунду? - любой веб сервер справится?...

# Высокая нагрузка что это?

- ▶  $53.328 * 10^9$  запросов в секунду? - средний CPU
- ▶ Более реалистично!
- ▶ 1 запрос в секунду? - любой веб сервер справится?...  
перекодирующий видеоролики? :)

Высокая нагрузка это:

Высокая нагрузка это:

Нагрузка, с которой не справляется  
железо



Высокая нагрузка это:

Нагрузка, с которой не справляется  
железо



# Когда это бывает?

## Достигнуты технические ограничения

- ▶ Сеть
- ▶ Память
- ▶ CPU
- ▶ Хранилище

## Дополнительно

- ▶ Недоиспользование железа
- ▶ Трудности масштабирования



Причина

# Архитектурные проблемы



# Недоиспользование железа

Рассмотрим типичный вебсервер на Perl, Python, Ruby...

# Недоиспользование железа

Рассмотрим типичный вебсервер на Perl, Python, Ruby...

## Задачи одного цикла

- ▶ Чтение запроса из сети.
- ▶ Парсинг запроса http.
- ▶ Валидация запроса, выбор контроллера.
- ▶ Запрос(ы) к хранилищу данных.
- ▶ Формирование ответа (template).
- ▶ Отправка ответа клиенту.

# Недоиспользование железа

Рассмотрим типичный вебсервер на Perl, Python, Ruby...

## Задачи одного цикла

- ▶ Чтение запроса из сети.
- ▶ Парсинг запроса http.
- ▶ Валидация запроса, выбор контроллера.
- ▶ Запрос(ы) к хранилищу данных.
- ▶ Формирование ответа (template).
- ▶ Отправка ответа клиенту.

## Традиционная реализация (apache)

- ▶ один процесс на один цикл
- ▶ один тред на один цикл

# Недоиспользование железа

## Начались разговоры о HighLoad?

- ▶ Увеличение числа процессов/тредов.
- ▶ Увеличение числа серверов.

# Недоиспользование железа

## Вернемся к рассматриваемому серверу

- ▶ Проблемы наступили при  $\approx 100$  запросах в секунду.
- ▶ Увеличили число процессов в работе.
  - Помогло.
- ▶ Новые проблемы при  $\approx 150$  запросов в секунду.
- ▶ Дальнейшее увеличение числа процессов помогает слабо.

# Недоиспользование железа

Что делать?



# Недоиспользование железа

Что делать?

- ▶ Переписывать?



# Недоиспользование железа

Что делать?

- ▶ Переписывать?
  - Мы над этим 3 года работали!

# Недоиспользование железа

## Что делать?

- ▶ Переписывать?
  - Мы над этим 3 года работали!
- ▶ Добавлять второй сервер?

# Недоиспользование железа

## Что делать?

- ▶ Переписывать?
  - Мы над этим 3 года работали!
- ▶ Добавлять второй сервер?
  - Это тоже не просто!  
(бизнеслогика)

## Спокойно!

- ▶ Провести измерения.
- ▶ Провести анализ архитектуры.
- ▶ Найти слабые места.

# Недоиспользование железа

## Измерения

Чтение запроса из сети.	15K RPS
Парсинг запроса, контроллер.	150K RPS/CPU
Запросы к хранилищу.	60K RPS
Формирование ответа	100K RPS/CPU
Отправка ответа клиенту.	15K RPS

## Недоиспользование железа

Итого

$$\frac{1}{\frac{1}{15 \cdot 10^3} + \frac{1}{150 \cdot 10^3} + \frac{1}{60 \cdot 10^3} + \frac{1}{100 \cdot 10^3} + \frac{1}{15 \cdot 10^3}} = 6000 RPS$$

# Недоиспользование железа

Итого

$$\frac{1}{\frac{1}{15 \cdot 10^3} + \frac{1}{150 \cdot 10^3} + \frac{1}{60 \cdot 10^3} + \frac{1}{100 \cdot 10^3} + \frac{1}{15 \cdot 10^3}} = 6000 RPS$$

Но, позвольте!

- ▶ У нас проблемы на 150 RPS!
- ▶ Тут что-то не так!

