Почему вы никогда не должны использовать MongoDB перевод

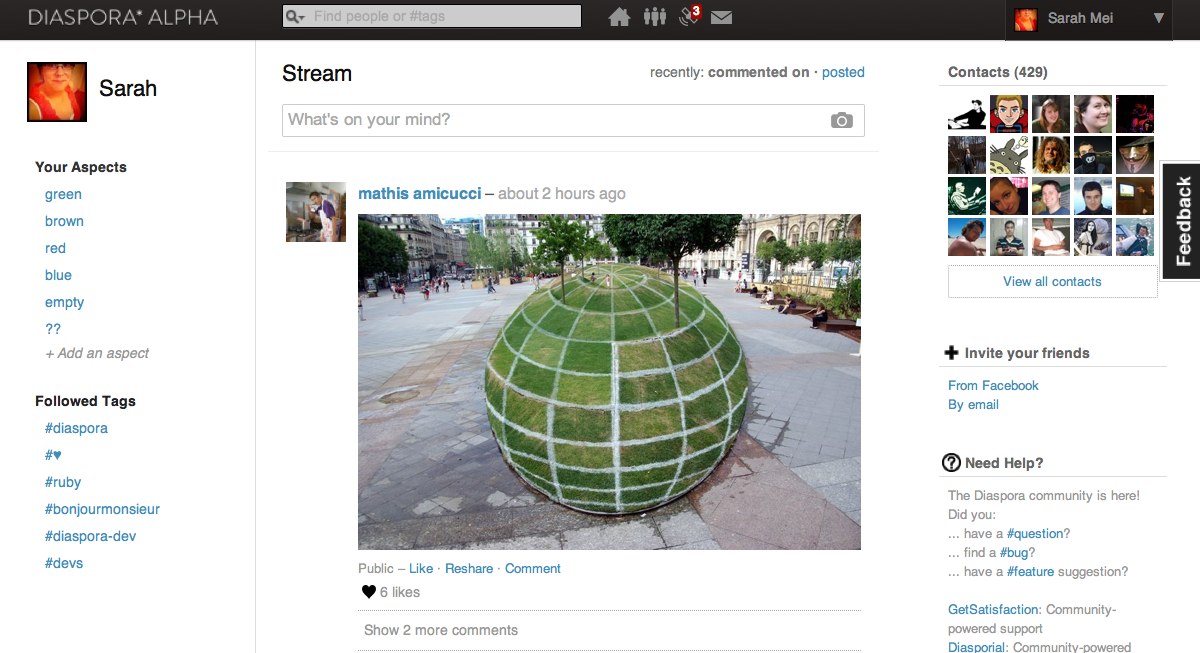
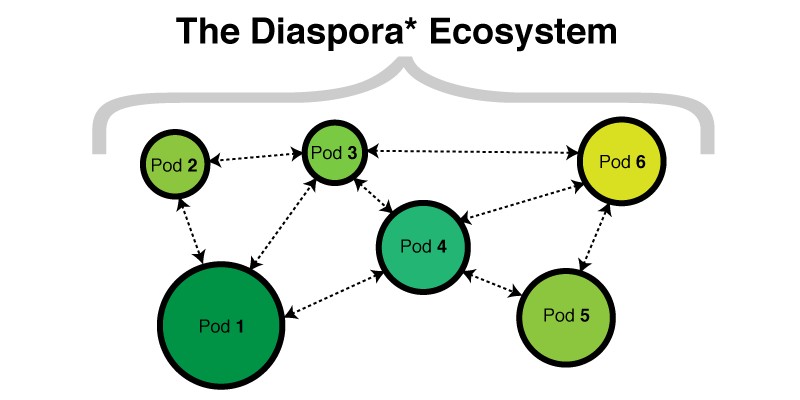
[Разработка веб-сайтов](https://habrahabr.ru/hub/webdev/)\*, [MongoDB](https://habrahabr.ru/hub/mongodb/)\*, [NoSQL](https://habrahabr.ru/hub/nosql/)\*

Дисклеймер от автора (автор — девушка): Я не разрабатываю движки баз данных. Я создаю веб-приложения. Я участвую в 4-6 разных проектах каждый год, то есть создаю **много** веб-приложений. Я вижу много приложений с различными требованиями и различными потребностями хранения данных. Я разворачивала большинство хранилищ, о которых вы слышали, и несколько, о которых даже не подозреваете.  
  
Несколько раз я делала неправильный выбор СУБД. Эта история об одном таком выборе — почему мы сделали такой выбор, как бы узнали что выбор был неверен и как мы с этим боролись.Это все произошло на проекте с открытым исходным кодом, называемым Diaspora.

Проект

Diaspora — это распределенная социальная сеть с долгой историей. Давным давно, в 2010 году, четыре студента Нью-Йоркского университета опубликовали на Kickstarter видео с просьбой пожертвовать $10,000 для того разработать распределенную альтернативу Facebook. Они отправили ссылку друзьям, семье и надеялись на лучшее.  
  
Но они попали в самую точку. Только что отгремел очередной скандал из-за приватности на Facebook, и когда пыль улеглась они получили $200,000 инвестиций от 6400 человек для проекта, в котором еще не было написано ни одной строки кода.  
  
Diaspora была одним из первых проектов на Kickstarter, которым удалось значительно превысить цель. Как результат, о них написали в газете New York Times, что обернулось скандалом, потому что на доске на фоне фотографии команды была написана неприличная шутка, но никто этого не заметил, пока фотографию не напечатали… в New York Times. Так я и узнала об этом проекте.  
  
