Меня хорошо слышно && видно?

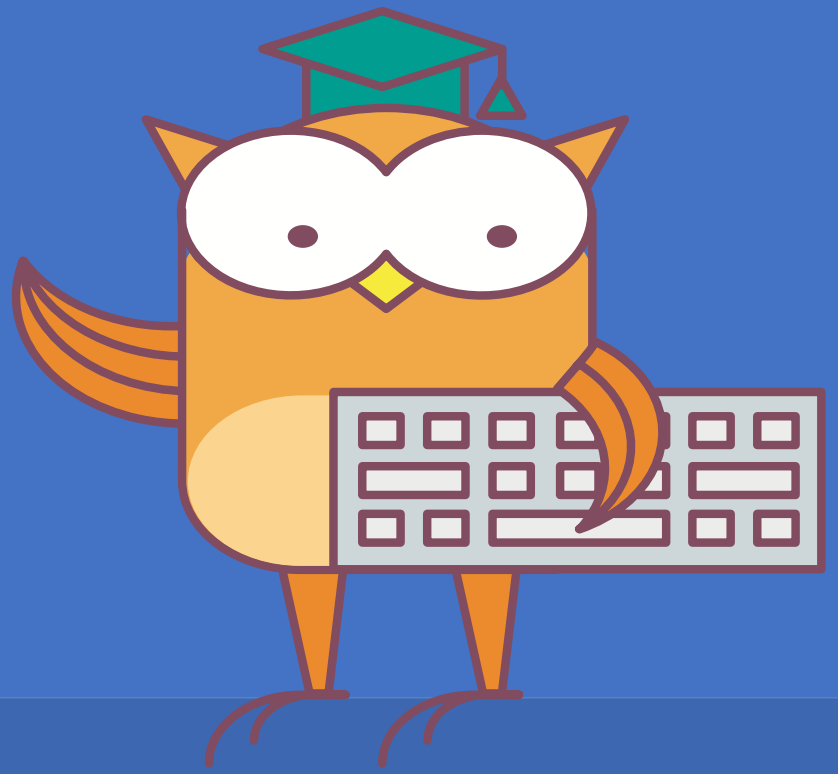
Напишите в чат, если есть проблемы!

Ставьте + если все хорошо

OTUS

Идемпотентные и коммутативные API

Архитектор ПО



Щетинников Стас

Карта вебинара

- Идемпотентность в API
- Надежная отправка сообщений
- Идемпотентность и коммутативность в очередях

01

Идемпотентность и коммутативность API

Идемпотентность API

Идемпотентность операции – можно послать несколько раз один и тот же запрос (сообщение), и состояние это не изменит.

Вообще retriability – возможность безопасно повторить запрос – очень полезная штука, и если есть возможность легко сделать запрос или событие идемпотентным – его стоит таким делать.

Идемпотентные операции

Есть приложение «Интернет-магазин». Когда пользователь нажимает кнопку «Оформить заказ», то происходит запрос
POST /api/v1/orders/
{
“products”: [{“id”: 42, “price”: “2500”}],
“shipping_to”: “Большая Филевская 14, кв. 88”
}

При этом деньги снимаются со счета в Личном кабинете, и происходит резервирование товара на складе.

Кнопка после нажатия остается активной. И иногда пользователи два раза нажимают на кнопку и происходит дублирование заказа со снятием двойной суммы.

Возможное решение

Пока не придет ответ сервера, не делаем кнопку «активной».

Что делать, если интернет отвалился в момент нажатия на кнопку «Оформить заказ»? Клиент получил ошибку таймаута, и считает, что запрос не прошел, а сервер его выполнил.

Возможное решение

Добавляем ключ Request-Id в запрос

```
POST /api/v1/orders/
X-Request-Id: de7efba4-267c-11ea-978f-2e728ce88125
{
    “products”: [{“id”: 42,  “price”: “2500”}],
    “shipping_to”: “Большая Филевская 16к1, кв. 14”
}
```

Если запрос с таким ключом уже пришел, то мы заказ не создаем.

Возможное решение

X-Request-Id – одна из возможных реализаций ключа идемпотентности.

Основная сложность в том, чтобы где-то хранить уже выполненные операции, но зато это относительно универсальное решение.

Ключи идемпотентности

https://stripe.com/docs/api/idempotent_requests - примеры ключей идемпотентности

Естественный ключ идемпотентности

Можно использовать естественный ключ идемпотентности.

Например, при создании заказа id можно генерировать на клиенте, тогда повторная попытка создать объект с тем же самым id не произойдет.

Еще больше идемпотентности!

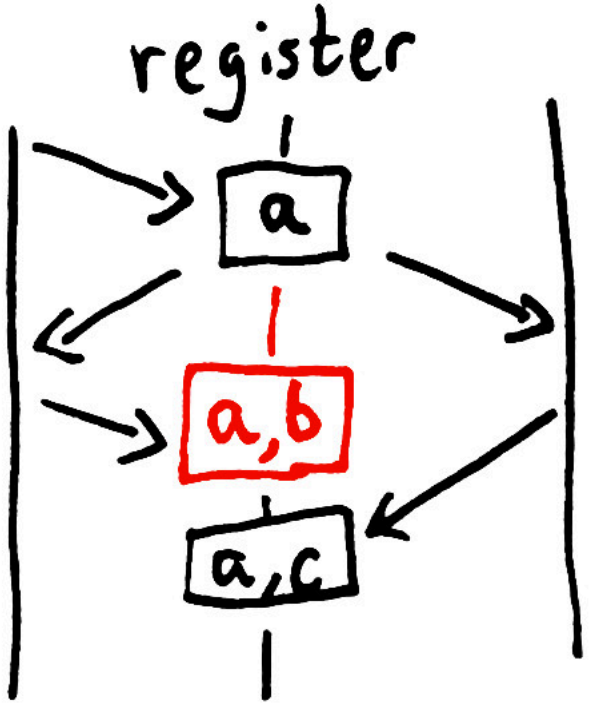
Пользователь нажал кнопку «Оформить заказ». Но ответ от сервиса был очень долгим. Клиент не стал дожидаться ответа от приложения и его полностью закрыл и выгрузил из памяти.

Когда он зашел в приложение, запрос еще не отработал и в списке заказов старого заказа не было. Клиент сформировал новый operation-id (client) и отправил еще один запрос. В результате создалось 2 заказа.

Еще больше идемпотентности!

Эта проблема похожа на ту, которую мы рассматривали на прошлом занятии с Lost Updates.

И решить ее можно с помощью оптимистических блокировок

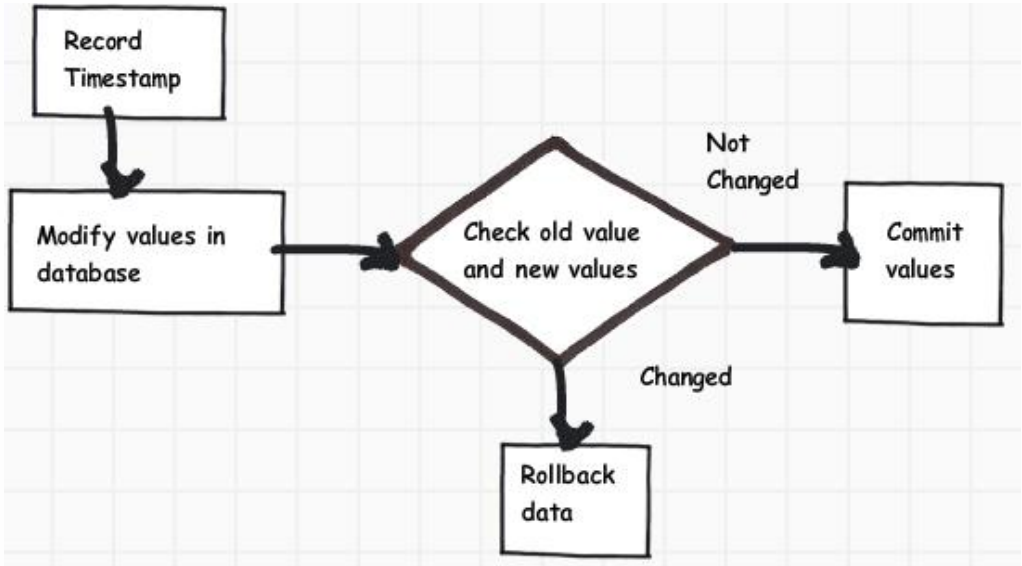


Еще больше идемпотентности!

Когда пользователь забирает для обновления какой-либо объект, он вместе с ним получает его версию.

И когда делает обновление отправляет в своем запросе ту версию, обновление которой он делает.

Сервер на своей стороне проверяет, если версия не совпадает с той, которая пришла в запросе, он запрос отклоняет.



Идемпотентность для коллекций

Делаем версионирование коллекции /api/v1/orders/.
Сервер присылает заголовок ETag с версией коллекции orders.
Клиент при изменении коллекции заголовок If-Match с версией, которую он знает.

/api/v1/orders/
Etag: 42

Сервер проверяет: если If-Match совпадает с версией на сервере, то запрос проходит. Если нет, то отвечает ошибкой.

POST /api/v1/orders/
If-Match: 42

429 Conflict

Идемпотентность для коллекций

Иметь «абстрактную» версию коллекции неплохо и довольно-таки универсально, но таки не всегда удобно.