# Недоиспользование железа

## Начинаем разбираться

- ▶ Хранилище выходит на свои RPS при достаточно большом числе соединений к нему.
- ▶ Либо хранилище надо располагать локально. неприемлемо.
- ▶ То же самое и с взаимодействием с клиентом.

# Недоиспользование железа

## Резюме ситуации, еще раз

- ▶ Имеется 100500 строк кода, над которым работали несколько лет.
- ▶ Этот код AS IS по результатам измерений может выдавать гораздо больше RPS чем в реальности.
- ▶ Проблемы начинаются на уровне RPS на порядок меньших, нежели расчетные.



# Недоиспользование железа

## Работа сервера

- ▶ Запросу выделяется CPU.
- ▶ Ожидаются данные запроса.
- ▶ Диспетчеризация.
- ▶ Запросы в БД.
- ▶ Ожидаются ответы из БД.
- ▶ Формируется ответ.
- ▶ Ответ отправляется клиенту.

# Недоиспользование железа

## Работа сервера

- ▶ Запросу выделяется CPU.
- ▶ Ожидаются данные запроса.
- ▶ Диспетчеризация.
- ▶ Запросы в БД.
- ▶ Ожидаются ответы из БД.
- ▶ Формируется ответ.
- ▶ Ответ отправляется клиенту.

## Магазин обуви

- ▶ Клиенту выделяется менеджер.
- ▶ Сопровождает клиента.
- ▶ Может сходить на склад за нужным размером.
- ▶ Ожидает товара со склада/решения клиента.
- ▶ Пробивает чек на кассе.
- ▶ Прощается с клиентом.

# Недоиспользование железа

## Измеряем

Ожидание запроса (данных) от пользователя.	70 мкс
Парсинг запроса, валидация.	6 мкс
Формирование запроса (запросов) в БД.	1 мкс
Ожидание ответа (ответов) из БД.	16 мкс
Соединение данных из БД с шаблоном.	10 мкс
Ожидание отправки данных клиенту.	70 мкс

# Недоиспользование железа

## Итого

- ▶ Код выполнялся:  $6 + 1 + 10 = 17$  мкс
- ▶ Чего-либо ожидали:  $70 + 16 + 70 = 156$  мкс

# Недоиспользование железа

## Итого

- ▶ Код выполнялся:  $6 + 1 + 10 = 17$  мкс
- ▶ Чего-либо ожидали:  $70 + 16 + 70 = 156$  мкс
- ▶ Код выполняется только 10% времени!
- ▶ И при этом тормозит!

# Недоиспользование железа

```
#include <unistd.h>

int main(int argc, char **argv) {
    int i;
    for (;;) {
        usleep(70);      usleep(7);
        usleep(17);      usleep(10);
        usleep(70);
    }
}
```

# Недоиспользование железа

## Итого

- ▶ Код, делающий только `sleep` в цикле неплохо грузит CPU  
- по моим измерениям - где-то 15% загрузки на CPU
- ▶ Запустив десяток таких “воркеров”, получаем примерно такую же нагрузку как на проблемном сервере.
- ▶ Понятно что пример синтетический (есть вопросы к реализации `usleep`).

# Недоиспользование железа

## Итого

- ▶ Код, делающий только sleep в цикле неплохо грузит CPU - по моим измерениям - где-то 15% загрузки на CPU
- ▶ Запустив десяток таких “воркеров”, получаем примерно такую же нагрузку как на проблемном сервере.
- ▶ Понятно что пример синтетический (есть вопросы к реализации usleep).

## Вернемся к нашему серверу

- ▶ Каждая отдельная часть имеет хорошую производительность достаточную для развития проекта еще на несколько лет вперед.
- ▶ Большую часть времени (90%) наш код проводит в ожидании.
- ▶ Что делать?