В результате успеха на Kickstarter парни бросили учиться и переехали в Сан-Франциско, чтобы начать писать код. Так они оказались в моем офисе. В то время я работала в Pivotal Labs и один из старших братьев разработчиков Diaspora тоже работал в этой компании, поэтому Pivotal предложили парням рабочие места, интернет, и, конечно, доступ к холодильнику с пивом. Я работала с официальными клиентами компании, а по вечерам зависала с парнями и писала код по выходным.  
  
Закончилось тем, что они оставались в Pivotal более двух лет. Тем не менее, к концу первого лета у них был минимальная, но работающая (в некотором смысле) реализация распределенной социальной сети на Ruby on Rails, использующая MongoDB для хранения данных.  
  
Довольно много баззвородов — давайте разберемся.

Распределенная социальная сеть

Если вы видели Facebook, то вы знаете все, что вам надо знать о Facebook. Это веб-приложение, оно существует в единственном экземпляре и позволяет вам общаться с людьми. Интерфейс Diaspora выглядит почти также, как Facebook.  
  
Лента сообщений посередине показывает посты всех ваших друзей, а вокруг нее куча мусора, на который никто не обращает внимания. Основное отличие Diaspora от Facebook невидимо для пользователей, это «распределенная» часть.  
  
Инфраструктура Disapora не располагается за одним веб-адресом. Есть сотни независимых экземпляров Diaspora. Код открыт, поэтому вы можете развернуть свои серверы. Каждый экземпляр, называется Стручок. Он имеет свою базу данных и свой набор пользователей. Каждый Стручок взаимодействует с другими Стручками, у которых так же своя база и свои пользователи.  
  
  
Стручки общаются с помощью API на основе протокола HTTP (сейчас это модно называть REST API — *прим. пер.*). Когда вы развернули свой Стручок, он будет довольно скучным, пока вы не добавите друзей. Вы можете добавлять в друзья пользователей в вашем Стручке, или в других Стручках. Когда кто-либо что-либо опубликует произойдет вот что:

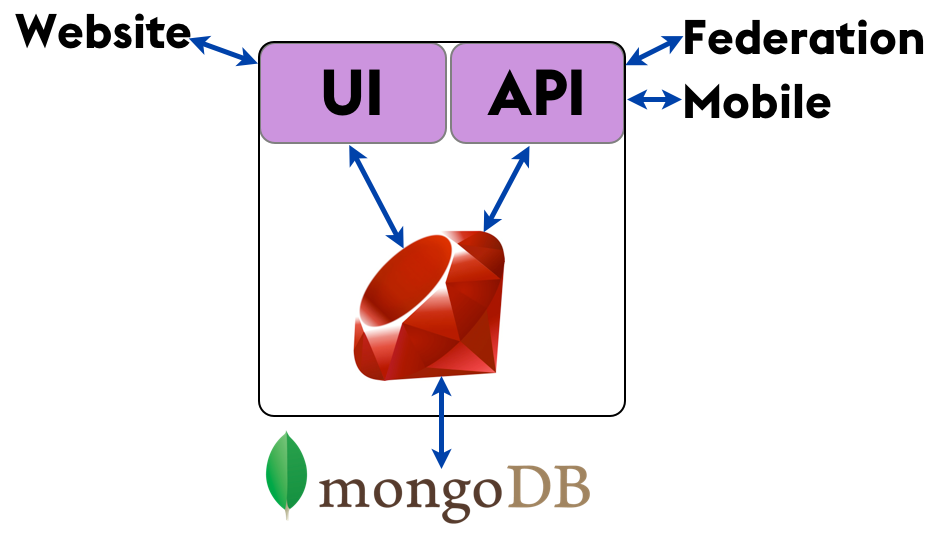
1. Сообщение сохранится в базе данных автора.
2. Ваш Стручок будет оповещен через API.
3. Сообщение сохранится в базе данных вашего Стручка.
4. В вашей ленте вы увидите сообщение вместе с сообщениями от других друзей.

Комментарии работают точно так же. Каждое сообщение может быть прокомментировано как пользователями из того же Стручка, так и людьми из других Стручков. Все, у кого есть разрешения просмотреть сообщение, увидят также все комментарии. Как-будто все происходит в одном приложении.

Кого это волнует?

Есть технические и юридические преимущества этой архитектуры. Основным техническим преимуществом является отказоустойчивость.  
  
*(такую отказоустойчивую систему надо иметь в каждом офисе)*  
  
Если один из Стручков падает, то все остальные продолжают работать. Система вызывает, и даже ожидает, разделение сети. Политические следствия этого — напрмиер если в вашей стране закрывают доступ к Facebook или Twitter, ваш локальный Стручок будет доступен другим людям в вашей стране, даже если все остальные будут недоступны.  
  
Основное юридическое преимущество — это независимость серверов. Каждый Стручок- юридически независимая сущность, управляемая законами той страны, где расположен Стручок. Каждый Стручок также может устанавливать свои условия, на большинстве вы не отдаете права на весь контент, как например на Facebook или Twitter. Diaspora — свободное программное обеспечение, как в смысле «даром», так и в смысле «независимо». Большинство тех, кто запускает свои Стручки, это очень заботит.  
  
Такова архитектура системы, давайте рассмотрим архитектуру отдельного Стручка.