Для того, чтобы защититься от повторного запроса в коллекцию, можно передавать не версию, а некоторый хэш состояния

Например, можно передавать количество заказов в коллекции или max(order_id).

Пример использования fingerprint

hash от содержания коллекции – fingerprint

https://cloud.google.com/compute/docs/reference/rest/v1/instances/setTags

Материалы

Статья про идемпотентность API

https://habr.com/ru/company/yandex/blog/442762/

02

Идемпотентность и коммутативность в очередях

Повторная обработка событий

Сеть не всегда надежна, поэтому возможны ситуации, когда сообщение обрабатывается дважды. Например, консьюмер не успел ack-нуть сообщение (или пометить у себя), но успел его обработать. Или по умолчанию во многих брокерах ack-и не синхронные, поэтому в случае если брокер развалится, он подумает, что сообщение не обработалось.

Идемпотентные сообщения

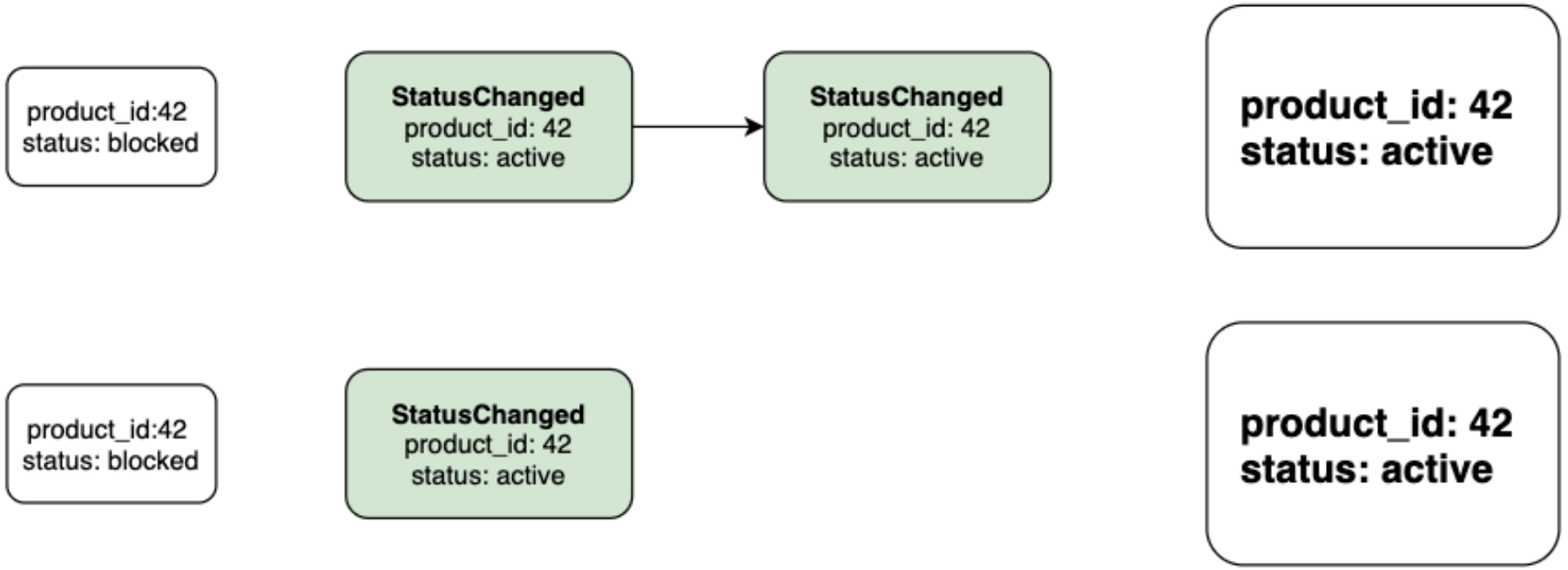
Самое простое решение передавать идемпотентные сообщения.

Например, события, которые передают новое состояние объекта чаще всего являются идемпотентными. Если мы их применим последовательно 2 раза не изменят состояния.

А вот события, которые передают только изменение состояния объекта, чаще всего при двойной обработке все испортят.

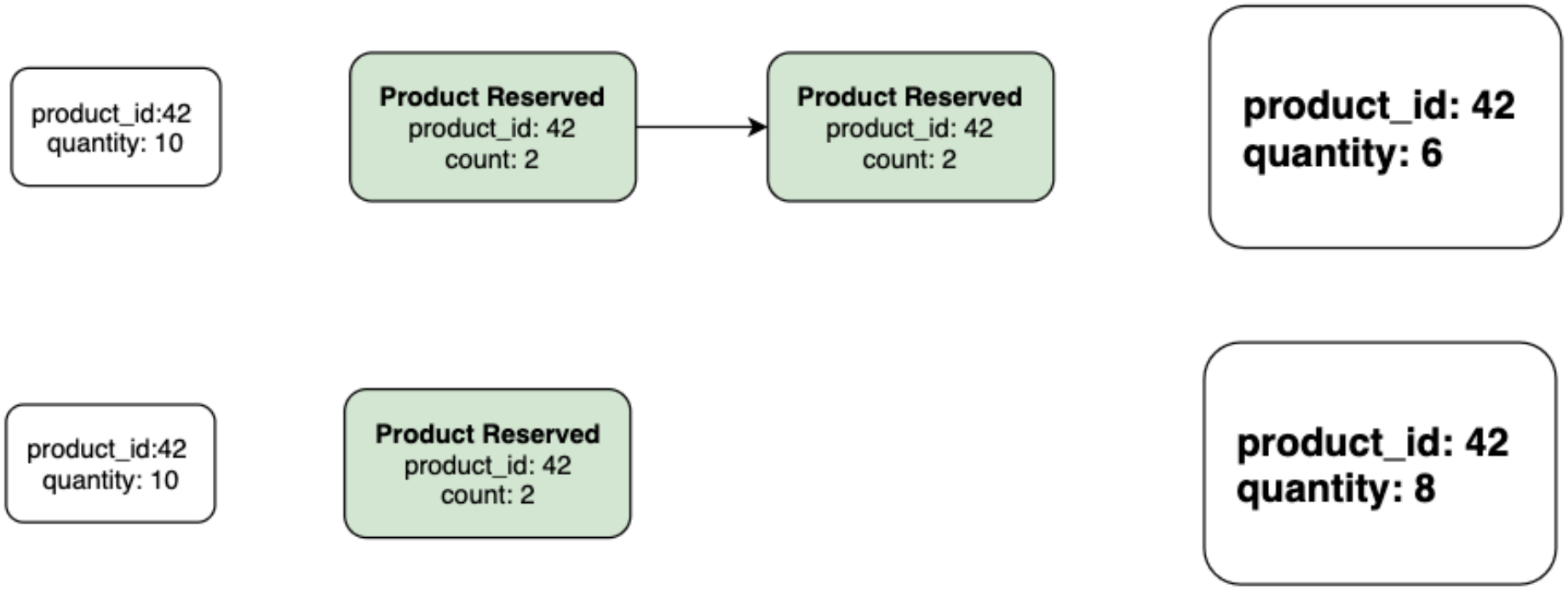
Идемпотентные сообщения

Например, события, которые передают новое состояние объекта чаще всего являются идемпотентными. Если мы их применим последовательно 2 раза не изменят состояния.



Идемпотентные сообщения

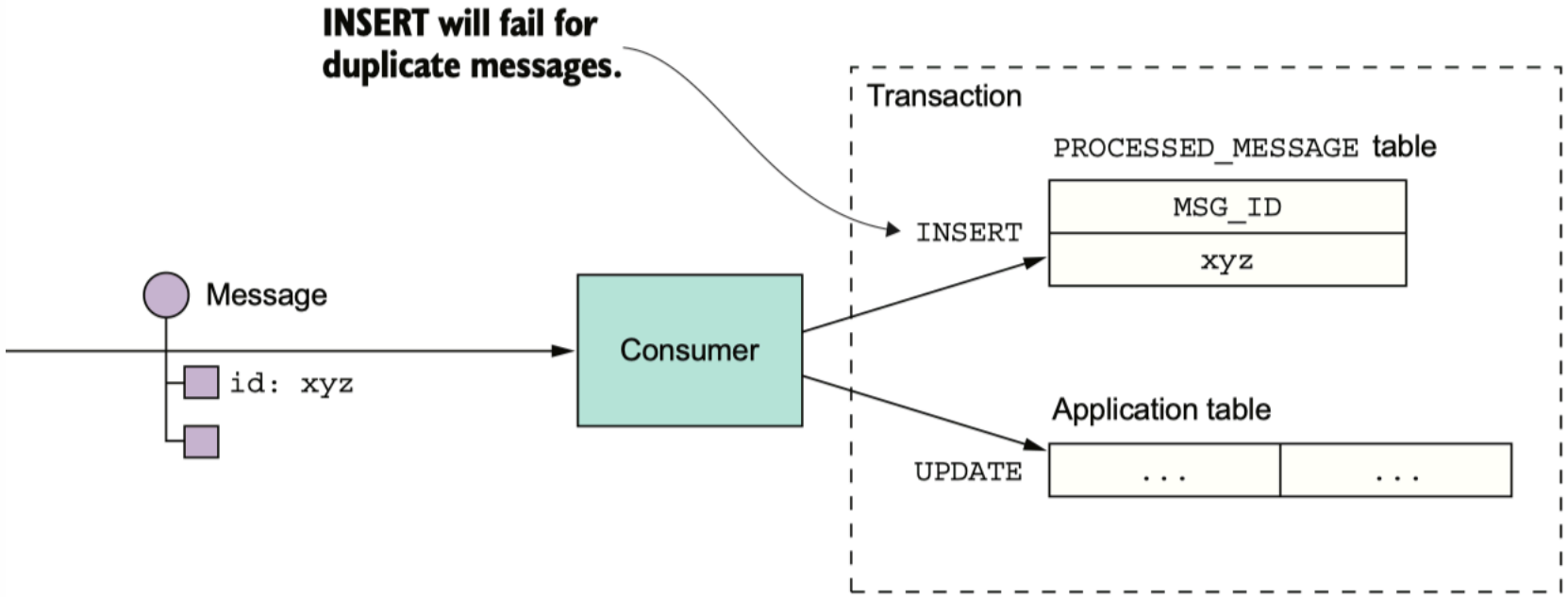
А вот события, которые передают только изменение состояния объекта, чаще всего при двойной обработке все испортят.