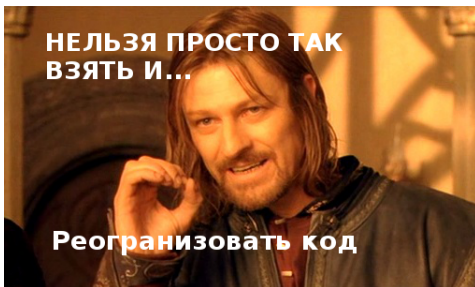
Недоиспользование железа

Просто реорганизовать код



# Недоиспользование железа

## Просто реорганизовать код



# Событийно-ориентированное программирование

*Компьютер — это конечный автомат. Треды для тех людей, которые не умеют программировать конечные автоматы.*

Алан Кокс



# Событийно-ориентированное программирование

## Избавимся от тредов!

- и процессов.



# От магазина обуви к продуктовому!

## Особенности продуктового магазина

- ▶ Клиенты собираются в очереди.
- ▶ Продавец обслуживает клиентов по очереди и без простоя.
- ▶ Простой продавца возможен только при отсутствии клиентов.
- ▶ Один продавец может обслужить в десятки раз больше клиентов.
- ▶ PROFIT?

# Событийно-ориентированное программирование

## Машина событий

- ▶ Исполнитель заявляет о готовности выполнить задачу.  
- подписывается на событие: “Свободная касса!”
- ▶ При появлении задачи исполнитель начинает ее выполнение немедленно.
- ▶ Если задача требует ожидания - ставит ожидающего в очередь и переходит к выполнению другой задачи.

# Событийно-ориентированное программирование

## Машина событий

- ▶ Исполнитель заявляет о готовности выполнить задачу.  
- подписывается на событие: “Свободная касса!”
- ▶ При появлении задачи исполнитель начинает ее выполнение немедленно.
- ▶ Если задача требует ожидания - ставит ожидающего в очередь и переходит к выполнению другой задачи.

## Типичная реализация

- ▶ Подписка на задачу - регистрация callback.
- ▶ Выполнение задачи - вызов callback.
- ▶ Ожидание - return из callback.

# Событийно-ориентированное программирование

## Перестроим наш сервер

- ▶ Ожидание запроса от пользователя.
  - заменится регистрацией callback "пришел запрос от пользователя"
- ▶ Диспетчеризация запроса - не изменится
- ▶ Формирование запросов в БД - не изменится
- ▶ Ожидание ответа из БД.
  - заменится регистрацией callback "пришел ответ из БД"
- ▶ Формирование ответа - не изменится
- ▶ Ожидание отправки данных клиенту.
  - заменится регистрацией callback "данные пользователю отправлены"





# Событийно-ориентированное программирование

## Переписываем одну страничку и изменяем

- ▶ Производительность одного сервера выросла в  $>10$  раз
- ▶ Одного CPU/коннекта к БД достаточно для еще нескольких лет роста нагрузки.
- ▶ Но слишком много переделок!

# Событийно-ориентированное программирование

## Переписываем одну страничку и изменяем

- ▶ Производительность одного сервера выросла в  $>10$  раз
- ▶ Одного CPU/коннекта к БД достаточно для еще нескольких лет роста нагрузки.
- ▶ Но слишком много переделок!

## Проблемы

- ▶ Сохранение контекста между callback
- ▶ Ветвления
- ▶ Обработка исключительных ситуаций

# Событийно-ориентированное программирование

## Что затронуто изменениями

- ▶ Интерфейс с вебсервером - некритично.
- ▶ Интерфейс с БД.
  - критично. Много кода бизнеслогики поехало в callbacks. Сложную логику практически невозможно реализовать. Требуется переписывание 90% проекта.
- ▶ Интерфейс с сетью (если он есть)
  - так же переехал в callbacks.

# Событийно-ориентированное программирование

## Плюсы Обувного магазина

- ▶ Контекст задачи можно хранить на текущем стеке (свалить на менеджера).
- ▶ Процесс можно прервать в любой момент.
- ▶ Произвольное ветвление (можно ходить по разным отделам в произвольном порядке).

## Плюсы продуктового магазина

- ▶ Менеджеры не простаивают

# Событийно-ориентированное программирование

## Соединим плюсы обоих подходов

Пишем свой планировщик (fibers).

