HighLoad++

Практическое создание крупного масштабируемого проекта web 2.0 с нуля

Дмитрий Бородин

Об авторе

В настоящее время – развлекательное приложение «Лице-Мер»:

- **5-е место в рунете по посещаемости** в категории «Общение». Выше по посещаемости - только 3 соц.сети, сами «Вконтакте» и «Мой мир».
- **Более 1.100.000 «уников» в сутки**.
- 20.000 SQL запросов/с.

В прошлом - php.spb.ru (DiMA), первый сайт на русском о PHP (1997г).

HighLoad++

Часть 1.

Приступая к разработке…

Традиционный подход к разработке: ТЗ, функционал, ООП.

Неправильные вопросы: выбор языка, базы, фреймворка.

На фоне **главных задач**, эти – **не существенны**.

HighLoad++

Слово «Highload» обманчиво

Проект, имеющий масштабирование, способен выдержать любую нагрузку, **даже если там ужасный код и не оптимизированный SQL**.

Думайте не о Highload, а об архитектуре.

Проблема первого шага [1]

- Все новое – принимается в штыки.
- Богатый опыт программирования – мешает.

Вас ждет кардинально другой подход к разработке, как администратора и архитектора ПО, а не профессионального программиста.

Проблема первого шага [2]

99% стартапов web 2.0 никогда не запустятся чисто по техническим причинам – проект не был правильно масштабирован с нуля.

Никто не умеет строить крупные web 2.0 проекты кроме тех, кто их уже имеет – запущенные и выжившие социальные сети с млн. пользователей.

При проектировании 100 млн. пользователей не должны вызывать проблем .

Честное горизонтальное масштабирование

Проект разбит на множество независимых компонент, способных выполняться на разных серверах (**честно**).

Все что можно, **заранее** разбито по множеству баз, серверов, очередей и т.д. Ничего на «потом» откладывать нельзя. Думайте об этом заранее.

Проект может наращивать системный администратор без участия программистов и правки кода, под контролем архитектора ПО.

Честное горизонтальное масштабирование

Масштабирование – это не репликация.

SQL кластеры и прочие «серьезные» решения – тупиковые, в масштабировании не применимы.

Часть 2.

Доказательства приведенной теории.

Технические причины, почему ваш стартап гарантированно никогда не запустится.

Примеры типичных страниц в любой социальной сети.

Невозможная задача №1

Мы делаем очередной клон самой популярной соц.сети.

100.000.000 пользовательских профайлов:

- Таблица с профайлами: ФИО, аватар, город…
- Таблица со списком друзей.

Данные разбиты на 200-300 SQL серверов.

Страница «Мои друзья»

Попробуем реализовать эту страницу.

- Список user_id друзей (100 шт.)
- Веб-страница должна показать друзьй по алфавиту: ФИО, аватар, пол, город.
- User_id из списка – случайные. Данные о каждом разбросаны случайно по 200 SQL.

Как получить данные?

Обращаться к 100 SQL серверам – нереально.

Каждая веб-страница может сделать 2-3 SQL запроса максимум.

User_id могут быть вообще случайны: из поиска по анкетам.

Вывод: традиционный подход программирования не применим.

Расширяем проблему

Замените слово «друзья» на другую сущность крупного проекта:

- запись в блоге/форуме
- статья в СМИ
- описание товара в магазине

Невозможная задача №2

На всех социальных сайтах есть подписка на разные события по друзьям. Друг или автор, на которого вы подписаны, сгенерировал некое событие:

- Загрузил фото, статью, файл и т.д.
- Оставил комментарий к существующему объекту (фото, статья и т.д.)

Эти данные должны мгновенно отображаться в вашей ленте новостей.

Вывод

Забудьте про суть проекта, его функционал, назначение и т.д.

Сегодня вы задумали обычный проект. Со временем захотите web 2.0.

Спроектируйте проект на бумаге заранее для всей web 2.0 функций.

Часть 3.

О главном:

- Правильное оперирование данными
- Многопоточность и атомарность в SQL/Memcache
- Отложенная обработка

Memcache – удивительно сложная система

«Невозможная задача»

Какую БД выбрать?

Выбор небольшой – MySQL и PostgreSQL.

Разницы – нет, т.к. используем 1% их возможностей.

Многие крупные проекты используют MySQL.

Стыдится использовать MySQL в серьезных проектах не нужно.

Призыв к той или иной БД – ошибка от непонимания задачи.

Какой язык выбрать?

Ответ: не важно.

Проблема и задачи масштабирования лежат в иной плоскости.

Рекомендую самые **простые и распространенные** инструменты:
PHP, php-fpm, nginx, MySQL, memcache, Redis.

Призыв к выбору языка/фреймворка – ошибка от непонимания задачи.

