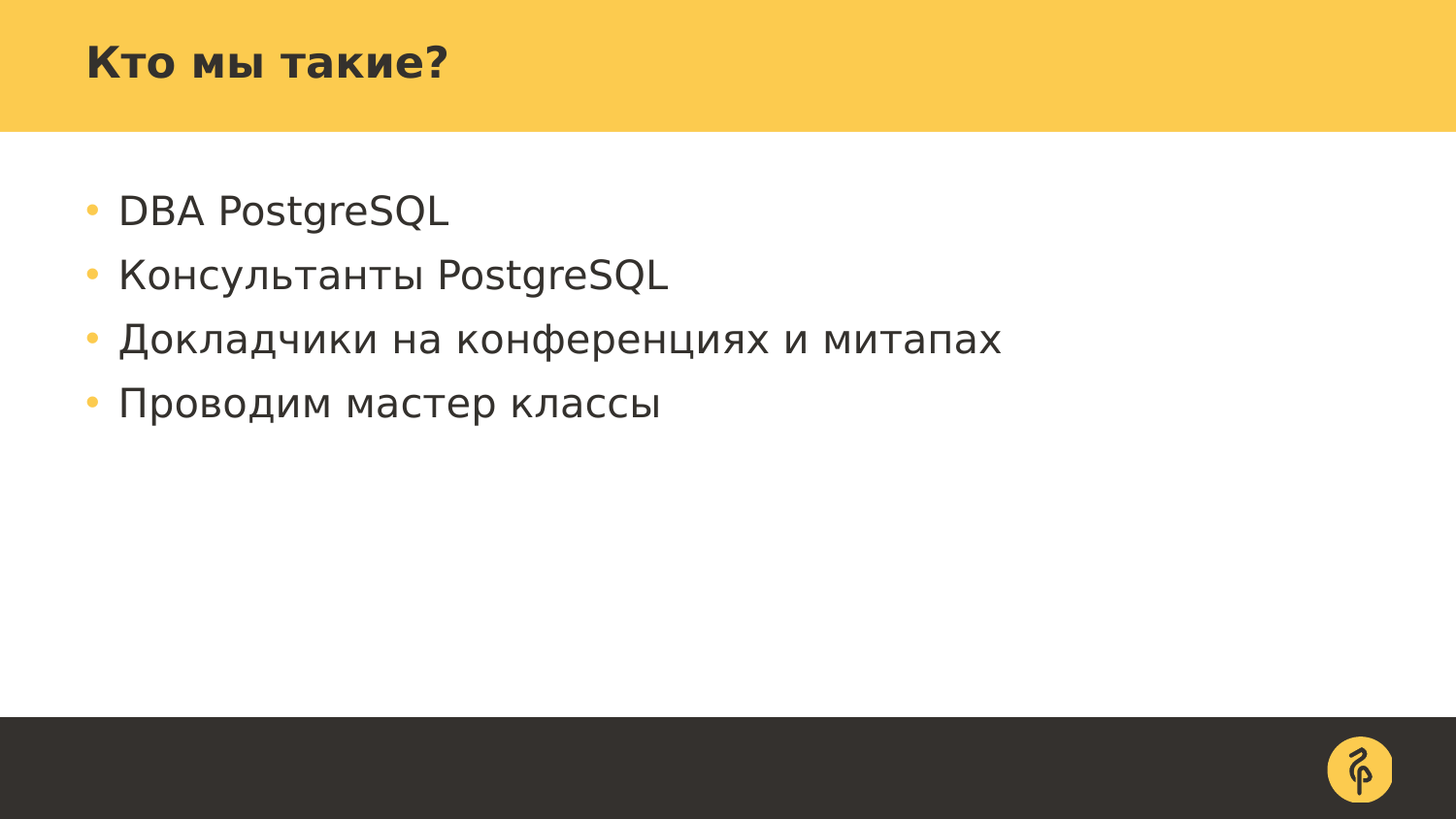
Управление нагрузкой на PostgreSQL, когда одного сервера уже мало. Андрей Сальников

* [Системное администрирование](https://habr.com/ru/hub/sys_admin/),
* [PostgreSQL](https://habr.com/ru/hub/postgresql/),
* [SQL](https://habr.com/ru/hub/sql/),
* [Серверное администрирование](https://habr.com/ru/hub/s_admin/),
* [Администрирование баз данных](https://habr.com/ru/hub/db_admins/)

**Предлагаю ознакомиться с расшифровкой доклада начала 2019 года Андрея Сальникова "Управление нагрузкой на PostgreSQL, когда одного сервера уже мало"**

Основные тезисы:  
1) Стандартные практики распределения нагрузки в PostgreSQL. Сначала мы обсудим причины возникновения высокой нагрузки на базу данных. Следующим этапом рассмотрим те методы распределения нагрузки.  
2) Будут рассмотрены вопросы того, как устроена репликация в PostgreSQL, какие есть различия между синхронными и асинхронными репликами, как правильно настраивать реплики.





Я сейчас начну с представления, кто я такой. Я – администратор баз данных из компании Data Egret. И мы работаем как удаленные администраторы баз данных у разнообразных клиентов. Я обычно называю самые известные по России. Это HeadHunter, Lamoda. У них нет своих DBA по PostgreSQL. И у них DBA – это мы. И помимо них у нас достаточно много других клиентов. И мы сталкиваемся с разным профилем нагрузки, и решаем эти проблемы с разной степенью эффективности у разных клиентов.

Также мы предоставляем услуги консультации, когда человеку нужно просто узнать какие-то аспекты о базе данных, но не нужна удаленная поддержка.

И так как Postgres является open source’рсным решением и там большое community, то принято обмениваться информацией. Мы делимся своим опытом работы на конференциях, рассказывая те или иные аспекты работы с базой данных Postgres.

То, о чем я буду говорить в той или иной мере применимо ко всем RDBMS базам данных. На данный момент такая жирная четверка – это две enterprise: Oracle и MSSQL, и две open source: MySQL с его форками и Posgres, у которого форков тоже достаточно большое количество.

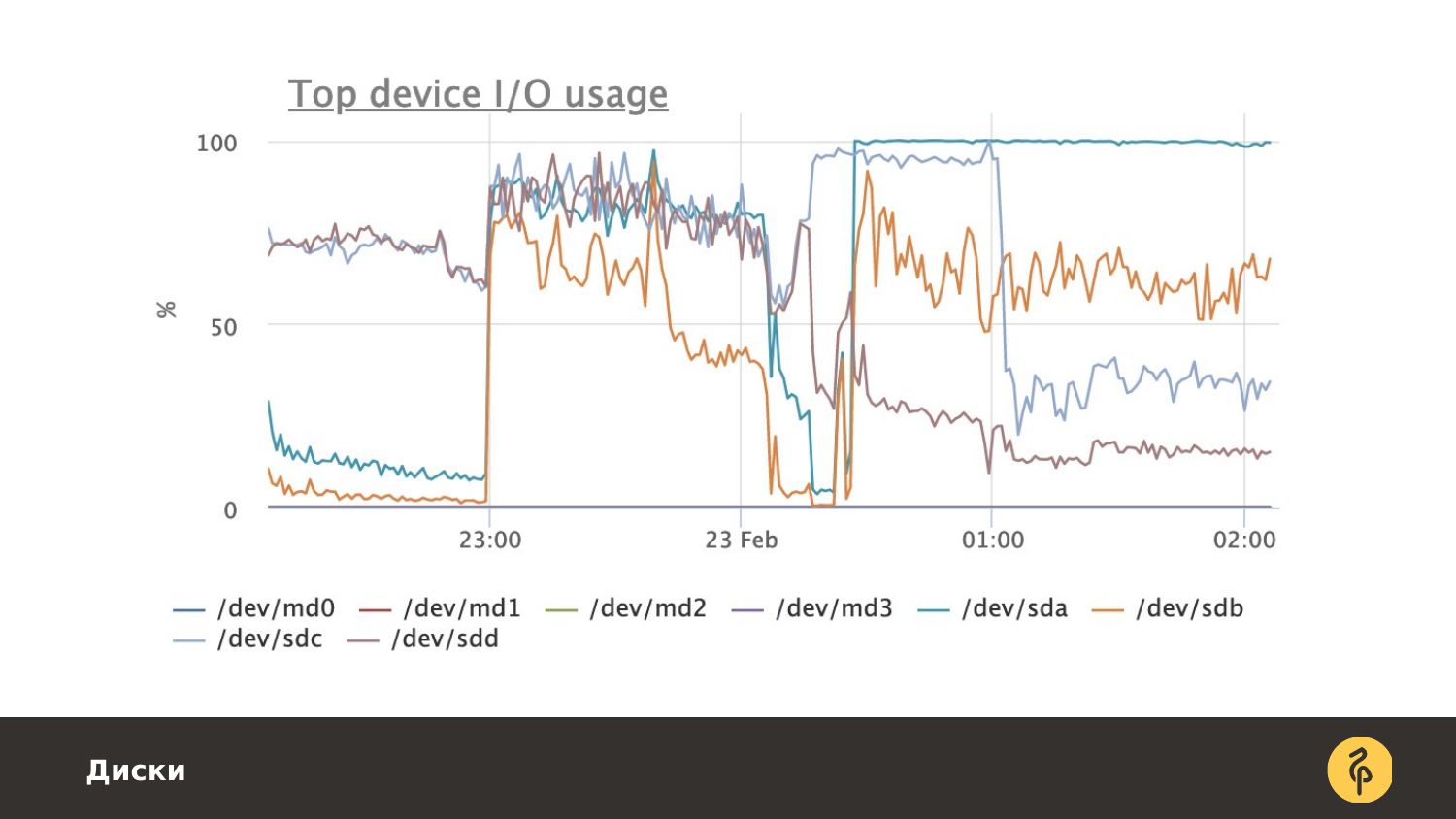
И помимо этого мы проводим какие-то обучения. Но это обычно корпоративный формат. И мы делаем это не так часто. Основная работа – это предоставление услуг администрирования и популяризация Postgres как базы данных.

Вернемся к теме доклада – управление нагрузкой на сервер и когда одного сервера уже мало.

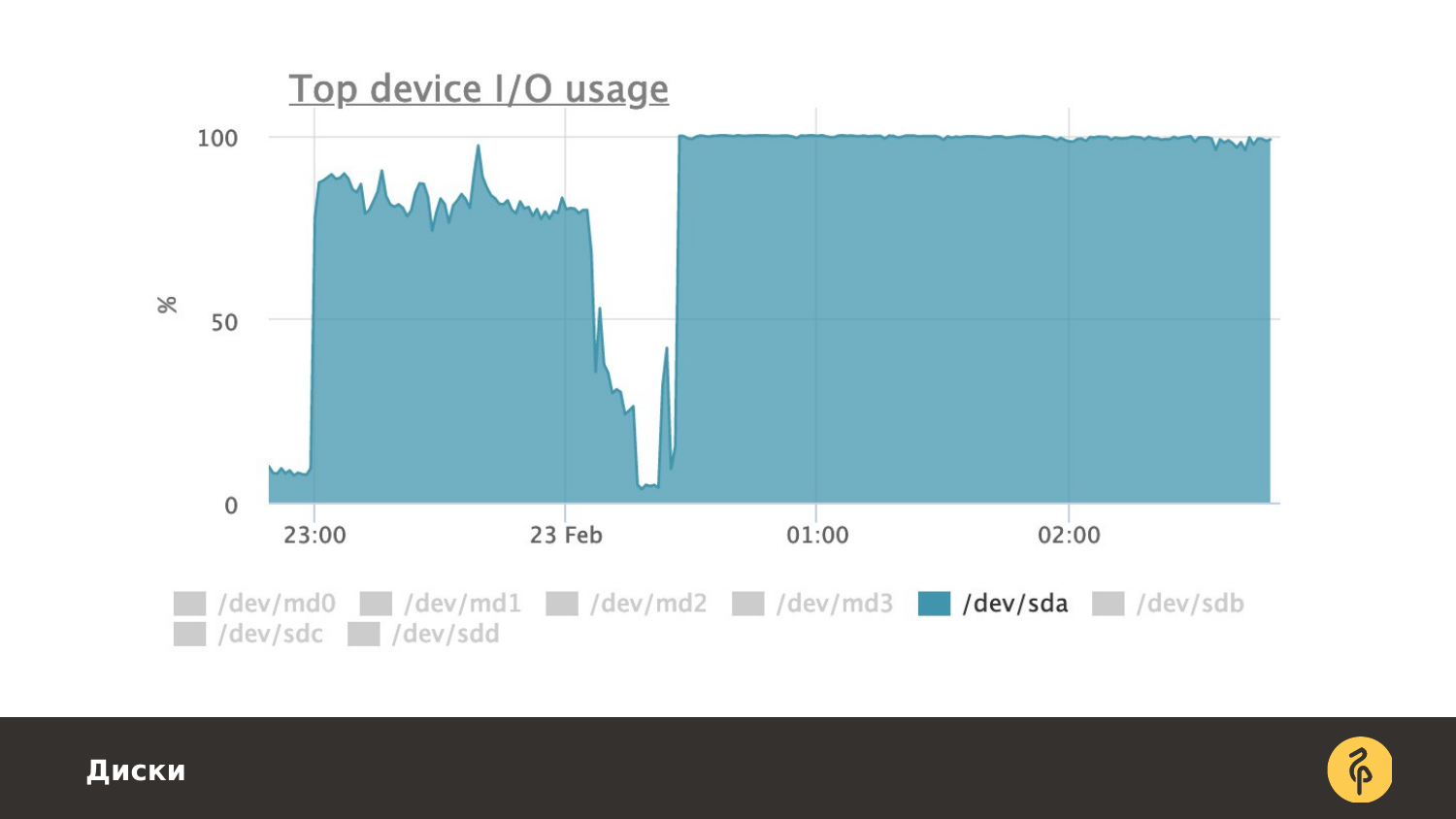
Очень часто нас на конференциях спрашивают о том, какие есть мультимастер-решения и как они работают. Я вам сразу скажу, чтобы развеять всякие наивные мысли, решения есть, но они вам только добавят тормозов. Ускорить ваш процесс они никак не смогут. Решений достаточно много. Делают их основные коммитеры в ванильный Postgres, но делают они их узкоспециализированными, обычно под своих клиентов, которые платят за это много денег. А в общем случае это никак не работает, поэтому все лечится тем, что мы оптимизируем работу основного сервера базы данных.



Начнем со страшилок. Как проявляется нагрузка на базу данных? Я не буду рассказывать о том, что нужно мониторить. Об этом есть куча докладов, просто покажу картинки-страшилки, когда мы в полку кладем утилизацию по дискам.