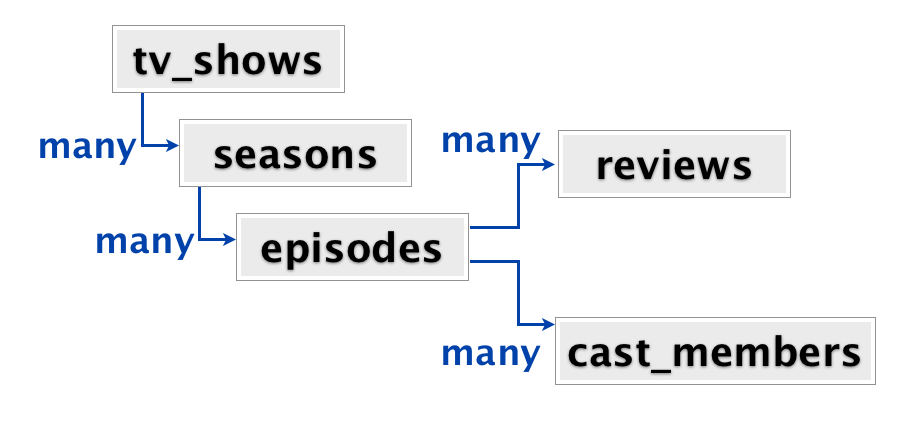
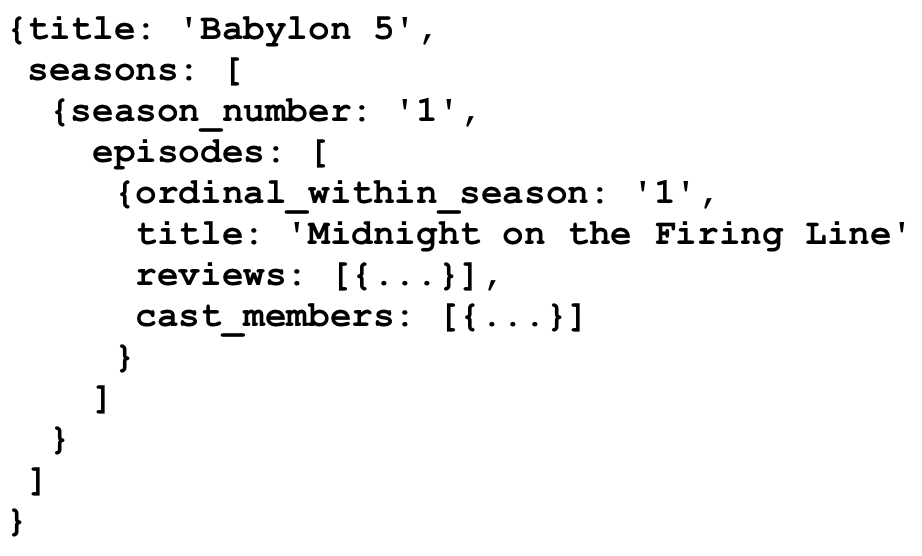
Это Rails приложение.

Каждый Стручок это Ruby on Rails приложение со своей базой на MongoDB. В некотором смысле это «типичное» Rails приложение — оно имеет пользовательский интерфейс, программный API, логику на Ruby и базу данных. Но во всех остальных смыслах оно совсем не типичное.  
  
API используется для мобильных клиентов и для «федерации», то есть для взаимодействия между Стручками. Распределенность добавляет несколько слоев кода, которые отсутствуют в типичном приложении.   
  
И, конечно, MongoDB — далеко не типичный выбор для веб-приложений. Большинство Rails приложений используют PostgreSQL или MySQL. (по состоянию на 2010 год — *прим. пер.*)  
  
Так вот код. Рассмотрим, какие данные мы храним.

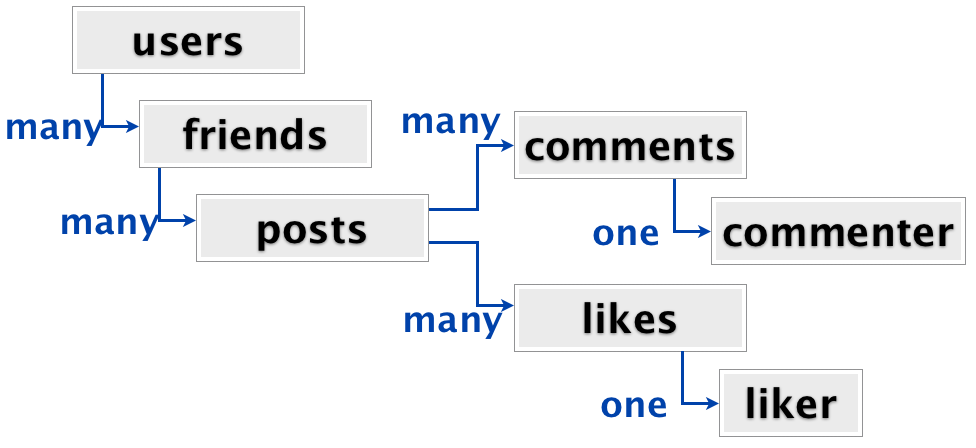
Я не думаю, что это слово означает, что вы думаете, что это означает

«Социальные данные» — это информация о нашей сети друзей, их друзей и их деятельности. Концептуально мы думаем об этом как о сети — неориентированном графе, в котором мы находимся в центре, и наши друзья находятся вокруг нас.  
  