Тщательность изучения

Без тщательно изучения выбранных инструментов ничего не работает.

Пример:

- Знание memcache на уровне команд set/get – это 0.5% того, что надо.
- Знание наизусть документации (+cas) – это 5% всего, что надо знать.

Остальные 95% опыта (проблем) возникнут **при настоящем Highload**.

О главном [1/3]

Основная задача, о которой нужно думать для создания горизонтально масштабируемого проекта – **как хранить и оперировать данными.**

Тщательный подход не к оптимизации SQL запроса и коду, а к минимизации кол-ва чтения и записи данных.

Основные правила highload так же надо соблюдать.

О главном [2/3]

Очень важно думать о **многопоточности** и **атомарности** операций.

Это нужно применять и в обычном программировании, но в highload без этого вообще ничего не работает.

Главные инструменты решения проблем: SQL и Memcache/Redis.

Как показывает практика, об этом никто не задумывается. А к непонятным редким багам относятся философски.

О главном [3/3]

Последняя из важнейших оптимизаций – все что можно (и нельзя) **отложите в фоновую обработку** cron-скриптам.

Более подробно – чуть позже.

Пример оперирования данными

К тезису «О главном [1]».

Допустим, для решения одной задачи, нужно получать данные рандомно из БД. В таком случае вы займетесь оптимизациями “SELECT … ORDER BY rand()”, настройками БД и т.д.

Суть правильного подхода к высоким нагрузкам – вы не должны хотеть использовать “ORDER BY” вообще.

Не создавайте себе проблемы.

Memcache – это не кеш!

- Полноценная и надежная база данных с уникальными возможностями, недоступными в SQL.
- Никакой многопоточности. Долой эксперименты.
- Полезен в любых проектах, даже не масштабируемых.
- Важнейшая роль в хранении денормализованных данных.
- **Один из инструментов решения «невозможной задачи №1».**

Тонкости Memcache

Memcache – с виду прост. На деле - в 100 раз сложнее, чем вы думаете!

На примере класса для работы с блокировками.

Особая роль кодов возврата (помимо return) и разветвление логики.

Публично доступны классы и статьи – псевдонаучны и поверхностны.

Тема Memcache в Highload – отдельный мастер-класс на 8 часов.

Memcache::CAS()

Вся мощь в атомарности операций вашего кода на PHP.

CAS – это не транзакции и не блокировки, известные по SQL. Минусы транзакций и блокировок все понимают. У CAS их нет (есть другие).

Memcache и сессии

Сессии на PHP блокируют потоки.

Сессии в memcache - типичный антипаттерн и баги.

Идеальный вариант – версионные сессии на CAS():

- Нет блокировок
- Много потоков работает параллельно
- Данные не теряются при одновременной записи

Memcache и большие массивы

По аналогии с сессиями – в одном ключе удобно хранить большие сложные многомерные массивы:

- Много потоков читают и пишут
- Выполняют атомарные операции, без нарушения логики

Memcache Highload

MemcacheDB, как и SQL, упирается в лимиты: iostat, CPU, MEM.

Готовьтесь к дроблению и миграции до начала разработки.

Важная опция: период записи журнала на диск:

- По умолчанию – 5 минут, проект каждые 5 минут висит по 10 секунд.
- Рекомендую – 30 секунд, подписание почти незаметно.

SQL: атомарность и многопоточность

Важность обдумывания многопоточности.

Основное заблуждение: транзакции решают проблемы.

Далее – характерная задача…

Характерная задача

Изначально некий скрипт работал в один поток с неким абстрактным ресурсом. Пример ресурса: отправка SMS (не чаще 1 sms/сек – иначе бан) или проигрывание нескольких нот (случайные мелодии на 5 секунд).

Потом скрипт стали запускать в много потоков.

Как не допустить порчи абстрактного ресурса, если он сам не имеет средства защиты от параллельной подачи ему команд разными потоками?

Нужно очень простое и короткое решение в 2-5 строк кода.

Типичное ошибочное решение

С файлами: прочитать файл, если там нет флага «Занято», записать его туда и начать работу.

С SQL: сначала по SELECT получить флаг и если его нет (либо «Свободно»), записать туда по UPDATE/INSERT флаг «Занято».

Типичный антипаттерн.

Решения задачи

- SQL – **блокировки**: SELECT … FOR UPDATE, LOCK TABLES, GET_LOCK.
- SQL – **транзакции**
- SQL – **псевдоочереди** (Primary Key, Auto_increment или ORDER BY)
- SQL – **атомарность** (UPDATE TABLE SET flag='BUSY' WHERE flag='FREE')
- **Memcache** – атомарность add() или cas()
- **Файловая система**: flock() – причем очень легко незаметно ошибиться! Трюки от атомарных операций с ФС: mkdir и пр.
- **Операционная система**: shared memory и прочие трюки - открыть порт и т.д.