Idempotent reciever

Из продюсера
- отправлять сообщения вместе с ключом идемпотентности (msg_id)

В консьюмере:
- Завести табличку с отработанными сообщениями (processed_message)
- Коммиты в табличку с данными и processed_table происходят транзакционно
- Если сообщение уже было, то ничего не делать



Коммутативные сообщения

К сожалению, порядок сообщений также может портиться.

Даже в случае, если сам брокер сообщений предоставляет гарантии порядка внутри себя, все-равно у нас есть:

- Конкурентные продюсеры
  От нескольких продюсеров сообщения могут приходить в разном порядоке
- Конкурентные консьюмеры
  Консьюмеры могут обрабатывать сообщения в разном порядке.

Убираем конкурентность

Один из вариантов – это убрать конкурентность из продюсеров или консьюмеров.

Очевидно, что нам важен порядок событий (или сообщений), относящихся к какой-то одной конкретной сущностью.

Polling publisher

Если у нас есть transactional outbox, мы можем тогда в »безопасном» режиме написать отдельный сервис, который забирает события из БД и помечает их как отправленные. И отправляет их гарантированно в том порядке, в котором надо.

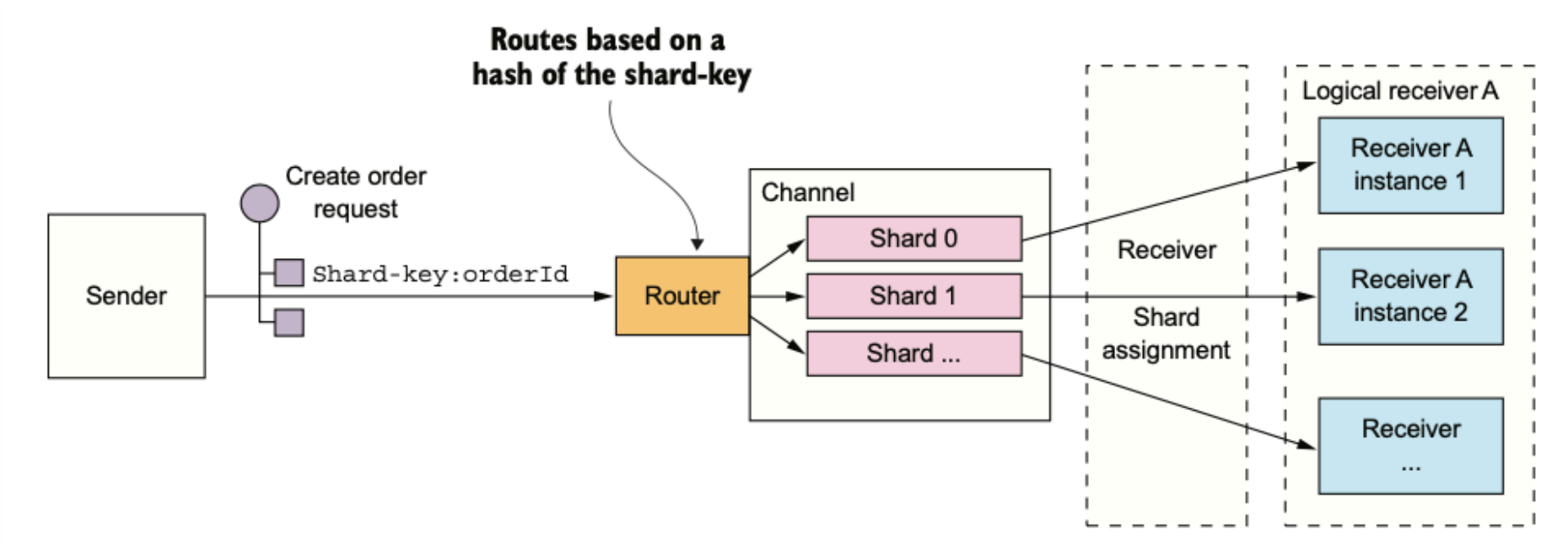
Для увеличения нагрузки, можем запустить несколько инстансов сервиса (асинхронной джобы), каждый из которых забирает свой набор объектов. Обязательно, чтобы все события для одного объекта, попали только к одному инстансу.

```
SELECT * FROM OUTBOX ORDERED BY ... ASC

BEGIN
 DELETE FROM OUTBOX WHERE ID in (....)
COMMIT
```

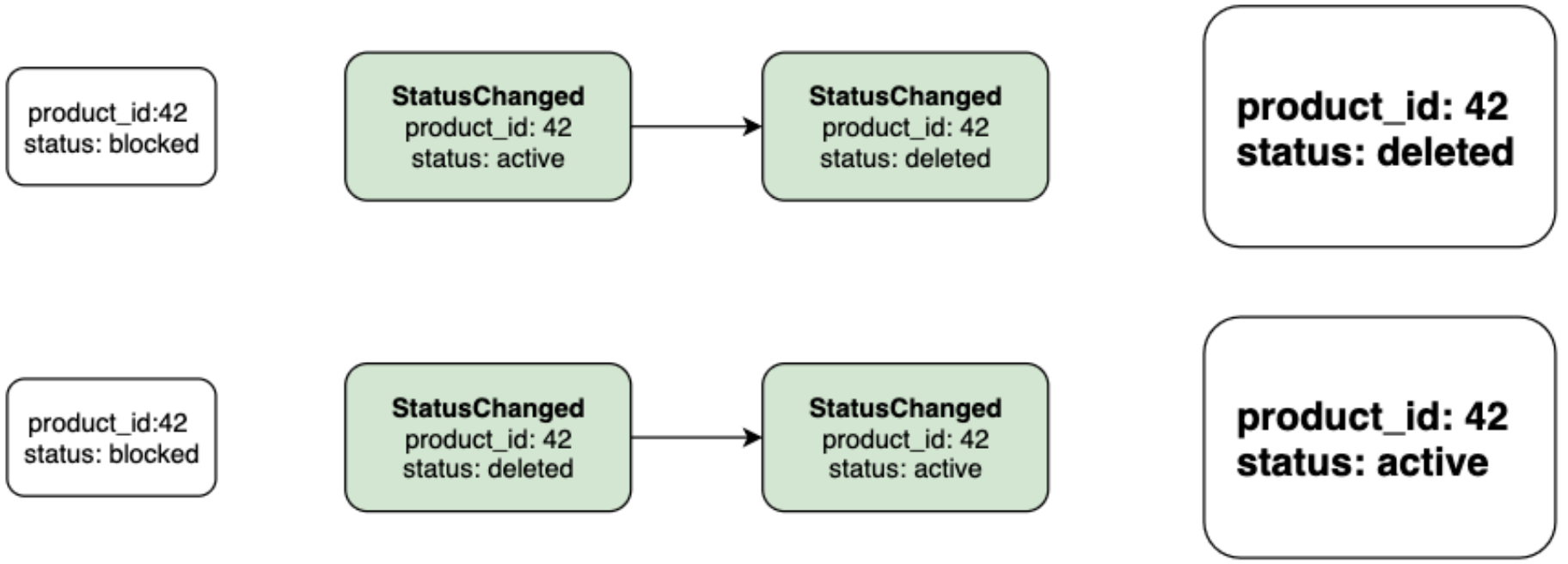
Sharded consumers

Для того, чтобы убрать конкурентность со стороны консьюмеров – делаем тоже самое, каждый получатель присасывается к своему шарду. Шарды сделаны таким образом, чтобы события (сообщения) про один объект были в одном шарде.



Коммутативные сообщения

Например, сообщения, которые передают состояния, не обладают коммутативностью, а значит, последовательное применение приведет к разным результатам.