- ▶ Невытесняющая многозадачность
  - ▶ Простое порождение “процессов”
  - ▶ Простое API
- Три основных метода
- ▶ Создать процесс (create, async)
  - ▶ Передать управление планировщику (yield, cede)
  - ▶ Разбудить выбранный процесс (wakeup, ready)

# Событийно-ориентированное программирование

## Интеграция планировщика с машиной событий

Структура кода теперь выглядит так:

- ▶ Подписка на событие (регистрация callback).
- ▶ Передача управления планировщику (yield, cede).
- ▶ Событие будит (wakeur) текущий процесс.
- ▶ Процесс обрабатывает события используя данные переданные в callback.

# Событийно-ориентированное программирование

## Интеграция планировщика с машиной событий

Структура кода теперь выглядит так:

- ▶ Подписка на событие (регистрация callback).
- ▶ Передача управления планировщику (yield, cede).
- ▶ Событие будит (wakeur) текущий процесс.
- ▶ Процесс обрабатывает события используя данные переданные в callback.

## Итого

Вернулись к (почти) традиционному виду программы.

# Событийно-ориентированное программирование

## Вернемся к нашему серверу

- ▶ Добавляем машину событий
- ▶ Добавляем библиотеку fibers
- ▶ Переписываем интерфейс с вебсервером (врапперы).
- ▶ Переписываем интерфейс с БД (врапперы).
- ▶ Переписываем другие сетевые обращения (если есть).



# Событийно-ориентированное программирование

## Вернемся к нашему серверу

- ▶ Добавляем машину событий
- ▶ Добавляем библиотеку fibers
- ▶ Переписываем интерфейс с вебсервером (врапперы).
- ▶ Переписываем интерфейс с БД (врапперы).
- ▶ Переписываем другие сетевые обращения (если есть).
- ▶ Итого: переписываем около 5% кода.
- ▶ PROFIT!

# Библиотеки и языки

# Библиотеки и языки

- ▶ Perl

# Библиотеки и языки

- ▶ Perl  
Coro + AnyEvent

# Библиотеки и языки

- ▶ Perl  
Coro + AnyEvent
- ▶ Python

# Библиотеки и языки

- ▶ Perl  
Coro + AnyEvent
- ▶ Python  
fibers + twisted

# Библиотеки и языки

- ▶ Perl  
Coro + AnyEvent
- ▶ Python  
fibers + twisted
- ▶ PHP5

# Библиотеки и языки

- ▶ Perl  
Coro + AnyEvent
- ▶ Python  
fibers + twisted
- ▶ PHP5  
появился оператор yield, fiber



# Что дальше?

- ▶ Используем fiber'ы/event-машины в том языке к которому привыкли
- ▶ Рассматриваем существующие варианты
  - ▶ Node.JS
    - отказались от парадигмы fibers
    - но в последнее время появились подвижки.

# Что дальше?

- ▶ Используем fiber'ы/event-машины в том языке к которому привыкли
- ▶ Рассматриваем существующие варианты
  - ▶ Node.JS
    - отказались от парадигмы fibers
    - но в последнее время появились подвижки.
  - ▶ Tarantool...



# Tarantool

- ▶ Полноценный app-сервер
- ▶ fibers, libev
- ▶ БД на борту
  - in-memory
  - disk
- ▶ Сокеты, диск, http-сервер, очереди

# Недостатки подхода

- ▶ Для больших проектов одного CPU все-таки маловато
- ▶ Реализации fiber'ов для традиционных ЯП плохо масштабируются по CPU/хостам.

# Перспектива

- ▶ Erlang
  - хорошее масштабирование по CPU и хостам
  - очень качественное решение
  - высокий порог вхождения
- ▶ Go
  - более низкий порог вхождения



# Наутакси

Крупнейший бэкэнд Яндекс.Такси.

