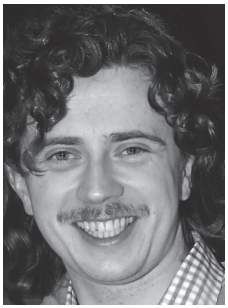
Издательство
Олега Бунина

Учебник по высоким нагрузкам

Highload-инструкторы

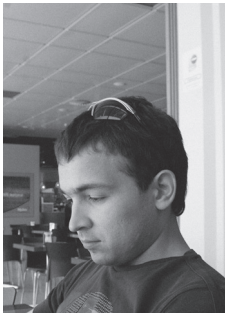


Олег Бунин

Известный специалист по Highload-проектам. Его компания «Лаборатория Олега Бунина» специализируется на консалтинге, разработке и тестировании высоконагруженных веб-проектов. Сейчас является организатором конференции HighLoad++ (www.highload.ru). Это конференция, посвященная высоким нагрузкам, которая ежегодно собирает лучших в мире специалистов по разработке крупных проектов. Благодаря этой конференции знаком со всеми ведущими специалистами мира высоконагруженных систем.

Константин Осипов

Специалист по базам данных, который долгое время работал в MySQL, где отвечал как раз за высоконагруженный сектор. Быстрота MySQL — в большой степени заслуга именно Кости Осипова. В свое время он занимался масштабируемостью MySQL 5.5. Сейчас отвечает в Mail.Ru за кластерную NoSQL базу данных Tarantool, которая обслуживает 500–600 тысяч запросов в секунду.



Максим Лапшин

Решения для организации видеотрансляции, которые существуют в мире на данный момент, можно пересчитать по пальцам. Макс разработал одно из них — Erlyvideo (erlyvideo.org). Это серверное приложение, которое занимается потоковым видео. При создании подобных инструментов возникает целая куча сложнейших проблем со скоростью. У Максима также есть некоторый опыт, связанный с масштабированием средних сайтов (не таких крупных, как Mail.Ru). Под средними мы подразумеваем такие сайты, количество обращений к которым достигает около 60 миллионов в сутки.

Константин Машуков

Руководитель отдела анализа и синтеза компании «Лаборатория Олега Бунина». Константин пришел из мира суперкомпьютеров, где долгое время «пилил» различные научные приложения, связанные с числодробилками. В качестве бизнес-аналитика участвует во всех консалтинговых проектах компании, будь то социальные сети, крупные интернет-магазины или системы электронных платежей.

Академия высоких нагрузок

Впервые опубликовано в журнале Хакер (www.xakep.ru)

Олег Бунин, Максим Лапшин, Константин Осипов и Константин Машуков

Каждый программист хочет стать лучшим, получать все более интересные и сложные задачи и решать их все более эффективными способами. В мире интернет-разработок к таким задачам можно отнести те, с которыми сталкиваются разработчики высоконагруженных систем.

Большая часть информации, опубликованная по теме высоких нагрузок в интернете, представляет собой всего лишь описания технических характеристик крупных систем. Мы же попробуем изложить принципы, по которым строятся архитектуры самых передовых и самых посещаемых интернет-проектов нашего времени.

Учебник по высоким нагрузкам
Урок 1

От авторов

Основным направлением деятельности нашей компании является решение проблем, связанных с высокой нагрузкой, консультирование, проектирование масштабируемых архитектур, проведение нагрузочных тестирований и оптимизация сайтов. В число наших клиентов входят инвесторы из России и со всего мира, а также проекты «ВКонтакте», «Эльдорадо», «Имхонет», Photosight.ru и другие. Во время консультаций мы часто сталкиваемся с тем, что многие не знают самых основ — что такое масштабирование и каким оно бывает, какие инструменты и для чего используются. Эта публикация открывает серию статей «Учебник по высоким нагрузкам». В этих статьях мы постараемся последовательно рассказать обо всех инструментах, которые используются при построении архитектуры высоконагруженных систем.

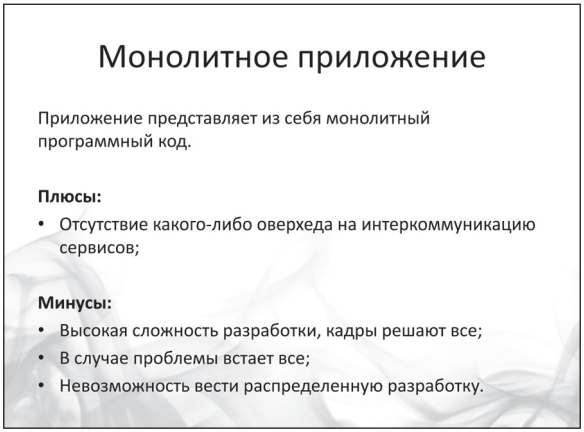
Учебник по высоким нагрузкам. Урок первый

Мы начнем описывать построение архитектуры высоконагруженных систем с самых основ. Не будем рассказывать ни о каких ноу-хау и постараемся не разделять возможные решения на «правильные» и «неправильные». Конечно, у нас есть излюбленные концепции, однако мы планируем дать наиболее полное представление о целом наборе приемов, методов, подходов, схем и инструментов, которые можно использовать.

В рамках нашего цикла мы будем объяснять, зачем нужен тот или иной инструмент, что он умеет делать, в чем может помочь, какие подводные камни могут возникнуть при его использовании, куда дальше копать и т. д.

Монолитные приложения и сервис-ориентированная архитектура

Итак, прежде всего нужно определиться с терминологией, чтобы правильно понимать друг друга. Рассмотрим два принципиально разных подхода к построению архитектуры веб-проектов.

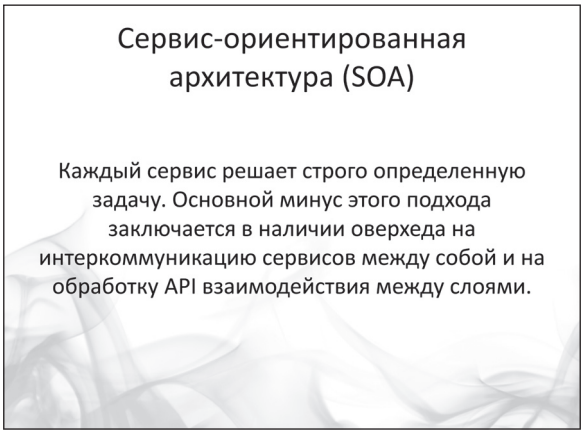


Монолитное приложение — это один большой кусок. Такие приложения могут работать очень быстро, поскольку в них нет никаких оверхедов, связанных с тем, что данные перегоняются из одного блока в другой, от одного сервиса к другому или каким-то образом конвертируются.

Один из самых известных примеров монолитного приложения — крупнейший почтовый сервис CommuniGate Pro. Во всяком случае, несколько лет назад он был монолитным и очень быстро работал. Однако он представлял собой немасштабируемое приложение, которое тем не менее вполне держало миллион пользователей.

Оно было написано одним человеком, и в какой-то момент это стало проблемой. Подключить новых программистов сравнимого уровня оказалось почти невозможным. Это основная беда монолитной архитектуры — большие сложности в масштабируемости разработки. Речь идет именно о распараллеливании разработки, а не о масштабировании программного решения.

Гораздо чаще в интернете используется так называемая сервис-ориентированная архитектура. Она подразумевает разделение программного обеспечения, сайта на некие сервисы, каждый из которых отвечает за что-то одно. При этом все сервисы обмениваются друг с другом данными по определенному протоколу.



Рассмотрим, например, Facebook. Он построен почти классически. Есть различные сервисы, каждый из которых реализует строго определенный набор функций. К примеру, служба со-

общений и ленты новостей (то, что, казалось бы, является ядром сети) представляют собой отдельные сервисы, которые тесно интегрированы. Возьмем сервис авторизации. Мы можем обратиться к нему, передать ему куку и спросить: «Это валидная кука? Если валидная, то кому она принадлежит?»

Наверное, одним из самых известных проектов, при построении которого сервисный подход используется по максимуму, является Amazon. Этот большой интернет-магазин сталкивается с задачами, которые характерны для любого крупного проекта. Когда в компанию пришел новый технический директор, он принял гениальное решение: «Мы будем делать сервисы!» В результате Amazon является не только и не столько крупным интернет-магазином, сколько поставщиком cloud-сервисов. Теперь при создании какого-либо продукта, крупного сайта всегда возникает вопрос: разрабатывать ли собственную систему хранения, или использовать систему от Amazon, которая изначально построена так, чтобы ее можно было купить?

Один из самых больших плюсов сервис-ориентированной архитектуры — возможность вести распределенную разработку. Тут надо понимать, что под распределенной разработкой имеют в виду вовсе не такую разработку, когда члены команды находятся в разных точках земного шара — один в Таиланде, а другой в Москве. Ситуация намного шире. Скажем, вы наняли в Москве пятнадцать программистов, а шестнадцатого нанять не можете. Пока вы нанимаете шестнадцатого, первый уже увольняется. Или другой пример. Когда возникают какие-либо ограничения, связанные с командой, некоторые задачи приходится передавать аутсорсерам. И эта другая команда, состоящая из незнакомых людей, которой вы уже не можете управлять, начинает коммитить что-то прямо в ваш репозиторий, туда же, куда и ваши основные девелоперы. Это проблема, которую очень сложно решить. Именно поэтому «монолитный» подход к монолитному приложению осложняет масштабирование разработки, когда речь, допустим, идет о едином PHP-приложении. Например, обычный CGI script в каком-то смысле является монолитным приложением. Когда нужно, чтобы этот CGI script «пилило» несколько человек, уже начинаются трудности.

Если у вас сервис-ориентированная разработка, вы можете прийти к сторонней команде и сказать: «Ребята, нам надо запилить эту штуку. Мы API продумали, оно должно быть таким». И можно действительно кусок вашего приложения отдать на разработку другим людям. Легко может оказаться, что задача уже давно кем-то решена и доступна в виде готового для использования решения.

Проект именно так и эволюционирует: берется монолитное приложение, разбивается на отдельные сервисы, каждый из которых отвечает за строго определенный набор задач, после чего для этих сервисов задается способ коммуникации. Он может быть каким угодно: от REST API по HTTP до простых запросов к базе данных. Неважно, что именно используется в качестве общей шины.

Другой пример: софт Erlyvideo, разработкой которого занимается Максим Лапшин. Это среднего размера софтина на языке программирования Erlang. Писать подобные видеостриминговые штуки сложнее, чем сайты, потому что надо одновременно держать в голове множество тонкостей. В такие проекты всегда трудно привлекать людей, а на обучение программиста, который бы смог написать то, что не сломало бы код на продакшне при выкатывании, нужно минимум полгода. В случае с «Эрливидео» та самая проблема двух гениев решена инструментально. Инструмент помогает вести параллельную разработку в одном коде, в одном приложении, в одном репозитории. Однако это, скорее, исключение из общего правила, обусловленное компактностью кода.

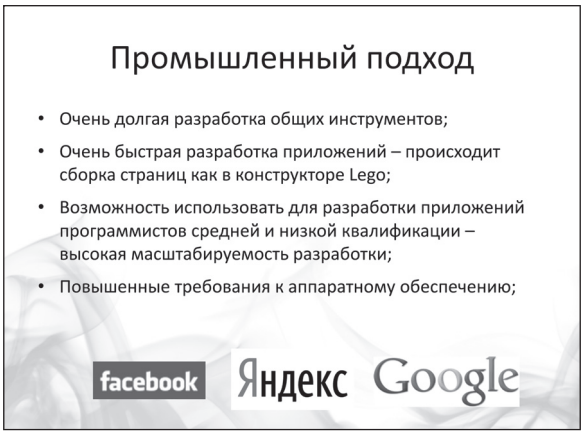
Ремесленный и промышленный подход

Условно говоря, существует два подхода к разработке высоконагруженных систем: ремесленный и промышленный. В чем разница? В том, где разрабатываются средства масштабирования — те инструменты, которые гарантируют, что ваша система будет выдерживать огромные нагрузки.

При использовании промышленного подхода эти средства разрабатываются отдельно от бизнес-логики. При ремесленном подходе средства и бизнес-логика разрабатываются одновременно. И там, и там имеются сервисы. Но в одном случае есть один большой слой, который отвечает за то, что все это будет работать. Объясним на примере.

Разработчики Google в большинстве своем не специализируются на разработке высоконагруженных систем. Если у кого-нибудь из них спросить: «Как ты будешь делать эту систему? Как она будет выдерживать миллионы пользователей?», то, скорее всего, услышишь: «Это очень просто. Я сделаю запрос к системе big data, и она быстро вернет ответ». Он не знает, что происходит внутри — ему это не надо. Если посмотреть на схему, то он работает на «зеленом» уровне и использует уже готовые разработки, сделанные отдельной командой инженеров, которые непосредственно занимаются высокими нагрузками. Так работает большинство крупных компаний.

Хорошим примером приложения, при разработке которого использовался этот подход, может послужить Google+. Это приложение было создано за совершенно фантастический срок в пару месяцев и приняло на себя, наверное, одну из самых чудовищных нагрузок по росту числа пользователей за всю историю интернета.



Аналогичным образом устроено большинство крупных проектов — Facebook, Google, «Яндекс». В «Яндексе» тоже есть эта волшебная шина данных, к которой идут запросы, и «Ян-

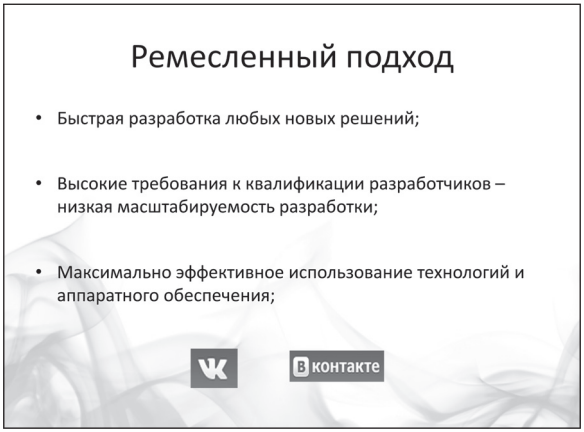
декс» никогда не падает целиком, за исключением тех случаев, когда возникают проблемы с дата-центром. В «Яндексе» падают куски страниц. Почему? Потому что какой-то сервис или какое-то хранилище перестает отвечать. Если, например, упал сервис погоды, то на главной странице будет отображаться все, кроме погоды. Именно при таком подходе одни специалисты отвечают за масштабирование, сборку, коммуникацию, а другие программируют, к примеру, отображение погоды или курса валют.

В качестве примера сайта, созданного с использованием ремесленного подхода, можно привести «ВКонтакте». Когда для «ВКонтакте» разрабатывается какой-либо отдельный сервис, например чат, разработчику передают все полномочия, специально оговаривая, что этот чат должен быть масштабируемым.

Независимо от того, имеются ли общие примитивы по хранению данных и прочее, в целом разработчик самостоятельно принимает решения. Он работает не только над самой бизнес-логикой, но и над ее масштабированием.

Плюсы и минусы ремесленного подхода интуитивно понятны. Если вы хотите использовать такой подход, вам опять же нужны очень хорошие, гениальные программисты, вам нужны люди, которые досконально разбираются во всех тонкостях, в том, что и как работает. Если спросить Павла Дурова, сколько человек из его команды обладает навыками по созданию высоконагруженных систем, он ответит: «Все!»

Помимо высоких требований к квалификации, которые не позволят, например, расшириться в два-три раза, для ремесленного подхода характерно также отсутствие оверхеда на общую шину, по которой гоняются общие данные, и максимально полное и эффективное использование машинных ресурсов. Когда «ВКонтакте» пару лет назад создавал чат по аналогии с Facebook, у него, по-моему, было всего две-три машины с установленным ejabberd-сервером.



Точно так же дело обстоит и с поиском. У «ВКонтакте» отдельный сервис поиска — очень быстрый, его переписывали множество раз. Хотя он очень большой, им занимается всего несколько человек, которые выполняют все необходимые операции вручную. Нет никакого магического сервиса, которому можно сказать: «Отдай индекс и разложи его на 100 серверов». Они делают это сами, руками.