Коммутативные сообщения

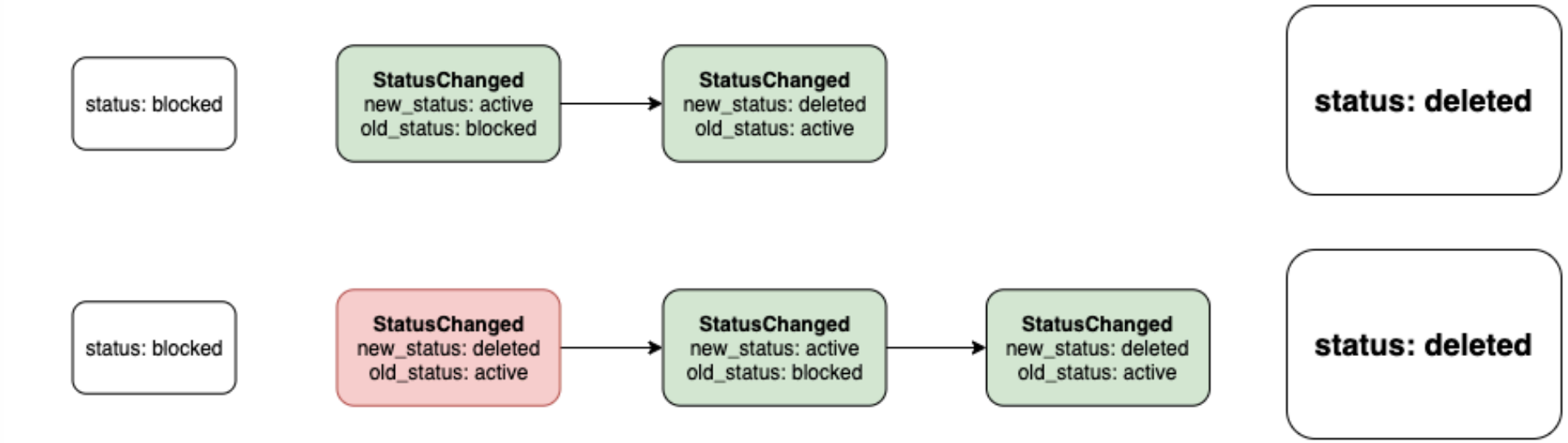
Важно понимать, что мы в дальнейшем делаем с сообщениями, и насколько нам важен правильный порядок.

Например, если мы просто сохраняем сообщения в базу (в качестве лога), и никак не обрабатываем, то мы можем их сохранять as is.

Паттерн Compare-and-set

Для решения проблемы идемпотентности и коммутативности, можем скрестить события, которые несут изменения с событиями, которые несут состояния.

В этом случае, продюсер, прежде чем применять событие сравнивает текущее состояние и если оно совпадает, то применяет событие, в ином случае возвращает событие назад в очередь.



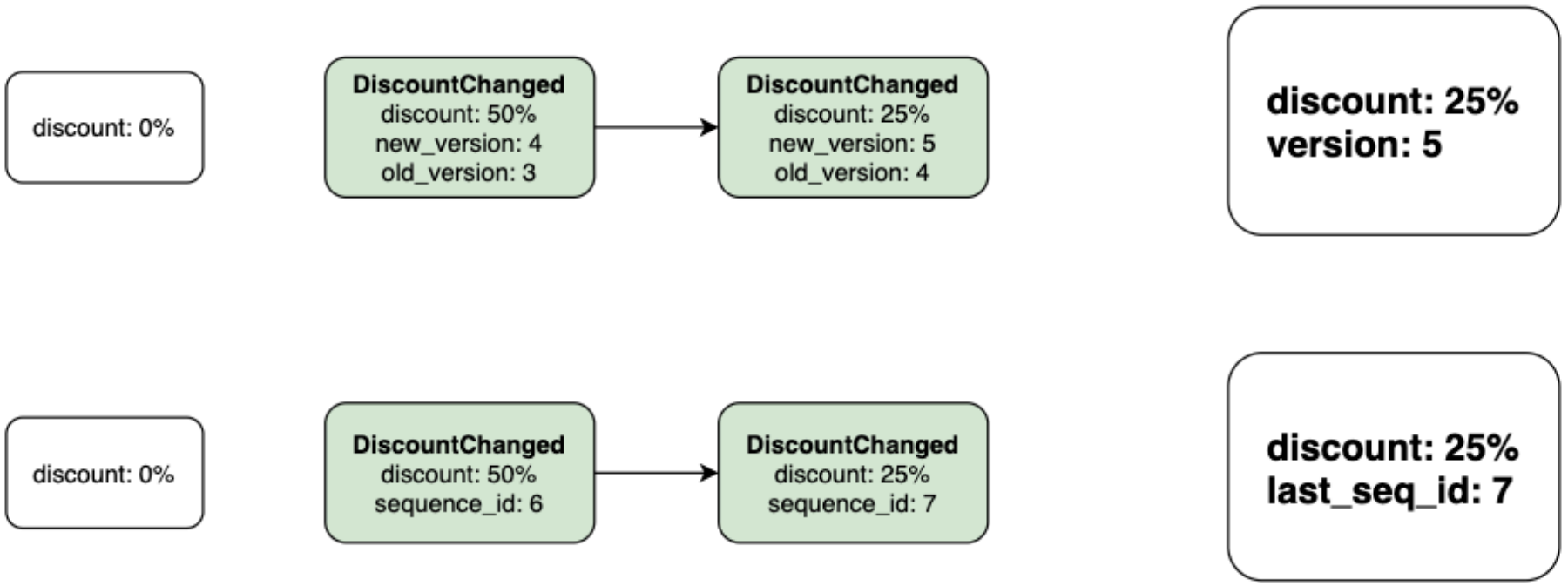
Паттерн Compare-and-set

Потенциально compare-and-set может приводить к неправильному порядку, но это не всегда важно.

Версионирование

Когда крайне важен порядок сообщений можно использовать более сложную схему, похожу на репликацию данных из БД.

Во все события (сообщения) прокидываем либо sequence_id, либо переход из старой версии в новую.

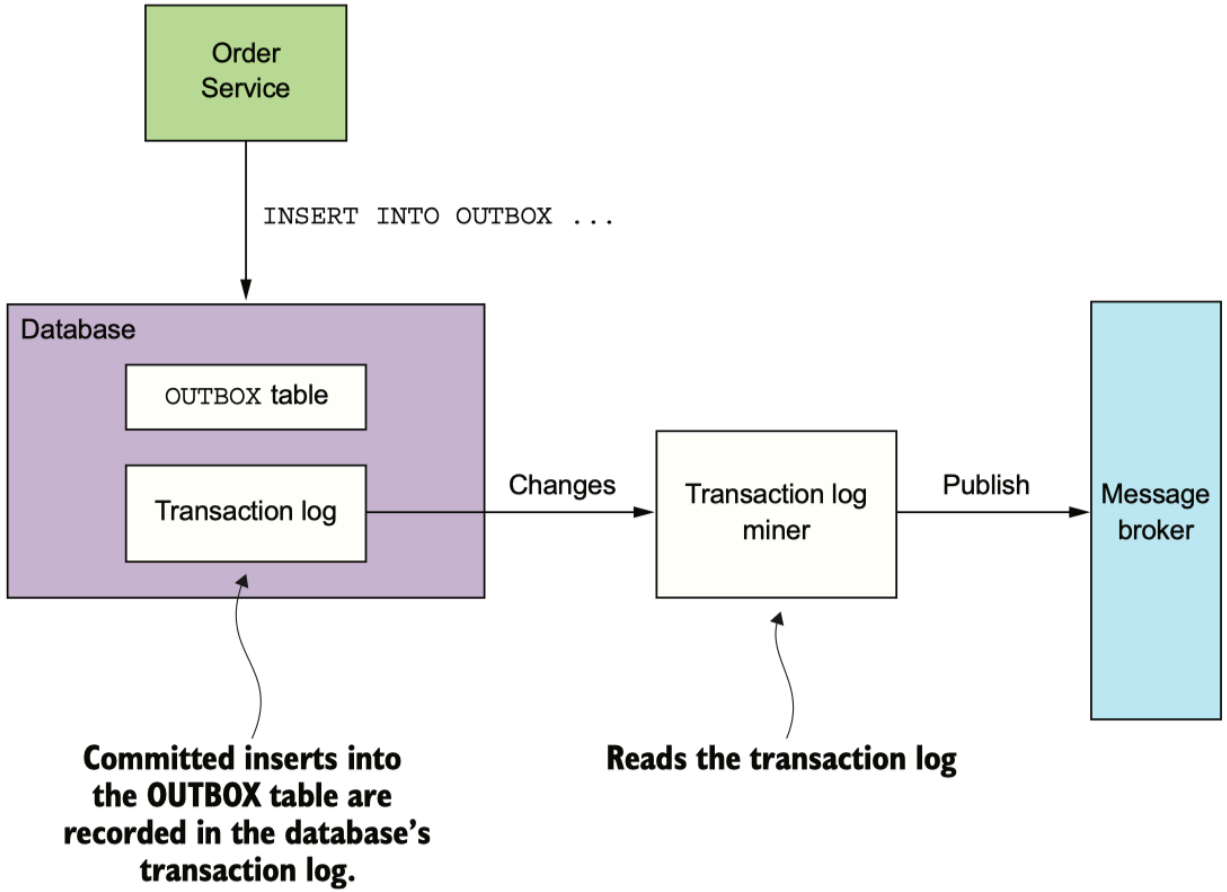


Версионирование

Реализация такой схемы довольно-таки не проста, и требует большой аккуратности. Например, если хоть одно событие пропадет, то все сломается.