Лучшее решение задачи

- UPDATE … SET flag=1 WHERE flag=0
- SELECT … FOR UPDATE
- Memcache::lock() и unlock() – на основе собственного класса

Лично решите эту задачу 20-ю разными способами.

И вам откроется темная сторона силы.

Статистика по задаче

Из примерно 200 хороших программистов (новичков не приглашали) в год на собеседовании в нашу компанию:

- 60% не ответили вообще (100% допустили типичную ошибку)
- 30% придумали плохие решения: LOCK TABLE или flock()
- 10% придумали решения с атомарностью SQL и/или очередями.
- Ни один человек не назвал SELECT .. FOR UPDATE или memcache.

Вывод: никто не задумывается об этом в обычной жизни.

HighLoad++

Часть 4

Конкретные технические подробности в Highload и масштабировании.

Практика использования SQL и memcache в больших масштабах.

Не используйте “KETAMA” в больших проектах.

Проблемы, затыки, решения, трюки, оптимизация SQL и Memcache

SQL Highload [1]

Все запросы – простейшие выборки по Primary key.

Без JOIN или вложенных запросов.

Доп. индексы в InnoDB могут глючать (опасность).

SELECT count(*) никогда не применяйте, слишком медленно.

Альтернатива для Select count(*).

SQL Highload [2]

Auto_increment замените на Sequence, для возможной репликации SQL.

Репликацию SQL для шардинга не используйте.

Применение Sphinx не нужно, когда можно обойтись простым SQL. Репликация решает задачу по поиску десятков млн. профайлов.

Следите за размерами таблиц. Будьте готовы их разбить.

Это должен уметь делать администратор без участия программиста.

SQL Highload [3]

Проблема **отката транзакций**: запросы на 2 сервера + memcache.

Транзакции – зло. По возможности замените на SQL-атомарность и Memcache::lock() по действиям пользователя.

Снижайте уровень изоляции транзакций, опция transaction_isolation. Необходимо изучить теорию уровней изоляции транзакций.

За годы собеседований на этот вопрос отвечают крайне редко.

SQL Highload [4]

Правильное оперирование данными: **SQL потребляет мало CPU**.
Пропускная способность сервера упирается только в жесткие диски.

Карта спотов и миграция пользователей между серверами.

Нагрузки по серверам заранее не предугадать.

SQL Highload [5]

Очень редко обсуждаемая опция **innodb_flush_log_at_trx_commit**.

- 1 или 2 – часто сохранятся, сильно грузить диск, надежно.
- 0 – реже сохраняться, меньше загрузка диска, не надежно.

Жертвуем надежностью, выжимаем большую скорость.

Остальные советы традиционны: настроить буферы в памяти, отключить кеш запросов, разобраться с методами записи данных и пр.

SQL кеш [1]

Паттерн сохранения в кеш результатов (чтение и обновление).

Типичные массовые ошибки:

- **не правильно составлен ключ/тег** в имени кешируемого объекта
- **забыли уничтожить кеш** при обновлении.

Заведите константу, отключающую кеш по всему проекту с целями отладки. Все сразу заработало? Баг найден за 5 секунд!

SQL кеш [2]

Самый сложный и часто встречаемый баг – это кусок кода, который вы забыли написать.

Согласно статистике, 30% от всех багов – «баг отсутствия кода».

Нет кода – нечего отлаживать. Но баг то есть!

Часто связан с неправильным кешированием.

Полезность unit-тестов для самоконтроля.

Лавинообразные нагрузки

Пусть, **100.000.000** пользователей разбиты в таблицы по 1000 пользователей (это называется «спот»). Таблица – файл на диске. Таблицы – это список профайлов или личных сообщений.

Иногда SQL начинает зависать => пользователи шлют еще запросы **=>** больше PHP-потоков подвисает **=>** подвисают и CRON-скрипты, не обрабатывая других пользователей **=>** SQL прекращает исполнять любые соединения **=> полный паралич всей системы**.

Защита от лавины

Благодаря разбиению пользователей на мелкие пачки (споты), выявляем зависшие из них и зависшие user_id. Для этого мониторим PROCESSLIST всех SQL серверов раз в секунду.

Заносим в черный список зависшие споты и user_id.

Легкая защита: **не принимаем никаких запросов** от user_id. Команда CRON-скриптам игнорировать разбор и возвращать их в очередь.

Железная защита: **убиваем все запросы**, связанные с зависшими спотами.

Причина лавины

Их всего три. Первые – наиболее вероятны.

Физическая. Сервер уперся в лимит по скорости чтения/записи жесткого диска. При попытке обработать очередной спот, таблица которого не в ОЗУ, происходит долгое ожидание жестких дисков. См. `iostat`.