Что касается повышенных требований к аппаратному обеспечению, то, если мы говорим о промышленном подходе, крупному бизнесу куда предпочтительнее вложить заранее известную сумму, пусть даже и очень большую, и получить за нее необходимый рабочий функционал. Некоторые считают, что это слишком дорого. Гораздо дешевле будет нанять дорогих специалистов, которые создадут систему, занимающую не тысячу серверов, а только 20. Те, кто придерживается этой позиции, утверждают, что тысяча серверов — это слишком высокая плата за безболезненное масштабирование, поэтому лучше нанять людей, которые будут следовать ремесленному подходу, используя уникальные инструменты.

Остановимся на еще одном немаловажном преимуществе ремесленного подхода. Дело в том, что крупные компании часто являются пионерами. Допустим, DynamoDB был разработан в Amazon для масштабирования их собственной системы работы со складом. После появления DynamoDB опенсорсным сообществом были написаны Cassandra, Hadoop и так далее. Все они существуют благодаря компаниям, использующим ремесленный подход, которые нуждаются в собственных сервисах типа DynamoDB, но не в состоянии разработать их самостоятельно. Таким образом, подобный обмен происходит постоянно. Google создает какое-то новое решение. Это решение опенсорсится, потом много раз переписывается, после чего его подхватывают десятки компаний, таких как «ВКонтакте», DeNA (Япония) и многие другие. Они делают из этого решения действительно качественный софт, которым могут пользоваться все.

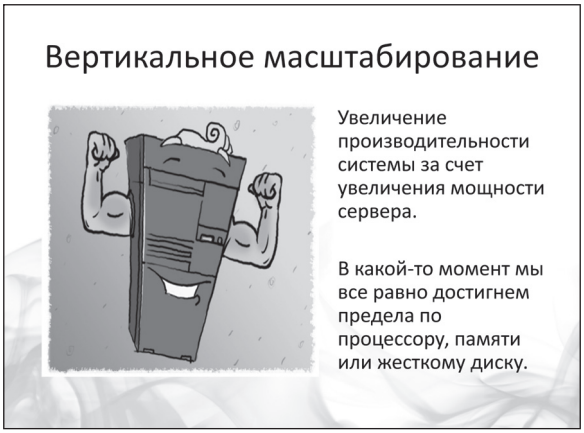
У крупных компаний из-за этого часто возникает предубеждение, которое можно выразить фразой «Все, что сделано не нами, нам не подходит» или «Not invented here». Крупные компании могут себе позволить привлекать профессоров и целые университеты, чтобы они вели какое-либо направление. Например, Google translate курирует знаменитый профессор, помоему, из Стэнфорда, который всю жизнь занимался теорией машинного перевода. То же самое происходит и в Microsoft, и других компаниях такой величины.

Профессионалы есть в компаниях обоих типов. Но это разного рода профессионалы. Профессионалы в тех компаниях, которые практикуют ремесленный подход, действительно знают, какие опенсорсные средства нужно прикрутить, а что написать самим, чтобы все заработало прямо завтра. В больших корпорациях, скорее всего, есть не только гуру, которые сидят и «пилят» big data и web scale, но и огромное количество специалистов, которые занимаются инновациями именно в области usability, новых сервисов и прочего.

Далее мы будем говорить в основном о сервисной архитектуре и об инструментах, которые можно применять в масштабировании.

Масштабирование архитектурного решения

Для начала рассмотрим самые основы — вертикальное и горизонтальное масштабирование. В чем состоит концепция вертикального масштабирования?



Вертикальное масштабирование заключается в увеличении производительности системы за счет увеличения мощности сервера. По сути, при вертикальном масштабировании задача увеличения производительности отдается на аутсорс производителям железа. Специалисты, которые делают большую железку, предлагают некое обобщенное решение, которое будет работать быстрее. Вы заменили машину — и у вас стало работать быстрее.

Какие здесь есть опасности? Один из главных недостатков — вертикальное масштабирование ограничено определенным пределом. Параметры железа нельзя увеличивать бесконечно. В какой-то момент станет нужна уже тысяча серверов. Закупить столько железа будет либо невозможно, либо нецелесообразно. Кроме того, стоимость машин с более высокими характеристиками не обязательно возрастает линейно. Следующая по мощности машина может стоить уже даже не в два раза дороже, чем предыдущая. Поэтому вертикальное масштабирование требует крайне аккуратного планирования.

Альтернативный подход — горизонтальное масштабирование

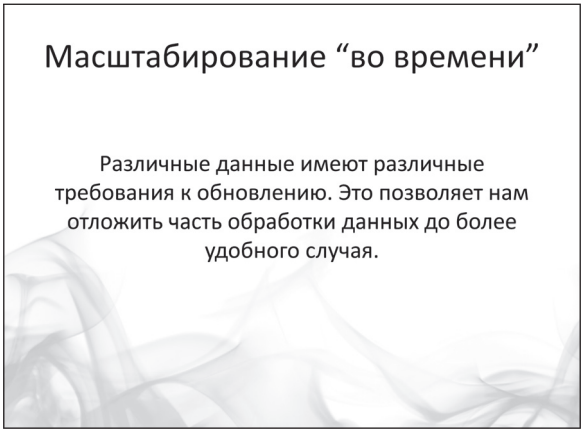
Основной его принцип — мы подключаем дополнительные серверы, хранилища и учим нашу программную систему использовать их все. Именно горизонтальное масштабирование является сейчас фактически стандартом.

Однако на самом деле вертикальная компонента присутствует практически всегда, а универсального горизонтального масштабирования как такового не существует. Известен также такой термин, как диагональное масштабирование. Оно подразумевает одновременное использование двух подходов, то есть вы сразу и покупаете новое железо, тем самым выигрывая время, и активно переписываете приложения. Например, такой подход принят в Stack Overflow.

И еще один способ масштабирования. Как вообще выполняется любое масштабирование, как проектируются высоконагруженные системы? Первое, что необходимо сделать, — это изучить предметную область, как данные движутся внутри системы, как они обрабатываются, откуда и куда текут. Интернет-проект — это в каком-то роде система для различного представления разных данных с разными свойствами.

При этом некоторые данные всегда должны быть актуальными, а на другие можно «забить» и показывать их обновление не сразу. Приведем простейший пример. Пользователь, который постит сообщение в свой блог, должен тут же увидеть это сообщение, иначе он подумает, что что-то сломалось. А во френд-ленте друзей пользователя это сообщение может появиться и через минуту.

Зная такие особенности, можно применять прекрасный метод, который называют отложенной обработкой. Его суть заключается в том, что данные обрабатываются в тот момент, когда это наиболее удобно. Например, пять-шесть лет назад, когда железо было не таким производительным, мы все запускали cron'ы по обработке статистики по ночам. В дальнейшем мы будем говорить в том числе и об инструментах, которые используются для реализации масштабирования во времени, например об очередях.



Трехзвенная структура

Чтобы говорить на одном языке, приведем еще одно определение — определение так называемой трехзвенной структуры системы. Три звена — это фронтенд, бэкенд и хранение данных.

Каждое звено выполняет свои функции, отвечает за различные стадии в обработке запросов и по-разному масштабируется.

Первоначально запрос пользователя приходит на фронтенд. Фронтенды отвечают, как правило, за отдачу статических файлов, первичную обработку запроса и передачу его дальше (своему апстриму, бэкенду). Второе звено, куда приходит запрос (уже предварительно обработанный фронтендом), — это бэкенд. Бэкенд занимается вычислениями — именно он отвечает за то, чтобы вычислить, обработать, переработать, повернуть, перевернуть, перекрутить, смасштабировать и так далее. На стороне бэкенда, как правило, реализуется бизнес-логика проекта.

Следующий слой, который вступает в дело обработки запроса, — это хранение данных, которые обрабатываются бэкендом. Это может быть база данных, файловая система, да и вообще что угодно. В наших статьях мы планируем подробно описать, как масштабируется каждое из этих звеньев. Начнем с фронтенда. Для чего он нужен, мы расскажем в следующем номере. :)

Урок 2
Масштабирование фронтендов

Напомним, на чем мы остановились в прошлый раз. При обработке запросов пользователя и обработке данных на стороне сервера выполняются операции, которые условно можно отнести к трем группам:

- предварительная обработка запроса,
- основные вычисления,
- хранение данных.

В трехзвенной архитектуре за каждое из этих действий отвечает отдельное звено. Предварительную обработку данных обеспечивает фронтенд, основные вычисления — бэкенд, хранение данных — база данных, файловая система, сетевое хранилище или что-то еще.

Фронтенд — первое звено на серверной стороне, которое и начинает обработку запроса. Зачем нужны фронтенды? Как правило, это легкие и быстрые веб-серверы, практически не занимающиеся вычислениями. Программное обеспечение фронтенда принимает запрос; далее если может, то сразу отвечает на него или, если не может, проксирует запрос к бэкенду.

Какие запросы обрабатывает фронтенд и почему?

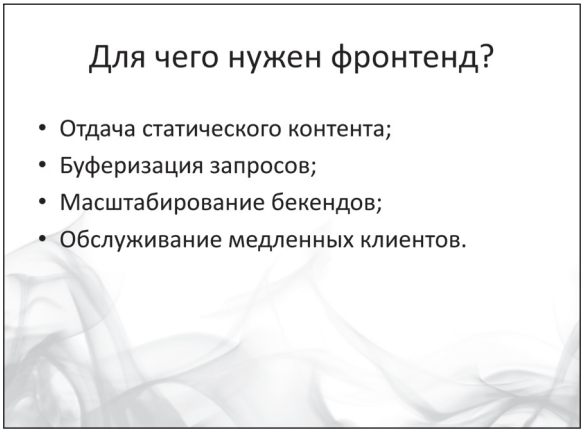
Обычно фронтенд представляет собой легковесный веб-сервер, разработчики которого сделали все для того, чтобы каждый запрос обрабатывался максимально быстро при минимальных затратах ресурсов. Например, у nginx на 10 тысяч неактивных keep-alive-соединений уходит не более 2,5 мегабайт памяти. В правильных веб-серверах даже файлы с дисков отдаются сразу в память, минуя загрузку (такого эффекта можно достичь, включив, например, опцию sendfile в nginx).

Так как фронтенд (в каноническом понимании) не обрабатывает данные, то ему и не нужно большое количество ресурсов на обработку запроса. Однако тяжеловесные PHP- или Perl-процессы с многочисленными загруженными модулями могут требовать по несколько десятков мегабайт на соединение. При разработке самой первой версии nginx шла настоящая борьба за каждый килобайт, выделяемый на обработку запроса. Благодаря этому nginx тратит на обработку запроса около 8–10 килобайт, в то время как mod_perl может распухнуть до 200 мегабайт. Это означает, что на машинке с 16 гигабайтами оперативки удастся запустить всего лишь 40 mod_perl'ов, однако та же самая машинка сможет обрабатывать несколько тысяч легких соединений.

Отдача статики

Правило простое: там, где не нужно отправлять запрос на бэкенд, где не нужно что-либо вычислять (очевидно, что существует класс запросов, для обработки которых это не требуется), все должно отдаваться фронтендом. Отсюда следует первое применение фронтенда — отдача дизайнерской статики, картинок, CSS-файлов, то есть всего, что не требует вычислений. В конфигурационном файле nginx (одно из наиболее удачных решений для фронтенда) вы прописываете, какие именно запросы должны отдаваться с локального диска, а какие передаваться дальше. Наличие фронтендов — это первый признак высоконагруженной системы.

Почему это не просто важно, а очень важно? Посмотрите на любую страницу, например в Facebook. Попробуйте посчитать количество картинок на ней, затем подключаемых CSS- и JavaScript-файлов — счет пойдет на сотни. Если каждый из этих запросов отправлять бэкенду, то никакой памяти и производительности серверов не хватит. Используя фронтенд, мы сокращаем требуемые для обработки запроса ресурсы, причем зачастую в десятки и сотни раз.



Как вариант, фронтенд может отвечать за отдачу хранящихся на диске бинарных данных пользователей. В этом случае бэкенд также не участвует в обработке запроса. Если бэкенд отсутствует, фронтенд напрямую обращается к хранилищу данных.

В качестве примера рассмотрим хранилище видеофайлов пользователей, которое размещено на десяти серверах с большими быстрыми дисками. Запрос на видеофайл пользователя приходит на фронтенд, где nginx (или любая другая аналогичная программа) определяет (например, по URI или по имени пользователя), на каком из десяти серверов лежит требуемый файл. Затем запрос отправляется напрямую на этот сервер, где другой, локальный, nginx выдает искомый файл с локального диска.

В крупных системах таких цепочек nginx'ов или подобных быстрых систем может быть довольно много.

Кеширование

Кеширование — вторая сфера применения фронтенда, некогда очень и очень популярная. В качестве грубого решения можно просто закешировать на некоторое время ответ от бэкенда.

Nginx научился кешировать относительно недавно. Он кеширует ответы от бэкенда в файлы, при этом вы можете настроить и ключ для кеширования (включив в него, например, куки пользователя), и множество других параметров для тонкого тюнинга процесса кеширования.

Соответствующие модули есть у большинства легких веб-серверов. В качестве ключа в этих модулях применяется, как правило, смесь URI- и GET-параметров.

Отдельно стоит упомянуть о потенциальных проблемах кеширования на фронтенде. Одна из них — одновременная попытка вычислить просроченное значение кеша популярной страницы. Если вы кешируете главную страницу, то при сбрасывании ее значения вы можете получить сразу несколько запросов к бэкенду на вычисление этой страницы.

В nginx имеется два механизма для решения подобных проблем. Первый механизм построен на директиве proxy_cache_lock. При ее использовании только первый запрос вычисляет новое значение элемента кеша. Все остальные запросы этого элемента ожидают появления ответа в кеше или истечения тайм-аута. Второй механизм — мягкое устаревание кеша, когда при определенных настройках, заданных с помощью директив, пользователю отдается уже устаревшее значение.

Строго говоря, кеширование на фронтенде — довольно сомнительный прием, ведь вы лишаетесь контроля над целостностью кеша. Вы обновляете страницу, но фронтенд об этом не знает и продолжает отдавать закешированную устаревшую информацию.

Вычислительная логика на стороне клиента

На стороне клиента теперь выполняется огромное количество JavaScript-кода, это способ «размазать» вычислительную логику. Фронтенд отдает браузеру клиента статику, и на стороне клиента проводятся какие-то вычисления — одна часть вычислений. А на стороне бэкенда выполняются более сложные задачи — это вторая часть вычислений.

Для примера приведу все тот же Facebook. При просмотре новостной ленты выполняется огромное количество кода на JavaScript. В «маленьком» браузере работает довольно серьезная «машинка», которая умеет очень многое. Страница Facebook загружается в несколько потоков и постоянно продолжает обновляться. За всем этим следит JavaScript, работающий у вас в браузере. Если вспомнить первый урок, то мы говорили о монолитной архитектуре. Так вот, использовать ее сейчас зачастую невозможно, поскольку приложения выполняются много где: и на стороне клиента, и на стороне сервера и так далее.

Однако попытки создать монолитные приложения, «размазанные» между браузером и сервером, все же предпринимались. Так, в качестве инструмента для написания приложений на Java-сервере, позволяющего прозрачно переносить их на клиентскую сторону, был разработан GWT (Google Web Toolkit).

Сюда также относятся всякие штуки от Microsoft типа Web Forms, которые якобы должны сами генерировать на JavaScript все, что нужно. Тем не менее про все эти решения можно сказать одно: они работают очень мучительно. На данный момент практически не существует легких в использовании и хороших средств, которые волшебным образом избавляли бы от необходимости писать отдельное приложение на JavaScript.

Масштабирование бэкендов

Одна из основных функций фронтенда — балансировка нагрузки между бэкендами, точнее, не столько балансировка, сколько проксирование.