- ▶ 20.000+ водителей непрерывно шлют свое местоположение
- ▶ > 1 млн заявок на заказы в мес

# Наутакси

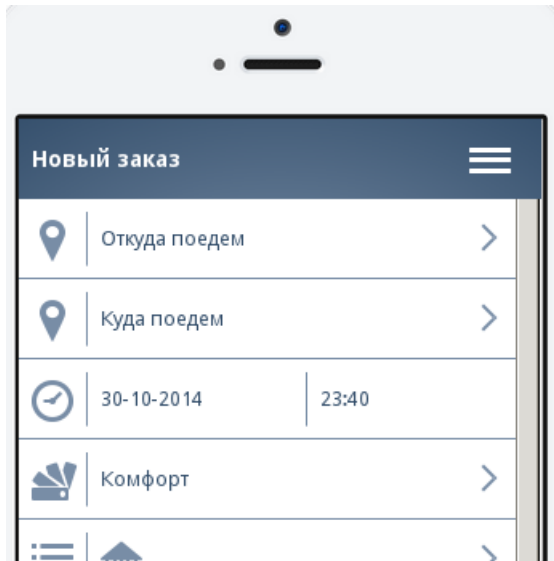
Крупнейший бэкэнд Яндекс.Такси.

- ▶ 20.000+ водителей непрерывно шлют свое местоположение
- ▶ > 1 млн заявок на заказы в мес

Проект включает в себя:

- ▶ Вебсервер на Coro+AnyEvent
- ▶ Большую коллекцию очередей на Tarantool
- ▶ Систему доставки сообщений на Tarantool
- ▶ Хранилище данных на PostgreSQL

# Наутакси



Проект обслуживает:

- ▶ Мобильное приложение клиента
- ▶ Мобильные приложения водителя
- ▶ Систему управления диспетчерскими
- ▶ Мониторинг
- ▶ Все помещается на двух хостах на Hetzner



# Наутакси

## Фи́чи

- ▶ Можно заказать такси (подача от 5 минут)
- ▶ и эвакуатор

## Ссылки

- ▶ [AppleStore://Наутакси](#)
- ▶ [PlayMarket://Наутакси](#)

Дзен

# HighLoad и базы данных

Что делать?

- ▶ Переезжать на другое железо?
- ▶ Переходить на другую БД?

# HighLoad и базы данных

## Что делать?

- ▶ Переезжать на другое железо?
- ▶ Переходить на другую БД?
- ▶ Опять смотреть на архитектуру!

## Вопрос

Что представляет из себя любая БД?

# Философия: что такое БД?

# Философия: что такое БД?

- ▶ Массив записей.

# Философия: что такое БД?

- ▶ Несколько массивов записей.



# Философия: что такое БД?

- ▶ Несколько массивов записей.
- ▶ Средства ускорения поиска по массиву (индексы).

# Философия: что такое БД?

- ▶ Несколько массивов записей.
- ▶ Средства ускорения поиска по массиву (индексы).
- ▶ И...

# Философия: что такое БД?

- ▶ Несколько массивов записей.
- ▶ Средства ускорения поиска по массиву (индексы).
- ▶ И...ВСЕ!

# Виды поисков по массиву

- ▶ Сканирование:  $O(N)$
- ▶ Индексированный:  $O(\ln N)$  (иногда  $O(1)$ ).



# Выбор из двух массивов: JOIN

```
SELECT
    *
FROM
    "users "
JOIN
    "roles" ON "users"."id" = "roles"."uid"
WHERE
    "roles"."id" IN (10, 20, 30)
    AND "users"."id" IN (30, 40, 50)
```

# Выбор из двух массивов: JOIN

- ▶ Выбираются записи первого массива  $K1 \cdot O(\ln N1)$   
 $K1$  определяется на стадии выборки из первого индекса
- ▶ Сопоставляются записям второго:  $K2 \cdot O(\ln N2)$   
На этом шаге становится известным  $K2$
- ▶ Результат фильтруется:  $O(K2)$  по условиям, если таковые еще не отфильтрованы.

Итого

$$K1 \cdot O(\ln N1) + K2 \cdot O(\ln N2) + O(K2)$$

# Пути оптимизации?

- ▶ Попытаться предсказать что меньше  $K1$  или  $K2$
- ▶ Последовательные элементы из индекса иногда можно выбрать за время  $\ln N + K1 \cdot c$
- ▶ Фильтрация на стадии JOIN
- ▶ И... все?!

# Пути оптимизации?

- ▶ Попытаться предсказать что меньше  $K1$  или  $K2$
- ▶ Последовательные элементы из индекса иногда можно выбрать за время  $\ln N + K1 \cdot c$
- ▶ Фильтрация на стадии JOIN
- ▶ И... все?!
- ▶ Денормализация.