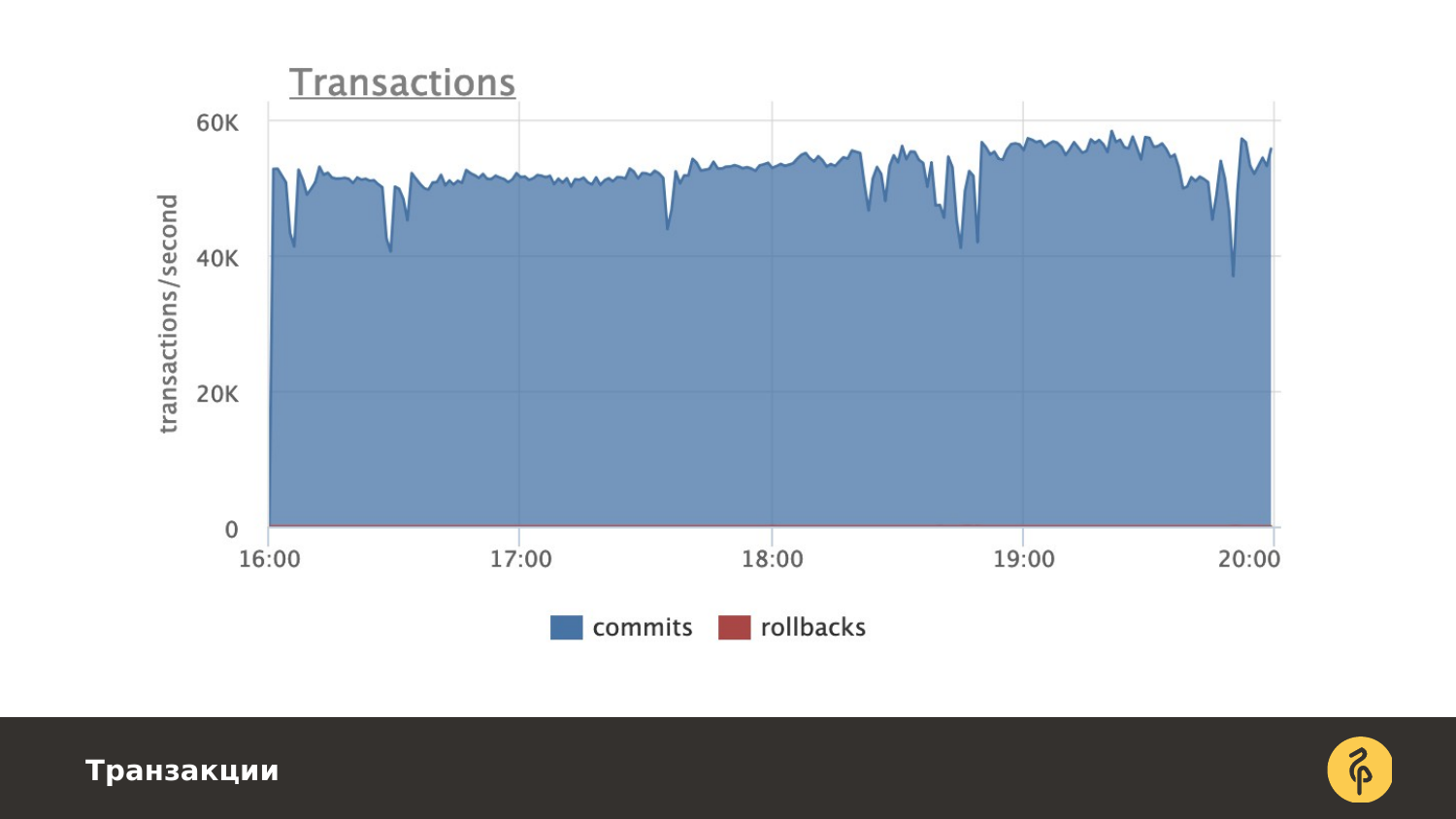


Я отдельно выделил диск, на котором крутится база данных.



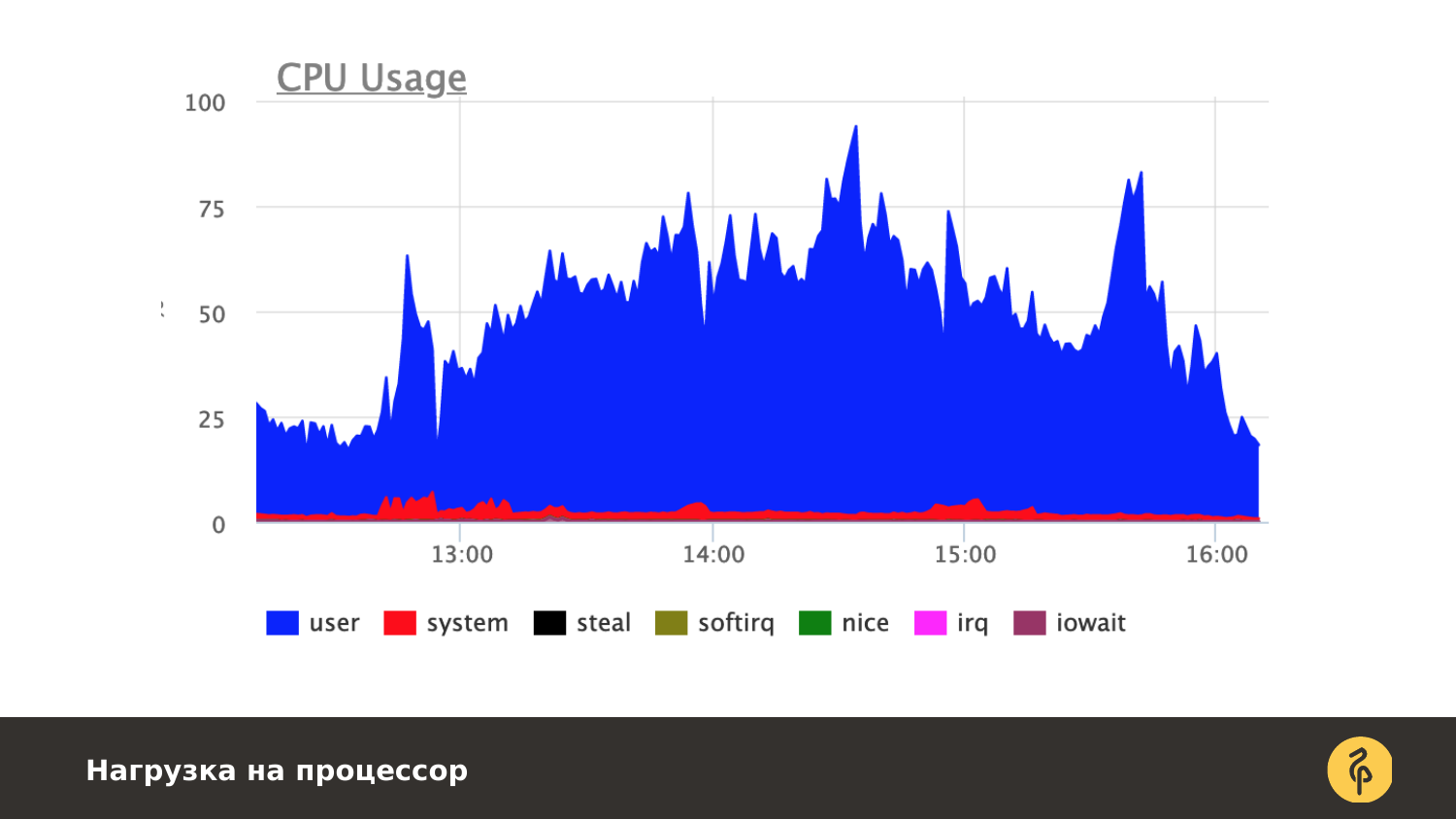
И тут, как вы видите, у него утилизация просто в полку. Это график с реальной базой данных нашего клиента. Это может быть по разным причинам, когда мы или генерируем очень много временных файлов для обработки запросов, или запускаем тяжелую аналитику, что, в принципе, синоним, или у нас не влезают операционные данные в нашу оперативку и нам приходится постоянно бомбить диск.

Вопрос. Предположите, что тут происходило? На самом деле тут у клиента возникли проблемы с местом на диске, и мы ему raid 1 пересобрали в raid 0. Это было на работающей продуктовой базе. Это мастер-сервер. И она работала, держала нагрузку, но пересборка положила в полку утилизацию по диску. Но бывают ситуации, когда и запросами так кладут в полку. Просто я все графики брал с наших клиентов, там таких уже ситуаций нет.

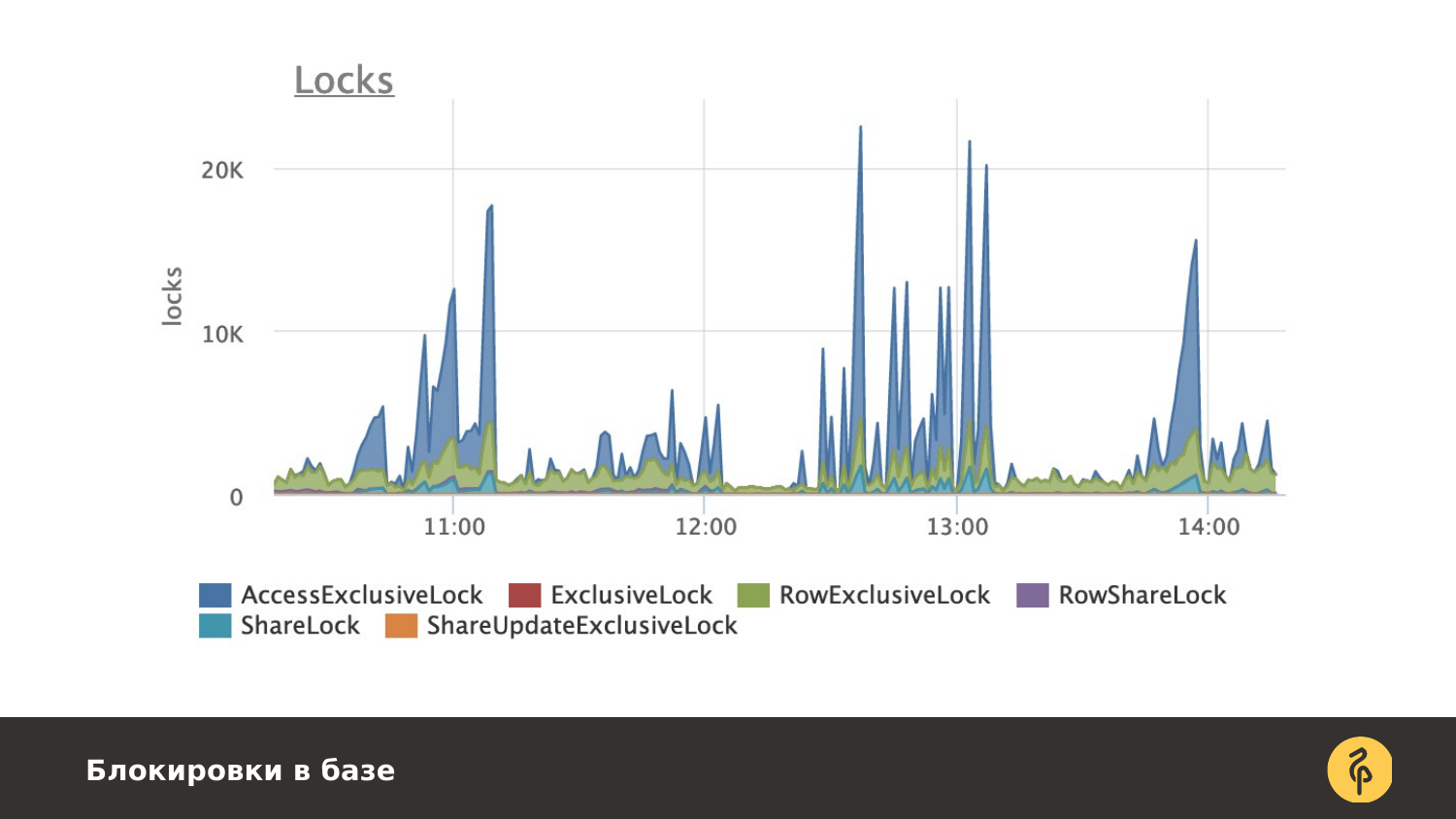


Это нездоровое количество транзакций, потому что если вы работаете правильно с базой данных, то у вас транзакций держатся на нагруженных системах – 20 000-30 000. Если у вас свыше 40 000, значит, вы делаете что-то не так и читаете кучу ненужных данных. Но тем более он продолжает работать.

Вот график той же самой работающей базы, у которой диск в полку был. У этих ребят также в среднем 50 000 транзакций в секунду. Нормально все работает.



Когда мы в полку кладем наши процессоры, это тоже от того, что мы бомбим базу какими-то безумными запросами и нужно что-то с этим делать. Бывают и другие вещи. Но в основном – это особенности работы приложения с базой данных и от того, что на этом стыке пытается сделать приложение с базой данных. Возникают проблемы. Иногда они на стороне базы данных, иногда они на стороне приложения. Нужно копаться и с той, и с другой стороны. Т. е. мы, как администраторы базы данных, можем что-то покрутить в базе данных, но нам нужна обратная связь от разработчиков, потому что если проблема в приложении, то это нужно править на стороне приложения. И тут мы бессильны.



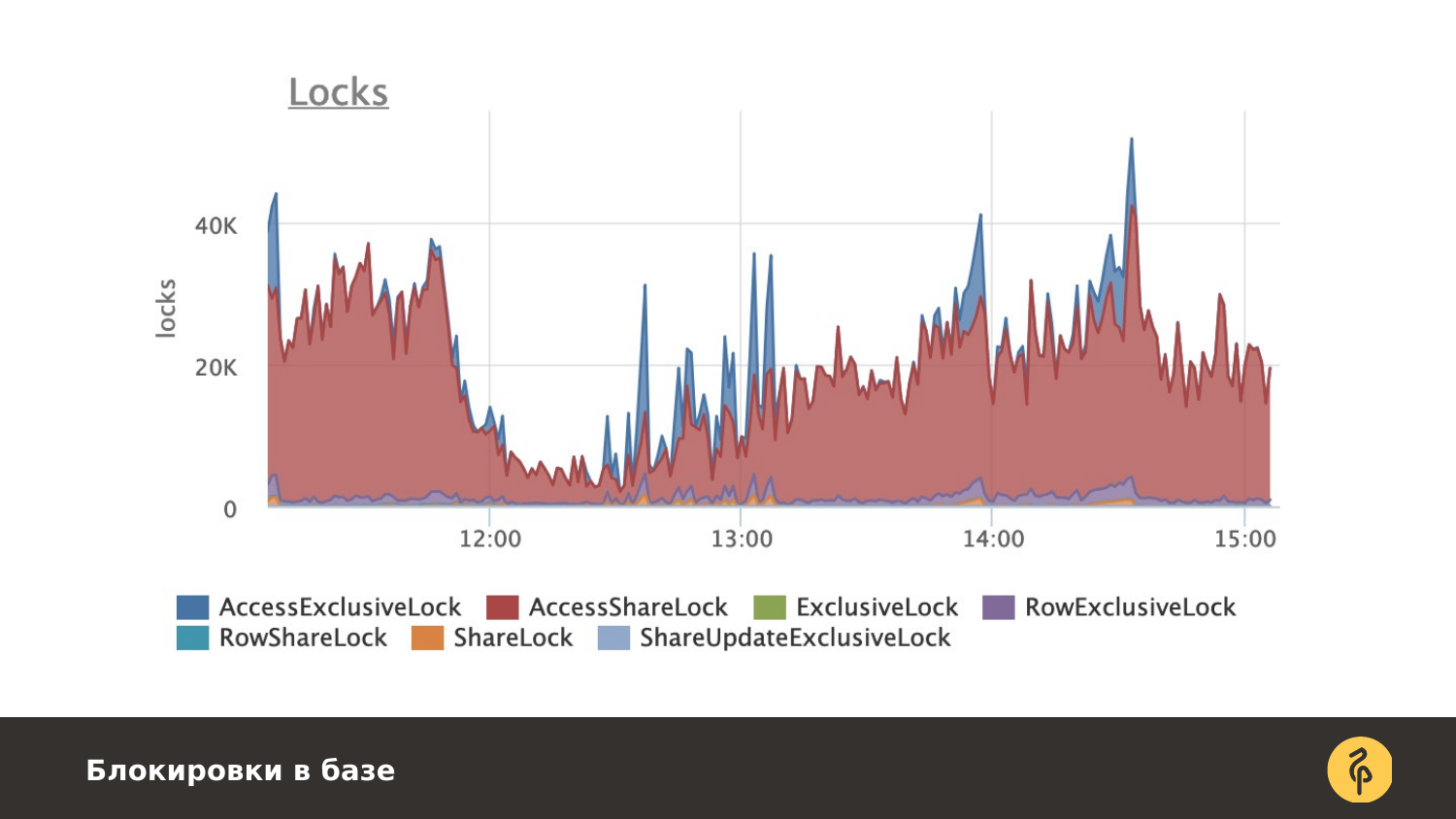
Вот это тоже нездоровая картина. Это блокировки по базе данных разного типа. Все базы данных, даже очереди самые банальные, даже файловые системы берут блокировки в том или ином виде на объекты. Потому что необходимо упорядочить запись и чтение. И есть операции, которые друг с другом конфликтуют. И для этого придумывают разное количество блокировок.

Есть нестрашные блокировки, есть страшные блокировки. Полезете в любую базу данных – страшными будут называться «exclusive», нестрашные будут называться «share».

Когда у вас есть пики по 20 000 блокировок и они эксклюзивные (вот эти синие пики), то это значит, что у вас совсем-совсем все нездорово.