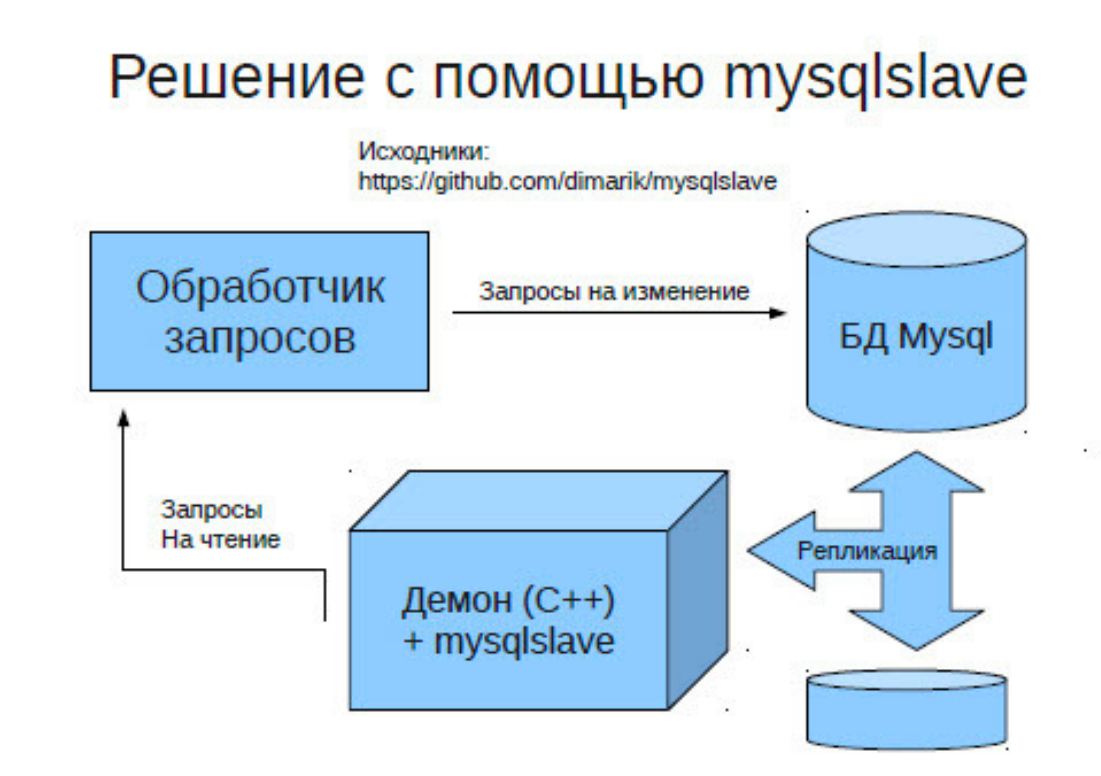
Transaction Log Miner

- Читаем WAL-лог базы данных или прикидываемся репликой
- Можем читать outbox table, можем читать сразу данные



Transaction Log Miner

Такая схема используется в Mail.Ru (mytarget) - https://habr.com/ru/company/mailru/blog/219015/