*(Фотографии с rubyfriends.com. Благодаря Мэтт Роджерс, Стив Klabnik, Нелл Shamrell, Катрина Оуэн, Сэм Ливингстон-серый, Джош Сассер, Акшай Khole, Прадьюмна Dandwate и Хефзиба Watharkar за вклад в # rubyfriends!)*  
  
Когда мы храним социальные данные, мы сохраняем как топологию, так и действия.   
  
Вот уже несколько лет мы знаем, что социальные данные не являются реляционными, если вы храните социальные данные в реляционной базе данных, то вы делаете это неправильно.  
  
Но какие есть альтернативы? Некоторые утверждаю что графовые базы данных подходят лучше всего, но я не буду их рассматривать, так как они слишком нишевые для массовых проектов. Другие говорят что документарные идеально подходят для социальных данных, и они достаточно мейнстримные для реального применения. Давайте рассмотрим почему люди считают, что для социальных данных гораздо лучше подходит MongoDB, а не PostgreSQL.

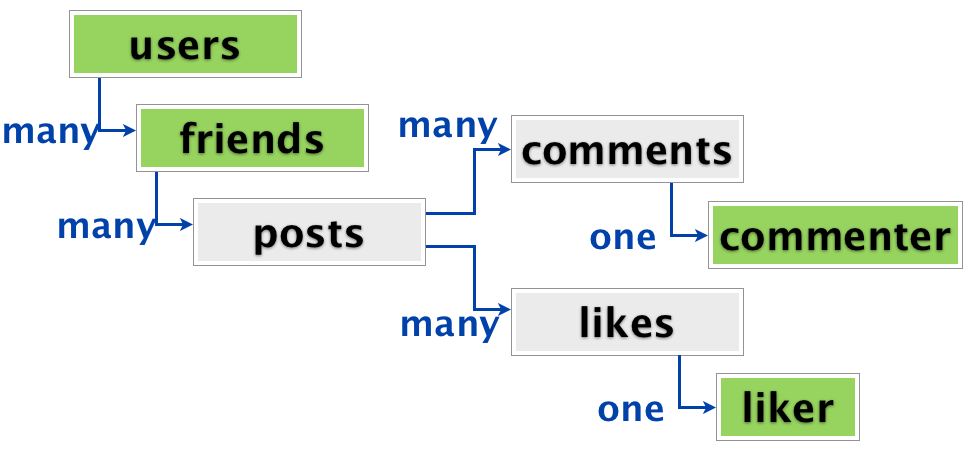
Как MongoDB хранит данные

MongoDB — это документарная база данных. Вместо хранения данных в **таблицах**, состоящих из отдельных **строк**, как в реляционных базах, MongoDB сохраняет данные в **коллекциях**, состоящих из **документов**. Документ — это большой JSON объект без заранее определенного формата и схемы.  
  
Давайте рассмотрим набор связей, которые вам необходимо смоделировать. Это очень похоже на проекты в Pivotal, для которых использовалась MongoDB, и это лучший вариант использования для документарной СУБД, который я когда когда-либо видела.  
  
В корне мы имеем набор сериалов. В каждом сериале может быть много сезонов, каждый сезон имеет много эпизодов, каждый эпизод имеет много отзывов и много актеров. Когда пользователь приходит на сайт, обычно он попадает на страницу определенного сериала. На странице отображаются все сезоны, эпизоды, отзывы и актеры, все на одной странице. С точки зрения приложения, когда пользователь попадает на страницу мы хотим получить всю информацию, связанную с сериалом.  
  
Эти данные можно смоделировать несколькими способами. В типичном реляционном хранилище, каждый из прямоугольников будет таблицей. У вас будет таблица tv\_shows, таблица seasons с внешним ключом в tv\_shows, таблица episodes с внешним ключом в seasons, reviews и cast\_members таблицы с внешними ключами в episodes. Таким образом, чтобы получить всю информацию о сериале нужно выполнить соединение пяти таблиц.  
  
Мы могли бы также моделировать эти данные в виде набора вложенных объектов (набор пар ключ-значение). Множество информации о конкретном сериале это одна большая структура вложенных наборов ключ-значение. Внутри сериала, есть множество сезонов, каждый из которых также объект (набор пар ключ-значение). В пределах каждого сезона, массив эпизодов, каждый из которых представляет собой объект, и так далее. Так в MongoDB моделируют данные. Каждый сериал является документом, который содержит всю информацию, об одном сериале.  
  
Вот пример документа одного сериала, Вавилон 5:  
  
У сериала есть название и массив сезонов. Каждый сезон — объект с метаданными и массивом эпизодов. В свою очередь каждый эпизод имеет метаданные и массивы отзывов и актеров.   
  
Похоже на огромную фрактальную структуру данных.  
  
*(Множество множеств множеств множеств. Вкусные фракталы.)*  
  
Все данные нужные для сериала хранятся одним документом, так что можно очень быстро получить всю информацию сразу, даже если документ очень большой. Есть сериал, называемый «General Hospital», который насчитывает уже 12000 эпизодов в течение 50+ сезонов. На моем ноутбуке, PostgreSQL работает около минуты, чтобы получить денормализованные данные для 12000 эпизодов, в то время как извлечение документа по ID в MongoDB занимает доли секунды.  
  
Так во многих отношениях, это приложение реализует идеальный вариант использования для документарной базы.