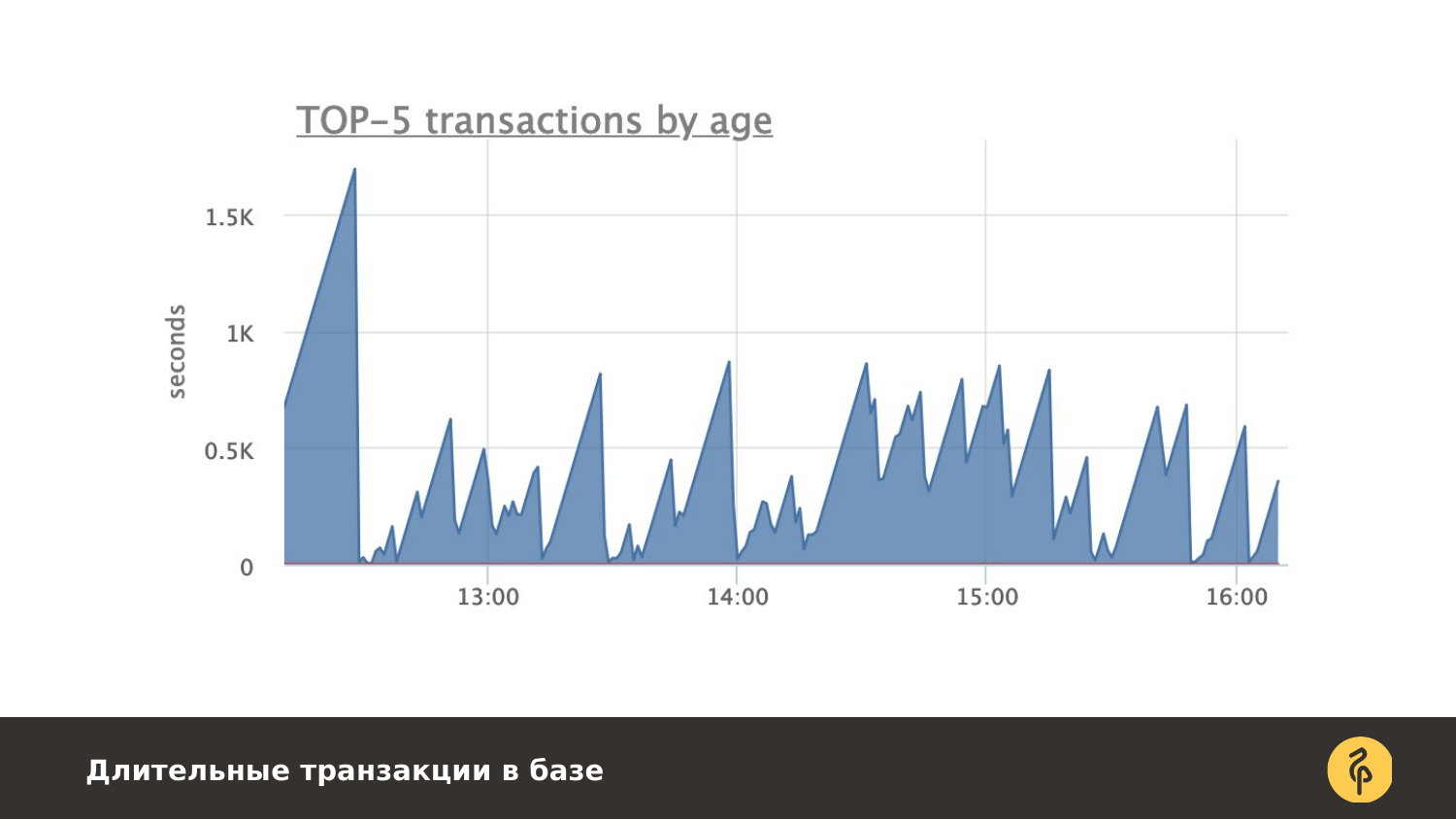
Я упомянул о двух наших клиентах: HH и Lamoda. Как вы думаете, сколько у них там пики? У них 10-20-50 эксклюзивных блокировок в секунду, не больше.

И на этом графике еще обрезан большой пласт читающих блокировок. Если мы добавим этот пласт, то это выглядит вот так.

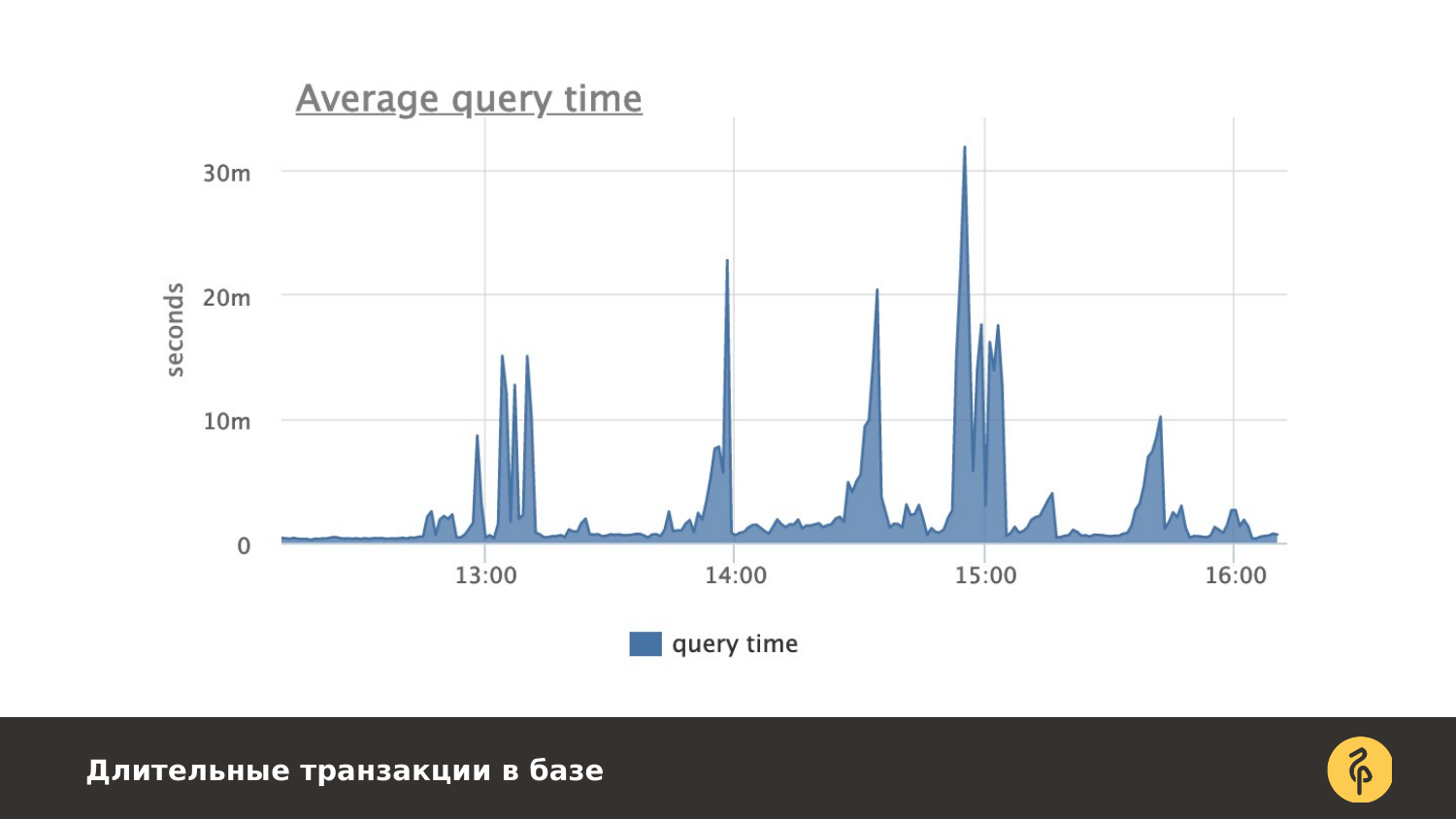


И, в принципе, база с этим нормально работает. Но все механизмы блокировок, которые пишутся, обычно реализуются внутри самого кода базы данных. И в какой-то степени опираются на операционную систему. И теперь, когда у нас такое большое количество блокировок, мы получим тормоза на том, что мы будем пытаться встать в эту очередь блокировок, разбирая, какие эксклюзивные, а какие не эксклюзивные. И это будет занимать достаточно большое время.

К примеру, я недавно столкнулся с ситуацией, когда поднимался бэкенд базы данных Postgres и он не мог стартануть, потому что при старте ему нужно было вклиниться и встать в очередь блокировок. И делал он это средствами операционки. Postgres опирается на Posix, и, соответственно, в этом отношении полностью доверяет операционке. Он попросил операционку – поставь мое желание взять блокировку в очередь. Она лезла и смотрела очередь, видела, что там до фига и говорила: «Нет, я ухожу» и уходила в sleep. И бэкенд зависал, и даже не становился полноценным. Просто поднимался и висел в ожидании, когда сможет в очередь вклиниться. Вот это самая главная боль, которую несут разработчики базы данных.

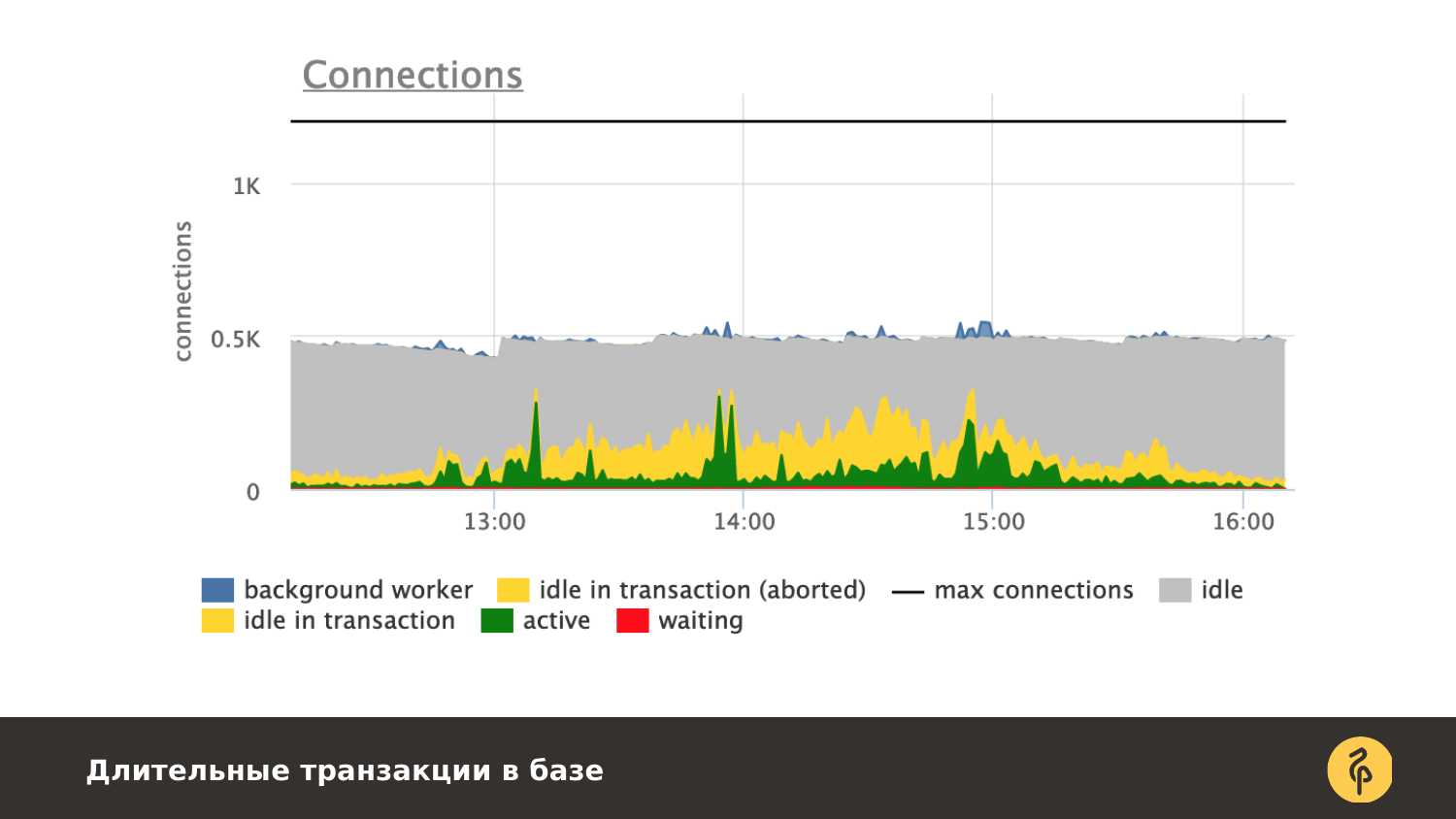


Тут нужно в совокупности смотреть графики. Это график времени транзакций.



А это график среднего времени выполнения запроса на той же самой базе и в те же самыми timestamps.

И как видите, время выполнения запроса укладывается в 30 миллисекунд. Это очень хорошо. А большая часть за миллисекунду выполняется. Но при этом у нас довольно длительные транзакции. И такие вещи в любой RDBMS базе данных несут прямое зло как базе данных, так и тормоза приложению. И это зло будет выражаться в этой картинке блокировок, потому что транзакция берет себе какой-то набор блокировок. Если вы пишите, то там будут эксклюзивные блокировки. Если читаете, то там будут шаренные блокировки. И их всегда будет достаточно много.



И еще одна картинка-страшилка. Она относится тоже к разработчикам, которые работают с базой данных. Сейчас я дам некоторые пояснения.

То, что серенькое – это idle-соединение. Idle-соединение с базой данных возникает, когда мы используем какой-нибудь pool connection. И эта вещь правильная, потому что когда мы пытаемся получить очередной коннект от базы данных, то запускаем процессы создания нового бэкенда и подключение к нему. Это занимает некоторое время, потому что нам нужно инициировать новый код, скопировать туда системные каталоги, в которых отражена структура нашей базы данных, структура хранимых процедур, индексов, типов, чтобы бэкенд мог добросовестно обслуживать ваше соединение. Это частенько добавляет существенный overhead, особенно, если мы используем базу данных под веб-сервисы, когда нам нужны миллисекундные ответы.

Поэтому логичное решение – это держать эти бэкенды уже поднятыми и по мере необходимости их использовать. И эта серенькая ровненькая полочка – это как раз наш резерв, которым мы играемся.

Зеленым – это то, что у нас действительно работает. Это какие-то операции, которые мы делаем в базе данных: читаем в нее, пишем из нее. Это реально работающие транзакции.

Желтым – это когда мы в базе данных сделали какую-нибудь операцию, которая поставила блокировки. И на стороне приложения ушли заниматься другими вещами, при этом забыв закрыть транзакцию, забыв отпустить блокировки, которые возникли.

Что мы можем в приложении сделать? Мы можем вычитать данные и пойти отправлять почту. Почта отправляется у нас секунду.

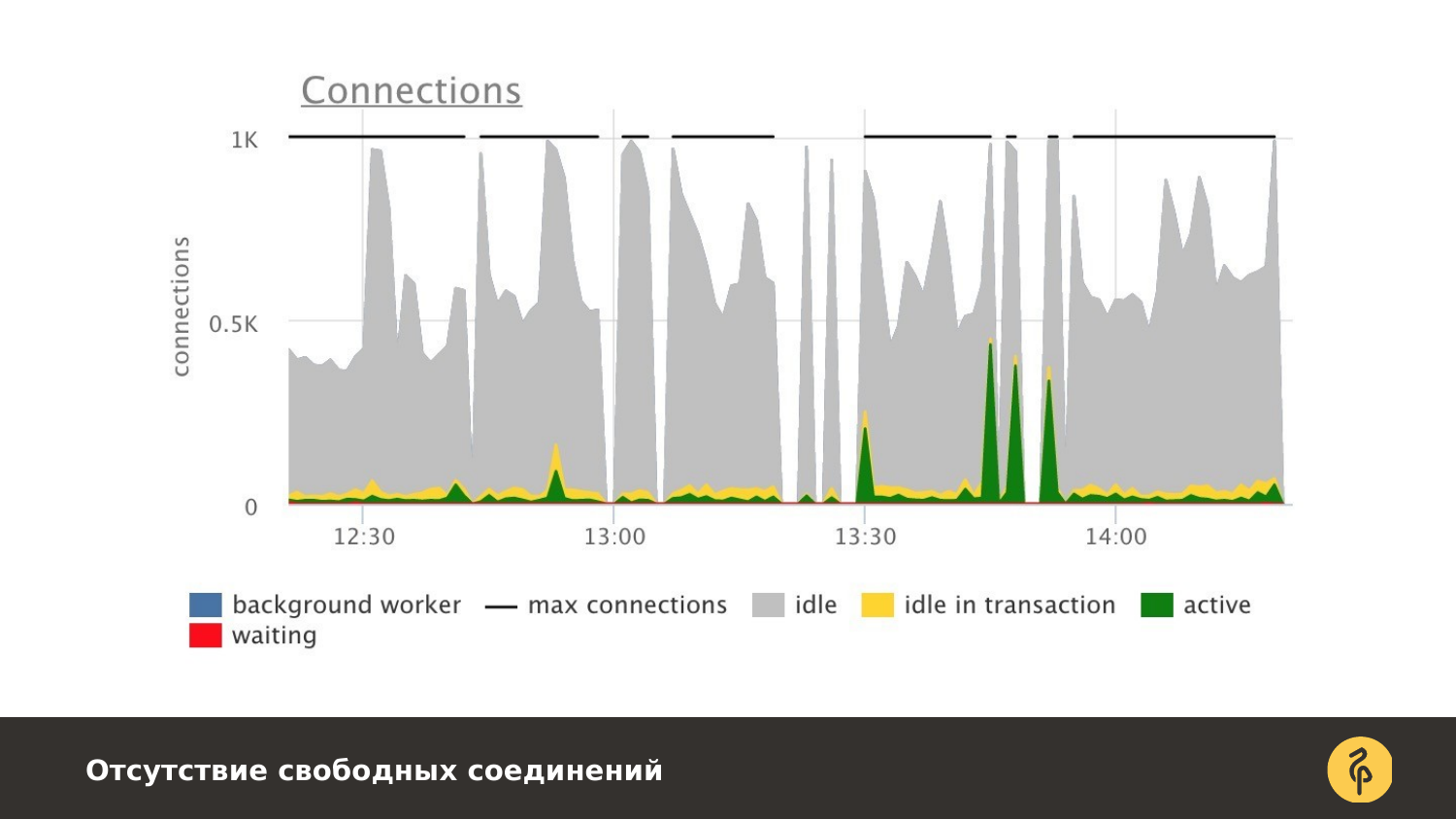
Вот здесь висит долгая транзакция, которая ничего не делает. Чем она плоха для базы данных? Во-первых, порождает большое количество блокировок. Во-вторых, из-за специфики хранения данных в Postgres (у него MVCC), он из таблицы реально не удаляет никогда строчки. Когда вы делаете update/delete, вы просто помечаете строчку как удаленную или записываете новую строчку с новым идентификатором, или ничего не пишите, но там копится мусор из таких старых строчек, которые уже никуда не нужны.