Логическая. Очень большая таблица. Например, спам на 1.000.000 сообщений. Обычно летающие запросы стали слишком тяжелы, CPU+iostat неожиданно взлетелы. Необходимо следить за размерами таблиц.

Редкая. Вы просто уперлись в возможности сервера.

SQL шардинг [1]

Представим, что мы делаем клон самой популярной социальной сети.

Имеется 100.000.000 пользователей: профайлы, сообщения, друзья, сведения о файлах и т.д.

Что такое шардинг.

SQL шардинг [2]

Типы данных.

Пользовательская информация: друзья, профайлы, сообщения, статьи, записи в блогах, файлы. «Социальность», порожденная пользователями.

Общая информация: списки пользователей по признакам, карта спотов, конфиги, логи проекта, cron-мониторинг, служебные сообщения.

Справочники: таблицы для поиска пользователей, каталог общих объявлений, баннерная вертушка, каталог товаров магазина и т.д.

SQL шардинг [3]

Пользовательская информация: хранится по спотам. Споты добавляются по мере роста числа пользователей.

Общая информация: лежит в одной базе. Нагрузки мизерные. В масштабировании не нуждается. Обычные скрипты не читают эту базу.

Справочники: из-за очень интенсивной нагрузки заводим множество SQL-реплик. Поиск даже по десяткам млн. пользователей летает!

SQL шардинг [4]

Друзья, сообщения, профайлы и т.д. – пусть будет всего 50 типов таблиц.

В каждом споте (набор 50 таблиц) хранится информация только о какой-то 1000 пользователей из 100 млн. Итого имеется 100 000 000/1000 = 100 000 спотов, т.е. 100 000*50 = **5 000 000 SQL таблиц**.

Заведем 150 серверов. На каждом по 1 инстансу MySQL: 100 000 000/150 = от 600 000 до 700 000 пользователей, 600 спотов, **30 000 таблиц**.

Проблема переполнения каталога файлами.

Невозможное возможно

Очевидный вывод из шардинга –JOIN невозможен.

Как решить «Невозможную задачу №1», чтобы вывести на экран ФИО, аватар и город абсолютно случайных 100 пользователей из 100 000 000?

Необходимо разместить в супер быстрой памяти только то, что реально нужно: ФИО, аватар и город. Ничего лишнего! Иначе супер быстрая память станет медленной.

Авторайзер [1]

Запустим 150 инстансов MemcacheDB, по числу SQL серверов. В них хранится только важное: ФИО, аватар и т.д.

Имея 100 разных user_id читаем данные с авторайзеров.

Еще разок кешируем.

Главная задача архитектора ПО: пресекать запись ненужных данных, следитьза за iostat, разбивать на более мелкие.

Карта спотов авторайзеров.

Авторайзер [2]

Не храним невосполнимых данных.

Скрипт миграции по серверам и восстановления из SQL.

Имеющиеся бекап системы для memcacheDB не работают.

Ketama в memcache – зло

«Ketama» отлично подойдет небольшому проекту и только для кеша.

Для авторайзеров использовать ее нельзя:

- Сервера не могут умирать и помечаться недоступными.
- Непоследовательное распределения ключей по серверам.

Функцию getMulti эмулируем php-кодом.

Отложенная обработка

90% информации пишем не в SQL и Memcache, а в очереди.

Иерархия очередей.

Зависание нижней очереди не рушит более важный процесс.

Привер. Обновление поля => очередь => запись в авторайзер => очередь => запись в профайл SQL => расчет ТОПов => очередь => статистика, контроль спама и т.д.

Мониторинг очередей, устранение проблем.

HighLoad++

Часть 5.

Технические характеристики [1]

- Более **1.100.000 «уников»** в сутки.
- Около 60.000 онлайн пользователей.
- База пользователей более **8.000.000.**
- Около 100 очень слабых серверов.
- 20.000 SQL/с и 3000 запросов/с на балансере.
- около 100 серверов, 600МБит/с трафик.

Технические характеристики [2]

- Разработка – 3 месяца. **Запуск – 2,5 месяца назад** (6 августа).
- Более месяца проект находится в ТОПе ВКонтакте по посещаемости на **первом месте**.
- Другие проекты: журнал **Вкурсе** и **www.Truegle.ru**

Технические характеристики [3]

PHP 5.3, MySQL, php-fpm, APC/xcache (зависит от версии и глючности PHP), nginx, memcache с патчами.

Очереди – в Redis.

Настоящий php-модуль для работы с Redis (не существует в мире).

Ближайшие планы.

Серийный выпуск масштабируемых проектов.

HighLoad++

Спасибо за внимание!

С удовольствием отвечу на ваши вопросы.

Дмитрий Бородин

Skype: borodin777

dima777@gmail.com