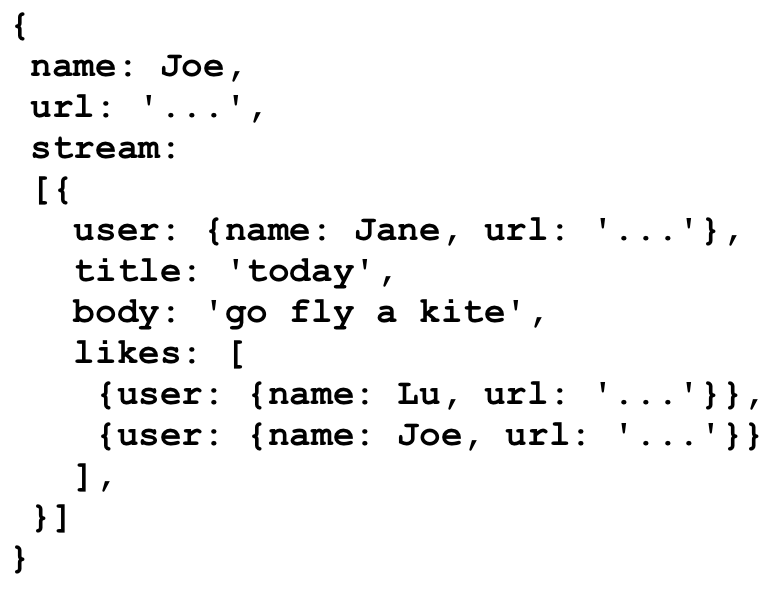
Хорошо. Но как насчет социальных данных?

Верно. Когда вы попадаете в социальную сеть, есть только одна важная часть страницы: ваша лента активности. Запрос ленты активности получает все посты от ваших друзей, отсортированных по убыванию даты. Каждый пост содержит содержит вложения, такие как фотографии, лайки, репосты и комментарии.  
  
Вложенная структура ленты активности выглядит очень похоже на сериалы.  
  
  
Пользователи имеют друзей, друзья имеют посты, посты имеют комментарии и лайки, каждый из которых имеет связан с одним комментатором или лайкером. С точки зрения связей это не сильно сложнее структуры телесериалов. И как в случае сериалов мы хотим получить всю структуру разом как только пользователь войдет в соцсеть. В реляционной СУБД это было бы соединение семи таблиц, чтобы вытащить все данные.  
  
Соединение семи таблиц, ух. Внезапно сохранение всей ленты пользователя в виде одной денормализованной структуры данных, вместо выполнения джоинов, выглядит очень привлекательно. (В PostgreSQL, такие джоины действительно медленно работают вызывают боль — *прим. пер.*)  
  
В 2010 году команда Diaspora приняла такое решение, статьи Esty об использовании документарных СУБД оказались весьма убедительными, даже несмотря на то, что они публично отказались от MongoDB в последствии. Кроме того, в это время, использование Cassandra в Facebook породило много разговоров об отказе от реляционных СУБД. Выбор MongoDB для Disapora был в духе того времени. Это не было неразумным выбором на тот момент, учитывая знания, которые они имели.

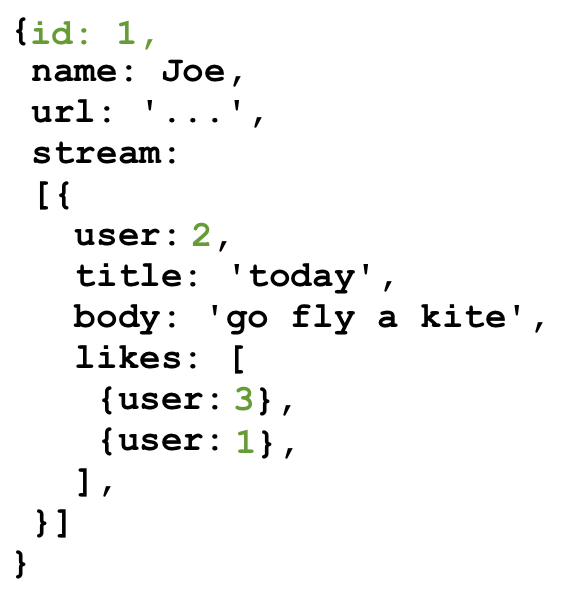
Что могло пойти не так?

Существует очень важное различие между социальным данным Diaspora и Mongo-идеальных данных о сериалах, которое никто не заметил на первый взгляд.  
  
С сериалами, каждый прямоугольник на схеме отношений имеет разный тип. Сериалы отличаются от сезонов, отличаются от эпизодов, отличаются от отзывов, отличаются от актеров. Никто из них не является даже подтипом другого типа.  
  
Но с социальными данными, некоторые из прямоугольников в диаграмме отношений имеют один и тот же тип. На самом деле, все эти зеленые прямоугольники одного типа — они все пользователи диаспоры.  
  
  
Пользователь имеет друзей, и каждый друг может сами быть пользователем. А может и не быть, потому что это распределенная система. (Это целый пласт сложности, который я пропущу на сегодняшний день.) Таким же образом, комментаторы и лайкеры также могут быть пользователями.  
  
Такое дублирование типов значительно усложняет денормализацию ленты активности в единый документ. Это потому, что в разных местах в документе, вы можете ссылаться на одну и ту же сущность — в данном случае, один тот же пользователь. Пользователь, который лайкнул пост также может быть пользователем, который прокомментировал другую активность.

Дублирование данных дублирование данных

Мы можем по-разному смоделировать это в MongoDB. Самый простой способ — дублирование данных. Вся информация о пользователе копируется сохраняется в лайке к первому посту, а затем отдельная копия сохраняется в комментарии ко второму посту.Преимущество в том, что все данные присутствует везде, где вам это нужно, и вы все еще можете вытащить весь поток активности еще в одном документе.  
  