Но когда мы в idle in transaction подвесили сессию, застолбив за своей транзакцией определенный айдишник этой транзакции, то внутренние механизмы чистки этих ненужных строчек у Postgres останавливаются. Это для них порог, т. е. тот самый автовакуум, про который любят говорить. Он доходит туда, останавливается и ничего не делает. Это выльется для вас в то, что таблицы и индексы будут пухнуть. И когда эта транзакция все-таки закончится (может быть), то у вас вместо одной таблички в мегабайт полезных данных будет табличка в 10 мегабайт, в 300 мегабайт, в 400 мегабайт и до гигабайтов может даже долезть куча бесполезных данных. А полезной информации всего один мегабайт будет. Это специфика движка Postgres, с которой надо считаться.

И красным – это waiting. Это такая штука, когда после эксклюзивной блокировки другие процессы не могут параллельно что-то делать с этой строчкой. Они встают в очередь и ждут, когда эксклюзивная блокировка отпустит или строчку, или таблицу. Они допустимы в небольших количествах, но лучше стараться, чтобы их не было.

И когда у вас желтый график такой большой, то это очень большая проблема. Совсем от него вы не избавитесь. И не избавитесь в следующих случаях: когда вы используете какой-нибудь язык, у которого нет нормальной многопоточности. К примеру, Ruby, Python. Они многопоточность эмулируют, выстраивая свои задачи в очередь. И если вы в этих задачах работаете с базой данных, то они сходили в базу данных, что-то сделали, потом встали в очередь. И в итоге у нас повисла idle in transaction.

В данном случае нужно это разбивать и запускать на отдельных instances отдельными приложениями, т. е. как-то разделять на кусочки. Но в любом случае вы от этого не избавитесь однозначно в таких языках. В других можно избавиться. И если нет вот этих желтых линий, то у вас все хорошо будет с базой данных.



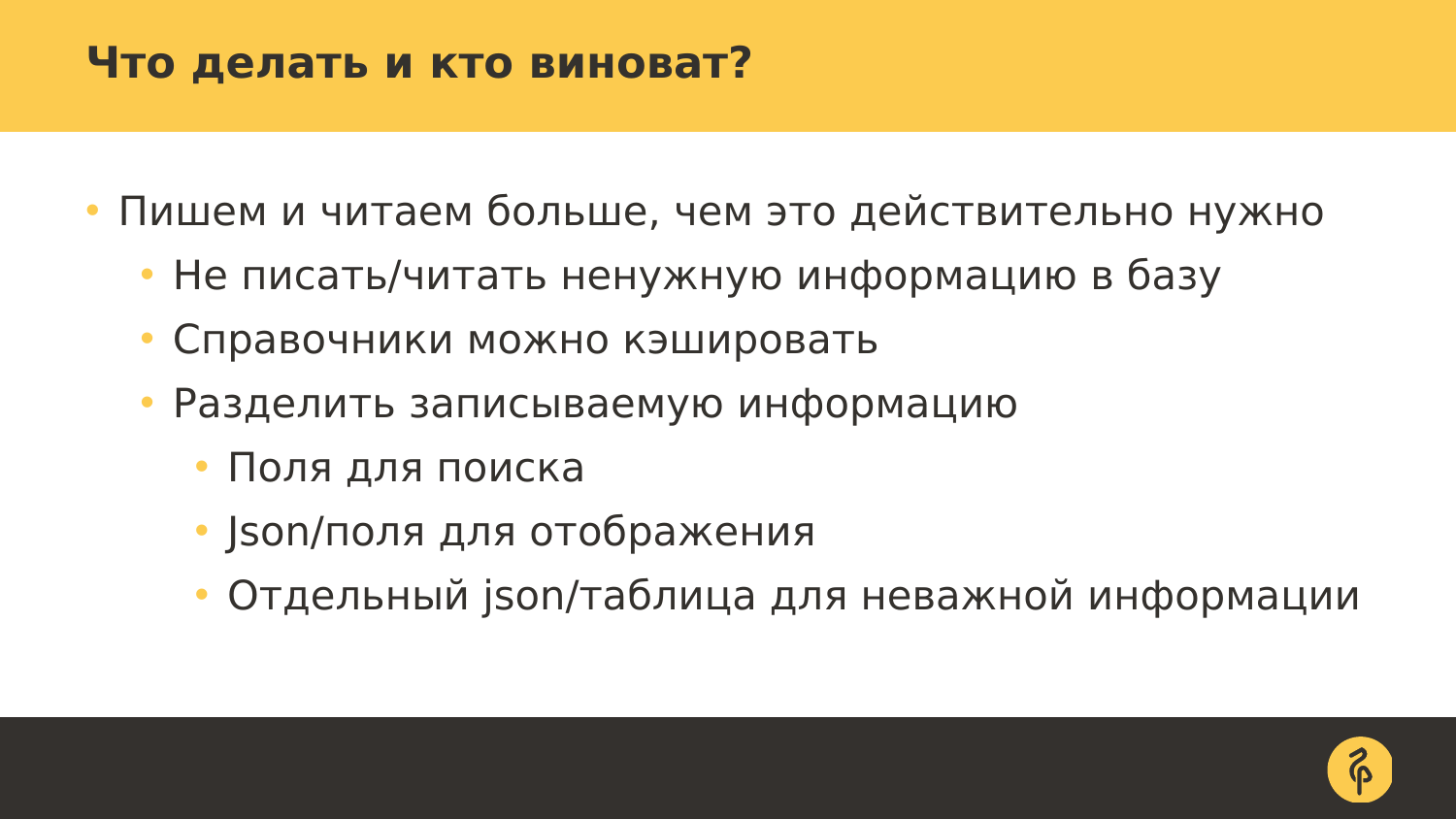
Последняя картинка-страшилка. Это разговор о том, когда мы не используем pool connection соединений. И есть pools, встроенные в драйвера языков программирования. Hibernate может поднимать свои pools. Но когда у нас куча разных софтов ломятся в базу данных, это не единый pool. И обычно ставят прослойку между базой данных и приложениями, которая делает pool соединений сама.

И если ее не поставить, то мы можем пробивать потолок в максимальном количестве соединений в базу данных. И, как вы видите, тут этот потолок пробивается именно idle-соединениями. Т. е. куча приложений пришло, которые зарезервировали себе соединение. И когда приходит очередное приложение, и требует себе новое соединение, у нас его нет, база не может его отдать. В данном случае это был мониторинг, который пытался метрики снять. Он пришел, все соединения забиты. Он ничего не снял. И тут у нас такие дырки в графике.

Это о том, что у нас убивает производительность и примерно, как это происходит в контексте картинок.



Сейчас более детальные причины, которые вам будут полезны, наверное, как разработчикам. И со стороны разработки вы можете над ними поработать, потом придя к себе домой или в офис.



Разберем очень частые причины у всех.

Пишем в базу данных и читаем из базы данных больше, чем надо. Бывает две ситуации. Пишем слишком мало, но это является уже проблемой разработки, а проблема базы – это когда пишем много. Мы замусориваем базу данных бесполезными логами, которые можно было просто в лог-файлы складывать.

Когда вы проектируете свою работу с базой данных, то необходимо четко понимать, какие данные вам действительно там нужны, т. е. к которым вы будете на самом деле обращаться, а какие данные не нужны.

У меня есть отличный пример из практики, когда у одного из наших клиентов была база в 2,5 терабайта. И из этой базы 2 терабайта занимала одна табличка – transaction log. Это была платежная система. Они били пяткой в грудь и говорили, что эти данные им нужны. Через три месяца они попросили удалить данные дальше, чем за полгода. Я подготовил эту операцию. Удалил данные. И в итоге удалил 1,5 терабайта. А потом еще через полгода они пришли и сказали: «Нам вообще не нужны эти данные. Мы поняли это». И я удалил оставшиеся 0,5 терабайта. И база из 2,5 терабайт стала в 0,5 терабайта размером. Я привожу этот пример, потому что тут цифры впечатляющие.

Менее впечатляющие примеры происходят у нас ежедневно. И постоянно в процессе разработки люди понимают, что те или иные данные не нужны. Если вы это поняли, то не стесняйтесь, удаляйте их. Это мы сейчас говорили про не писать.

Не читать ненужную информацию. Это в основном касается ORM-систем. Потому что есть такая там штука как предварительное чтение и кэширование информации. И эти механизмы, когда вы описали классы, которые ложатся на несколько табличек и у вас инициируется приложение в поток, который должен прочитать эти данные. И он начинает все это читать.

А реально конечному пользователю, который пользуется вашим приложением, нужны 2 цифры из, допустим, 490 строк, которые вы вычитали. Зачем вычитали эти 490 строк, когда вам нужно работать с двумя, непонятно. При этом у них размер может быть в два-три мегабайта. Но об этом будет чуть дальше.

Эти вещи вы можете абсолютно спокойно контролировать, потому что вы знаете, что хотите показывать пользователю. Если вы тащите из базы данных больше, чем надо, то значит, вы это делаете зря, и она у вас гибнет под нагрузкой, отдавая ненужную информацию и насилуя диски.

Есть другой аспект. Есть такая вещь как справочники, которые меняются, как правило, достаточно редко. А запросы к ним долбятся достаточно часто. Это бывает в контексте, когда мы открыли какую-нибудь формочку. Есть там, допустим, выпадающий список, которым мы показываем какие-то поля для фильтрации, взятые из справочника.

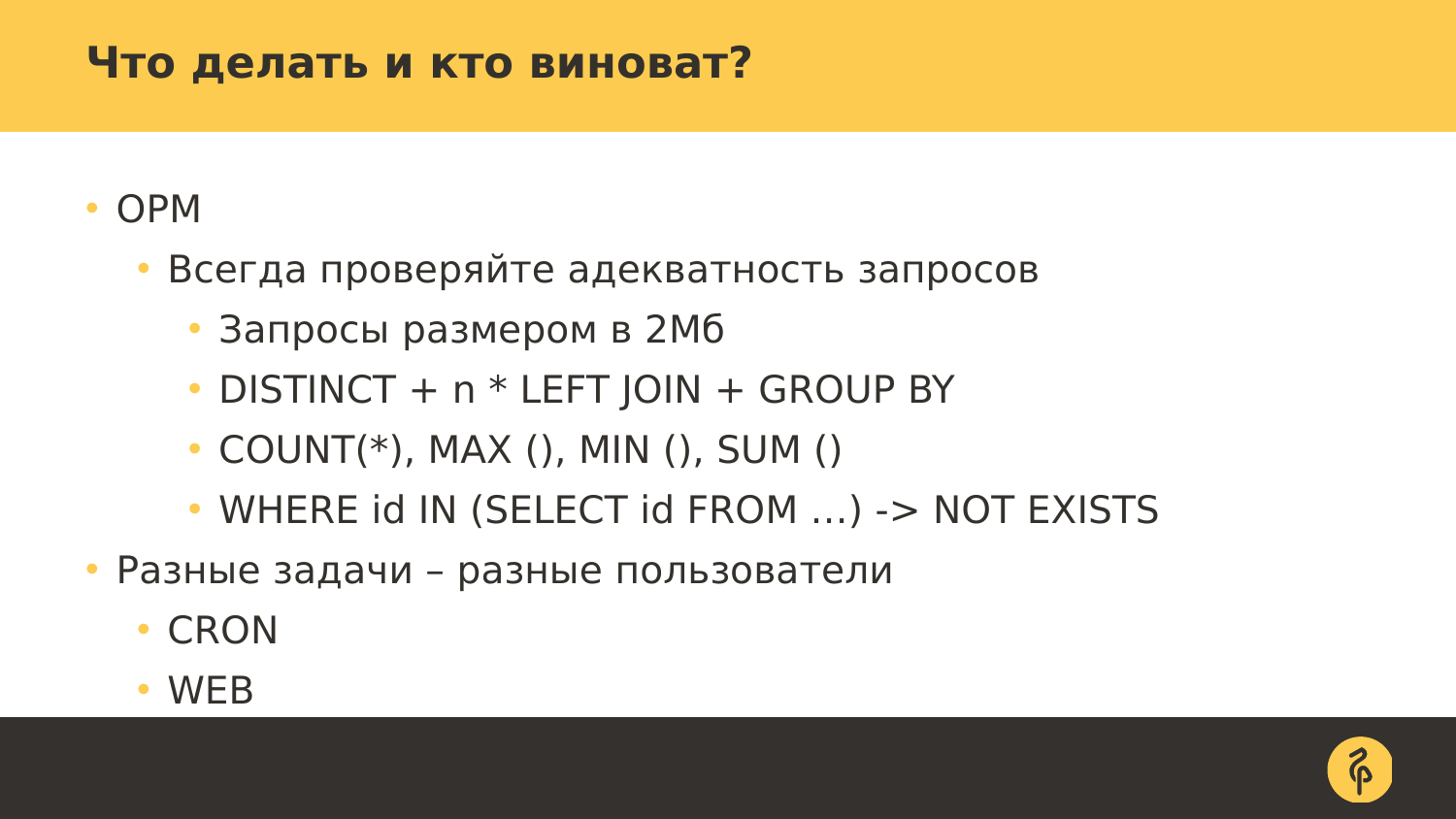
И зная то, что у нас справочник этот меняется раз в год, мы его можем читать один раз при старте нашего приложения. Вычитали этот справочник, оставили у себя в памяти и все, не лезем в базу. Иначе мы начинаем генерировать на каждый такой фильтр кучу запросов, которые ползут в базу данных. Они маленькие. Они работают быстро, но они забивают ее время, которое мы можем потратить на действительно полезные вещи. И кэширование справочников довольно частенько срезает очень сильно загрузку по процессору и сетевой трафик. Хотя, казалось бы, мелкая вещь – 3-4 строчки обычно из справочника берут, но вот так вот.

Нормализация данных. Тут вопрос двоякий. Нельзя слишком сильно нормализировать данные, нельзя слишком сильно денормализировать данные.

В каком аспекте нормализация хорошо работает? Возьмем пользователя, который логинется в систему. И нам нужно проверить, когда он был в последний раз, какой у него баланс и еще какую-то небольшую информацию. И еще есть большая портянка о том, кто он такой: пол, рост, вес. Это, кстати, относится все к пункту «Читаем и записываем больше, меньше». И эту портянку нам не надо тянуть, поэтому имеет смысл оптимизировать эту всю дополнительную информацию, которая требуется один раз, когда человек заполняет форму. И второй раз, когда делается рассылка, и мы к нему хотим обратиться со всеми эпитетами, которые он пожелал.