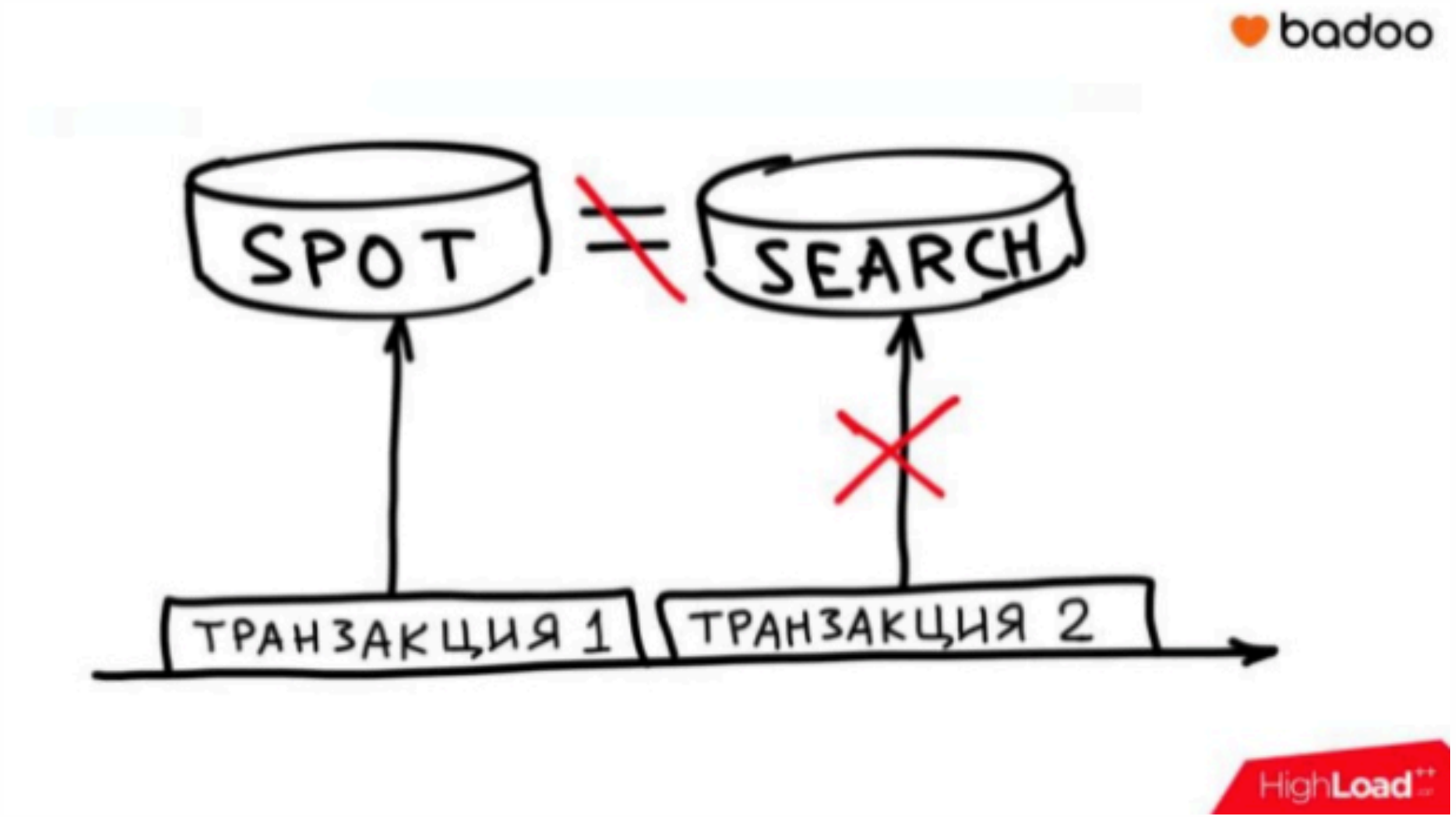
Transaction Log Miner

Такая схема используется в Rambler - https://www.youtube.com/watch?v=oByOmhOmOh4

Синхронизация данных

http://www.highload.ru/2017/abstracts/2930.html

Синхронизация данных



http://www.highload.ru/2017/abstracts/2930.html

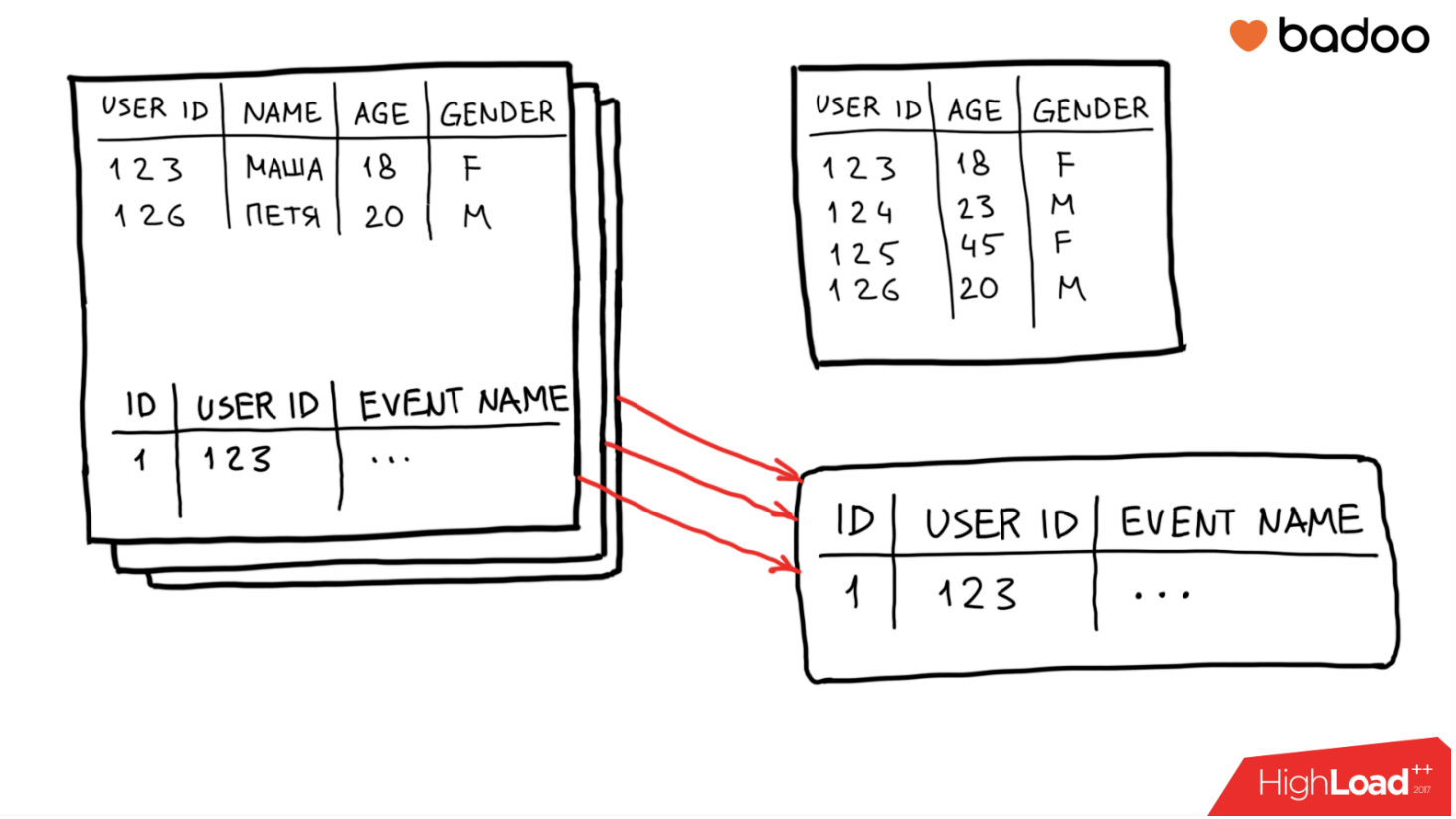
Синхронизация данных

http://www.highload.ru/2017/abstracts/2930.html

Синхронизация данных

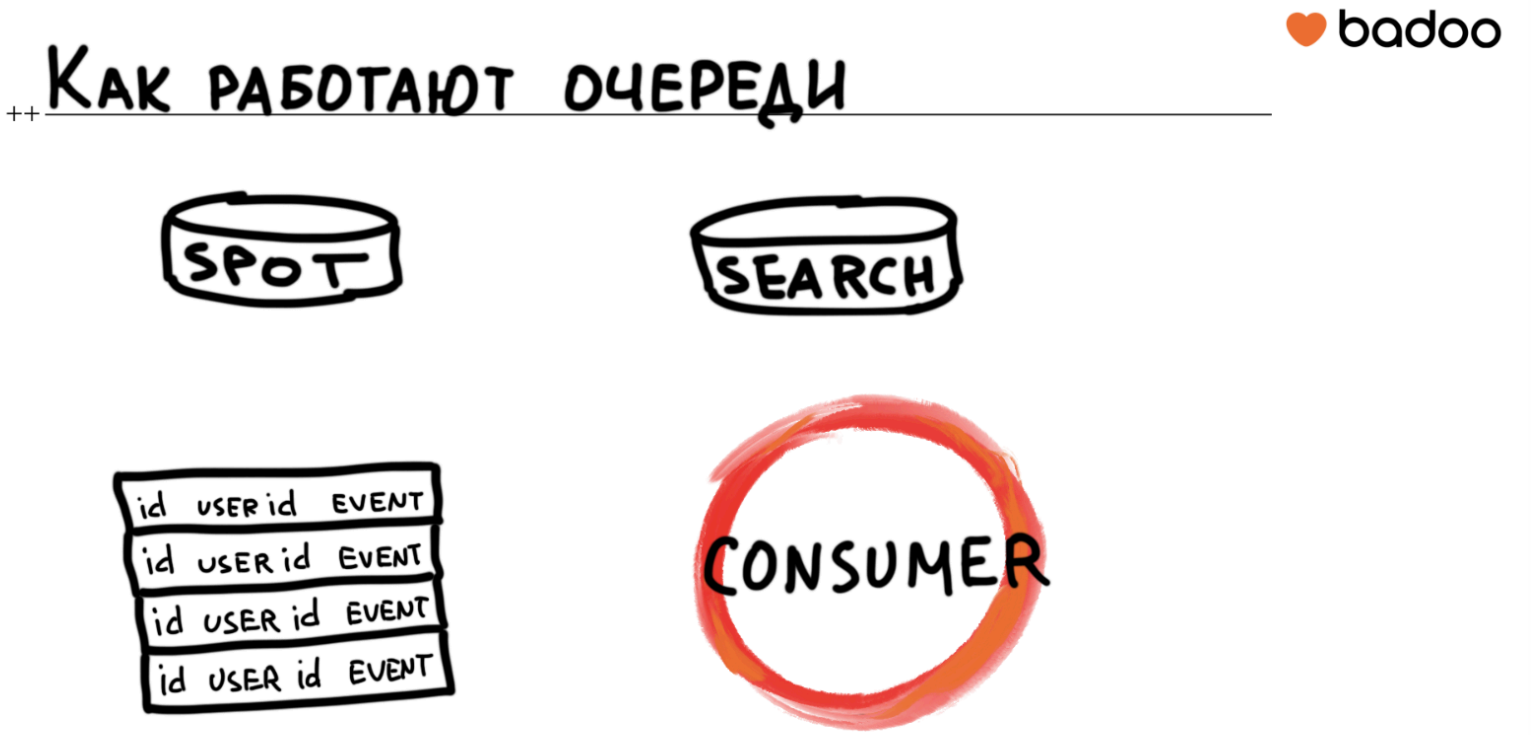
http://www.highload.ru/2017/abstracts/2930.html

Синхронизация данных



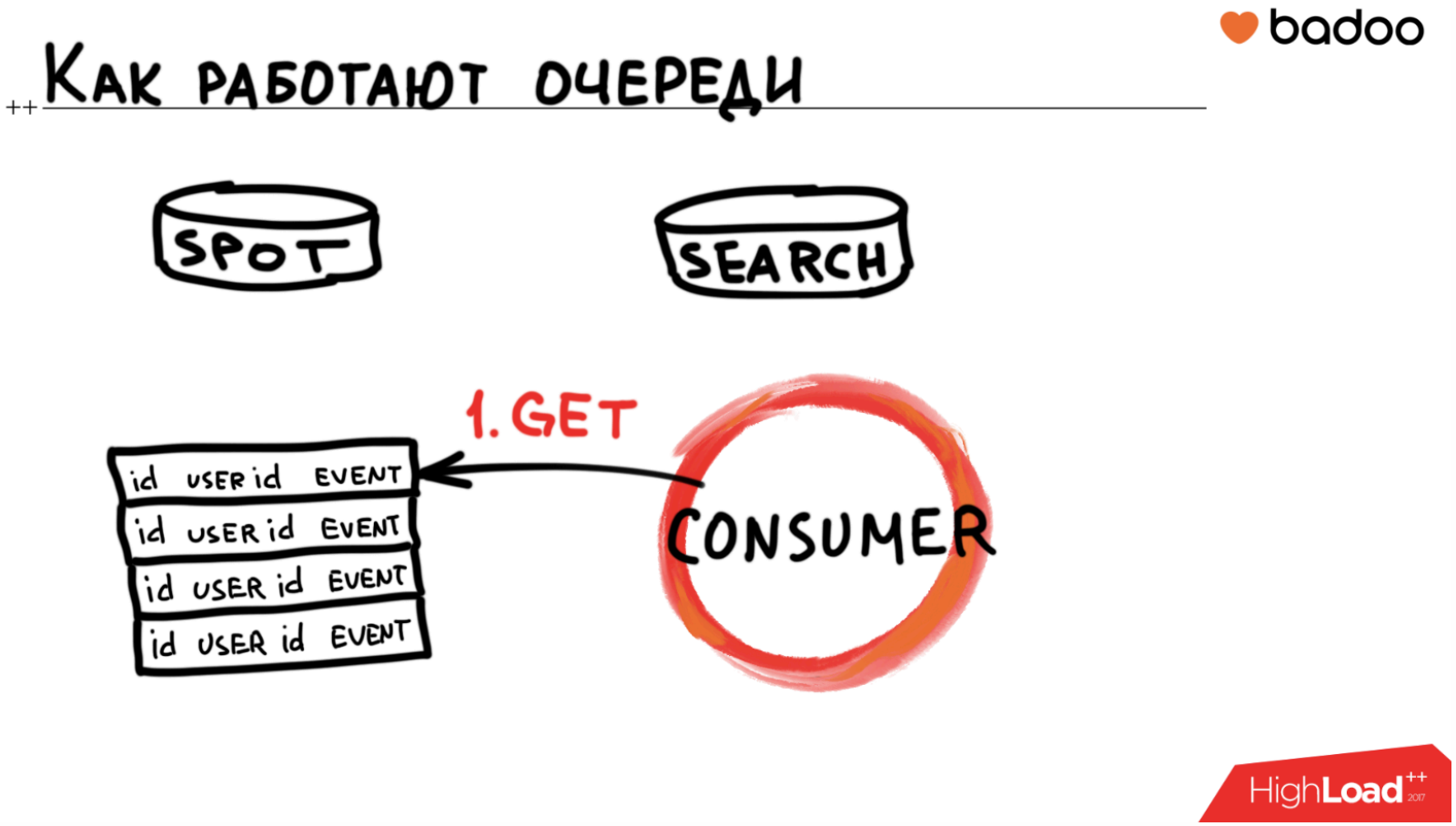
http://www.highload.ru/2017/abstracts/2930.html