Примерно так выглядит плотностью денормализованная лента активности.  
  
  
Все копии пользовательских данных встроены в документ. Это лента Джо, и у него есть копии пользовательских данных, в том числе его имя и URL, на верхнем уровне. Его лента, содержит пост Джейн. Джо лайкнул пост Джейн, так что в лаках к сообщению Джейн, сохранена отдельная копия данных Джо.  
  
Вы можете понять, почему это привлекательно: все данные вам нужно уже находится там, где вам это нужно.  
  
Вы также можете видеть, почему это опасно. Обновление данных пользователя означает обход всех лент активности, чтобы изменить данные во всех местах, где они хранятся. Это очень сильно подвержено ошибкам, и часто приводит к несогласованности данных и загадочным ошибкам, особенно при работе с удалениями.

Неужели нет надежды?

Существует другой подход к решению проблемы в MongoDB, который будет знаком тем, кто имеет опыт работы с реляционными СУБД. Вместо дублирования данных вы можем сохранять ссылки на на пользователей в ленте активности.  
  
При этом подходе вместо встраивания данных там, где они нужны, вы даете каждому пользователю ID. После этого вместо встраивания данных пользователя вы сохраняете только ссылки на пользователей. На картинке ID выделены зеленым:  
  
*(MongoDB фактически использует идентификаторы BSON — строки, похожие на GUID. На картинке числа, чтобы легче было читать.)*  
  
Это исключает нашу проблему дублирования. При изменении данных пользователей, есть только один документ, который нужно изменить. Тем не менее, мы создали новую проблему для себя. Потому что мы больше не можем построить ленту активности из одного документа. Это менее эффективное и более сложное решение. Построение ленты активности в настоящее время требует, чтобы мы 1) получить документ ленты активности, а затем 2) получили все документы пользователей, чтобы заполнить имена и аватары.  
  
Чего не хватает MongoDB — это операции соединения как в SQL, которая позволяет написать один запрос, объединяющий вместе ленту активности и всех пользователей, на которых есть ссылки из ленты. В конечном итоге приходится вручную делать джоины в коде приложения.

Простые денормализованные данные

Вернемся к сериалам на секунду. Множество отношений для сериалов не имеют много сложностей. Потому что все прямоугольники на диаграмме отношений являются разными сущностями, весь запрос может быть денормализован в один документ, без дублирования и без ссылок. В этой базе данных, нет связей между документами. Поэтому не требуются джоины.  
  
В социальной сети нет самодостаточных сущностей. Каждый раз когда вы видите имя пользователя или аватар вы ожидаете что можно кликнуть и увидеть профиль пользователя, его посты. Сериалы не работают таким образом. Если вы находитесь на эпизоде 1 сезон 1 сериала Вавилона 5, вы не ожидаете, что будет возможность возможность перейти на эпизод 1 сезон 1 General Hospital.

Не. Надо. Ссылаться. На. Документы.

После того, как мы начали делать уродливые джоины вручную в коде Diaspora, мы поняли что это только первая ласточка проблем. Это был сигнал что наши данные на самом деле реляционные, что существует ценность в этой структуре связей и мы двигаемся против базовой идеи документарных СУБД.   
  
Дублируете ли вы важные данные (тьфу), или используете ссылки и делаете джоины в коде приложения (дважды тьфу), если вам нужны ссылки между документами, то вы переросли MongoDB. Когда апологеты MongoDB говорят «документы», то они имеют ввиду вещи, которые вы можете напечатать на бумаге и работать таким образом. Документы имеют внутреннюю структуру — заголовки и подзаголовки, параграфы и футеры, но не имеют ссылок на другие документы. Самодостаточный элемент слабоструктурированных данных.  
  
Если ваши данные выглядят как набор бумажных документов — поздравляю! Это хороший кейс для Mongo. Но если у вас есть ссылки между документами, то у вас на самом деле нет документов. MongoDB в этом случае — плохое решение. Для социальных данных это действительно плохое решение, так как самая важная часть — связи между документами.  
  
Таким образом социальные данные не являются документарными. Это означает что на самом деле социальные данные… реляционны?

Опять это слово

Когда люди говорят «социальные данные не реляционны», это означает не то, что они имеют ввиду. Они имеют ввиду одну из двух вещей:

1. «Концептуально, социальные данные более граф, чем набор таблиц.»

Это абсолютно верно. Но есть на самом деле очень мало понятий в мире, которые, естественно, могут быть смоделированы как нормированные таблицы. Мы используем эту структуру, потому что это эффективно, поскольку это позволяет избежать дублирования, и потому, что, когда это действительно становится медленным, мы знаем, как это исправить.

2. «Гораздо быстрее получить все социальные данные когда они денормализованы в один документ»