Эту портянку нам необязательно каждый раз читать. И мы ее можем отложить в отдельную табличку или поле Json, которые вообще не будем тянуть из базы до тех пор, пока нам не понадобятся. Т. е. тут или в какой-нибудь небольшой Json складываем или в отдельную табличку с дополнительной информацией о пользователе. И это можно продлить дальше на другие аспекты. Просто я стараюсь приводить такие примеры, которые будут достаточно понятными.

Об этой части я частично рассказал. У нас есть поля, по которым мы можем искать, есть поля, которые мы должны показывать и есть поля, которые используются крайне редко. И это все дело можно разделить. Для поиска поля всегда должны быть у нас отдельные. И довольно частенько мы должны проиндексировать. А поля, которые мы показываем, они уже неиндексированные, но их можно хранить полями. Если они все время достаются одинаково, мы их можем запихнуть в Json, который будем на стороне приложения распарсивать и совать в нужные части формочки. И еще один Json, который нужно редко. Если он совсем не нужен, то его лучше не писать.



Про ОРМ я рассказал. Есть еще одна ОРМ, когда мы наверчиваем кучу join. Мы смотрим на работу с базой данных сквозь призму своего языка разработки. Как правила, данные, которые записаны в таблицах, транслируем на наши классы, объекты, что и в каком языке используется.

Я часто с Java сталкивался, поэтому могу в контексте hibernate говорить. Есть Hibernate, который реализует у нас, что мы хотим получить из класса. Он из этого делает запрос и отправляет в базу данных. И получается запрос, если в 2 мегабайта, то это еще хорошо. У нас есть личная табличка рекордов у каждого, кто и какой длины запрос встретил. 2 гигабайта – это запрос, это не предел ОМР.

В последний раз, когда я разбирал запрос, сгенерированный в Python, я насчитал у него 490 колонок, которые он вытаскивает. Это бухгалтерская система. И там было 5 СНИЛСов по одному и тому же человеку. Т. е. за ОРМ нужно следить, потому что подгадить он любит всегда. Если ОРМ не подгадил, то что-то странное случилось, значит, у вас приложение не работает.

И 2 мегабайта текста запроса нужно еще распарсить. У вас база данных будет тратить время на то, чтобы просто тупо разобрать этот запрос, а не выполнить его. А выполнять будет миллисекунду.

Следующая проблема, если вы видите запросы, которые у вас нагенерировались или которые вы сами написали, т. е. если вы видите в совокупности или первые два слагаемых, или вторые два слагаемых, то значит, у вас запрос написан неправильно, надо его переписать. Эту формулу сфотографируйте себе. И если увидели это в запросе, то значит, вы можете это переписывать смело. Он будет работать медленно. Его надо оптимизировать. Потому что LEFT JOIN, который порождает дублирующие строки и потом DISTINCT над этим, значит, что вы что-то неправильно там решили собирать.

GROUP BY – это другая ипостась того же самого DISTINCT. Т. е. крайне редко, когда это действительно нужно. А запросов таких я вижу очень много в своей практике.

И другая вещь – COUNT, MAX, MIN нужны бухгалтерам, когда они считают зарплату. Делают они это раз в месяц. Если вы это делаете каждую секунду, то задумайтесь, нужно ли вам это делать. Потому что любая из этих операций агрегирующих функций, встречающихся в запросе, она за собой влечет чтение почти всей таблицы. По крайней мере того скопа данных, который вы хотите перебрать. И это может оказаться достаточно большим объемом данных. И быстро такие запросы никогда не будут работать. И индексами вы их вряд ли как-нибудь подопрете. Потому что тут в любом случае необходимо будет перечитать данные. Единственное, что там MIN, MAX помогут в некоторых ситуациях. Но помимо этих агрегирующих функций есть куча других. Там целый зоопарк.

Рецепт, как оптимизировать запрос. Если у вас есть условие «WHERE id IN (SELECT id FROM)», то оно всегда будет работать достаточно медленно. Проще его переписать на «NOT EXISTS». И запихнуть id, который до скобочек, вовнутрь скобочек.

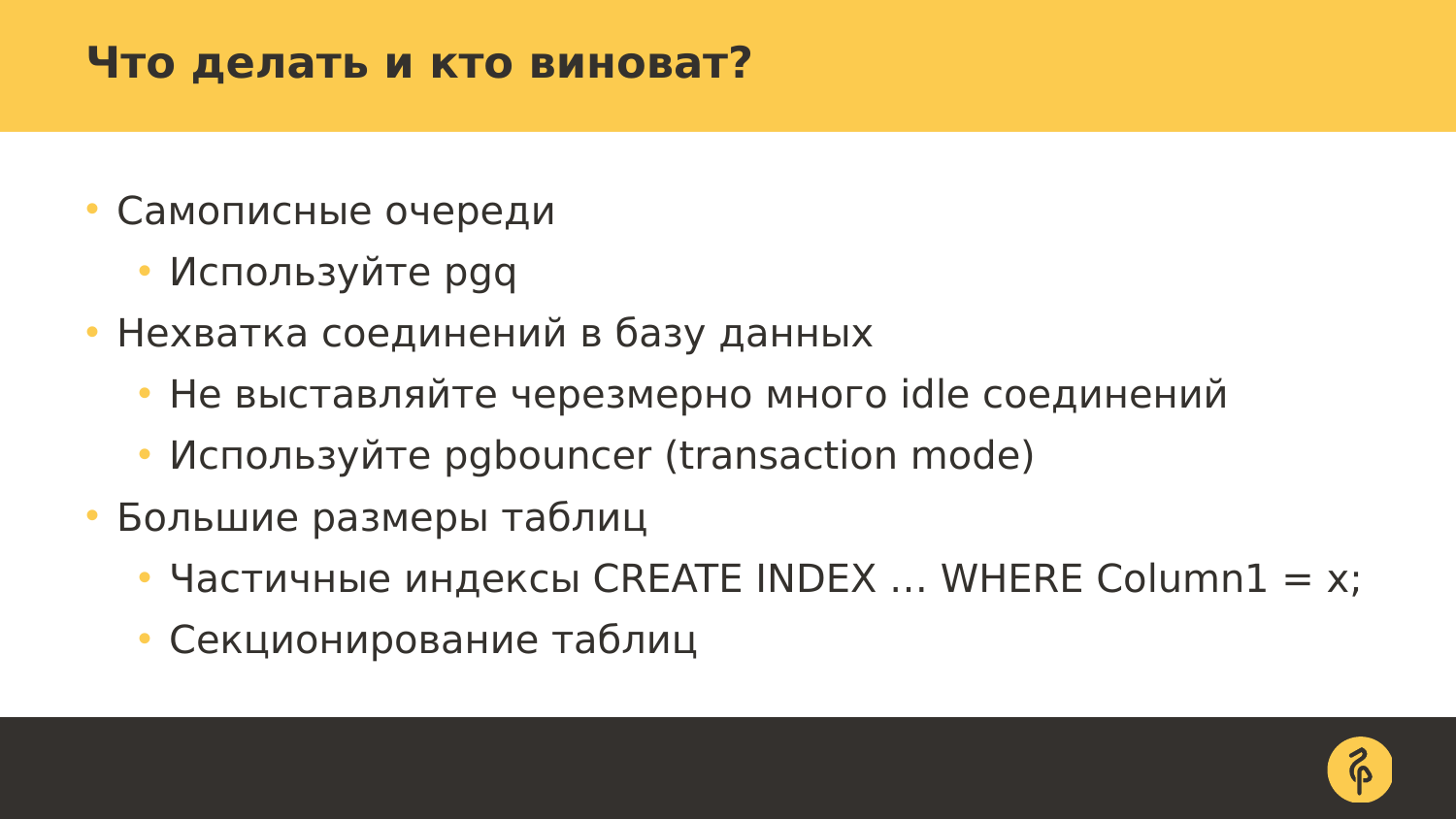
Объясню, почему. В первом случае SELECT в скобочках должен выполниться и вернуть вам весь массив этих ids. И потом вы пойдете сверять id с каждым, проверяя – попадает ли он в этот диапазон.

В случае NOT EXISTS и запихнув это условие в скобочку, вы можете включить оптимизацию по индексам. И если у вас индексировано оба поля, то база прыгнет в индекс и сверит: «Ага, и тут и тут есть id, идем дальше». И не будет возвращать весь скоп условий по другим частям WHERE, которые вы там впишите. Если у вас ничего, кроме WHERE нет, то табличка, в скобочках которая, будет целиком каждый раз у вас читаться.

Разные задачи – разные пользователи. Слишком много пользователей в базу данных не надо делать. Но если у вас есть приложение, которое занимается рассылкой почты или какими-то другими долгоиграющими заданиями и есть бэкенд, который должен отдавать быстро фронтенду данные, то лучше их разделять. Потому что потом на уровне операционной системы можно играться приоритетами и cron-задачам (это таски длительные) понижать им приоритет по дискам, по процессору, отдавая больший приоритет нашему обслуживанию веба или тому, что нужно достаточно быстро. Например, в контексте сайта нам нужно веб обслуживать.

Очень хорошо и полезно иметь разных пользователей. Если у вас несколько вебов, работающих по-разному, то там желательно тоже пользователей разделить, чтобы вы могли определять более важную часть от более неважной. И средствами операционки вы можете повышать-понижать приоритет сессии того или иного пользователя. Без этого, когда у вас база будет долбиться одним пользователем на все задачи, у вас будет там каша. И вы не сможете выделить нужные PIDs в операционке и понизить им приоритеты. Это полезная практика, которая может вам действительно помочь, когда пошла большая нагрузка и нужно понижать.

К примеру, те же HH уже знают, что если делают cron …, то у них отдельные пользователи. Они на них вешают. И у них в cron мы прописали понижение приоритетов. Пришла задача кроновская, понижаем приоритеты, потому что основная наша цель – обслуживать клиентов веба.



Последний пункт. Каждый в своей жизни должен написать очередь. Если у вас есть опыт общения с очередями, которые вне базы: с Kafka, RabbitMQ, ZeroMQ, то пользуйтесь ими.

Но бывает задача, когда нам нужна очередь в рамках базы данных. И в таком случае не пишите своих очередь. Skype и SkyTools за вас это написали и называется это PGQ. Они попрыгали на всех граблях из разного металла и с разной длиной ручек. Поэтому это идеально проверенный проект. Т. е. она обрабатывает 20 000-30 000 событий в секунду достаточно нормально.

Единственный ее недостаток – плохая документация. Там придется немного помучится, прежде чем научиться ею пользоваться. Или просто спросите тех, кто ею пользовался, чтобы подсказали как ею пользоваться.

Но самому писать не стоит. Я писал. Каждый DBA, кто работал в нашей команде, писал свои очереди в Postgres. Мы тоже попрыгали на этих граблях.

Нехватка соединений в базу данных. Это к тому вопросу, когда я показывал, что в приложении забили idles. И меня даже мониторинг не мог подключиться к базе данных, чтобы снять метрики с внутренних вьюшек.

Не выставляйте у себя в приложении слишком много idle-сессий. Они не нужны. Делайте небольшой запас. Если вы предполагаете, что приложение у вас должно иметь 10 активных коннектов, ставьте 15, чтобы у вас было 15 сессий, т. е. в idle оно должно держать не больше 15 сессий. Не перебарщивайте с этим, потому что база данных должна работать, а не простаивать просто так.

И используйте pgbouncer как прослойку между базой данных и вашим приложением. Для приложения он будет выглядеть как база данных. Для базы данных он выглядит как клиент, но он достаточно хорошо умеет балансировать соединение.

Смысл имеет [PgBouncer](https://www.pgbouncer.org/) в transaction-режиме, т. е. когда он пуллит конкретные транзакции: от start transaction до commit или rollback. Это позволяет не пересоздавать каждый раз сессию в базе данных, затрачивая ресурсы на форк… бэкенда и инициализацию его – это раз. И два – он достаточно грамотно организует очереди. И, как правило, мы получаем даже прирост в скорости обработки запросов базой данных по сравнению, если бы мы напрямую долбились в нее.

Есть еще [Pgpool-II](https://www.pgpool.net/mediawiki/index.php/Main_Page), но это не очень удачное решение, поэтому я его не рекомендую.

Ребята из Яндекса насиловали PgBouncer с разных сторон, но у них там иногда сумасшедшие нагрузки. И они решили написать свой. Называется он [Odyssey](https://github.com/yandex/odyssey). Я пару месяцев назад с ними разговаривал. Он у них только пока во внутреннем пользовании, т. е. там багов еще большая куча. И они вылезают на разных языках с разным профилем нагрузки. И они постоянно его фиксят-фиксят, но, я надеюсь, где-то через год [Odyssey](https://github.com/yandex/odyssey) заменит Pgbouncer. Пока PgBouncer – безальтернативный вариант.

Большие размеры таблиц. Это всегда плохо, потому что если мы COUNT решим на 200 гигабайтной табличке, то нам надо будет эти 200 гигабайт прочитать. Даже если мы MIN без индекса там решим найти, то нам придется ее прочитать и отсортировать еще.

А если мы захотим создать индекс на 200 гигабайтной табличке, он у нас будет весить 100 гигабайт. А потом неожиданно закончится место на дисках. Поэтому есть несколько решений.

Ленивое решение – мы смотрим на наши запросы, которые пуляются в эту большую табличку. Частенько это бывает такой профиль нагрузки. Возьмем те же самые транзакции платежные. И мы стараемся обрабатывать те транзакции, которые не обработаны. И понятное дело, что транзакции, которые обработаны, их большее количество в этой таблице. Поэтому мы можем создать частичный индекс, который будет работать по необработанным транзакциям и в нем проиндексируем айдишник.

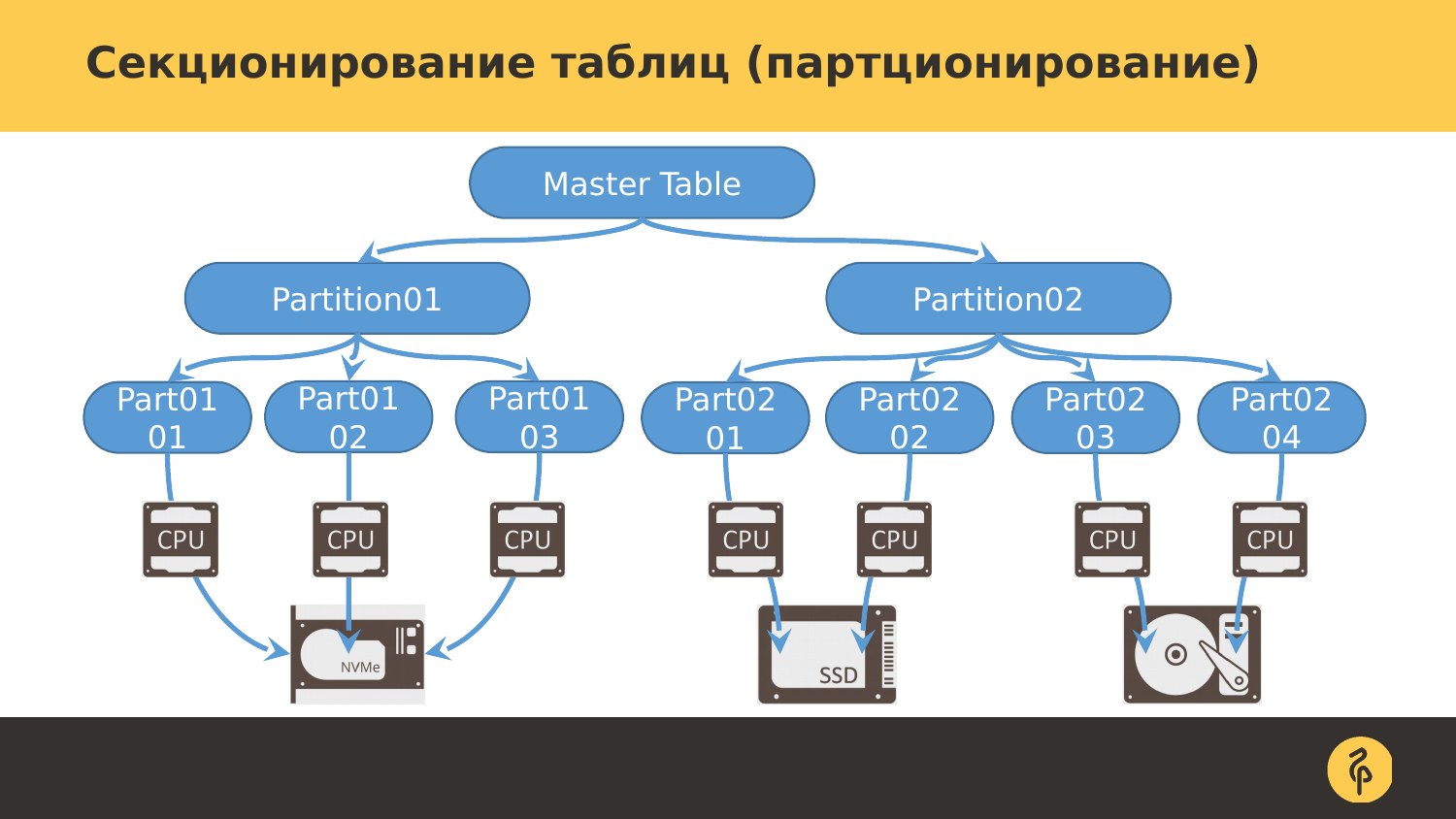
И если мы в запросе будем указывать вот это условие, то тогда мы полезем в этот индекс. Какие у него будут преимущества? То, что он будет не 100 гигабайт, а 2 гигабайта на 200 гигабайтной табличке. Соответственно, поиск по нему нужного айдишника будет быстрым. Но это имеет смысл делать, если на большой табличке у нас реально 90 % долбятся в 10 % таблички. Если мы обрабатываем все 200 гигабайт, то тут это не поможет.