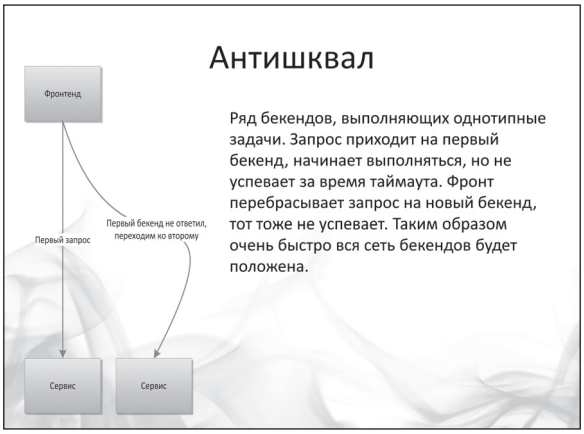
Огромный сайт «ВКонтакте» взаимодействует с внешним миром с помощью 30–40 фронтендов, за которыми скрываются многие тысячи бэкендов, выполняющих вычисления. В настройках фронтендов прописываются так называемые апстримы (upstreams), то есть серверы, куда следует отправлять тот или иной запрос.

Правил для роутинга запросов довольно много. Эти правила позволяют организовать довольно сложную логику перебрасывания запросов. Например, запросы с URI /messages/ отправляются на обработку в кластер серверов для работы с сообщениями, а /photo/ — на фотохостинг и так далее, причем все эти запросы минуют вычисляющие бэкенды.

Иногда встречаются и умные фронтенды, которые учитывают текущую нагруженность бэкендов при проксировании запросов, например, выбирая для проксирования наименее нагруженный бэкенд. Некоторые фронтенд-серверы умеют перезапрашивать другой бэкенд, если первый не смог обработать запрос.

При использовании этих функций стоит учитывать проблему антишквала. В чем она состоит?

Проблема с антишквалом



Допустим, есть ряд бэкендов, выполняющих однотипные задачи. Запрос приходит на первый бэкенд, начинает выполняться, но не успевает до окончания тайм-аута. Умный фронтенд перебрасывает запрос на новый бэкенд, тот тоже не успевает. Таким образом, очень быстро вся сеть бэкендов ляжет.

Варианты решения:

I. Промежуточное звено с очередью, из которого бэкенды сами забирают задачи. Проблемы этого варианта:

1. Смешение подходов — использование асинхронных методов для решения синхронной задачи.
2. Дальнейшее выполнение запроса, когда фронтенд отключился и больше не ждет ответа.
3. Исчезновение задач, которые попали на тормозящий бэкенд (это решается рестартом очереди).

II. Умные запросы от фронтенда:
   1. Первый запрос к первому бэкенду идет с тайм-аутом в одну секунду. Второй запрос идет с тайм-аутом две секунды, третий — три секунды, а четвертого уже нет, то есть мы ограничиваем количество запросов.
   2. Бэкенд может определять, не перегружен ли он (раз в секунду спрашивать LA и кешировать его). В начале обработки запроса выполняется проверка. Если LA слишком высокий, фронтенду отдается Gone Away (штатная ситуация — переход к другому бэкенду).

В любом случае бэкенд получает информацию о том, сколько времени ее ответ будет ждать фронтенд, сколько времени запрос будет актуален.

Медленные клиенты

Перейдем теперь к еще одной из основных сфер применения фронтендов и поговорим о так называемом обслуживании медленных клиентов.

Представьте, что вы заходите на страницу, например, РБК (rbc.ru) и начинаете ее загружать. Страницы у них по одному-два мегабайта. Соединение не очень хорошее — вы на конференции, в роуминге, используете GPRS, — и вот эта страница загружается, загружается, загружается… Раньше такое было повсеместным и сейчас тоже случается, хотя и гораздо реже.

Рассмотрим, как происходит обработка запроса в nginx. Браузер клиента открывает соединение с одним из процессов nginx'а. Затем клиент передает этому процессу данные запроса. Одновременно процесс nginx может обрабатывать еще тысячи других соединений. Для каждого соединения существует свой входной буфер, в который закачивается запрос пользователя.

Только полностью записав запрос в буфер, nginx открывает соединение с противоположной стороной — бэкендом — и начинает проксировать запрос ему. (Если запрос очень большой, то данные в отведенную под буфер память не поместятся и nginx запишет их на диск — это один из параметров для тюнинга nginx.)

Этот же механизм действует и в обратном направлении — фронтенд буферизует ответ, полученный от бэкенда, и потихоньку отдает клиенту.

Если бы пользователь напрямую общался с процессом бэкенда, процесс бы вычислил ответ, причем моментально, за десятую долю секунды, а потом ждал, пока пользователь скушает его по одному килобайту. Все это время процесс бэкенда был бы занят и не принимал бы других запросов.

В том числе и для решения этой задачи устанавливаются легкие фронтенды. Таких фронтендов много, необязательно использовать nginx. Для бэкенда фронтенд выглядит как обычный очень быстрый браузер. Он очень быстро получает ответ от бэкенда, сохраняет этот ответ и потихоньку отдает конечному пользователю, то есть решает пресловутую проблему последней мили. Держать две минуты соединение на фронтенде — это гораздо дешевле, чем держать процесс на бэкенде.

Таким образом, мы описали основные задачи, которые решает фронтенд.

Масштабирование фронтендов

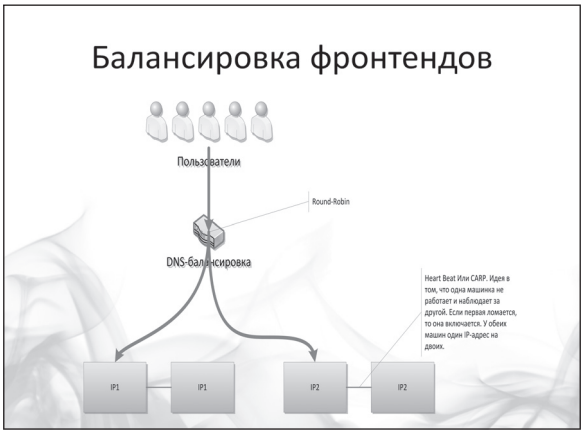
Одним из важнейших условий того, чтобы все работало и проект можно было масштабировать горизонтально, является возможность поставить дополнительные сервера. Обеспечить эту возможность непросто. Каким-то образом вы должны выставить в интернет большое количество серверов и направить пользователей на те из них, которые работают.

Кроме того, увеличение количества серверов вызывает и другие трудности. Допустим, один сервер выходит из строя раз в год. Но при наличии двух серверов сбои будут возникать раз в полгода. Если серверов уже тысяча, неисправности случаются постоянно. На каждом этапе нужно обеспечивать бесперебойную работу системы, когда ломается одна из множества одинаковых деталей.

Когда запрос уже попал в вашу систему (мы говорим про бэкенды и прочее), тут уже вы вольны программировать, как хотите. Но до того, как запрос попадает от фронтенда к бэкенду, он сначала должен попасть на фронтенд. Браузер пользователя должен к какому-то компьютеру послать какие-то данные. Отдельная сложная задача — сделать так, чтобы это было хорошо, просто и надежно.

Она имеет два аспекта. Первый из них — это технология. Раньше, например в 2001 году, технология балансировки была реализована элементарно. Когда вы заходили на spylog.ru, DNS в зависимости от того, откуда вы и кто вы, выдавал вам www1.spylog.ru, или www2.spylog.ru, или www3.spylog.ru. Сегодня большинство веб-сайтов давно уже не прибегает к этому способу. Они используют либо IPVS, либо NAT.

Таким образом, один аспект задачи состоит в том, как послать данные на работающую машину.



Второй аспект заключается в том, как понять, какая машина работает. Для этого необходим мониторинг. В простейшем случае этот мониторинг представляет собой проверку того, отвечает ли машина на ping. При более глубоком рассмотрении оказывается, что сама эта проблема разделяется на несколько других.

Вы начинаете мониторинг. Допустим, вы обнаруживаете, что у машины живая сетевая карта, но сгорел диск. То есть вы хотите сделать балансировку, распределить нагрузку, а машина банально тормозит. Мониторинг и роутинг как раз и позволяют решить задачу балансировки.

DNS-балансировка

Вернемся к первому аспекту — к отправке запроса на ваши фронтенды. Самый простой способ, который используется до сих пор, — это DNS-балансировка, то есть эти несколько машин, куда нужно отправлять пользователя, зашиты в DNS.

Начинать проще всего с TTL в пять или в одну минуту (то есть с минимального, который разумно выставить). Пока у вас три-пять фронтендов, что, на самом деле, тоже немало, это на довольно долгое время убережет вас от проблем. Когда же их больше…

Вы, конечно, можете возразить, что часть провайдеров любит кешировать. В этом случае TTL длительностью пять минут превращается в проблему.

Однако трудности возникнут в любом случае, какое бы решение вы ни выбрали. Если вы, например, от DNS-балансировки перешли к выделенной железке, к IPVS, появятся проблемы с нагрузкой этой железки. Они также могут быть связаны с надежностью дата-центра или дистрибуцией контента. Тут очень много аспектов.

Из всего вышесказанного можно вывести правило, которое применимо при разработке любой крупной системы, — решаем проблемы по мере их появления, каждый раз выбирая наиболее простое решение из всех возможных.

Отказоустойчивость фронтенда

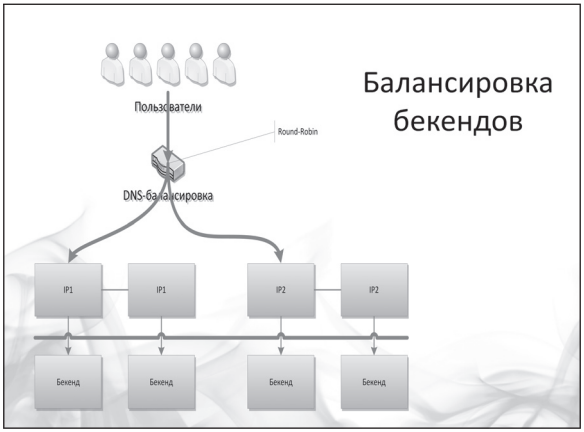
Рассмотрим чуть более сложный способ, который часто используется и имеет кучу вариантов. Как он реализуется? Ставим рядом две машинки, у каждой из которых две сетевых карты. С помощью одной каждая из них «смотрит в мир», с помощью другой они слушают и мониторят друг с друга. Внешние сетевые карты имеют одинаковые IP-адреса. Весь поток идет через первую машину. Как только одна из них умирает, поднимается IP-адрес на второй. Именно так реализованы CARP (во FreeBSD), Heartbeat (в Linux) и другие соединения подобного рода.

Такая схема долгое время работала в Rambler, и, насколько я понимаю, она используется повсеместно. У вас есть DNS-балансировка, разбрасывающая пользователей на пары серверов, в каждой из которых серверы контролируют друг друга.

Перейдем к балансировке бэкендов. Она осуществляется на уровне фронтенда. У него есть простой сервис, который знает все свои так называемые upstream'ы и логику, по которой между ними разбрасываются запросы. В подавляющем большинстве случаев это происходит случайным образом. Но можно задать какие-то обратные связи, посылать запрос не на ближайший upstream, а на тот, который меньше всего нагружен, и так далее.

Речь идет о том, что для множества сайтов хватает всего двух фронтендов, причем с избытком. Nginx — это очень быстрая штука. А бэкенды, которые вы пишете, — это ваша бизнес-логика, и она может работать сколь угодно оптимально или неоптимально. Их, как правило, гораздо больше.

Обычный масштаб чаще всего предполагает наличие двух или четырех фронтендов и двадцати бэкендов. При этом вопрос о том, как отправить запрос тому бэкенду, который лучше всего его обслужит, остается по-прежнему актуальным.



Пожалуй, на этом о масштабировании фронтендов всё. В следующем уроке мы поговорим о том, как масштабировать бэкенды.

Стрелочки на рисунке направлены от фронтендов к бэкендам. Этот рисунок иллюстрирует интересную технологию Mongrel2, используемую в мире Ruby. Ее разработкой занимается известный в Ruby-сообществе Зед Шоу, который и предложил перевернуть привычную схему обработки запросов с ног на голову.

> *Согласно его схеме, не фронтенды ходят к бэкендам и предлагают им обработать какой-то запрос, а наоборот. Фронтенды накапливают у себя очередь на запросы, а огромное количество бэкендов эти фронтенды опрашивает: «Дай чего-нибудь обработать, дай на подумать» — и возвращает ответ. Таким образом, мы получаем масштабируемую асинхронную обработку.*

> *Nginx [engine x], написанный Игорем Сысоевым, объединяет в себе HTTP-сервер, обратный прокси-сервер и почтовый прокси-сервер. Уже длительное время nginx обслуживает серверы многих высоконагруженных российских сайтов, таких как «Яндекс», Mail.Ru, «ВКонтакте» и «Рамблер». Согласно статистике Netcraft, в июне 2012 года nginx обслуживал или проксировал 10,29% самых нагруженных сайтов.*

Урок 3.
Масштабирование бекенда

5.1. Функциональное разделение
5.2. Классическое горизонтальное масштабирование
  5.2.1. Низкая степень связности кода
  5.2.2. Share-nothing для горизонтального масштабирования: плюсы и минусы
  5.2.3. Слоистость кода
  5.2.4. Минимизация использования сложных запросов сразу к нескольким таблицам
5.3. Кеширование
  5.3.1. Общие рекомендации при использовании кеширования
  5.3.2. Проблема инвалидации кеша
  5.3.3. Проблема старта с непрогретым кешем

Начнем наш третий урок, посвященный бизнес-логике проекта. Это самая главная составляющая в обработке любого запроса. Для таких вычислений требуются бэкенды — тяжелые серверы с большими вычислительными мощностями. Если фронтенд не может отдать клиенту что-то самостоятельно (а как мы выяснили в прошлом номере, он без проблем можем сам отдать, к примеру, картинки). На бэкенде отрабатывается бизнес-логика, то есть формируются и обрабатываются данные, при этом данные хранятся в другом слое — сетевом хранилище, базе данных или файловой системе. Хранение данных — это тема следующего урока, а сегодня мы сосредоточимся на масштабировании бекенда.

Сразу предупредим: масштабирование вычисляющих бэкендов — одна из самых сложных тем, в которой существует множество мифов. Облачные вычисления решают проблему производительности — уверены многие. Однако это верно не до конца: для того чтобы вам действительно могли помочь облачные сервисы, вы должны правильно подготовить ваш программный код. Вы можете поднять сколько угодно серверов, скажем, в Amazon EC2, но какой с них толк, если код не умеет использовать мощности каждого из них. Итак, как масштабировать бэкенд?

Функциональное разделение

Самый первый и простой способ, с которым сталкиваются все, — это функциональное разбиение, при котором разные части системы, каждая из которых решает строго свою задачу, разносятся на отдельные физические серверы. Например, посещаемый форум выносится на один сервер, а все остальное работает на другом.

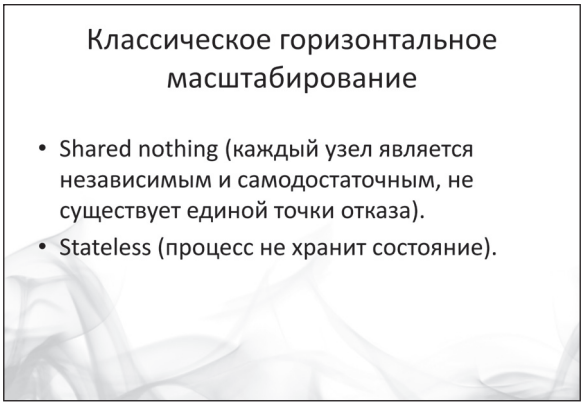
Несмотря на простоту, о подобном подходе многие забывают. Например, мы очень часто встречаем веб-проекты, где используется только одна база MySQL под совершенно различные типы данных. В одной базе лежат и статьи, и баннеры, и статистика, хотя по-хорошему это должны быть разные экземпляры MySQL. Если у вас есть функционально не связанные данные (как в этом примере), то их целесообразно разносить в разные экземпляры баз данных или даже физические серверы. Посмотрим на это с другой стороны. Если у вас есть в одном проекте и встроенная интегрированная баннерокрутилка, и сервис, который показывает посты пользователей, то разумное решение — сразу осознать, что эти данные никак не связаны между собой и поэтому должны жить в самом простом варианте в двух разных запущенных MySQL. Это относится и к вычисляющим бэкендам — они тоже могут быть разными. С совершенно разными настройками, с разными используемыми технологиями и написанные на разных языках программирования. Возвращаясь к примеру: для показа постов вы можете использовать в качестве бэкенда самый обычный PHP, а для баннерной системы вы можете запустить модуль к nginx'у. Соответственно, для постов вы можете выделить сервер с большим количеством памяти (ну PHP все-таки), при этом для баннерной системы память может быть не так важна, как процессорная емкость.