# Усложняем задачу

- ▶ Увеличиваем число таблиц в JOIN
- ▶ Вводим дополнительные условия WHERE
- ▶ Даже оценить ресурсы на выполнение запросов становится сложно.

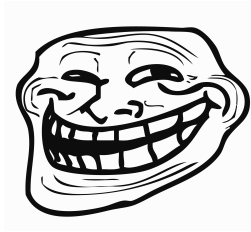


Поговорим об интересном

# Почему популярны NoSQL?

Поговорим об интересном

Почему популярны NoSQL?



# Почему популярны NoSQL?

# Почему популярны NoSQL?

- ▶ Они более быстрые?

# Почему популярны NoSQL?

- ▶ Они более быстрые?

см. соседние доклады: PostgreSQL быстрее MongoDB в 10 раз.

# Почему популярны NoSQL?

- ▶ Они более быстрые?  
см. соседние доклады: PostgreSQL быстрее MongoDB в 10 раз.
- ▶ Позволяют хранить слабоструктурированные данные?

# Почему популярны NoSQL?

- ▶ Они более быстрые?  
см. соседние доклады: PostgreSQL быстрее MongoDB в 10 раз.
- ▶ Позволяют хранить слабоструктурированные данные?  
hstore, arrays, composite появились в PostgreSQL много лет назад.



# Почему популярны NoSQL?

- ▶ Они более быстрые?  
см. соседние доклады: PostgreSQL быстрее MongoDB в 10 раз.
- ▶ Позволяют хранить слабоструктурированные данные?  
hstore, arrays, composite появились в PostgreSQL много лет назад.
- ▶ Шардинг/репликация?

# Почему популярны NoSQL?

- ▶ Они более быстрые?  
см. соседние доклады: PostgreSQL быстрее MongoDB в 10 раз.
- ▶ Позволяют хранить слабоструктурированные данные?  
hstore, arrays, composite появились в PostgreSQL много лет назад.
- ▶ Шардинг/репликация?  
решения для PostgreSQL существуют в избытке.

# Почему популярны NoSQL?

- ▶ Они более быстрые?  
см. соседние доклады: PostgreSQL быстрее MongoDB в 10 раз.
- ▶ Позволяют хранить слабоструктурированные данные?  
hstore, arrays, composite появились в PostgreSQL много лет назад.
- ▶ Шардинг/репликация?  
решения для PostgreSQL существуют в избытке.
- ▶ Может быть NoSQL имеют лучшие возможности по предсказанию плана выполнения запроса?

# Почему популярны NoSQL?

- ▶ Они более быстрые?  
см. соседние доклады: PostgreSQL быстрее MongoDB в 10 раз.
- ▶ Позволяют хранить слабоструктурированные данные?  
hstore, arrays, composite появились в PostgreSQL много лет назад.
- ▶ Шардинг/репликация?  
решения для PostgreSQL существуют в избытке.
- ▶ Может быть NoSQL имеют лучшие возможности по предсказанию плана выполнения запроса?  
большинство NoSQL не умеет JOIN.
- ▶ Что еще?

# Философия: что такое БД?

- ▶ Массив(ы) записей.
- ▶ Индексы.

# Почему популярны noSQL?

# Почему популярны noSQL?

- ▶ Они не умеют JOIN.

# Почему популярны noSQL?

- ▶ Они не умеют JOIN.
- ▶ Это заставляет программиста помнить о том что такое БД.



# Почему популярны noSQL?

- ▶ Они не умеют JOIN.
- ▶ Это заставляет программиста помнить о том что такое БД.
- ▶ Положительно влияет на архитектуру приложения.

# О БД Tarantool

# О БД Tarantool

- ▶ Это не БД.

# О БД Tarantool

- ▶ Это не БД.
- ▶ Это APP-сервер с БД на борту.



# О БД Tarantool

- ▶ Это не БД.
- ▶ Это APP-сервер с БД на борту.
- ▶ JOIN не нужны: денормализация.





Конференция разработчиков  
высоконагруженных систем