Синхронизация данных



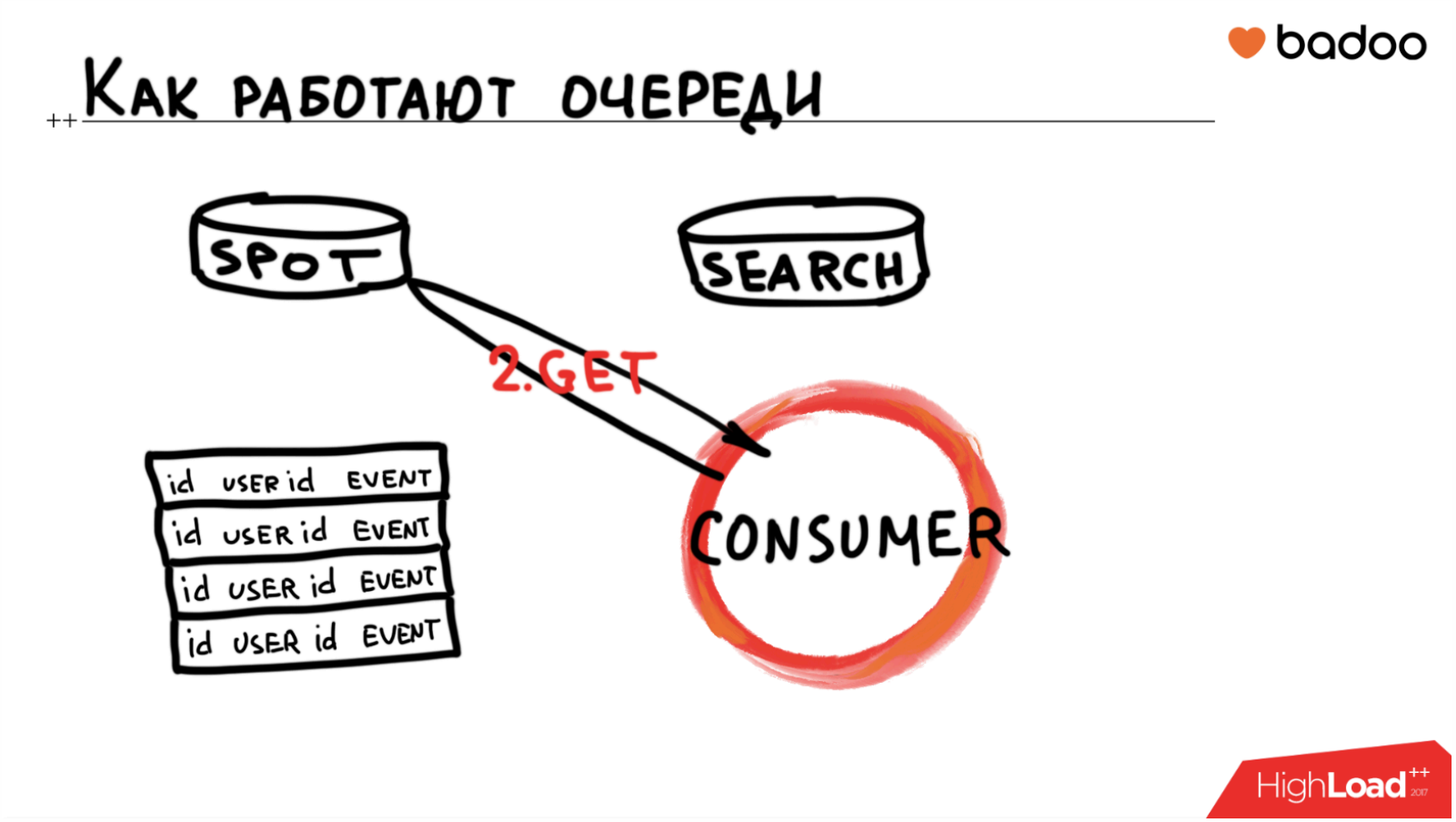
http://www.highload.ru/2017/abstracts/2930.html

Синхронизация данных



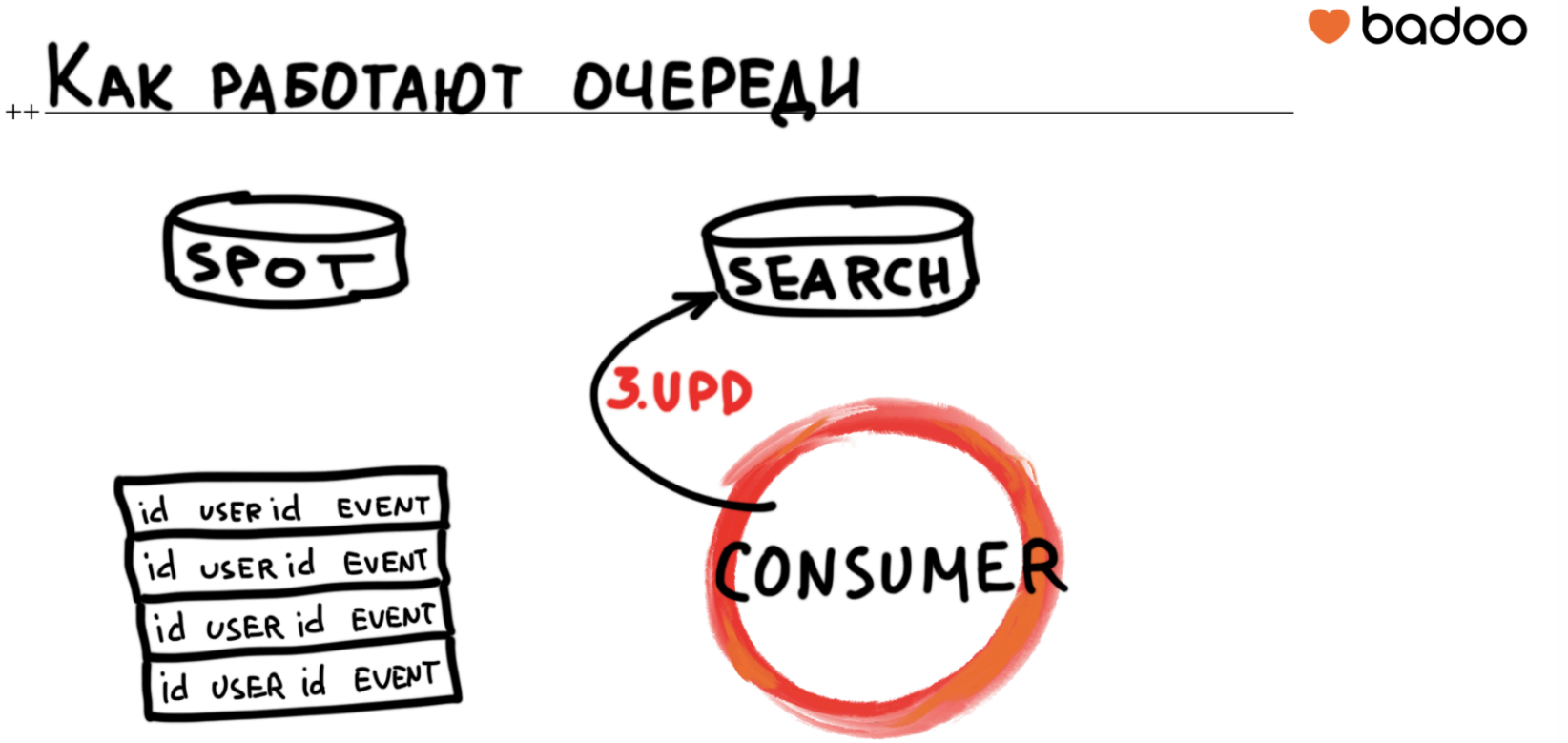
http://www.highload.ru/2017/abstracts/2930.html