Сделаем выводы: функциональное разбиение бэкенда целесообразно использовать в качестве простейшего метода масштабирования. Группируйте сходные функции и запускайте их обработчики на разных физических серверах.

Обратимся к следующему подходу.

Классическое горизонтальное масштабирование

О том, что такое горизонтальное масштабирование, в принципе, мы уже знаем. Если вашей системе не хватает мощности, вы просто добавляете еще десять серверов, и они продолжают работать. Но не каждый проект позволит провернуть такое. Есть несколько классических парадигм, которые необходимо рассмотреть на раннем этапе проектирования, чтобы программный код можно было масштабировать при росте нагрузки.



Концепции Shared Nothing и Stateless

Рассмотрим две концепции — Shared Nothing и Stateless, которые могут обеспечить возможность горизонтального масштабирования.

Подход Shared Nothing означает, что каждый узел является независимым, самодостаточным и нет какой-то единой точки отказа. Это, конечно, не всегда возможно, но в любом случае количество таких точек находится под жестким контролем архитектора. Под точкой отказа мы понимаем некие данные или вычисления, которые являются общими для всех бэкендов.

Например, какой-нибудь диспетчер состояний или идентификаторов. Другой пример — использование сетевых файловых систем. Это прямой путь получить на определенном этапе роста проекта узкое место в архитектуре. Если каждый узел является независимым, то мы легко можем добавить еще несколько — по росту нагрузки.

Концепция Stateless означает, что процесс программы не хранит свое состояние. Пользователь пришел и попал на этот конкретный сервер, и нет никакой разницы, попал пользователь на этот сервер или на другой. После того как запрос будет обработан, этот сервер полностью забудет информацию об этом пользователе. Пользователь вовсе не обязан все свои следующие запросы отправлять на этот же сервер, не должен второй раз приходить на него же. Таким образом, мы можем динамически менять количество серверов и не заботиться о том, чтобы роутить пользователя на нужный сервак.

Наверное, это одна из серьезных причин, почему веб так быстро развивается. В нем гораздо проще делать приложения, чем писать классические офлайновые программы. Концепция «ответ — запрос» и тот факт, что ваша программа живет 200 миллисекунд или максимум одну секунду (после чего она полностью уничтожается), привели к тому, что в таких распространенных языках программирования, как PHP, до сих пор нет сборщика мусора.

Описанный подход является классическим: он простой и надежный, как скала. Однако в последнее время нам все чаще и чаще приходится отказываться от него.

Критика концепций Shared Nothing и Stateless

Сегодня перед вебом возникают новые задачи, которые ставят новые проблемы. Когда мы говорим про Stateless, это означает, что каждые данные каждому пользователю мы заново тащим из хранилища, а это подчас бывает очень дорого. Возникает резонное желание положить какие-то данные в память, сделать не совсем Stateless. Это связано с тем, что сегодня веб становится все более и более интерактивным. Если вчера человек заходил в веб-почту и нажимал на кнопку «Reload», чтобы проверить новые сообщения, то сегодня этим уже занимается сервер. Он ему говорит: «О, чувак, пока ты сидел на этой страничке, тебе пришли новые сообщения».

Возникают новые задачи, которые приводят к тому, что подход с Shared Nothing и отсутствием состояния в памяти иногда не является обязательным. Мы уже сталкивались неоднократно с ситуациями наших клиентов, которым мы говорим: «От этого откажитесь, положите данные в память» и наоборот «Направляйте людей на один и тот же сервер». Например, когда возникает открытая чат-комната, людей имеет смысл роутить на один и тот же сервер, чтобы это все работало быстрее.

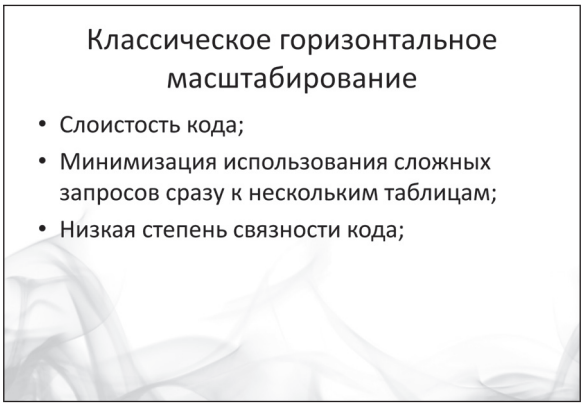
Расскажем про еще один случай, с которым сталкивались. Один наш знакомый разрабатывал на Ruby on Rails игрушку наподобие «Арены» (онлайн драки и бои). Вскоре после запуска он столкнулся с классической проблемой: если несколько человек находятся в рамках одного боя, каждый пользователь постоянно вытаскивает из БД данные, которые во время этого боя возникли. В итоге вся эта конструкция смогла дожить только до 30 тысяч зарегистрированных юзеров, а дальше она просто перестала работать.

Обратная ситуация сложилась у компании Vuga, которая занимается играми для Facebook. Правда, когда они столкнулись с похожей проблемой, у них были другие масштабы: несколько миллиардов SELECT'ов из PostgreSQL в день на одной системе. Они перешли полностью на подход Memory State: данные начали храниться и обслуживаться прямо в оперативной памяти. Итог: ребята практически отказались от базы данных, а пара сотен серверов оказались лишним. Их просто выключили: они стали не нужны.

В принципе, любое масштабирование (в том числе горизонтальное) достижимо на очень многих технологиях. Сейчас очень часто речь идет о том, чтобы при создании сервиса не пришлось платить слишком много за железо. Для этого важно знать, какая технология наибо-

лее соответствует данному профилю нагрузки с минимальными затратами железа. При этом очень часто, когда начинают размышлять о масштабировании, то забывают про финансовый аспект того же горизонтального масштабирования. Некоторые думают, что горизонтальное масштабирование — это реально панацея. Разнесли данные, все разбросали на отдельные серверы — и все стало нормально. Однако эти люди забывают о накладных расходах (оверхедах) — как финансовых (покупка новых серверов), так эксплуатационных. Когда мы разносим все на компоненты, возникают накладные расходы на коммуникацию программных компонентов между собой. Грубо говоря, хопов становится больше. Вспомним уже знакомый тебе пример. Когда мы заходим на страничку Facebook, мощный JavaScript идет на сервер, который долго-долго думает и только через некоторое время начинает отдавать вам ваши данные. Все наблюдали подобную картину: хочется уже посмотреть и бежать дальше пить кофе, а оно все грузится, грузится и грузится. Надо бы хранить данные чуть-чуть «поближе», но у Facebook уже такой возможности нет.

Слоистость кода



Еще пара советов для упрощения горизонтального масштабирования. Первая рекомендация: программируйте так, чтобы ваш код состоял как бы из слоев и каждый слой отвечал за какой-то определенный процесс в цепочке обработки данных. Скажем, если у вас идет работа с базой данных, то она должна осуществляться в одном месте, а не быть разбросанной по всем скриптам. К примеру, мы строим страницу пользователя. Все начинается с того, что ядро запускает модуль бизнес-логики для построения страницы пользователя. Этот модуль запрашивает у нижележащего слоя хранения данных информацию об этом конкретном пользователе. Слою бизнес-логики ничего не известно о том, где лежат данные: закешированы ли они, зашардированы ли (шардинг — это разнесение данных на разные серверы хранения данных, о чем мы будем говорить в будущих уроках), или с ними сделали еще что-нибудь нехорошее. Модуль просто запрашивает информацию, вызывая соответствующую функцию. Функция чтения информации о пользователе расположена в слое хранения данных. В свою очередь, слой хранения данных по типу запроса определяет, в каком именно хранилище хранится пользователь. В кеше? В базе данных? В файловой системе? И далее вызывает соответствующую функцию нижележащего слоя.

Что дает такая слоистая схема? Она дает возможность переписывать, выкидывать или добавлять целые слои. Например, решили вы добавить кеширование для пользователей. Сделать это в слоистой схеме очень просто: надо допилить только одно место — слой хранения данных. Или вы добавляете шардирование, и теперь пользователи могут лежать в разных базах данных. В обычной схеме вам придется перелопатить весь сайт и везде вставить соответствующие проверки. В слоистой схеме нужно лишь исправить логику одного слоя, одного конкретного модуля.

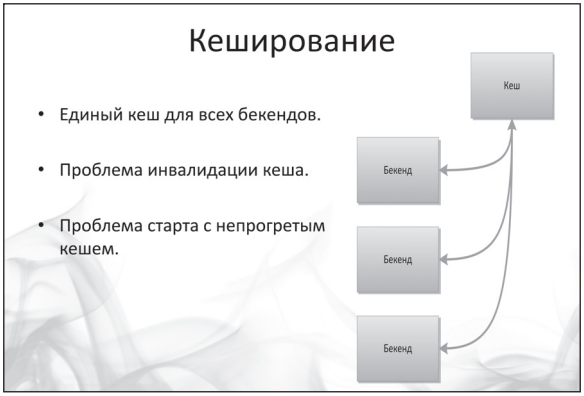
Связность кода и данных

Следующая важная задача, которую необходимо решить, чтобы избежать проблем при горизонтальном масштабировании, — минимизировать связность как кода, так и данных. Например, если у вас в SQL-запросах используются JOIN'ы, у вас уже есть потенциальная проблема. Сделать JOIN в рамках одной базы данных можно. А в рамках двух баз данных, разнесенных по разным серверам, уже невозможно. Общая рекомендация: старайтесь общаться с хранилищем минимально простыми запросами, итерациями, шагами.

Что делать, если без JOIN'а не обойтись? Сделайте его сами: сделали два запроса, перемножили в PHP — в этом нет ничего страшного. Для примера рассмотрим классическую задачу построения френдленты. Вам нужно поднять всех друзей пользователя, для них запросить все последние записи, для всех записей собрать количество комментариев — вот где соблазн сделать это одним запросом (с некоторым количеством вложенных JOIN'ов) особенно велик. Всего один запрос — и вы получаете всю нужную вам информацию. Но что вы будете делать, когда пользователей и записей станет много и база данных перестанет справляться? По-хорошему надо бы расшардить пользователей (разнести равномерно на разные серверы баз данных). Понятно, что в этом случае выполнить операцию JOIN уже не получится: данные-то разделены по разным базам. Так что придется делать все вручную. Вывод очевиден: делайте это вручную с самого начала. Сначала запросите из базы данных всех друзей пользователя (первый запрос). Затем заберите последние записи этих пользователей (второй запрос или группа запросов). Затем в памяти произведите сортировку и выберите то, что вам нужно. Фактически вы выполняете операцию JOIN вручную. Да, возможно вы выполните ее не так эффективно, как это сделала бы база данных. Но зато вы никак не ограничены объемом этой базы данных в хранении информации. Вы можете разделять и разносить ваши данные на разные серверы или даже в разные СУБД! Все это совсем не так страшно, как может показаться. В правильно построенной слоистой системе большая часть этих запросов будет закеширована. Они простые и легко кешируются — в отличие от результатов выполнения операции JOIN. Еще один минус варианта с JOIN: при добавлении пользователем новой записи вам нужно сбросить кеши выборок всех его друзей! А при таком раскладе неизвестно, что на самом деле будет работать быстрее.

Кеширование

Следующий важный инструмент, с которым мы сегодня познакомимся, — кеширование. Что такое кеш? Кеш — это такое место, куда можно под каким-то ключом положить данные, которые долго вычисляют. Запомните один из ключевых моментов: кэш должен вам по этому ключу отдать данные быстрее, чем вычислить их заново. Мы неоднократно сталкивались с ситуацией, когда это было не так и люди бессмысленно теряли время. Иногда база данных работает достаточно быстро и проще сходить напрямую к ней. Второй ключевой момент: кэш должен быть единым для всех бэкендов.



Второй важный момент. Кеш — это скорее способ замазать проблему производительности, а не решить ее. Но, безусловно, бывают ситуации, когда решить проблему очень дорого. Поэтому вы говорите: «Хорошо, эту трещину в стене я замажу штукатуркой, и будем думать, что ее здесь нет». Иногда это работает — более того, это работает очень даже часто. Особенно когда вы попадаете в кэш и там уже лежат данные, которые вы хотели показать. Классический пример — счетчик количества друзей. Это счетчик в базе данных, и вместо того, чтобы перебирать всю базу данных в поисках ваших друзей, гораздо проще эти данные закешировать (и не пересчитывать каждый раз).

Для кеша есть критерий эффективности использования, то есть показатель того, что он работает, — он называется Hit Ratio. Это отношение количества запросов, для которых ответ нашелся в кеше, к общему числу запросов. Если он низкий (50–60%), значит, у вас есть лишние накладные расходы на поход к кешу. Это означает, что практически на каждой второй странице пользователь, вместо того чтобы получить данные из базы, еще и ходит к кешу: выясняет, что данных для него там нет, после чего идет напрямую к базе. А это лишние две, пять, десять, сорок миллисекунд.

Как обеспечивать хорошее Hit Ratio? В тех местах, где у вас база данных тормозит, и в тех местах, где данные можно перевычислять достаточно долго, там вы втыкаете Memcache, Redis или аналогичный инструмент, который будет выполнять функцию быстрого кеша, — и это начинает вас спасать. По крайней мере, временно.

Проблема инвалидации кеша

Но с использованием кеша вы бонусом получаете проблему инвалидации кеша. В чем суть? Вы положили данные в кэш и берете их из кеша, однако к этому моменту оригинальные данные уже поменялись. Например, Машенька поменяла подпись под своей картинкой, а вы зачем-то положили одну строчку в кэш вместо того, чтобы тянуть каждый раз из базы данных. В результате вы показываете старые данные — это и есть проблема инвалидации кеша. В общем случае она не имеет решения, потому что эта проблема связана с использованием данных вашего бизнес-приложения. Основной вопрос: когда обновлять кеш? Ответить на него подчас непросто. Например, пользователь публикует в социальной сети новый пост — допустим, в этот момент мы пытаемся избавиться от всех инвалидных данных. Получается, нужно сбросить и обновить все кеши, которые имеют отношение к этому посту. В худшем случае, если человек делает пост, вы сбрасываете кэш с его ленты постов, сбрасываете все кеши с ленты постов его друзей, сбрасываете все кеши с ленты людей, у которых в друзьях есть те, кто в этом сообществе, и так далее. В итоге вы сбрасываете половину кешей в системе. Когда Цукерберг публикует пост для своих одиннадцати с половиной миллионов подписчиков, мы что — должны сбросить одиннадцать с половиной миллионов кешей френдлент у всех этих subscriber'ов? Как быть с такой ситуацией? Нет, мы пойдем другим путем и будем обновлять кэш при запросе на френдленту, где есть этот новый пост. Система обнаруживает, что кеша нет, идет и вычисляет заново. Подход простой и надежный, как скала. Однако есть и минусы: если сбросился кэш у популярной страницы, вы рискуете получить так называемые race-condition (состояние гонок), то есть ситуацию, когда этот самый кэш будет одновременно вычисляться несколькими процессами (несколько пользователей решили обратиться к новым данным). В итоге ваша система занимается довольно пустой деятельностью — одновременным вычислением n-го количества одинаковых данных.

Один из выходов — одновременное использование нескольких подходов. Вы не просто стираете устаревшее значение из кеша, а только помечаете его как устаревшее и одновременно ставите задачу в очередь на пересчет нового значения. Пока задание в очереди обрабатывается, пользователю отдается устаревшее значение. Это называется деградация функциональности: вы сознательно идете на то, что некоторые из пользователей получат не самые свежие данные. Большинство систем с продуманной бизнес-логикой имеют в арсенале подобный подход.