И секционирование таблиц. Я до сих пор привыкаю к этому слову. Я пришел из мира Oracle, там был partition. И я привык «партиционирование» говорить. Но по-русски правильно – секционирование.

Секционирование таблиц нам помогает разбить табличку на кусочки небольшие. И об этом я подробней расскажу на следующих слайдах.



Чем оно может помочь нам?

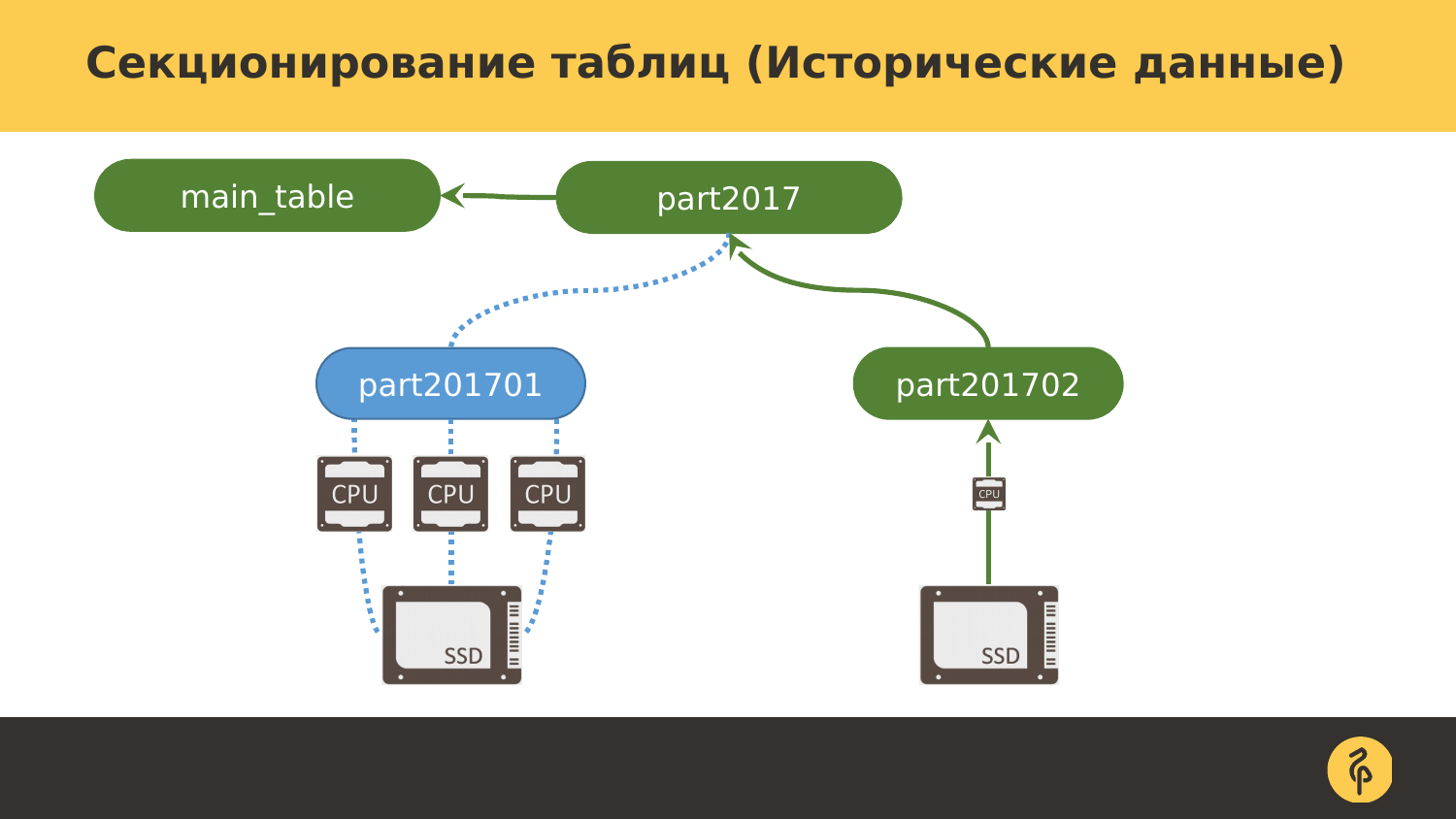


Это принципиальная схема партиционирования. Заключается она в том, что мы таблицу можем логически разбить на куски, независимых данных.

Два основных рабочих примера. Это идущие по времени вперед данные, когда мы нарезаем их кусками: по дню, по месяцу, по году, в зависимости от плотности наших данных. И второй профиль использования партиционирования, когда у нас в таблице хранятся разные клиенты. И, допустим, каждого клиента мы хотим обрабатывать быстро. И мы можем отделить клиентов, генерирующих много данных в одну партицию, а маленьких в другую партицию. И когда у нас будет приходить запрос по такому жирному клиенту, у которого много данных, мы будем проваливаться только в его данные и будем только с ними работать. И это сразу автоматом будет отрезать от нас другие данные, которые нам не нужны.

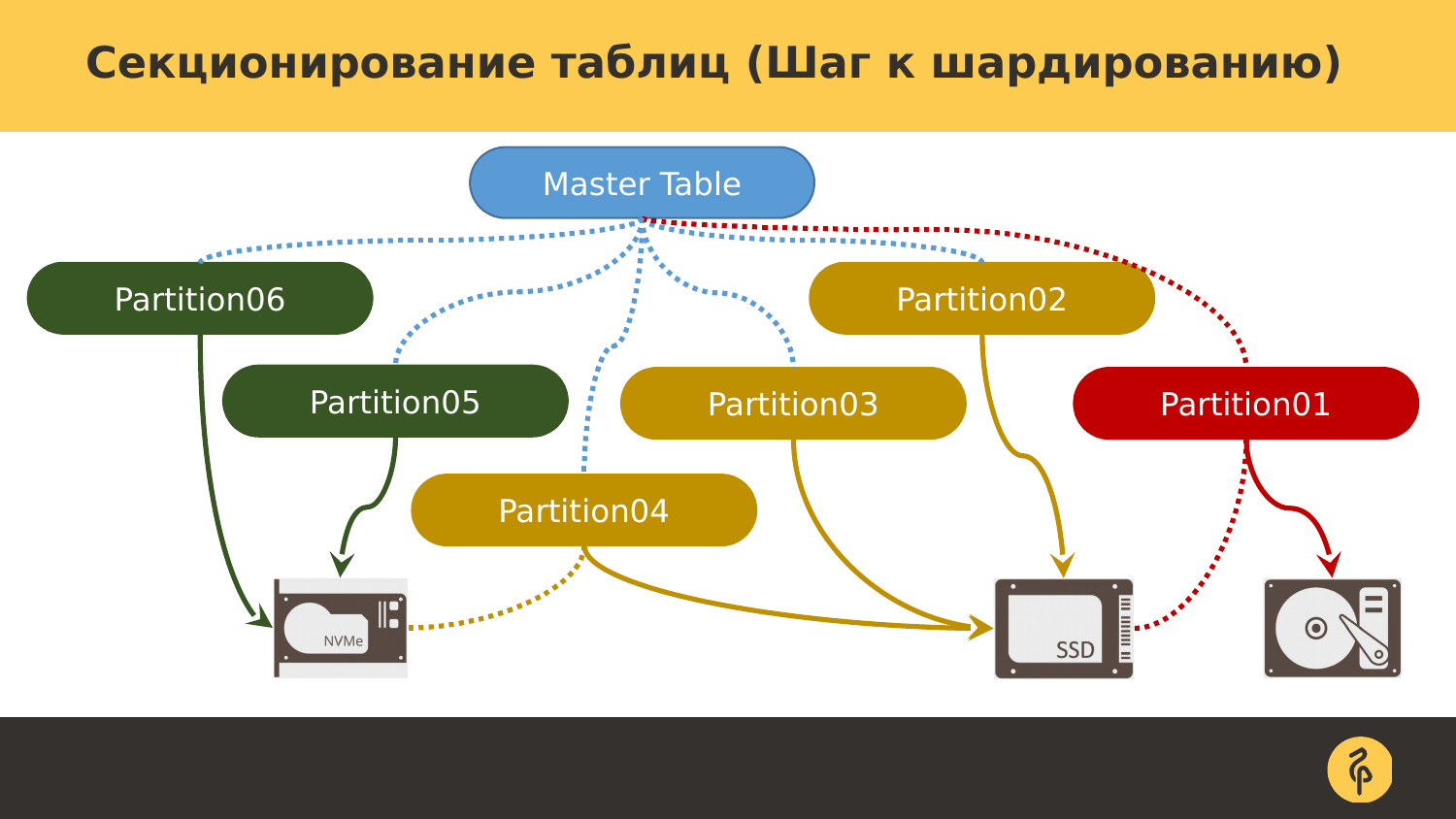
Вот два профиля. С учетом развития баз данных, в каждой базе данных это по своему реализовано, но так как у нас многоядерные системы, то уже все базы так или иначе обрабатывают параллельно запросы по этим таблицам и могут читать их параллельно, что отображено на уровне процессоров.

И последний уровень – это больше касается именно архивных таблиц и когда у нас есть проблемы с оборудованием. Допустим, старые данные, которые нужно аналитикам редко, мы можем хранить на дешевых дисках: на HDD или SSD. А актуальные и горячие данные за последние месяц или неделю храним на NVMe диске. И периодически просто перебалансируем партиции – менее нужные уносим на дешевое оборудование, а более нужное оставляем на более дорогом. В реальности у большинства наших клиентов нет такого зоопарка дисков, у нас везде стоят NVMe, как правило.



Чем хорошо партиционирование? Когда мы в этой партиционированной таблице обращаемся к маленьким табличкам как к единой части через одно имя и прописываем в условии WHERE поле, по которому было партицирована таблица, и она попадает под правила партиционирования, то база данных без поиска всяких индексов идет сразу в нужную партицию за нужными данными.

На выхлопе нам это дает меньше трат процессорного времени, потому что мы читаем уже более-менее нужные нам данные. И меньше трат дисковых ресурсов. И это позволяет нам повысить нагрузку.



Зеленые данные – это более актуальные данные. Желтые – это более старые данные, которые нам нужны для аналитики. И красные данные – это те данные, которые нам уже не нужны ни аналитики, ни для оперативной работы. И мы их можем удалить. Причем удаление – это drop table, который работает быстро и не насилует базу вычищением строк.

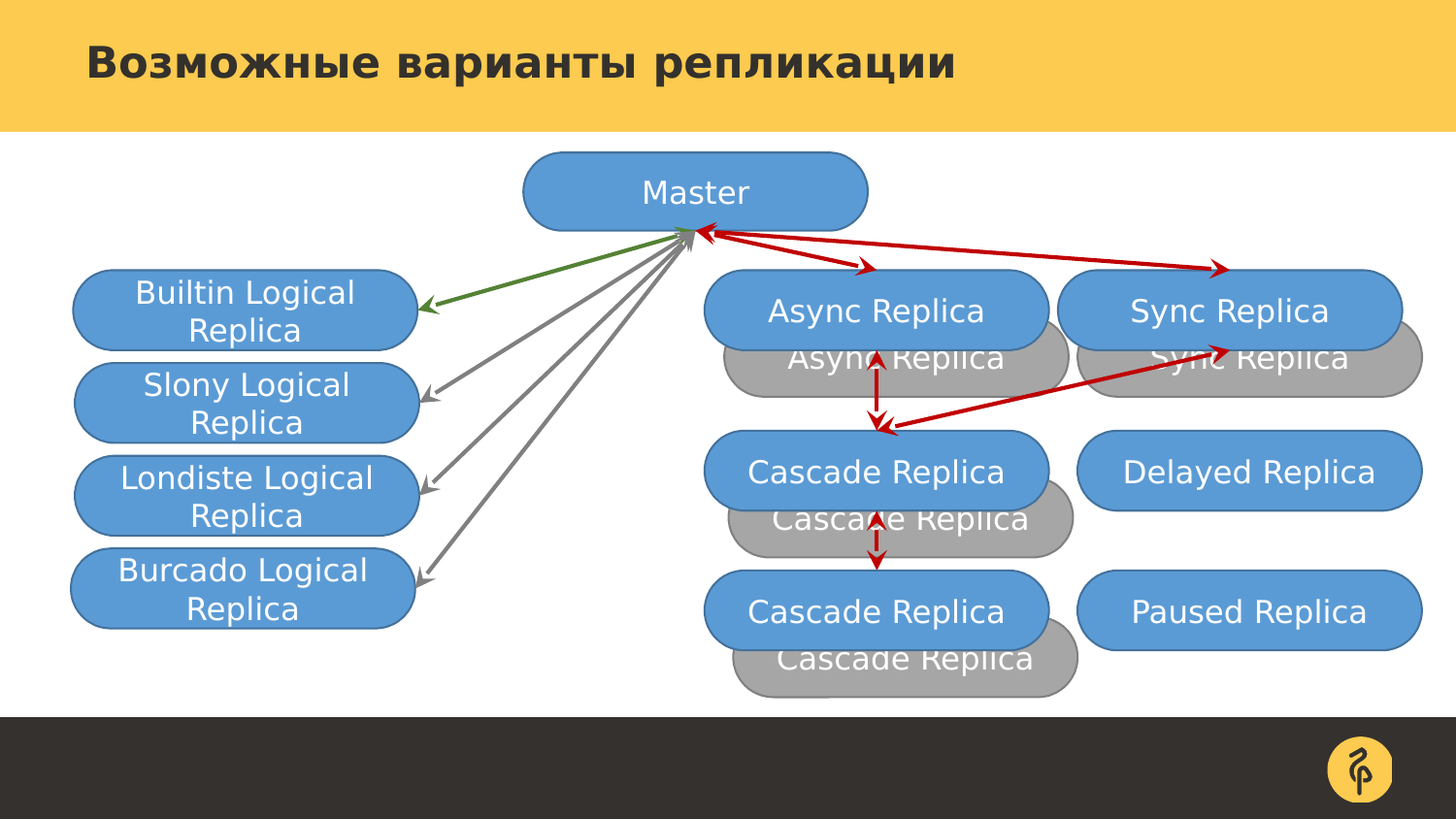
На это можно посмотреть с другой стороны. Когда мы запускаем какие-то компании, которые нужно обрабатывать, то это тоже вещь по времени. Но мы тут делим по какому-нибудь критерию компании. Допустим, мы – рекламное агентство и запустили рекламную компанию по Coca-Cola, Pepsi, Nike, Adidas и т. д. И спортивные они отработали компании, и остались нужны только аналитикам. Причем время работы этих компаний может быть разным. Они не обязательно должны уйти друг за другом по времени. Важна их актуальность на данный момент.

И, допустим, спортивные компании нам не нужны, а нужны только аналитикам. Т. е. мы их отцепили на медленные ресурсы, потому что аналитики делают медленные и длинные запросы. А по газированным напиткам они у нас еще работают. Нам нужно еще получать быстро данные. Мы их держим на быстрых дисках. Когда они станут неактуальными, они станут желтыми. А когда они перестанут нужными быть аналитикам, их можно будет вообще удалить, потому что не нужно хранить всякое барахло в базе.