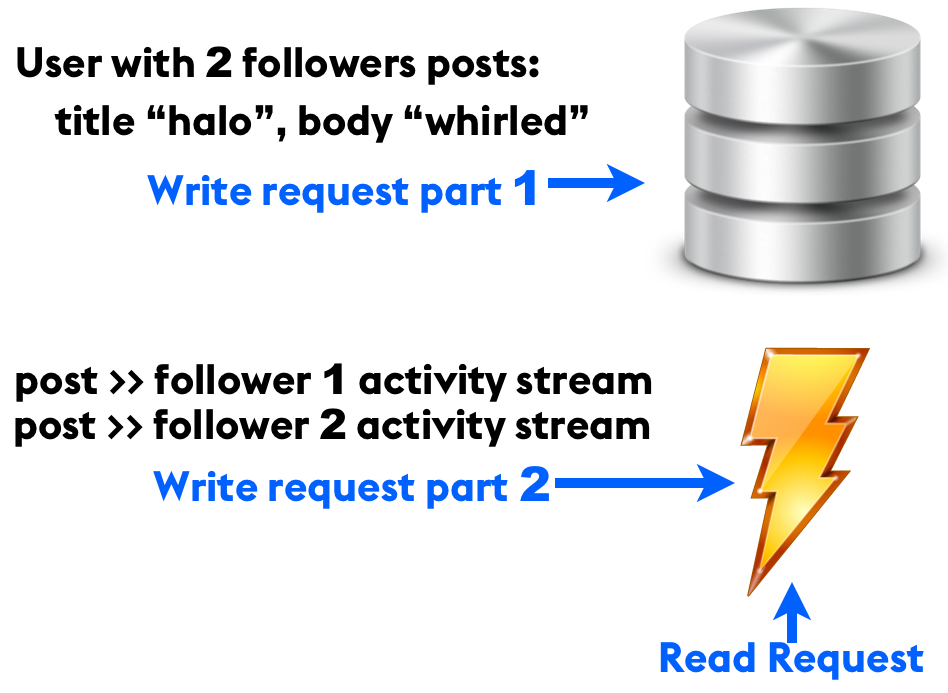
Это также абсолютно верно. Когда ваши социальные данные в реляционном хранилище, вам нужно сделать соединение многих таблиц, чтобы получить ленту активности для конкретного пользователя, и что медленнее с ростом объемов таблицы. Тем не менее, у нас есть хорошо понятное решение этой проблемы. Оно называется кеширование.  
  
На конференции [All Your Base Conf](http://allyourbaseconf.com/) в Оксфоре, где я сделала доклад по теме этого поста, [Neha Narula](http://nehanaru.la/) представила замечательный доклад о кешировании, который я рекомендую посмотреть. Вкратце, кеширование нормализованных данных — это сложная, но вполне изученная проблема. Я видела проекты, в которых лента активности была денормализована в документарной СУБД, как MongoDB, что позволяло получать данные гораздо быстрее. Единственная проблема — инвалидация кеша.

«Есть только две трудные задачи в области информатики: инвалидация кеша и придумывание названий.»

*Фил Карлтон*

Оказывается, инвалидировать кеш на самом деле довольно трудно. Фил Карлтон написал большую часть SSL версии 3, X11 и OpenGL, так что он знает кое-что о компьютерной науке.

инвалидация кеша как сервис

Но что такое инвалидация кеша, и почему это так сложно?  
  
Инвалидация кеша это знание когда ваши данные в кеше устарели и требуется их обновить или заменить. Вот типичный пример, который я каждый день вижу в веб-приложениях. У нас есть долговременное хранилище, обычно PostgreSQL или MySQL, и перед ними мы имеем слой кеширования, на основе Memcached или Redis. Запрос на *чтение* ленты активности пользователя обрабатывается из кеша, а не напрямую из базы данных, что делает выполнение запроса очень быстрым.  
  
  
*Запись* — гораздо более сложный процесс. Предположим что пользователь с двумя друзьями создает новый пост. Первое что происходит — пост записывается в базу данных. После этого фоновый поток записывает пост в закешированную ленту активности обоих пользователей, которые являются друзьями автора.  
  
Это очень распространенный паттерн. Твитер держит в in-memory кеше ленты последних активных пользователей, в которые добавляются посты когда кто-то из фолловеров создает новый пост. Даже небольшие приложения, которые используют нечто, вроде лент активности, так делают (см: соединение семи таблиц).  
  
Вернемся к нашему примеру. Когда автор меняет существующий пост, обновление обрабатывается также, как и создание, за исключением того что элемент в кеше обновляется, а не добавляется к существующему.  
  
Что произойдет, если фоновый поток, обновляющий кеш, поток прервется посередине? Может упасть сервер, отключатся сетевые кабели, приложение перезапустится. Нестабильность является единственным стабильным фактом в нашей работе. Когда такое случается данные в кеше становятся несогласованными. Некоторые копии постов имеют старое название, а другие — новое. Это нелегкая задача, но с кэшем, всегда есть ядерный вариант.  
  
  
Всегда можно полностью удалить элемент из кеша и пересобрать его из *согласованного долговременного* хранилища.  
  
Но что если нет долговременного хранилища? **Что если кеш — единственное что у вас есть?**  
  
В случае MongoDB это именно так. Это кеш, без долговременного согласованного хранилища. И он **обязательно станет** несогласованным. Не «согласованным в конечном счете» (*eventually consistent*), а просто несеогласованным все время (Этого не так сложно добиться, достаточно чтобы обновления происходили чаще, чем, чем среднее время достижения согласованного состояния — *прим. пер.*). В этом случае у вас нет вариантов, даже «ядерного». У вас нет способа пересобрать кеш в согласованном состоянии.  
  
Когда в Diaspora решили использовать MongoDB, то объединили базу с кешем. База данных и кеш — очень разные вещи. Они основаны на разном представлении о стабильности, скорости, дублировании, связях и целостности данных.

Преобразование

Как только мы поняли, что случайно выбрали кеш для базы данных, что мы могли сделать?  
  
Ну, это вопрос на миллион долларов. Но я уже ответила на вопрос на миллиард долларов. В этом посте я говорила о том, как мы использовали MongoDB в сравнении с тем, для чего оно было разработано. Я говорила об этом так, как будто вся информация была очевидна, и команда Dispora просто не в состоянии провести исследование, прежде чем выбрать.   
  