Проблема старта с непрогретым кешем

Еще одна проблема — старт с непрогретым (то есть незаполненным) кешем. Такая ситуация наглядно иллюстрирует утверждение о том, что кэш не может решить проблему медленной базы данных. Предположим, что вам нужно показать пользователям 20 самых хороших постов за какой-либо период. Эта информация была у вас в кеше, но к моменту запуска системы кэш был очищен. Соответственно, все пользователи обращаются к базе данных, которой для построения индекса нужно, скажем, 500 миллисекунд. В итоге все начинает медленно работать, и вы сами себе сделали DoS (Denial-of-service). Сайт не работает. Отсюда вывод: не занимайтесь кешированием, пока у вас не решены другие проблемы. Сделайте, чтобы база быстро работала, и вам не нужно будет вообще возиться с кешированием. Тем не менее даже у проблемы старта с незаполненным кешем есть решения:

1. Использовать кеш-хранилище с записью на диск (теряем в скорости);
2. Вручную заполнять кэш перед стартом (пользователи ждут и негодуют);
3. Пускать пользователей на сайт партиями (пользователи все так же ждут и негодуют).

Как видите, любой способ плох, поэтому лишь повторимся: старайтесь сделать так, чтобы ваша система работала и без кеширования.

Урок 4. Масштабирование во времени

В прошлых уроках мы говорили о том, как писать программы так, чтобы их можно было запустить в нескольких экземплярах и тем самым выдерживать большую нагрузку. В этом уроке мы поговорим о еще более интересных вещах — как использовать для построения технической архитектуры знания о бизнес-логике продукта и как обрабатывать данные тогда, когда это нужно, максимально эффективно используя аппаратную инфраструктуру.

Отложенные вычисления

Когда пользователь вводит запрос на сайт, необходимо дать ему ответ, и для этого сначала придется проделать соответствующую работу. Скажем, если человек сделал модификационный запрос (например, создал новый пост), то вам предстоит провернуть огромный объем работы. Недостаточно просто положить пост. Нужно обновить счетчики, оповестить друзей, разослать электронные уведомления. Хорошая новость: делать все сразу необязательно.

Тот же самый Facebook после того, как вы публикуете новый пост, делает еще одиннадцать разных вещей — и это только то, что видно снаружи невооруженным глазом. Причем все эти операции исполняются в разное время. Например, электронное письмо с уведомлением можно послать вообще минут через десять.

Этот принцип мы и возьмем на вооружение. Любой маленький сайт, начиная расти по нагрузке, сталкивается с тем, что, оказывается, больше нельзя делать все необходимые операции синхронно в функции обработки самого модификационного запроса. В противном случае пользователь не получит моментальный ответ.

Современные языки веб-программирования часто не позволяют в явном виде реализовать такой трюк, поскольку в них не предусмотрена возможность делать что-либо после ответа. Однако есть и исключения. В популярном веб-сервере Apache существует более десяти стадий обработки запроса, включая трансляцию URI, авторизацию, аутентификацию и собственно обработку запроса. Сюда же входят и стадии, которые выполняются уже после того, как ответ пользователю отправлен. Некоторые фреймворки (например, mod_perl) позволяют перехватить эти стадии и повесить на них ваши собственные функции. Можно передать в эти функции данные для обработки и спокойно обрабатывать их уже после того, как пользователь получил ответ.

Асинхронная обработка

Однако иногда описанного выше подхода с постобработкой данных недостаточно. Действия, которые надо совершить с ними, могут занимать слишком много времени, а ресурсы вебсервера небезграничны.

В таком случае помогает следующий архитектурный паттерн — сохраните данные в некое промежуточное хранилище, а затем обработайте их с помощью отдельного асинхронного процесса. Термин «асинхронность» означает в общем случае разнесенность во времени. То есть данные собираются сейчас, а обрабатываются тогда, когда будет удобно.

Например, очень часто асинхронно обсчитывается статистика уникальных посетителей — раз в день, как правило по ночам, в часы наименьшей загрузки, запускается скрипт, который берет весь массив данных, накопленных за день, обрабатывает их и сохраняет уже в другом виде. Такой подход применяется при разработке баннерных сетей, счетчиков и других подобных проектов. Часто в часы наименьшей нагрузки выполняются и различные обслуживающие процедуры: оптимизация баз данных, бэкапы.

Обратите внимание — мы уже используем наше знание о бизнес-процессах в проекте. Мы знаем, что детальную статистику за день пользователи готовы подождать, и учитываем этот факт при проектировании архитектуры проекта. Более углубленное использование этих знаний мы с вами разберем дальше.

Очереди

Рассмотрим подробнее инструменты, позволяющие отложить на потом те вещи, которые «не горят». Одно из уже упоминавшихся решений — промежуточное хранилище. Но существует целый класс однотипных задач, для которых хотелось бы, чтобы это хранилище обладало определенными свойствами. Это уже не просто данные для обработки — тут важен порядок, важна очередность.

Для подобных целей используют инструмент, называемый очередями, — особый вид хранилища, поддерживающий логику FIFO (первый вошел — первый вышел). Получаются очереди сообщений и очереди задач, которые надо делать. Например, вместо того чтобы слать e-mail, можно поместить в очередь задачу: «Послать e-mail». Какой-то фоновый скрипт, который запущен, вытянет из очереди новый запрос. «О, надо послать e-mail. Сейчас пошлю его».

Очереди — довольно продвинутый и часто встречающийся инструмент. Например, даже когда пользователь в Windows кликает мышкой на кнопку в приложении, то приложение не принимается немедленно обрабатывать это событие (ведь в этот момент приложение может быть занято другим действием). Операционная система присылает приложению сообщение, содержащее описание совершенного пользователем действия. Сообщение ставится в очередь и будет обработано в порядке поступления. В результате если вы два раза кликните мышкой на два разных пункта меню, то сначала откроется одно, потом другое.

Для использования очередь есть ряд инструментов, один из наиболее популярных сейчас — RabbitMQ (www.rabbitmq.com), написанный на Erlang'е. Причем несмотря на колоссальные возможности по обслуживанию очередей сообщений, начать использовать его крайне просто. Практически для любого языка программирования (PHP, Python, Ruby и так далее) есть готовые библиотеки.

В крупных веб-системах могут использоваться одновременно десятки очередей: очередь на отправку электронной почты, очередь для обновления счетчиков, очередь для обновления френдлент пользователей.

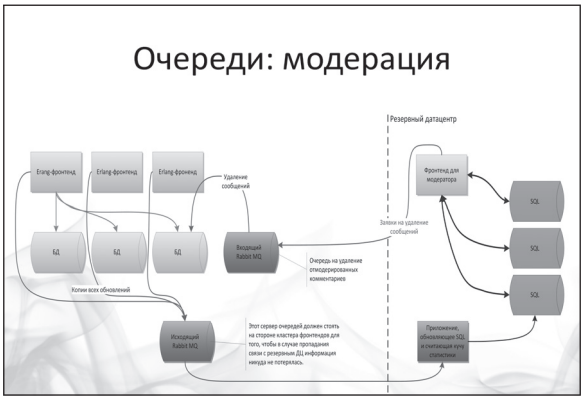
По большому счету очереди — это пример межсервисной коммуникации, когда один сервис (публикация поста) ставит задачи другому сервису (рассылка электронной почты). Рассмотрим это подробнее на конкретном примере.

Пример премодерируемой социальной сети

Рассмотрим пример большого проекта социальной сети уровня Facebook. Пользователи постят посты в огромном количестве, генерируются гигантские объемы различных операций. Но допустим, что это не просто Facebook, а социальная сеть с премодерацией всех сообщений. Задача стала еще сложнее — что делать?

Разработчики посмотрели: посты храним так, комментарии храним так. Всё в разных табличках, может быть даже в базах данных разного вида. Например, этих баз и табличек десять. Неужели модераторский софт должен будет ходить по десятку баз данных, вытаскивать обновления за последние пять минут из всех этих табличек? Объединять все изменения в один большой список и показывать модератору? А что, если модераторов несколько? А если данных очень много?

Вот тут на помощь и приходят очереди. Любой постинг приводит не только к добавлению сообщения в большую базу данных (где оно будет жить постоянно), но и к его попаданию в некую модераторскую систему. Взаимодействие сервисов позволяет навести тут порядок.



Нижний красный блок на слайде — это исходящий сервер очередей RabbitMQ, который получает сообщение. Этот сервер кем-то слушается с той стороны, и в результате приходящие данные перегруппировываются и уже складываются в другую базу данных или в другое место, специально предназначенное для модерации. Например, они группируются, и вместо модерации может идти, например, аналитическая система. Сам бог велел входящие данные преобразовывать, чтобы потом было удобно их обрабатывать и решать данную конкретную узкую задачу.

Но вернемся к нашему примеру. Все эти сообщения каким-то образом просматриваются, и, так как схема базы данных заточена под модерацию, это происходит легко и просто. Нам что-то не нравится — удаляем это сообщение, и запрос точно так же уходит обратно, в другую очередь: «Это сообщение из тех, что ты мне прислал, удали». Есть такой же разборщик, который берет эту задачу с удаленным сообщением, ищет, где же оно там все-таки лежит, и удаляет.

Мы получили классический пример использования очереди.

Неконсистентность данных

Здесь возникают недостатки, характерные для любой системы, которая хранит данные в двух местах. Любая проблема с оборудованием, с выполнением этой сложной функциональности — и возникает неконсистентность данных. Например, сообщение попало в основную базу, но не было добавлено в очередь. Сообщение прошло модерацию, но оно реально удалено не было, потому что очередь, ответственная за хранение сообщений на удаление, вышла из строя.

Чтобы избежать этого, нужно писать программу таким образом, чтобы при повторном ее выполнении она доходила до конца. Этот принцип называется идемпотентностью — повторное действие не изменит наши данные, если в первый раз все было сделано правильно. Увы, это далеко не всегда можно применить.

Другое решение — логическое логирование действий. Создается некий блокнот, например файл, хранящийся на каком-то надежном носителе. Программы при выполнении оставляют там записи вида: «Я успел сделать то, то и то, но еще не успел сделать вот это» и «Я собираюсь сделать это». Если что-то пошло не так, подобный отчет позволит понять, на каком этапе произошел сбой, и исправить положение.

Алгоритм проектирования архитектуры

1. Рассмотрим решение нашей проблемы, исходя из конкретных условий бизнес-логики. Сначала составляем варианты использования проекта (Use cases), функциональное описание, в нашем случае — основные веб-сервисы. Описываем, что конкретно делает пользователь на той или иной странице.

Например, страница news feed пользователя:

- загружаются десять последних объектов-записей по времени объектов, опубликованных всеми пользователями, на которых подписан пользователь;
- для каждой из записей поднимается информация о пользователе-авторе (имя, аватар и ссылка на профиль);
- для каждой из записей на странице поднимаются три последних комментария, по каждому комментарию поднимается аватар и имя пользователя.

Отдельного упоминания заслуживает обсуждение потенциальных сценариев дальнейшего развития проекта. Например, сейчас нет обновления комментариев без перезагрузки, а потом будет — такую возможность нужно предусмотреть еще на этапе начального проектирования. Далее по этим описаниям планируются потоки данных с расчетом потенциальных объемов, требований к скорости в каждом случае. Важно также проговорить потенциальную степень деградации данных.

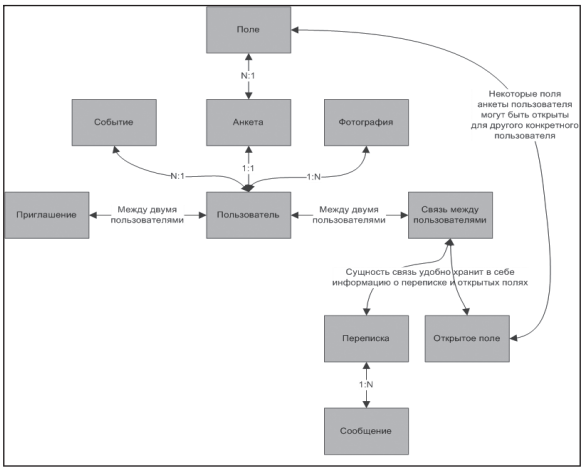
Например, у нас ожидается тридцать миллионов зарегистрированных пользователей в новой потенциальной социальной сети. По аналогии с существующими сетями предположим, что в день на сайт будет заходить зарегистрированных пользователей, то есть шесть миллионов пользователей.

Какое количество записей делает в день пользователь? Этот вопрос требует исследования, но допустим, что в среднем три сообщения в день кто-то больше, кто-то меньше. На практике большинство будет заходить на сайт только для чтения чужих сообщений, поэтому сократим в пять раз — пусть пишет по три сообщения каждый пятый пользователь.

Получается 3,6 миллиона записей в сутки. У каждого пользователя 100 подписчиков, то есть (если мы остаемся на схеме уведомлений об изменениях) 360 миллионов уведомлений в сутки.

С учетом пикового характера веб-трафика получаем 10 тысяч уведомлений в секунду — это может быть проблемой! Мы видим такую цифру и понимаем, что нам придется рассылать уведомления не в реальном времени.

Чтобы рассчитать объемы данных, допустим, что половина сообщений текстовые, а половина — графические. Размер текстового сообщения в среднем 200 байт, графического — 100 килобайт. Итого в день мы генерируем данных на 360 мегабайт текстовых сообщений и на 180 гигабайт графики. Это немало, и очевидно, что мы не можем просто положить тексты в SQL базу данных и делать по ней выборки. Максимум — мы можем делать выборки по некоей упрощенной информации, например таблицам с идентификаторами.



На этом этапе можно и нужно проговорить сущности и связи между ними. Например, как на приведенном выше рисунке. Это пример из реального проекта простой социальной сети, разработанной нашей компанией.

Здесь же мы проговариваем вопросы деградации, для этого продакт-менеджер должен ответить нам на следующие вопросы:

- Страшно ли, если запись друга появится в news feed пользователя не моментально, а через пять секунд? А если через десять? Через минуту? Через час? Какова допустимая задержка?
- Сколько последних записей мы выводим на одной странице? Десять? Двадцать? Могу ли я перейти к более старым записям?
- Должны ли новые записи появляться на странице без перезагрузки?
- Должны ли новые комментарии к записям, находящимся на странице, появляться без перезагрузки страницы?
- Страшно ли, если записи будут подгружаться пользователю постепенно, не сразу, сначала одна, потом еще пять, а потом вдруг раз — и загрузилась запись где-то в середине.

Здесь же мы прописываем скорость работы страницы (в нашем случае не более 0,2–0,5 секунды, например) и проговариваем какие-то особенности использования модуля. Например, в нашем случае с news feed это может быть:

- 99% пользователей просматривают ленту своих записей на одну-три страницы назад (обычно до последней прочитанной записи). Архив просматривается крайне редко. Можем ли мы как-нибудь использовать эту особенность?
- Нам не надо показывать пользователю сразу всю страницу, мы можем показать ему пару записей и, пока он смотрит на них, подгружать остальные.

Вот тут надо понаблюдать за работой конкурентов, например того же Facebook. Алгоритм работы примерно такой:

— сначала загружается обвязка;

— затем происходит запрос идентификаторов записей для данной news feed;

— затем в цикле, начиная от самых старых, запрашиваем подробности про записи.

Эту схему можно упростить — например часть данных в виде JSON загружать заранее при загрузке страницы. Если посмотреть страницы Facebook, вы увидите только JavaScript, все данные представлены в виде JSON-массивов. Это очень удобно — просто сделать AJAX-обновление страницы без перезагрузки.

2. Далее нужно сформулировать дополнительные технические требования к отказоустойчивости и скорости всей системы и отдельных веб-сервисов. Наконец, для каждого из веб-сервисов, исходя из конкретных особенностей данных и требований к каждому компоненту, проектируем архитектуру и подбираем технологии. Многие технологии имеют аналоги, и среди кластера технологий выбор стоит делать на основе предпочтений команды разработчиков. Что умеют, на том и надо работать.

Итак, у нас есть довольно существенный поток данных, обладающих, однако, определенными особенностями. Похоже, что стоит разделить сами записи и структуру news feed'ов.

Не все данные нужно показывать сразу, успокоил нас продакт-менеджер и дал пару минут, чтобы поместить новое сообщение пользователя во френдленты его друзей. Строго говоря, это может быть и не так исходя из бизнес-логики проекта. В этом случае мы с вами выбрали бы другую архитектуру для системы хранения френдлент.

При обсуждении продакт-менеджер сказал нам также, что рассылку почтовых уведомлений

мы можем отложить на потом. Мы рассчитали объем задач, прикинули систему хранения для очереди и приступили к реализации.