После того, как мы вычислили ОМР, когда мы не читаем и не пишем всякую ерунду в базу данных и когда мы научились не писать плохие запросы, и когда мы следим в 4 глаза за нашим ОМР, чтобы он нам не фигачил 2 гигабайтные, 2 мегабайтные запросы в базу данных, но при этом мы продолжаем упираться в производительность одного сервера, вот тогда появляется вопрос: «Что нам может помочь?».

В начале я вам сказал, что мультимастер-решения не работают. Точнее они работают, но они медленные. Они не ускорят вас никогда, по крайней мере для RDBMS баз данных.



Поможет репликация. Прежде чем рассказать о том, как она поможет, сделаю краткий обзор о том, какие репликации существуют.

Их можно поделить на две группы: логическая репликация и бинарная репликация.

Логическая репликация – это когда протокол репликации вам транслирует те самые запросы, которые пришли в базу данных. Вы сделали INSERT. И этот INSERT таким же INSERT’ом ушел по протоколу логической репликации. Сделали DELETE и этот же DELETE в нативном SQL ушел. Это в грубой форме объяснение логической репликации.

А бинарная репликация выглядит так. Произошел INSERT таблички данных. Изменились файлы и структура данных, которые хранятся. И эти блоки измененных файлов по протоколу бинарной репликации уходят на сервер. Т. е. мы вообще не знаем, что там за запрос произошел.

И теперь о том, какие существуют.

С версии 9.4 сделали логическую репликацию. Она была убогой. К версии 9.6 ее развили до текущего состояния. В 10-11 она не изменилась. Видимо, товарищи, которые ее писали, получили то, что им нужно от нее.

У Postgres есть своя встроенная логическая репликация. Единственное, в чем надо заморочиться – это распарсить этот лог архивный, чтобы получить оттуда запрос. Там есть нюансы.

На заре Postgres, когда не было никаких в нем репликаций, люди делали три триггеры на репликации, которые реагировали на изменения, которые происходят в таблицах. И через очередь изменений отдавали на другой сервер вот эти команды – какие данные изменились. И там происходили INSERTS и DELETES.

И постепенно появились три решения, независимые друг от друга. Это Slony, Londiste и Burcado.

Burcado – наименее используемый у нас в стране. У нас наиболее используемый – Slony. Londiste – почти убит, его уже давно не саппортят и не коммитят. И с появлением встроенной логической репликации эти три остаются только у тех, кто изначально ими пользовался.

Чем хороши логические репликации, помимо того, что вы в них видите нативный SQL? Они вам позволяют разделять данные из одной базы данных по нескольким потребителям. Или, наоборот, агрегировать несколько баз данных в одну. Допустим, когда вам нужно считать какие-то сводные отчеты. Для этих целей лучше логической репликации не придумать, потому что вы можете фильтровать на уровне протокола репликации и брать только нужные данные.

В принципе, в Postgres можно сразу же подписываться в текущей версии репликации на нужные таблицы и не получать информацию о том, что изменилось в других. И в логических самописных это тоже есть.

Что есть в бинарных репликациях? Есть асинхронная реплика.

Асинхронная реплика – это бинарная копия вашего Мастер-сервера, на которую поставляются по сети изменения, которые происходят на Мастер-сервере. И она консистентна с Мастер-сервером, но она отстает от него на некоторое время небольшое. Там не то же самое, что на Мастере-сервере. У Мастера-сервера может быть несколько асинхронных реплик.

Есть синхронная реплика. Синхронная реплика имеет то же самое состояние, что и Мастер-сервер. По факту синхронной репликой в Postgres вы себе устраиваете двухфазные коммит, потому что Postgres должен убедиться, что изменения, случившиеся на Мастере, как минимум, приплыли на реплику и, как максимум, применились там.

В последних версиях есть несколько уровней синхронности. От минимальной до максимальной. Их может быть несколько.

Проблема асинхронной метрики в том, что, если она у вас отвалится, у вас двухфазный коммит не будет работать, и все транзакции встанут колом, потому что нет синхронной реплики, с которой должно прийти подтверждение.

Нужны они крайне редко. В принципе, у наших клиентов нет синхронных реплик. Все нагруженные проекты справляются с асинхронными, просто понимая, что данные на 10-20 миллисекунд отстают.

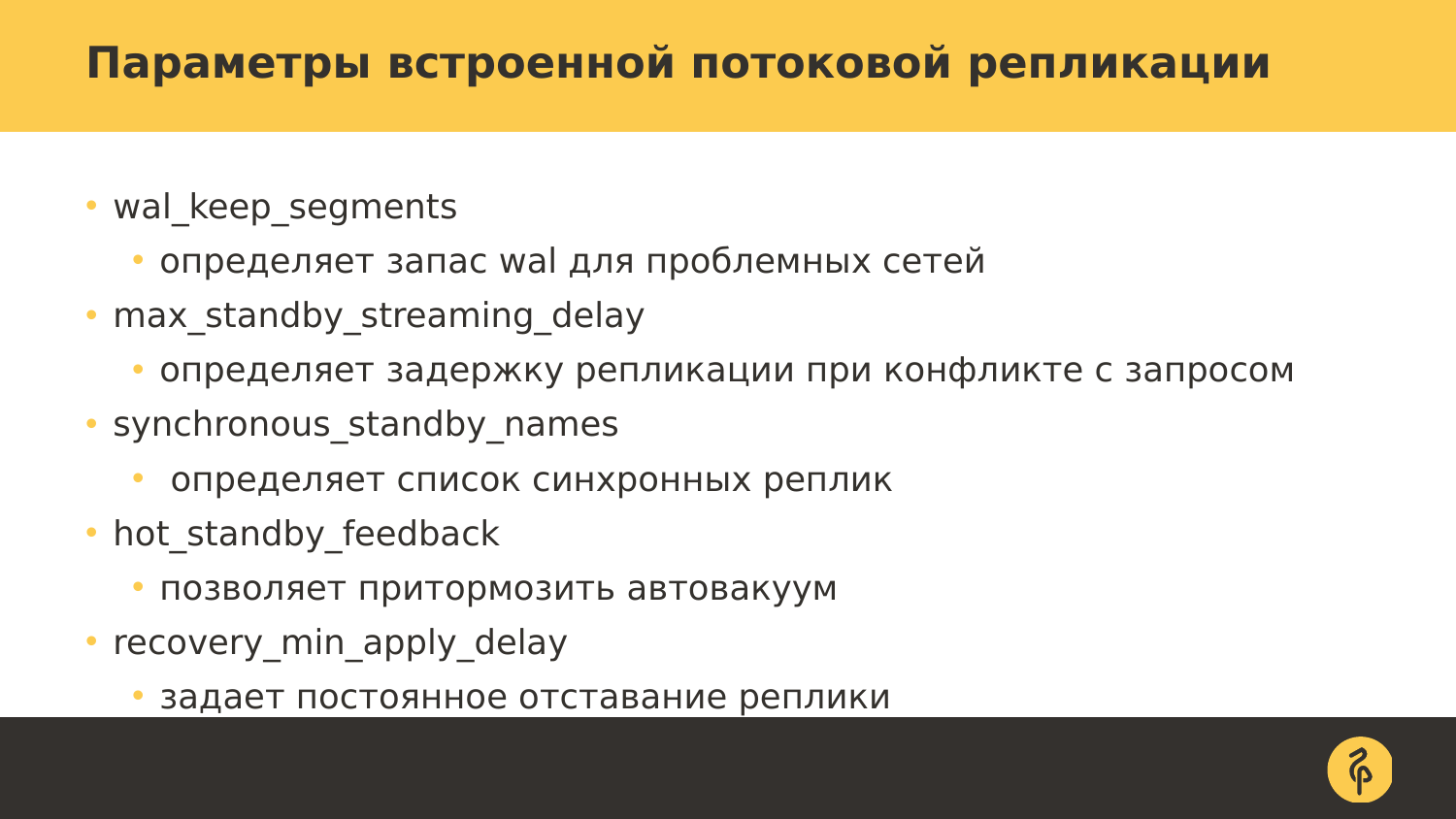
Есть каскадные реплики. Каскадная реплика может быть как синхронной, если она к синхронной прицеплена, так и асинхронной.

Почему она называется каскадной? Она берет свои изменения не с Мастер-сервера, а с одной из этих реплик. Это полезно, чтобы не нагружать Мастер-сервер, когда у нас 20 реплик, иначе мы забьем сетевой трафик, требуя от него изменения, потому что каждой реплике он будет рассылать все изменения, которые у него произошли. Вполне логично их рассылать по одной-двум репликам. И уже с них брать изменения для других реплик. Соответственно, каскадных реплик мы можем нацеплять тоже какое угодно количество. И они уже не будут влиять на Мастер-сервер.

И к каскадным репликам мы можем подцепить дальше каскадные реплики. Поставить такое развесистое дерево. Это принесет вам проблемы в случае, когда вы будете делать мажорное обновление Postgres, потому что вы задолбаетесь это все дело переналивать. И их тоже может быть какое угодно количество.

Есть еще два типа реплик. Это delayed реплика. Это реплика, которая отстает от мастера на заданную вами величину времени. Такие реплики полезны, когда у вас база в 8 терабайт и вы иногда делаете страшные вещи с базой данных, и не хотите потом эти 8 терабайт высасывать, восстанавливать снова. Поможет в этом реплика, которая у вас гарантирована на 6 часов позже, чем Мастер. Вы можете на ней прибить репликацию. Остановить процесс репликации. И вытащить те данные, которые попортили базу. Очень полезная вещь на больших базах данных. И имеет смысл на больших базах данных. В принципе, 200 гигабайт сейчас восстановить – не проблема.

И реплика на паузе. Мы можем на паузу поставить любую из этих реплик. Это просто приостановка проигрывания репликации. Это тоже полезно. Допустим, если мы не хотим держать задержанную реплику, а собираемся делать какую-то потенциально опасную миграцию данных, мы выбираем реплику, которая будет нам гарантом. Ставим ее на паузу. Делаем миграцию. И проверяем – хорошо ли прошла миграция. Если хорошо, то отпускаем реплику с паузы. Она догоняется до консистентного состояния с Мастером и все хорошо. Если мы напортачили, то мы можем на нее переключиться, сделав ее Мастером или ручками перенести попорченные данные обратно.



Нам интересна больше встроенная репликация для шардирования нагрузки. Что там у нас есть?

Есть wal\_keep\_segments. Он определяет запас вот этих изменений, которые в базе данных произошли. И если у нас по каким-то причинам произошел разрыв между Мастером и сервером по сетевым причинам, например, умер сетевой роутер, то один wal-файл по умолчанию – 16 мегабайт. Т. е. если у нас 1 000 сегментов, то получается, что 1,5 гигабайта данных мы можем пережить падение сети между двумя серверами. Когда она восстановится, реплика подцепится и за счет этих файлов догонится до актуального состояния.

Если их не хватает, то в случае сетевых проблем просто они …, и реплика, когда подцепится к Мастер-серверу, не сможет его догнать, потому что уже нет нужных файлов для того, чтобы проиграть всю цепочку изменений. Это очень важно, потому что по умолчанию он там «0» стоит. И если вы будете настраивать репликацию, то его нужно будет менять.

Max\_standby\_streaming\_delay. Определяет задержку репликации при конфликте с запросом. Что это такое? На реплике мы можем запускать запросы, которые выполняются. Но есть операции, которые происходят только на Мастер-сервере с изменением данных. Это связано в основном с автовакуумом, когда он приходит и чистит старые строчки, которые нам больше не нужны. Но в этот момент у нас может возникнуть на реплике ситуация, что выполняется запрос, который читает именно эти строчки, которые на Мастере больше не нужны.

И база для того, чтобы реплицироваться должна их удалить, поэтому этот параметр определяет задержку, на которую мы можем приостановить репликацию, чтобы дать отработать нашему запросу. Если запрос выполняется дольше, тогда Postgres решает, что репликация для него важнее и он прибивает запрос.

Synchronous\_standby\_names. На Мастер-сервере задаете имена синхронных реплик. Там такой развесистый список. Не все они могут бы синхронными. Они могут быть между собой меняться. Но вы указываете количество синхронных реплик. И это получается у вас двухфазным коммитом в случае одной синхронной реплики. Трехфазным в случае двух синхронных реплик и т. д. Лучше не используйте никогда этот параметр. Но знать врага надо в лицо.

Hot\_standby\_feedback. Это такая штука, которая используется при нагрузке, когда мы гоняем большие аналитические запросы на реплике. Тут становится понятно, что рано или поздно придут изменения, которые сделал автовакуум, вычищая мертвые строчки. И он прибьет наш многочасовой запрос.

Чтобы он этого не сделал, реплика может посылать Мастер-серверу запрос и говорить: «Я работаю с транзакциями, начиная от такой-то. Пожалуйста, пока не трогай, не вычищай остатки данных от этих транзакций у себя». И притормаживает автовакуум на Мастере.

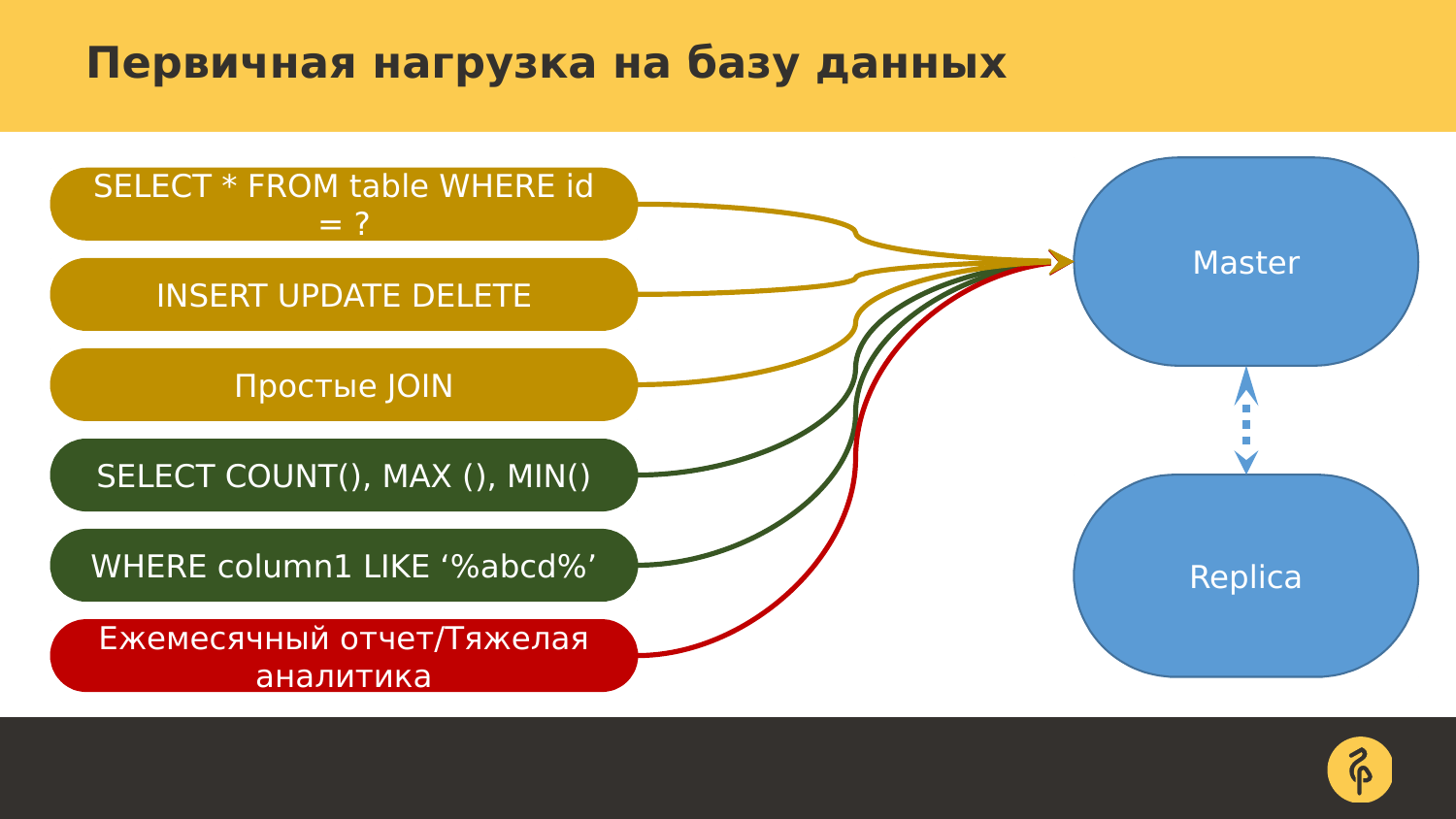
Вещь полезная и вредная. Не злоупотребляйте. Hot\_standby\_feedback нужен, когда мы параллелим быструю веб-нагрузку, чтобы у нас запросики точно отрабатывали. Или, когда нам нужна сложная аналитика. Но для действительно большой аналитики есть другие решения.