*Но это было совсем не очевидно.* Документация MongoDB говорит о том что хорошо, и вообще не говорит о том, что не хорошо.Это естественно. Все так делают. Но в результате потребовалось около 6 месяцев и много жалоб пользователей и много расследований, чтобы выяснить что мы использовали MongoDB не по назначению.  
  
Делать было нечего, кроме извлечения данных из MongoDB и помещения в реляционную СУБД, на ходу решая проблемы несогласованности данных. Сам процесс извлечения данных из MongoDB и помещения в MySQL был прямолинейным. Более подробно в [докладе на All Your Base Conf](https://speakerdeck.com/sarahmei/switching-data-stores-a-postmodern-comedy).

Ущерб

У нас были данные за 8 месяцев работы, которые превратились в 1.2 миллиона строк в MySQL. Мы провели восемь недель разрабатывая код для миграции и когда запустили процесс основной сайт ушел в даунтайм на 2 часа. Это было более чем приемлемым результатом для проекта в стадии pre-alpha. Мы бы могли уменьшить даунтайм, но мы закладывали 8 часов, так что два часа выглядело фантастикой.  
  
*(NOT BAD)*

Эпилог

Помните приложение для телесериалов? Это был идеальный вариант использования для MongoDB. Каждый сериал был одним самодостаточным документом. Нет ссылок на другие документы, нет дублирования, и нет способа сделать данные несогласованными.  
  
После трех месяцев в разработке все прекрасно работало с MongoDB. Но однажды в понедельник на планерке клиент сказал, что один из инвесторов хочет новую фичу. Он хочет иметь возможность кликнуть на на имя актера и посмотреть его карьеру в телесериалах. Он хочет список всех эпизодов во всех сериалах в хронологическом порядке, в которых этот актер снимался.   
  
Мы хранили каждый сериал в виде документа в MongoDB, содержащем все данные, в том числе актеров. Если этот актер встречался в двух эпизодах, даже в одном сериале, информация хранилась в двух местах. Мы не могли даже узнать что это один и тот же актер, кроме как с помощью сравнения имен. Для реализации фичи надо было обойти все документы, найти и дедуплицировать все экземпляры актеров. Ух… Надо было это сделать как минимум один раз, а потом поддерживать внешний индекс всех актеров, который будет испытывать те же проблемы с согласованностью, как и любой другой кеш.

Видите что происходит?

Клиент ожидает что фича будет тривиальной. Если бы данные были в реляционном хранилище, то это было бы действительно так. В первую очередь мы попытались убедить менеджера что эта фича не нужна. Но менеджер не прогнулся и мы придумали несколько более дешевых альтернатив, вроде ссылок на IMDB по имени актера. Но компания делала деньги на рекламе, поэтому им нужно было чтобы клиент не уходил с сайта.   
  
Эта фича подтолкнула проект к переходу на PosgreSQL. После общения с заказчиком выяснилось, что бизнес видит много ценности в связывании эпизодов между собой. Они предполагали просмотр сериалов, снятых одним режиссером и эпизодов, выпущенных в одну неделю и многое другое.  
  
Это было в конечном счете, проблема коммуникации, а не техническая проблема. Если эти разговоры, что произошли раньше, если бы мы взяли время, чтобы действительно понять, как клиент хочет видеть данные и что хочет делать с ним, то мы, вероятно, перешли бы PostgreSQL ранее, когда было меньше данных, и было легче.

Учиться, учиться и еще раз учиться

Из опыта я узнала кое-что: идеальный кейс MongoDB еще уже, чем наши данные о сериалах. *Единственное*, что удобно хранить в MongoDB — произвольные JSON фрагменты. "*Произвольные*" в этом контексте означает, что вам абсолютно *все равно* что внутри JSON. Вы даже не смотрите туда. там нет схемы, даже неявной схемы, как было в наших данных о сериалах. Каждый документ — набор байт, и вы не делаете никаких предположений о том, что внутри.  
  
На RubyConf я столкнулась с [Conrad Irwin](https://twitter.com/conradirwin), который предложил этот сценарий. Он сохранял произвольные данные, пришедшие от клиентов, в виде JSON. Это разумно. CAP теорема не имеет значения, когда в ваших данных нет смысла. Но в любом интересном приложении данные имеют смысл.  
  
От многих людей я слышала, что MongoDB используется как замена PostgreSQL или MySQL.Нет обстоятельств при которых это может быть хорошей идеей. Гибкость схемы (по факту отсутствие схемы — *прим. пер.*) выглядит как хорошая идея, но на самом деле это полезно только тогда, когда ваш данные не несут ценности. Если у вас есть неявная схема, то есть вы ожидаете некоторую структуру в JSON, то MongoDB — неверный выбор. Я предлагаю взглянуть на hstore в PostgreSQL (в любом случае быстрее, чем MongoDB), и изучить как делать изменения схемы. Они действительно не так сложны, даже в больших таблицах.

Найдите ценность

Когда вы выбираете хранилище данных, самое главное понять где в данных и связях находится ценность для клиента. Если вы пока еще не знаете, то нужно выбирать то, что не загонит вас в угол. Запихивание произвольных JSON данных в базу выглядит гибким решением, но настоящая гибкость заключается в простом добавлении функций для бизнеса.   
  
Делайте ценные вещи простыми.

Конец

Спасибо что дочитали досюда.