Строго говоря, использование серверов очередей не обязательно. Не умеете работать с RabbitMQ? Ничего страшного — используйте паттерн «Очереди», но храните список задач в обычном MySQL.

Использование очередей для достижения надежности

Усложним задачу из примера с почтовыми рассылками. Итак, воркеру, обслуживающему почтовые рассылки, нужно разослать 25 писем. Три послали, на четвертом закончилось место на диске. Хлоп, сломалось! Если вы пишете синхронную рассылку в едином PHP-коде, то у вас из-за отвалившегося почтовика может возникнуть 500-я ошибка для пользователя. Это вообще неприемлемо.

Это важный аспект — с помощью очереди сообщений можно позволить сломаться какому-то куску вашей системы. Например, в одном из проектов, который мы разрабатывали, запись в базу шла через очередь сообщений. Это было очень удобно. Можно было на решардинг и на другие обслуживающие операции выключить один из кусков базы данных на какое-то время. Все это время у нас тупо росла очередь сообщений и что-то не записывалось. Потом, когда базу чинят и снова подключают, очередь рассасывается.

Таким образом, очередь сообщений позволяет вам еще и функционально развязать куски всей вашей экосистемы и позволить кому-то сломаться без общей деградации на время.

На рисунке изображено два сервиса, которые полностью независимы друг от друга. Поломка одного не приводит к поломке другого. Допустим, сервису А нужно отправить что-нибудь в сервис Б. Он ставит задачу во внутреннюю очередь сервиса А. Раздающий демон (своего рода выходные ворота сервиса А) разбирает внутреннюю очередь и рассылает запросы вовне.

Входные ворота сервиса Б принимают запрос и пишут его во внутреннюю очередь сервиса Б. Воркеры сервиса Б обрабатывают задачи из внутренней очереди.

Подобная система не только практически неубиваема, она еще и восстанавливается после сбоя без потери данных :). Может сложиться впечатление, будто это нечто запредельное, но это не так. В крупных банковских системах подобные архитектуры встречаются на каждом шагу. Да и в веб-системах можно использовать что-то похожее для достижения независимости сервисов между собой.

В качестве резюме

Итак, мы изучили один из самых мощных паттернов в проектировании веб-систем — использование очередей.

Под очередями сообщений в проектировании веб-проектов могут пониматься две разные вещи. Во-первых, способ отложить задачу «на потом». Нам надо сделать что-то, но пользователю надо ответить прямо сейчас. Мы выходим за рамки PHP'шного «запрос — ответ» и делаем что-то чуть позже, чем отдали ответ.

Во-вторых, речь может идти о так называемой общей шине данных. У нас возникает межсервисная коммуникация, которая помогает разнести вызовы между сервисами, сделать их разными по времени и унифицировать общение между разными сервисами.

Удачи! В следующем номере самое сложное — масштабирование баз данных.

Урок 5.
Базы данных.
Последний пункт обязательной программы

Если твой сайт — это не домашняя страничка, то тебе нужно где-то хранить данные. Рано или поздно выясняется, что твоя СУБД перестает с этим справляться. Какие существуют подходы к масштабированию базы данных?

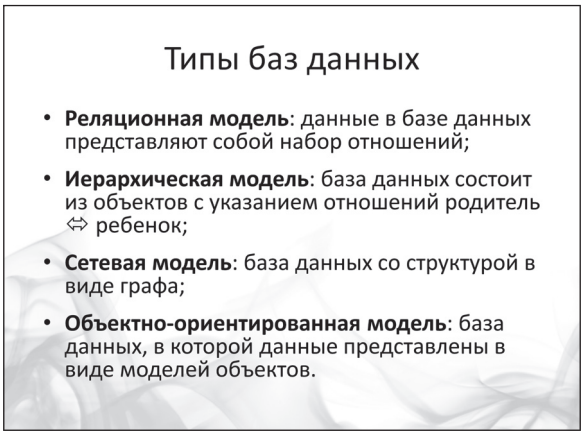
Подходов примерно столько же, сколько и для масштабирования фронтендов и бекендов, но ключевая мысль, с которой мы начнем — одна. Ты должен провести исследование предметной области, исследование потоков данных (подробно мы говорили об этом в прошлом уроке), и на основе результата этого исследования уже принимать решения о том, какие из видов масштабирования баз данных тебе подходят, какие нет.

Общего решения здесь нет. Подходит тебе синхронная репликация или нет? Подходит master-master или нет? Можешь ты поставить много баз данных и разбить данные между большим количеством экземпляров или нет? Все это зависит от конкретного приложения, от вашего пользователя и того, как ты хочешь показывать ему данные.

Чтобы говорить о конкретных решениях, нужно научиться анализировать предметную область своих данных. Для базы данных нужно определить модель представления данных, язык доступа к данным, на котором программист будет с ней общаться. Эту работу важно проделать на самом раннем этапе, хотя бы потому, что это напрямую влияет на выбор СУБД.

Различные типы баз данных

Для того, чтобы лучше понимать, как нам масштабировать базу данных, вспомним, какие, собственно СУБД у нас бывают? Если использовать классификацию по используемой модели представления данных, то получится четыре группы, представленных на слайде.



Какие сейчас в этом смысле направления, тенденции? В принципе, мы немного откатились назад, лет на 30. Мы сейчас заново проходим графовые базы данных, сетевые базы данных, иерархические модели.

Итак, теоретически, выбор СУБД выглядит так: изучаете предметную область и определяете наиболее подходящую модель представления своих данных. На основе этого выбираете оптимальную для себя систему.

Это правильно, но на деле, в реальной разработке (и это правило относится ко всем сложным проектам) предпочтение нужно отдавать тем инструментам, которые знают ваши главные специалисты. Тем не менее, это не отменяет того, что специализированные решения можно применять для отдельных типов хранимых данных — в зависимости от их характера.

В данный момент для веба есть базы данных общего назначения. Это MySQL и PostgreSQL. Если рассматривать еще и специализированные решения, то список получится на 30-40 позиций. Это и Mongo, и Redis, и тот же Neo4j. Однако в общем случае для основных ваших данных вам нужен только MySQL или PostgreSQL.

Почему? База данных — это не только то, что вы видите. Это еще и экосистема вокруг этого продукта, которая и заставляет его работать, расти и развиваться. Поясним на примере, почему это важно.

Допустим, вам хочется сделать полностью автоматический шардинг. Вы смотрите на автошардинг, сделанный в MongoDB, вам он нравится. Какие с этим могут возникнуть проблемы? Точно такие же проблемы, какие могут возникнуть с любой базой. У вас растет нагрузка, растет количество данных. MongoDB начинает, грубо говоря, тупить. И вот тут возникает главный вопрос — как и где придется решать такие проблемы? Это необходимо учитывать еще на этапе выбора СУБД.

Решение таких проблем упирается в развитость экосистемы вокруг используемого продукта. Есть ли сообщество, есть ли развитие продукта, есть ли кто-нибудь, кому я могу послать сообщение об ошибке или я использую СУБД на свой страх и риск? Именно поэтому лучше пользоваться более популярными продуктами с хорошей поддержкой.

Тюнинг запросов

Тюнинг запросов и оптимизация базы данных вообще — отдельная большая область знаний. Кратко перечислим основные направления, в которых можно проводить исследование.

Первая область связана с особенностями конкретного сервера базы данных, с его архитектурой. Сюда входят те буферы, кэши, которые использует сервер; механизм открытия/закрытия таблиц; различные особенности и так далее. Как правило, все эти параметры настраиваются.

Почему этим нужно заниматься? Приведем пример — настройки по умолчанию для СУБД PostgreSQL рассчитаны на работу всего-лишь с несколькими мегабайтами памяти — они крайне неэффективны. Отсюда следует, что настраивать базы данных необходимо.

Второе направление — особенности интерпретации и оптимизации SQL-запросов, которые применяются в данном SQL-сервере. Изучив эту сторону вопроса, можно значительно оптимизировать запросы программного обеспечения к базе данных. Нередко скорость обработки многократно увеличивается от введения одного небольшого индекса.

Также стоит обратить внимание на особенности операций с базой данных в общем. Чем отличаются операции выборки от операции вставки и какие конкретно физические действия придется совершать серверу базы данных при выполнении тех или иных запросов. Это — отдельная большая тема для разговора.

Третья плоскость, в которой стоит искать способы ускорить работу базы данных — структура базы данных, структура конкретных таблиц, индексирование и другие подобные вопросы.

Мы же поговорим только об одном — о том, какие запросы можно использовать в высоконагруженной системе, а какие нет?

Мы должны использовать все те же подходы, о которых говорили на предыдущих уроках — share nothing и stateless. Подход очень простой — представь сразу, что твои таблицы расположены не на одном, а на десяти серверах. Что они разрезаны, самые старые новости лежат на одном сервере, а новые на другом. Далее мы будем подробно говорить о механизмах подобного разделения. Но сейчас надо представить, что такое разделение уже произошло.

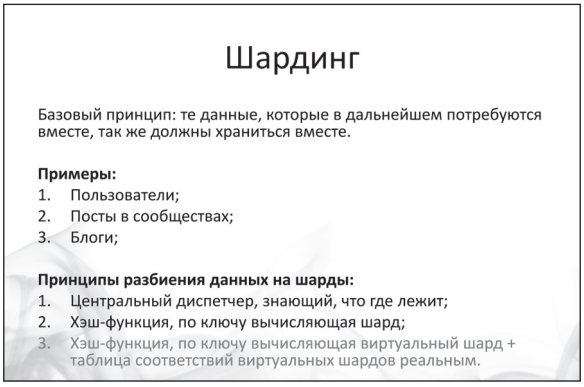
Теперь ответь на такой вопрос — как вы будете выполнять join'ы из таблиц, расположенных на разных физических серверах? Правильный ответ — вручную. Работа с базой данных в высоконагруженном проекте предполагает простые легкие конечные запросы. С заданием границ выборки, максимальным количеством извлекаемых элементов. Минимальное количество индексов, только самое необходимое, ведь индексы ускоряют выборки, но замедляют обновления БД.

Не рекомендуется использовать множество приятных внутренних механизмов СУБД. Объединения, пересечения — все это нужно делать в памяти бекенда. Нужно сделать join для двух таблиц? Сделай два отдельных запроса и перемножь результат в памяти. Не используй хранимые процедуры — как они будут работать, если исходные данные для них окажутся на разных серверах?

Следование этим простым рекомендациям может быть и замедлит несколько обработку страниц, зато сделает ее возможной, когда Ваш сайт вырастет.

Шардинг

Итак, ты смоделировал предметную область. Так или иначе, основная техника, которая используется при масштабировании СУБД — это шардинг.



Основной принцип простой. У вас есть 100 миллионов пользователей в вашей социальной сети, чате и так далее. Всю информацию, относящуюся к первым десяти миллионам, храните на одной машине, ко вторым десяти миллионам — на второй машине. У вас 10 машин. Шардинг — это разбиение, нарезка ваших данных по машинам.

Ключевым вопросом тут является принцип шардинга. По какому критерию разбивать данные? По пользователям, по комментариям, по товарам, по каталогу, по категориям товаров? Как выбрать принцип разбиения — это отдельный большой вопрос. Нужно анализировать приложение и его бизнес-логику. Главный принцип — данные должны быть максимально связаны в одном шарде и минимально связаны между шардами.

Шардинг — это некий компромисс между масштабированием и удобством доступа, удобством аналитики.

При любом шардинге так или иначе таких правил, которое мы привели выше, (первые 10, вторые 10, третьи 10), множество. Это называется принципом построения шардинга. Когда строится такое разбиение, нужно учесть множество факторов, но главное — это твои данные.

Например, у тебя такая структура, что данные только добавляются, никогда не удаляются. Наиболее простым способом для этого будет некая парадигма с ящиками. Один шард — это ящик. Ящик наполнился — открыли следующий, ящик наполнился — открыли следующий. Таким образом, мы данные каждый раз добавляем в новую и новую машину. Потом мы уже думаем, что делать со старыми ящиками, которые, может быть, даже никогда не используются. Опустошать эти ящики, выкидывать, кидать в конец.

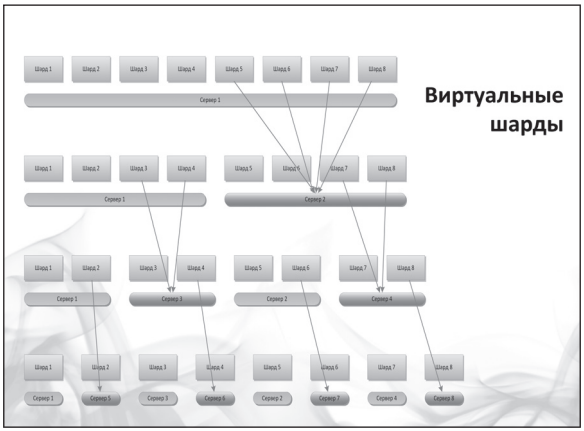
Рассмотрим второй принцип, который нужен в ситуации, когда характер нагрузки совсем другой. Допустим, что данные в каждом шарде могут расти или наоборот не расти по разным законам. Классический пример с Марком Цукербергом и Lady Gaga на Facebook. Если вы храните всё о Lady Gaga на компьютере № 69, рано или поздно этот компьютер переполнится.

Нужно думать, что делать со всеми этими данными. Или если вместе с Lady Gaga на этом же компьютере хранится 10 тысяч невинных обычных домохозяек, то рано или поздно хранение Lady Gaga на этом шарде приведет к тому, что домохозяйки получат низкое качество сервисов, потому что постоянно большой профиль нагрузки будет у Lady Gaga. Главная особенность такого сюжета — его непредсказуемость, поэтому нужна достаточно гибкая техника — виртуальные шарды.

Виртуальные шарды

Нужно предразбить пространство данных на заведомо огромное, но при этом равномерное по своей наполненности количество виртуальных шардов. Скажем, 100 тысяч виртуальных шардов. У тебя есть эта цифра, и изначально ты все эти шарды хранишь на небольшом количестве машин. Например, ты на каждой машине запускаете 10 MySQL'ей. В каждом MySQL'е ты запускаешь 100 баз данных, а всего у тебя 10, 20, 100 машин. Всё, предразбиение выполнено.

Постепенно вся эта система начинает наполняться, и можно достаточно беспроблемно (с помощью репликации) разнести данные на отдельные машины, на отдельные базы данных, на отдельные экземпляры серверов и так далее.



Эта техника называется “виртуальными шардами”. Разбиение данных по шардам — это некая договоренность, как мы будем класть элемент данных и как мы будем его потом искать. Универсального решения нет. Это некая договоренность между back-end’ом (бизнес-логикой) и системой хранения.

Виртуальные шарды — это некая прослойка, которая позволяет мне как back-end’у общаться всегда с конкретным шардом, не задумываясь о том, где физически находится этот виртуальный шард.

Получается, двойной процесс — принцип, похожий на схему работы виртуальной памяти в компьютере.. Пользователь вычисляется виртуально (например, по пользователю, по какому-то куску данных), определяется виртуальный шард. Затем берется некая таблица соответствий, по которой выясняется, где физически находится искомый шард.

Все это делается для того, чтобы в будущем, когда происходит рост каждого отдельно взятого шарда, мы могли легко и просто, не затрагивая ни бизнес-логику, ни программную часть, физически мигрировать данные с одной машины на другую.

При шардинге, как и при любой технике децентрализации все равно должна остаться какая-то центральная сущность, в которой хранится информация о том, как была проведена децентрализация. В нашем случае нужна информация, какой виртуальный шард на какой физической машине находится и какой пользователь к какому виртуальному шарду относится.

Выбор варианта центрального компонента зависит от сценария роста, от конкретного приложений. Один из них — иметь некоего диспетчера шардов, в котором хранится эта информация.

Второй — просто хранить в конфигурационном файле, если эта информация редко меняется. Вы просто распространяете этот конфигурационный файл по всему дата-центру, и везде, на любом конкретном компьютере у вас есть данные о том, что где лежит.

Третий способ — использование функционального принципа. У вас есть функция, которая однозначно выдает вам ответ. Все, что вам нужно, — это хеш-функция или некая комбинация хеш-функции и таблиц. Но принцип в том, что это функция. Это некие минимальные данные, которые редко (практически никогда) не обновляются. Вы можете использовать эти знания везде.