Recovery\_min\_apply\_delay. Когда мы работаем по идеологии CI/CD и выкатываемся каждый час, а через три часа обнаружили, что мы что-то угробили в базе данных, то такие реплики нам помогают. Устанавливаем 6 часов. И у нее стабильно на 6 часов позже состояние, чем у Мастера. Лучше 8 часов ставить.

Это о том, какие бывают репликации. И частично о том, как они бывают полезны.



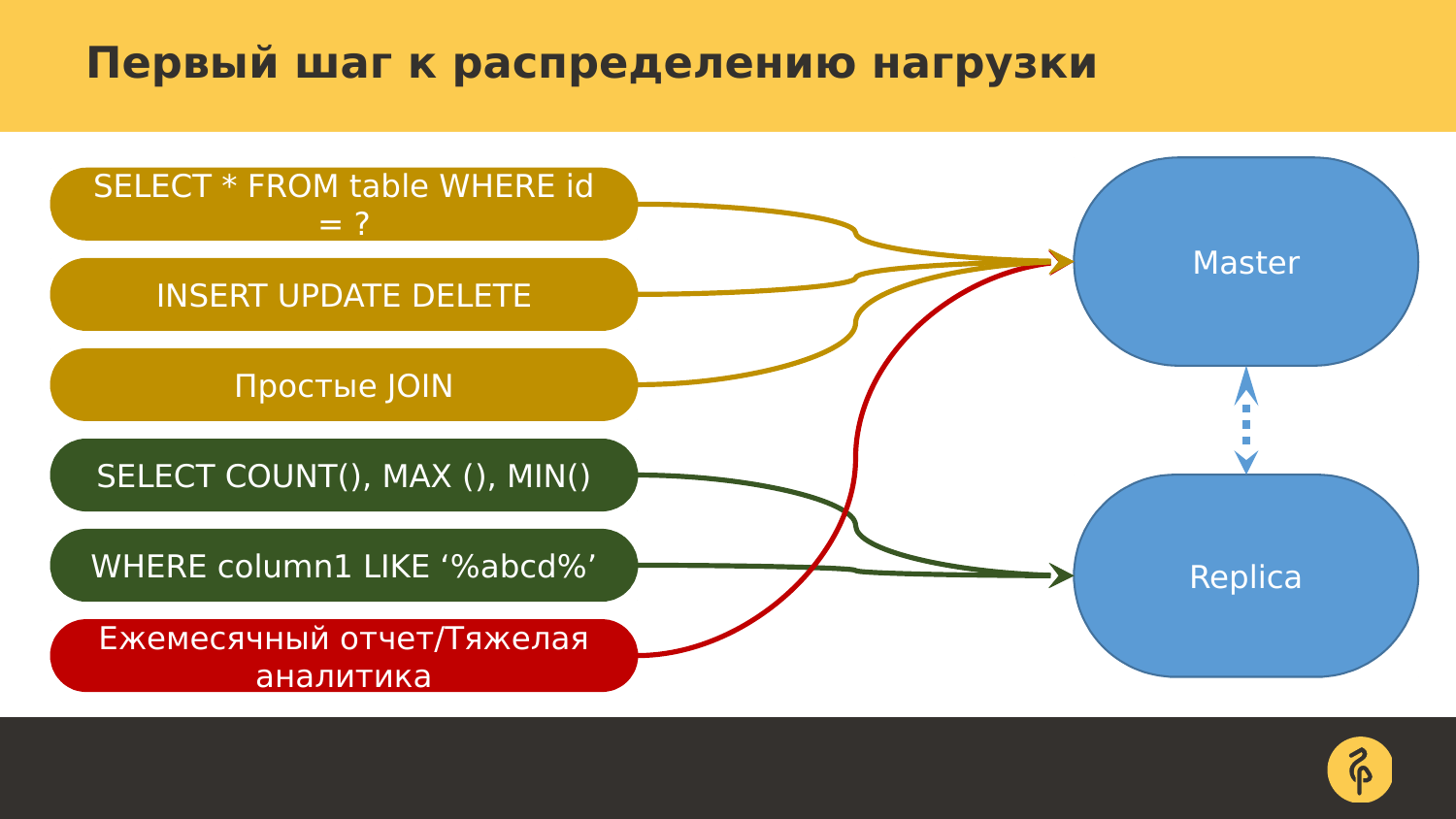
Последняя часть доклада осталась.



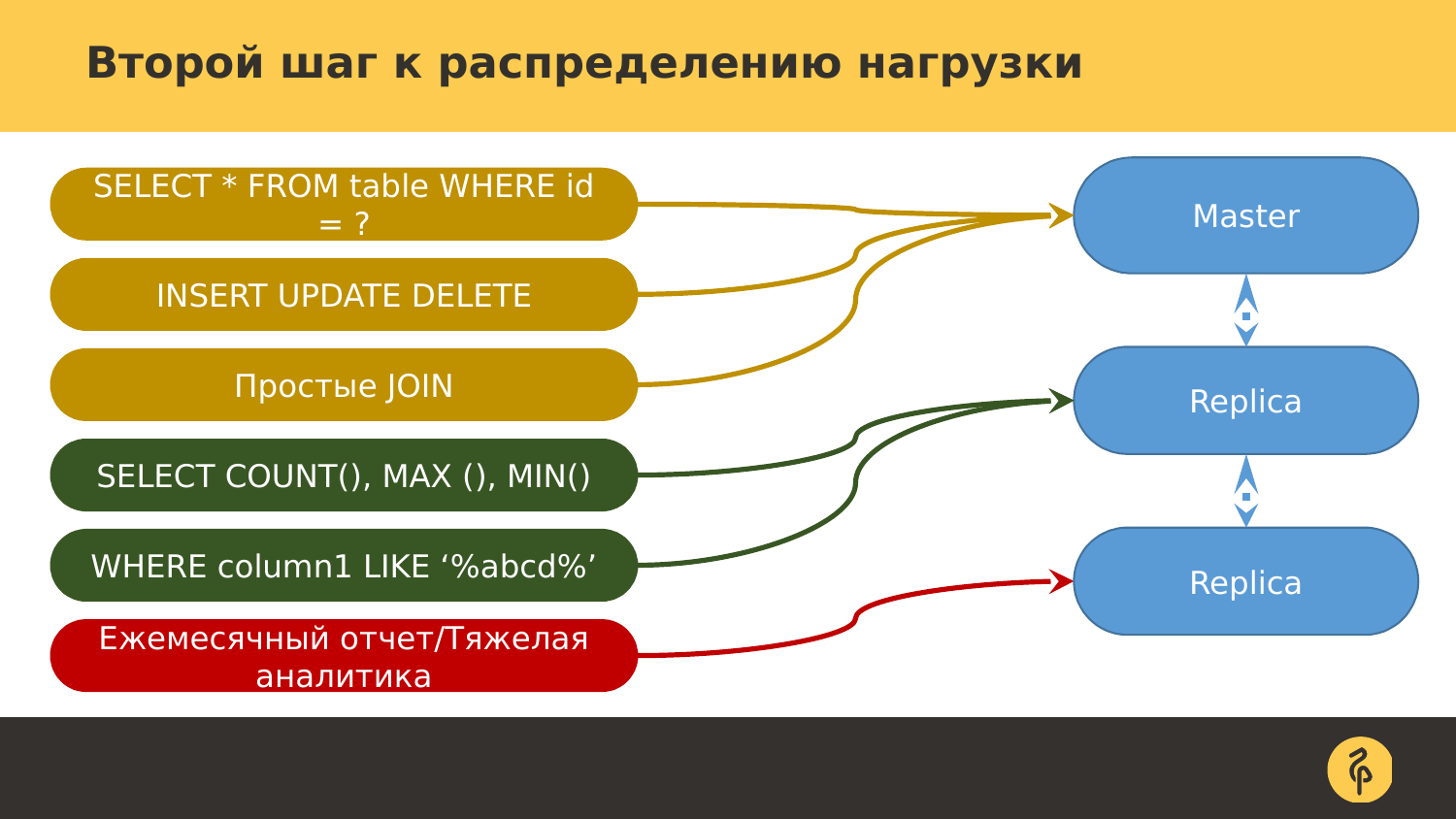
Когда мы начинаем работать с Postgres у нас вся нагрузка выглядит так, даже может быть без реплики. Я тут торопился и не сделал анимацию.

Если мы беспокоимся о безопасности, то у нас full story должна быть реплика, потому что если вдруг сдохнет Мастер-сервер, то мы должны куда-то переключиться. Но всю нагрузку делаем на Мастер-сервер. И он у нас от этого загибается.

Что мы можем в данном случае сделать? Мы можем выделить ту нагрузку, которую можем пустить на реплику, потому что не все запросы нам нужны на Мастере. К примеру, вот эти COUNT, MAX и какие-то невнятные поиски, которые будут работать долго всегда, их лучше отправить на реплику.



Эти запросы никогда не будут работать с миллисекундой. И отставание в 20 миллисекунд реплики при хороших настройках вполне позволительно, чтобы мы завернули все эти запросы на реплику. Этим мы разгрузили Мастер для нормального ответа по вебу. Потому что большая часть работы простая и она идет по айдишникам. Например, вставь новую строчку или обнови строчку, если пользователь что-то наменял. Иногда простые Join, потому что данные у нас нормализованными должны быть.

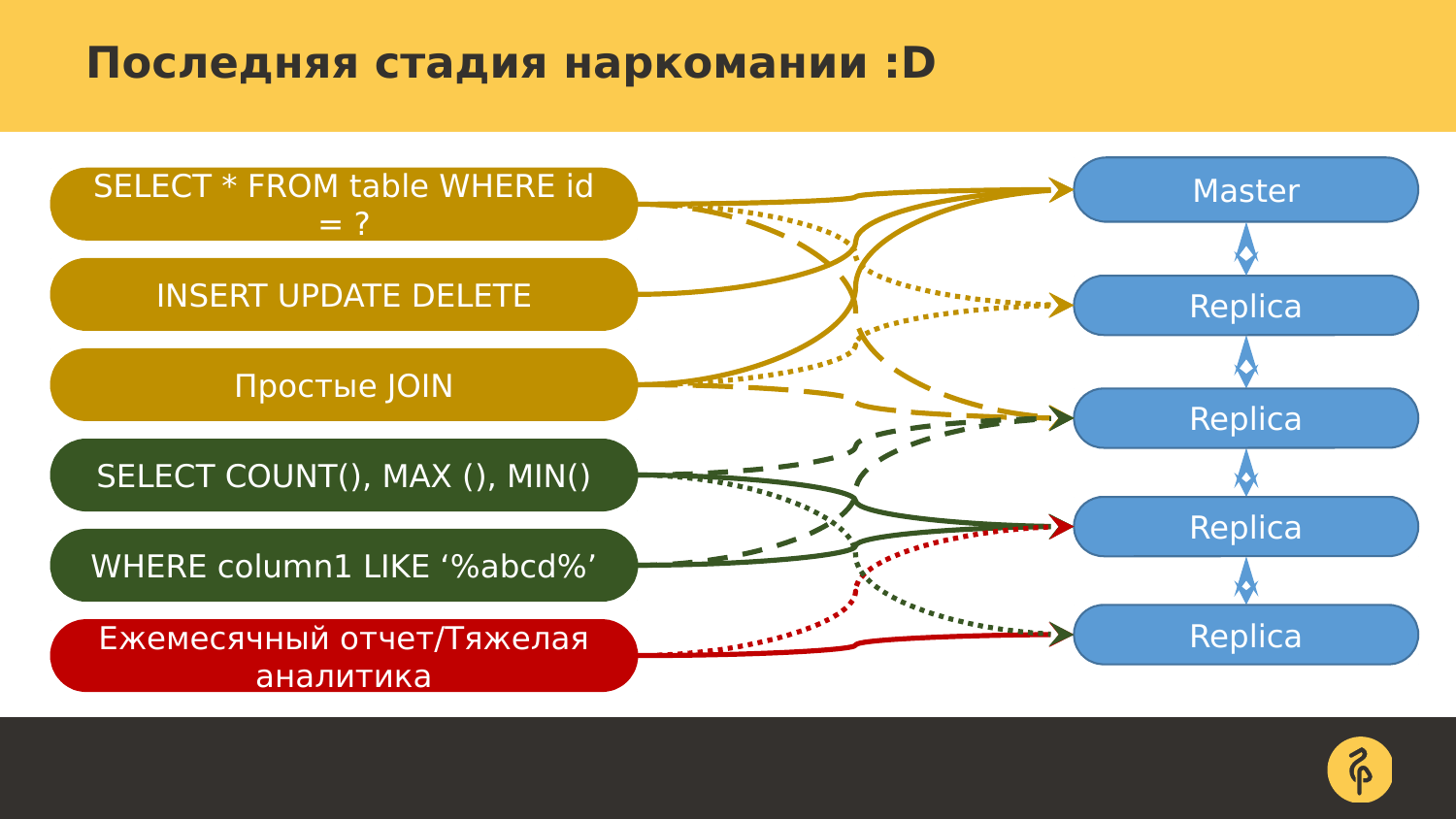


Но у нас есть еще какие-то тетеньки бухгалтеры, которые наш KPI считают и еще что-то. И есть тяжелые запросы. Для них можем создать третью реплику, если они очень сильно мешают. И эту реплику можно сделать max\_standby\_streaming\_delay. Если у нас двухчасовой запрос, то мы можем позволить ей отставать на 4 часы, чтобы он гарантированно выполнился.

Тут главное понимать, что если запрос длится 2 часа, то он никогда не работает с актуальными данными. И делать на Мастере ему нечего. Его можно спокойно завернуть на реплику. Все OLAP-запросы, которые вы делаете на своем сервисе, который обеспечивает веб-нагрузку, они там не нужны, их можно смело на реплику заворачивать. И там может быть несколько реплик, с подкрученными конфигами под конкретную OLAP-нагрузку. Это вполне нормальная рабочая схема.

Но, например, у HeadHunter случилось так, что им не хватило одного мастера. Им приходится читающую веб-нагрузку параллелить по репликам. И тогда наступает последняя стадия наркомании. Тут на стрелочки не смотрите, потому что это могут быть как каскадные, так и дерево из каскадных реплик. И так они все могут быть к Мастеру подцеплены.

В RDBMS базах данных пишущих запросов обычно не много, потому что если вы пишите больше, чем она может записать, то вам нужна не RDBMS, а любая другая база данных. Если для аналитики, то это ClickHouse, например.



И на чтении нам нужно масштабироваться. Мы добавляем реплик, обеспечиваем минимальную сетевую latency. Все длительные запросы, которые могли у нас тормозить транзакцию, репликацию, мы уже вынесли на реплики до этого, поэтому они в более-менее синхронном состоянии. Там задержки минимальные, позволительные. И пока у вас от фронтента до бэкенда дойдет запрос на поиск нужной информации, эти данные уже могут приплыть на реплику. Т. е. он будет идти дольше, чем протокол репликации. И мы отлично там читаем, просто параллелим запросы, читаем или с одной, или с другой реплики.

Может выясниться, что некоторые из вот этих смутных запросов могут быть нужны на более-менее актуальных данных, тогда частично мы можем их завернуть на актуальную реплику. Но основную часть нужно хранить все-таки на отдельной реплике. Так смешивать нужно только в крайних случаях, когда вы ограничены по железу. Лучше заведете отдельную реплику для этого.

А иногда мы понимаем, что это, наоборот, тяжелая аналитика, которую нужно завернуть на специально настроенную отстающую на долго реплику. И можем ошибиться там. Зарплату хочется вовремя получать. И поэтому можем ее отправить на менее отстающую реплику.

Схема «Последняя стадия наркомании» на HeadHunter вот так примерно выглядит. Там у них 11 серверов в каскаде на основной сервис. И это работает. У них такой сумасшедший монолит работает без проблем, как часы. Отчасти из-за того, что у их службы поддержки премия зависит от того, насколько не было downtime. У них есть разрешенное время downtime. И сколько они от этого разрешенного времени сэкономили, настолько у них хорошая премия. И они стараются. У них downtime.

У меня все на эту тему. Она больше обзорная, но, я думаю, что она будет полезна.