Центральный диспетчер

Есть компания “Badoo” (140 миллионов регистраций), сервис знакомств. По сведениям последнего года, они используют центральный диспетчер. У них нет никакой функции, которая по пользователю вычисляет, где конкретно хранится шард. В чем плюсы и минусы такого подхода?

Центральный диспетчер, при более сложной реализации, дает значительно лучшую утилизацию железа и возможность очень быстро обновлять серверный парк. Стоит центральный диспетчер и просто сообщает: «Вот тебе еще 10 серверов, заливай пользователей на них». Ты опять же можешь контролировать загрузку и знать, что это у тебя старая слабая машинка, на не миллион пользователей, и баста. Вот эта новая, супер — на нее 10 миллионов пользователей.

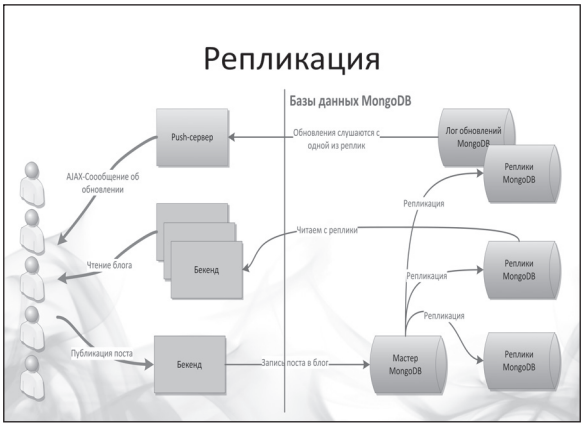
Репликация

Каждый из серверов баз данных может выйти из строя. Надо быть к этому готовым, и тут на помощь приходит техника репликации.

Репликация — это средство связи между машинами, между серверами баз данных. По большому счету, это транспорт. С помощью репликации можно эти данные перенести с одной машины на другую либо продублировать на двух машинах. Это средство организации этого разбиения.

На каждую машину можно посылать любой запрос. Все машины имеют общую копию данных — синхронная репликация. Эти данные с любой машины попадают на любую через некую трубу и эта труба позволяет иметь все эти реплики синхронными.

Какие тут принципиальные издержки, помимо того, что вам нужно резать данные? Каждые данные присутствуют в системе в двух-трех экземплярах. Каждая конкретная машина хранит то, за что непосредственно она отвечает, плюс она является запасом, бэкапом для какой-то другой машины.

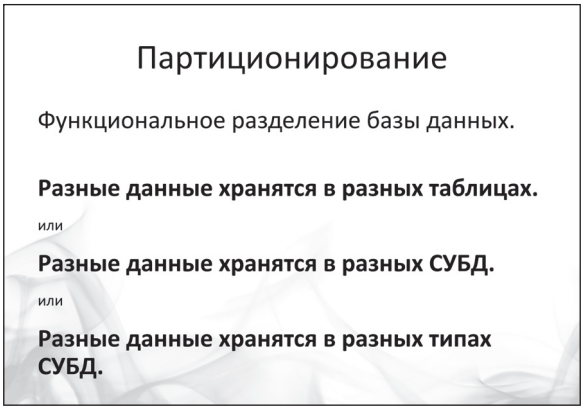


Основной принцип использования репликации, который чаще встречается, заключается (опять же, как неожиданно!) в использовании особенностей запросов к базе данных. Наиболее вероятный сценарий использования базы данных — редкие операции обновления и частые запросы на чтение. Организуется простая схема, когда операции обновления идут на центральную систему, а оттуда реплицируются (копируются) на несколько серверов, которые выполняют запрос на чтение.

Партиционирование

У нас есть несколько подходов, чтобы сделать так, чтобы смасштабировать базу данных. Первое — шардирование, когда мы бьем данные по кусочкам, раскладываем их по выбранному критерию на машинах.

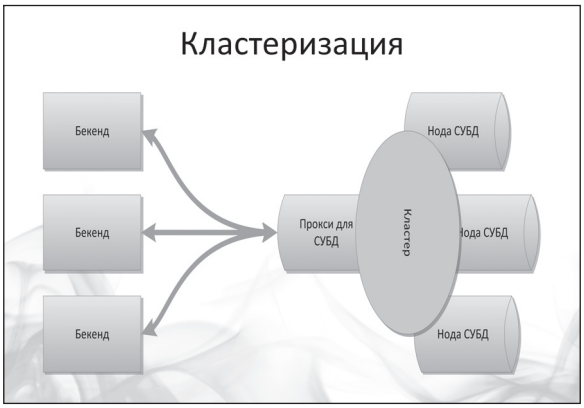
Второе — это партиционирование. Тоже бьем данные, но немного по другому принципу. То же самое, что функциональное разбиение бекендов. Все, что относится к форуму, лежит в одном месте. То, что относится к еще чему-то, лежит в другом месте. То, что относится к форуму, лежит в одной базе данных. То, что относится к новостям, — в другой базе данных.



Потом начинаем двигаться еще дальше. Мы начинаем использовать особенности наших данных. Мы начинаем хранить, например, новости в реляционной базе данных, а что-то еще — в NoSQL'ной базе данных.

Кластеризация

Кластеризация, про которую упоминалось. Вы в каком-то смысле аутсорсите проблему масштабирования базы данных на разработчика этой самой базы данных или на разработчика кластера. Для вас кластер выглядит как единое целое.



.Часто кластер, действительно, выглядит как единое целое, но это скорее дань клиент-серверной архитектуре, дань тому, что изначально базы данных были большие и толстые.

Существует множество коммерческих и бесплатных кластерных решений. Ты покупаешь кластер, его настраивают и далее это решение самостоятельно. Все внутренние процессы могут быть тебе даже не известны. Это хорошо, с одной стороны — за тебя все настроили профессионалы. С другой стороны это плохо — у тебя нет возможности что-либо исправить в случае ошибки. Ты просто не знаешь, как эта штука работает.

Все тоже самое можно реализовать с помощью репликационной модели, когда вы простонапросто соединяете базы данных в некую структуру. Данные в них движутся, существуют копии. Конкретные процессы репликации и шардинга в данном случае видны — поэтому возможно отстрелить лишние, при необходимости.

Денормализация

Рассмотрим еще один инструмент — денормализацию. Иногда для повышения эффективности хранения приходится размещать данные не самым оптимальным образом, то есть денормализуете. Например, можно их дублировать, можно хранить их в разных форматах. В примере с очередью модерации, как вы помните, мы хранили их в основной базе данных и отправляли еще куда-то. Система хранения, схема хранения, инструменты хранения отражают характер данных и предполагаемую модель их использования. Это происходит именно для того, чтобы ускорить обработку, ускорить построение страницы.

Хороший пример того, как данные не денормализуются, а надо было бы, — это френд-лента в «Живом Журнале». Отсюда (во всяком случае, несколько лет назад) все их проблемы: низкая скорость работы и ограничение на количество френдов у одного пользователя. Дело в том, что каждый раз френд-лента строится нормальным, честным SQL-запросом: «Дай мне все сообщения всех моих друзей, отсортируй».

В Facebook это не так. Там каждое сообщение может храниться в нескольких миллионах экземплярах — именно для того, чтобы обработка данных происходила быстрее, чтобы быстрее показать пользователю news feed.

Это и есть денормализация данных. Ничего страшного, что данные хранятся в двух-трех экземплярах. В этом есть, естественно, оборотная сторона: нужно знать об этом, уничтожать или каким-то образом обрабатывать правильно. Но это позволит ускорить работу по построению страниц.

Чтобы осуществить денормализацию данных, придется немного поломать существующую модель представления данных, а это усложнит ее анализ. Для этого есть огромное количество решений, которые хранят данные в денормализованном виде, но могут представлять их в реляционном виде.

Особенности хранимых процедур в MySQL

Поговорим о достоинствах и недостатках хранимых процедур в MySQL. Напомним, что использовать этот инструмент в масштабируемой базе данных надо очень аккуратно.

Хранимая процедура — это текст, записанный в системную таблицу, который будет регулярно читаться из таблицы, компилироваться и кешироваться к скомпилированном виде в каждом соединении к серверу. Поскольку на каждый скомпилированный объект требуется от 80КБ оперативной памяти, при использовании большого количества хранимых процедур в большом количестве соединений к серверу надо рассчитывать на рост оперативной памяти, необходимой для MySQL. К примеру, при 1000 активных соединений, каждое соединение использует 20 хранимых процедур по 100КБ, необходимо до 2ГБ дополнительно оперативной памяти для хранимых процедур.

Несмотря на этот недостаток, использование хранимых процедур является распространённой практикой при доступе к данным в крупных веб-проектах по следующим причинам:

- дополнительный уровень внутренней безопасности. Прикладной программист, разрабатывающий сервис сети, вызывает хранимую процедуру, а не SQL запрос, и таким образом не имеет прав и может не знать непосредственной схемы данных
- возможности изменения схемы данных без изменения приложений. Меняется только уровень хранимых процедур.

Производительность процедуры напрямую зависит от её сложности. Распространённой практикой является создание хранимых процедур, инкапсулирующих не более 1-2х запросов к БД. Использование более сложных процедур не распространено, т.к.:

- пропорционально увеличивается задержка на выполнение процедуры
- отладка в случае неполадок усложняется, т.к. в MySQL нет интегрированного отладчика хранимых процедур, не говоря уже о том, что в production может быть ещё важно понять какой конкретно запрос в хранимой процедуре начинает выполняться медленно, и наличие большой процедуры добавляет сложности в поиске проблемы.

Важно учитывать то, что при изменении хранимой процедуры одновременно инвалидируются все кеши всех активных соединений. Это может привести к серьёзным «провалам» в производительности, т.к. все соединения одновременно будут пытаться пересоздать свои скомпилированные копии хранимых процедур. Это следует учитывать, и не планировать массированные изменения во время прайм-тайм нагрузок.

При репликации хранимых процедур MySQL использует так называемый «unrolling», т.е. в replication log попадает не непосредственно вызов хранимой процедуры (CALL GetUserComet(480145)) а запросы, выполненные внутри хранимой процедуры. Т.е. реплика не выполняет саму процедуру, а только те запросы, которые хранимая процедура использует и которые изменяют данные.

Необходимо также иметь ввиду, что алгоритм выполнения хранимых функций и триггеров в MySQL существенно отличается от описанного выше, т.е. не следует эти знания применять для хранимых функций и триггеров.

Последний пункт обязательной программы

Вот, наверное, и все основы масштабирования баз данных. Используйте все приемы разумно, в той мере, в какой необходимо.

Почему базы данных — это “последний пункт”, спросишь ты? Все очень просто — в предыдущих пяти уроках мы рассмотрели основные архитектурные модули типичного высоко-

нагруженного проекта — фронтенд, бекенд, базу данных. Для каждого мы перечислили типичные подходы к масштабированию. Ты можешь уже приступить к созданию своего собственного высоконагруженного проекта

Но разработать проект — это еще не все. Проект надо поддерживать, эксплуатировать, организовать правильный хостинг, правильный мониторинг. Вот об этих, сервисных, но совсем немаловажных аспектах пойдет речь в следующих уроках. До встречи!

Урок 6. Надежность, эксплуатация, паттерны масштабируемых архитектур.

В последний урок мы решили добавить всё, что не успели рассмотреть в предыдущих, а также систематизировать основные изученные паттерны проектирования. Кроме того, остался последний пункт нашей обязательной программы – это эксплуатация. Мало разработать систему, нужно еще и поддерживать, а это целое искусство.

Как уже неоднократно говорилось, первое и главное отличие высоконагруженного проекта заключается в сложности и наличии большого числа взаимосвязанных компонент. Ключевое слово, на которое нужно сделать ударение – БОЛЬШОГО.

Представьте себе поисковый кластер Яндекса или хранение фотографий ВКонтакте – там используются тысячи машин. Серверный диск, при его активном использовании, в среднем выходит из строя раз в два года. Это означает, что из трех тысяч серверов прямо сегодня полетят диски у 4-х машин. Вывод – в большой крупной системе всегда что-то не работает, какой-то из серверов вышел из строя, какой-то из дисков сбоит. И это – нормальное состояние системы. Ты должен быть к нему готов, и мы в этой статье расскажем как.

Надежность

Надежность веб-системы (как и любой другой системы) заключается в способности сохранять в пределах установленной нормы значения всех параметров, характеризующих способность выполнять требуемые функции. В нашем случае – обрабатывать запросы пользователей.

В зависимости от ситуации, параметры установленной нормы можно определять по-разному. Конечно, хорошо, если сайт ответит на запрос за одну десятую секунды, но если он ответит на запрос за две секунды, ничего критичного не произойдет. С этим связан один из способов борьбы с резко возросшей нагрузкой, который называется деградация функциональности. В любом проекте есть целый ряд функций, почти незаметных пользователю, но требующих для своей реализации серьезных серверных ресурсов.

Например, скорость появления опубликованной редактором новости на сайте. Если это не новостное издание, то пострадает ли пользователь, если новость станет доступна только через десять минут после публикации? Нет, не пострадает, он даже этого не заметит. Но разработчикам это позволит внедрить кеширование или прегенерацию.

Возьмем в качестве примера онлайн-конструктор сайтов Setup.ru, разработанный в нашей компании. Созданный пользователем сайт не начнет работать, пока он не нажмет кнопку “Опубликовать”. Это звучит просто, но такое искусственное препятствие позволило значительно снизить нагрузку на систему, поддерживающую 356 тысяч пользовательских сайтов всего на нескольких серверах.

Подробно о деградации функциональности мы рассказывали в четвертом уроке, а сейчас вернемся к надежности. Одно дело - допустимое снижение скорости работы нашей системой, но как избегать проблем со скоростью во всех других случаях? Как избегать замедлений вне зависимости от того, какие диски полетели и куда?

Избыточность и дублирование

Способ ровно один – если ты хочешь быть готовыми к выходу из строя некоторых элементов – вводи избыточность этих самых элементов. Это, к сожалению, неизбежно. Если хочешь, чтобы фронтенд всегда был на связи – ставь два, если хочешь надежности в работе базы данных – настраивай репликацию.

Это как с денормализацией из прошлого урока – хранение дублированных данных, оптимизированных для выборок – да, придется хранить одни и те же элементы в нескольких экземплярах, зато быстро работает. Это неизбежное зло.

Чтобы понять, что именно нужно дублировать, нужно смоделировать проблемное состояние и заранее просчитать его.

Диск может выйти из строя? Значит надо хранить данные не в одном экземпляре, а в нескольких. И обновлять в реальном времени, закачивая файл на один из серверов в паре, сразу же закачивать и на другой. Пока файл не появится на обоих серверах, операция считается невыполненной.

Сервер бекенда может сгореть? Пусть серверов будет на один больше. Или на несколько штук больше, чтобы не вызвать цепную реакцию и не столкнуться с проблемой шквала/антишквала (смотрите предыдущие уроки).

Может умереть что-то на машине с базой данных? Настрой репликацию и пусть слейв всегда стоит под парами, готовый принять нагрузку. Должно ли программное обеспечение самостоятельно принимать решение о переключении на другой сервер базы данных? Это, на самом деле, совсем непростой вопрос. Потому что:

- Все экземпляры, допустим, бекенда, должны одновременно принять решение о переключении на новый сервер базы данных;
- Что делать, если старый сервер базы данных восстановится? Часть бекендов уйдет на старый сервер, часть на новый и гарантированно появится огромная проблема с целостностью данных.

Поэтому автоматическое переключение на реплику используется редко. Обычно на реплику переключают только чтение из базы данных, одновременно отключая (деградация функциональности, это тоже она) запись в базу данных. И, естественно, все это сопровождается взводом красных флажков в системе мониторинга. Впрочем, о системе мониторинга позже.

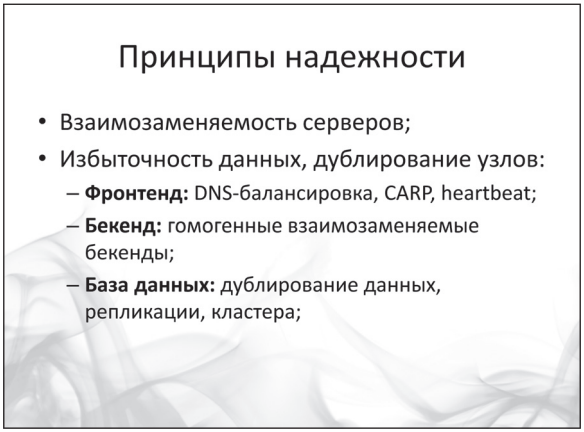
Принципы надежности

Мы уже упоминали в прошлых уроках все основные способы введения избыточности в прошлых уроках, поэтому просто перечислим их:

- Для фронтенда это балансировка или составление пар серверов, в которых один всегда запасной. Запасной сервер включается только в том случае, если ведущий перестал откликаться и обрабатывать запросы;
- Для бекенда это гомогенные взаимозаменяемые бекенды, отсутствие точек отказа (принцип shared nothing), отсутствие изменения состояния бекенда после обработки запроса (принцип stateless);
- Для баз данных это денормализация, репликации, кластера. Для бинарных данных это резервирование и дублирование информации.

Как в программном обеспечении вы избегаете единых точек отказа, то точно также надо избегать точек отказа и в железе. Старайтесь не использовать никакого уникального оборудования без крайней на то нужды. Вы удивитесь, но поиск Яндекса, обрабатывающий десятки тысяч запросов в секунду и хранящий колоссальное количество информации, построен на самых обычных серверах. И вот почему:

- Легко и просто достигается взаимозаменяемость компонент;
- Закупка новых серверов и комплектующих не является проблемой – Вам не нужно полгода ждать из США какую-то уникальную деталь.



Chaos Monkey

Говоря о надежности и отказоустойчивости, не можем не упомянуть один из самых интересных способов тестирования. Как проверить, что наша система продолжит работать при выходе из строя отдельных серверов или компонентов?

Инженеры компании Netflix придумали инструмент и назвали его Chaos Monkey. Работа обезьянки Хаоса состоит в том, чтобы хаотично прибивать случайный сервис или процесс на каком-то случайно выбранном сервере программной системы. Весь проект при этом должен продолжать работать, какой-бы конкретно процесс не убила Chaos Monkey.

Системные администраторы оценят простоту и элегантность решения – скрипт на пару строк, вызывающий “kill -9” для случайного процесса.

Мониторинг

Итак, применяя все изложенное в предыдущих уроках и начале этой статьи, мы построили масштабируемую высоконагруженную систему. Все узлы продублированы, введена некоторая избыточность оборудования. Это все?

Нет, не все. Второй ключевой аспект крупного проекта – вы должны знать о нем абсолютно все!

Все, что происходит в любой точке системы, любые аномалии в поведении отдельных элементов вашей программной системы должны оперативно детектироваться и анализироваться.

Обычно в мониторинг попадает некий джентельменский набор параметров типа значения load averages, количество чтений/записи с диска, свободное пространство на дисках, количество процессов, трафик.

Это важно, но этого недостаточно. Советуем также включить в мониторинг еще два вида параметров:

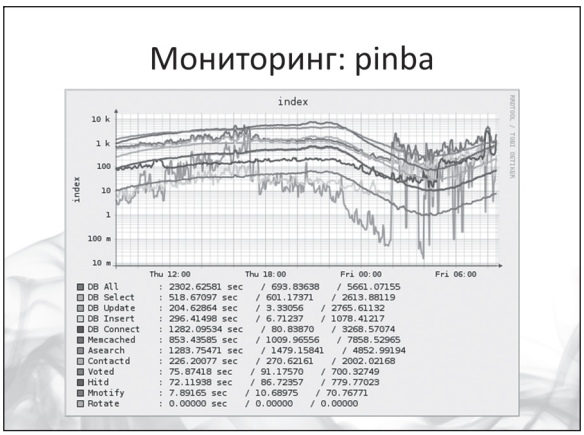
- Бизнес-параметры, например, количество регистраций за последнюю минуту, количество отправляемых сообщений, количество поисковых запросов и другие. То есть такие параметры, которые описывают поведение и работу бизнес-логики вашего проекта;
- Более детальные специализированные технические параметры, например, время отдачи отдельных страниц, задержка появления новых данных на реплике, время выполнения отдельных операций в базе данных. То есть технические параметры, которые описывают работу именно вашего проекта, учитывая его специфику и функциональность.

Мониторинг бизнес-параметров даст вам возможность отслеживать и реагировать на изменения в поведении ваших пользователей. Например, ссылка с главной страницы Яндекса вызовет огромный приток регистраций, вы отслеживаете этот процесс и своевременно начинаете подготовку новых серверов для новых шардов.

Или, во время выборов, пользователи начинают активно переписываться, а несколько групп резко растут в размерах. Наблюдения за такими параметрами поможет вам реагировать проактивно, например, подготовить для оппозиционеров отдельный сервер и перенести крупные группы туда :) Это удобно во всех отношениях :)

Мониторинг бизнес-показателей, кстати, это основной инструмент и, как это ни странно, прогнозирования нагрузки. Если вы знаете темпы роста своего сайта, то сможете подсчитать нагрузку через неделю или месяц.

Специализированный технический мониторинг дает вам информацию о работе отдельных подсистем вашего проекта, причем именно вашего проекта. Вы сможете увидеть залипания каких-то отдельных сервисов и реагировать именно на них. Например, картинки в фотоальбоме стали отдаваться очень медленно – один из серверов вышел из строя и синхронизация с его заменой занимает очень много времени. Или в одном из голосований количество проголосовавших выросло до миллиона и проверка на уникальность голоса теперь занимает слишком много времени.



Нормально ли так опускаться в детали? Да, нормально. Приведу пример – на последнейм конференции HighLoad++ 2012 (http://www.highload.ru/) Олег Илларионов из ВКонтакте рассказал о том, что они постоянно мониторят отдачу картинок и иногда вручную обрабатывают рост интереса к какой-то конкретной картинке.

Примером такой картинки Олег назвал аватарку Павла Дурова, что вовсе не удивительно! Кстати, аватарка Павла Дурова не лежит в стандартной системе хранения и доставки контента ВКонтакте – аватарка Павла Дурова – это статика, разложенная на все фронтенды, и отдается вместе со всеми CSS и JS-файлами и другими картинками оформления.

Это очень хороший пример крупной системы – простое решение проблемы, которую заметили с помощью хорошо простроенного мониторинга.

Восстановление

Итак, мы не просто построили крупную систему, мы построили классный мониторинг. Мы знаем все, что происходит в системе, в каждом ее уголке. Что делать, если мы обнаружили проблему? Все средства восстановления должны быть автоматизированы. Если ты сталкиваешься с какой-то проблемой более, чем один раз – автоматизируйте ее. Но большую часть автоматизируйте априори.

Синхронизация сервера для хранения картинок должна производиться скриптом. Изменение настроек – скриптом, настройка нового сервера – тем более. Подробнее об этом стоит почитать популярную ныне тему DevOps, а мы приведем несколько ключевых мыслей из доклада Александра Титова и Игоря Курочкина про организацию системного администрирования в компании Skype.

Существует притча, которая описывает эволюцию. Когда царь пришел к мудрецу и спросил его: «Как устроена Земля? Почему она не падает?», мудрец сказал: «Земля стоит на льве». Потом он спросил: «Хорошо. Почему лев не падает?» — «Лев стоит на слоне». — «Почему слон не падает?» — «Слон стоит на черепахе. Больше не спрашивайте меня, потому что дальше идут одни черепахи». Один сервер — это, по сути дела, оно и есть, основа всего.

С одним сервером все понятно. Он просто администрируется одним специалистом, который владеет полной информации о состоянии этого сервера – все умещается у него в голове. Дальше идет следующий уровень — 5 серверов. По большому счету, подходы те же самые, но уже начинается обобщение и автоматизация. Появляются скрипты, решающие ту или иную задачу системного администрирования. Больше 20-ти машин — это, в первую очередь, большая система. Уже на этом уровне возникают проблемы из-за разницы конфигураций. Разные

версии LIPC на одном сервере работают, на другом сервере не работают. Разные версии PHP, Ruby. Все это достаточно знакомо многим.

Начинает требоваться большое количество документации. Каждое изменение вы должны прописать – ведь когда различного рода неочевидных изменений станет много, вы начнете их забывать. На количестве машин больше 20-ти стоимость поддержки сильно превышает стоимость внесения изменений. Вы перестаете вносить изменения, вы постоянно исправляете ошибки.

В этот момент начинает приходить понимание, что надо управлять не одной машиной, а какой-то абстрагированной штукой — кластером. Потребуется ввести автоматическую установку машин, автоматическое управление конфигурациями, автоматическое развертывание сервисов, и автоматическую выкатку. Про автоматическую выкатку надо поговорить подробнее.

Deployment

Вопрос про deployment – это очень важная вещь. Нужно уметь перевыкатывать весь свой сайт на новое голое железо за 20 минут (без учета времени копирования данных). Вопрос владения своей инфраструктурой очень серьезный, особенно если возникает необходимость масштабироваться или, скажем, выкатываться в облако.

Допустим, сломался конкретный сервис. Поднимаем данные из бэкапов. Если на подъем всего проекта заново уходит день, возможно, этот день будет фатальным для вашего бизнеса. Amazon в этом плане очень правильно учит людей. Когда вы берете машину EC2, она может перезагрузиться в любую секунду, и там не останется ничего, никаких данных, и нужно иметь возможность выкатываться на голый Linux в течение минут.

Чтобы нормально жить в условиях меняющегося железа и нового подключающегося железа, вам необходимо бы владеть своей инфраструктурой и быть уверенными в том, что и инициализация новых серверов тоже будет работать. Это первая часть, что касается deployment.

Второе – это вообще процесс выкатывания обновления сайта. Пусть мы взяли админа, и админ нам обещает, что он в случае чего все выкатит за час. Поверим его обещаниям, хотя, конечно, не стоит :)

Дальше в дело вступают программисты. Программиста спрашивают: «Покажи на сайте новую вещь». Он говорит: «У нас апдейты выходят раз в неделю». Можно так работать? Нет, так работать нельзя.

Выкатывание новой версии на сайт должно производиться максимально легко. Опять же это нужно, чтобы вы могли быстро пробовать новые вещи, чтобы можно было легко и быстро поправить. За что не любят в вебе Java? Помимо прочих причин одна из проблем – надо долго компилировать. Нельзя взять и быстро выкатить на сайт, хотя это бывает нужно. Что бы кто ни говорил, но даже в самых больших и крупных проектах иногда бывает редактирование кода на продакшне. В этом просто кто-то признается, а кто-то не признается. В любом крупном проекте такое иногда бывает. Все зависит от того, как именно организован деплой.

Есть разные способы все это упростить, автоматизировать. Но в любом случае должно быть подконтрольно, никаких магических выкатывалок, никаких «оно как-то расползается по кластеру». Это означает, что ты не контролируешь процесс, а раз не контролируешь, то он неизбежно сломается, приведя к серьезным трудноуловимым ошибкам.

Существует большое количество инструментов для деплоймента, изучите и выберете оптимальный для конкретной ситуации. Например, библиотека Capistrano. Ты описываешь в специальном формате процесс выкатки, прописываете списки серверов, библиотеки кода, что и куда выкатывать, что и где перезапускать. И далее процесс выкатки упрощается до запуска управляющей команды.

Процесс отката

Мы упомянем также и о процессе отката – что делать, если обновление, которое вы только что выкатили, сломалось в боевых условиях? Нужно оперативно и быстро откатиться назад. Автоматизация процесса отката – это, конечно, чистая магия, о которой мы в нашей статье говорить не будем.

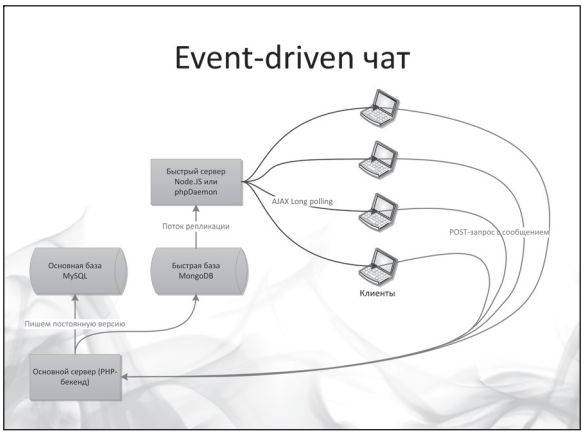
Но один совет для того, чтобы облегчить процесс отката, приведем. Основная проблема отката это изменения в базе данных. Отсюда вывод – любое изменение в базе данных должно быть оформлено в виде файла с конкретными SQL-командами. Этот файл должен быть положен в репозиторий и именно этот файл выполняет скрипт деплоймента для выкатки новой версии схемы базы данных. Вместе с этим файлом вы также можете класть и файл, содержащий SQL-команды для отката к предыдущему состоянию схемы СУБД.

Ясно, что это должны быть автономные атомарные команды. Допустим, ты добавляешь новое поле в регистрационную информацию пользователя. Значит SQL-файл, который нужно выполнить до выкатки требуемого кода на боевые сервера, содержит команду add column, а SQL-файл для отката к предыдущей версии этот же столбец удаляет.

Опять же, ясно, что программный код не должен ломаться от того, что пришел лишний столбец или, наоборот, нужно столбца в базе данных не существует. Операция деплоя не атомарна, ситуация, когда код уже новый, а изменения в СУБД до реплики не докатились, теоретически возможна.

На магии выкатки мы и заканчиваем наш учебник в плане систематизации основ построения высоконагруженных проектов. В качестве добавки рассмотрим еще несколько основных паттернов проектирования.

Чат



Сакральный чат – ну какой программист не писал его? Рассмотрим простую схему простой переписки между пользователями. Как реализовать моментальное появление новых сообщений в окне переписки?

Итак, каждый новый клиент устанавливает соединение с одним из “быстрых” серверов (например, Node.js), от него JavaScript, работающий в браузере клиента будет получать новые сообщения, обрабатывать их и выводить в окне переписки. Получение будет происходить в режиме AJAX Long Polling (постоянно открытое соединение).

Отправка на сервер нового сообщения идет на основной PHP-сервер, который записывает сообщение в постоянную базу данных для истории (например, MySQL) и “быструю” базу данных. В “быстрой” базе хранится только актуальная переписка, например, за последний день. В нашем случае будем использовать для горячей информации базу данных MongoDB.

А теперь ответ на вопрос, почему такой странный выбор – MongoDB и Node.JS. Все поступающие сообщения мы записываем в коллекцию для репликации, которую в качестве слейва слушают сервера Node.JS.

Каждый из Node.JS’ов знает идентификаторы всех клиентов, с которыми у него установлена связь. Если из потока репликации приходит сообщение для одного из своих клиентов, то забираем его и отправляем в браузер клиента.

Нам даже перекодировать ничего не надо, так как формат коммуникации между MongoDB-серверами, Node.JS и формат хранения данных в MongoDB – JSON.

Бинарный кластер

Еще один часто встречающийся паттерн в проектировании – использование бинарных кластеров. Под этим устойчивым словосочетанием понимается примерно следующее.

Весь пул бинарных файлов (картинки, видеофайлы) разделяется на самые обычные шарды. Но каждый из шардов состоит из двух и более серверов, информация на которых полностью дублируется.

Далее возможны варианты – вы можете объединить сервера в группе бинарного кластера с помощью heartbeat или CARP-решения (основной сервер и резервный) или просто распределить трафик на несколько серверов с помощью какой-нибудь простой балансировки.

С помощью подобного решения достигается одновременно и масштабируемость и надежность. Просто единицей масштабируемости является ни один сервер, а сразу группа серверов, именуемая бинарным кластером.

Пожалуй и все, остановимся на этом, так как о примерах можно говорить бесконечно. Что еще почитать? В России один из основных источников информации о высоконагруженных системам – конференции HighLoad++, организуемые Олегом Буниным. Расшифрованные доклады конференции выкладываются на сайт Клуба Профессионалов (http://www.profyclub.ru/), там вы сможете найти еще десятки примеров конкретных ситуаций, архитектур и проектов.

Удачи Вам!