Синхронизация данных



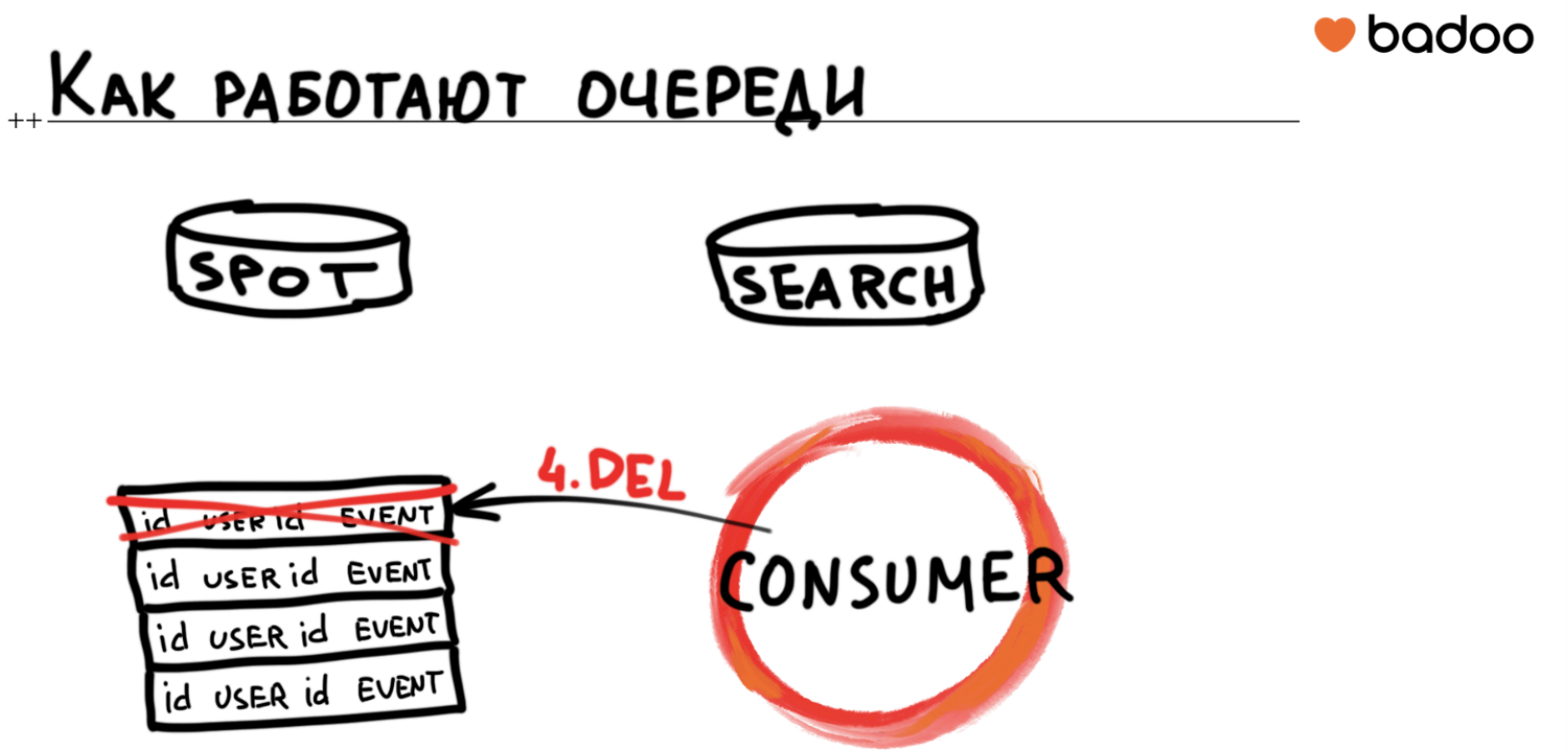
http://www.highload.ru/2017/abstracts/2930.html

Синхронизация данных



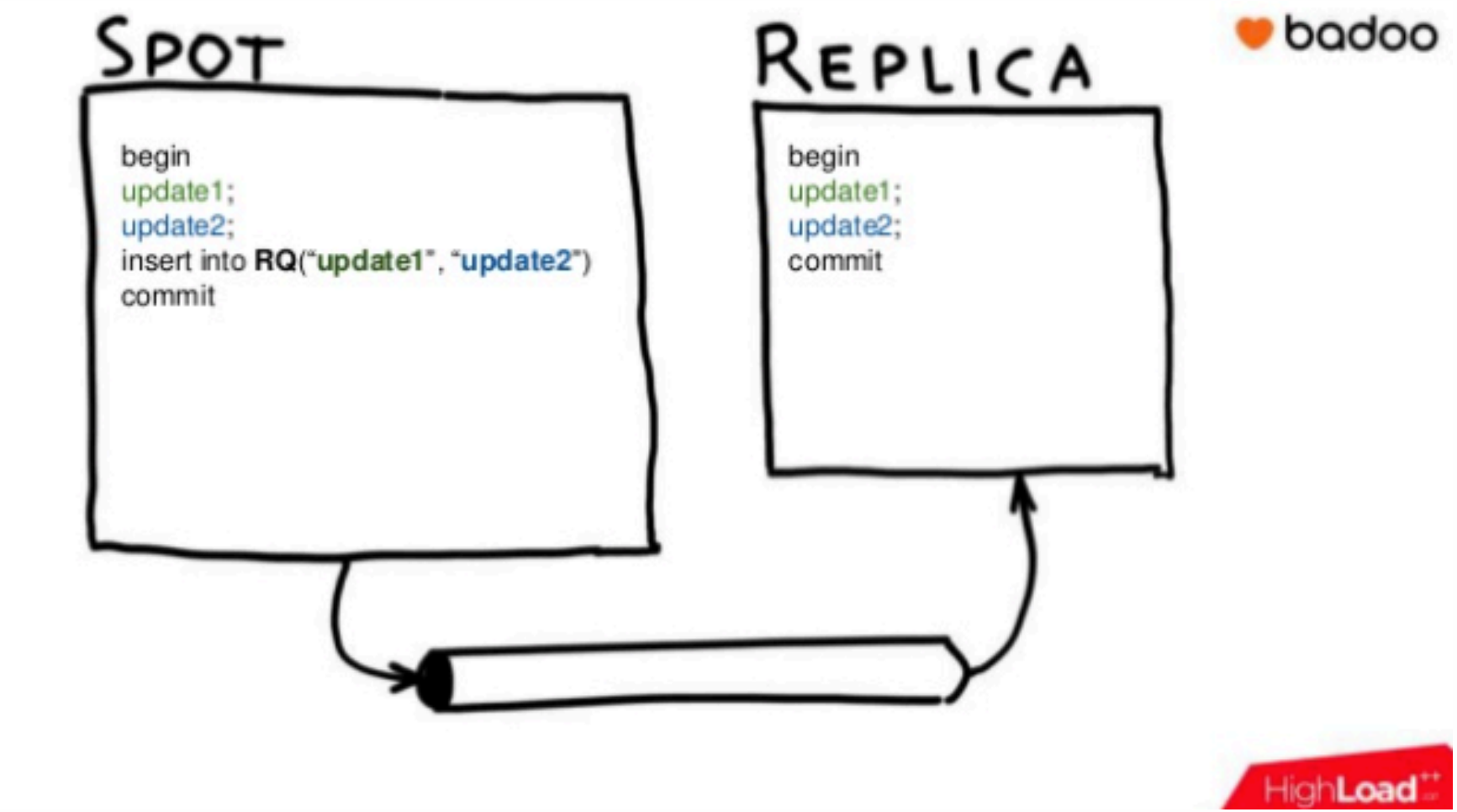
http://www.highload.ru/2017/abstracts/2930.html

Синхронизация данных



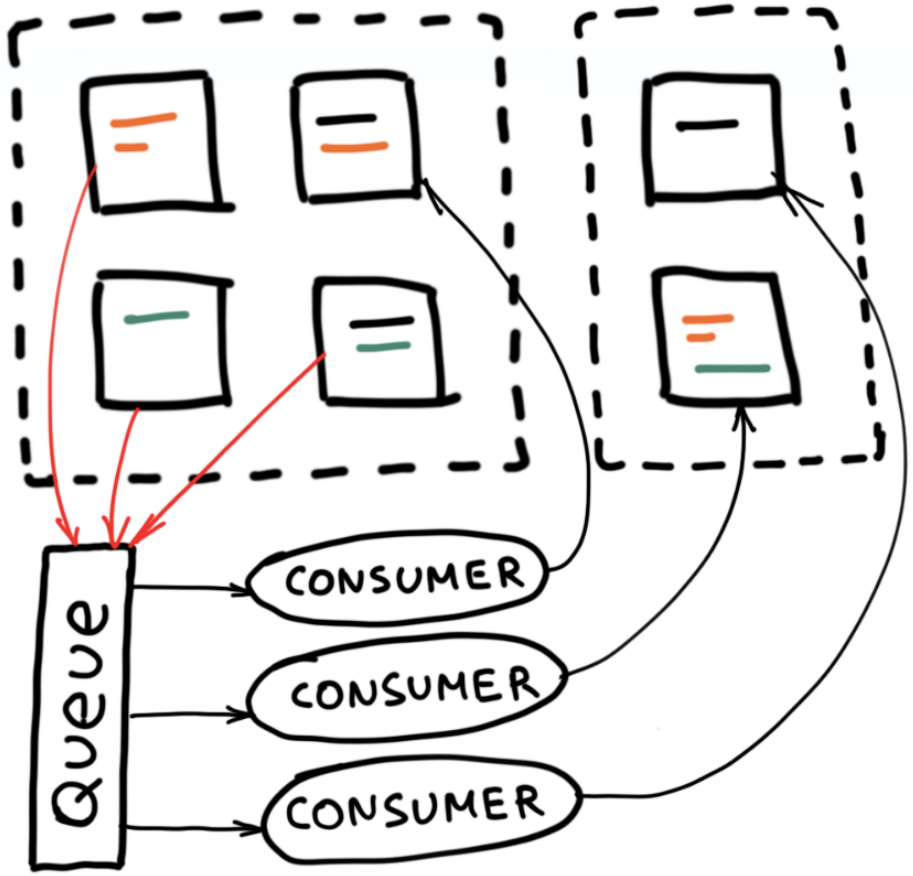
http://www.highload.ru/2017/abstracts/2930.html

Синхронизация данных



http://www.highload.ru/2017/abstracts/2930.html

Синхронизация данных



http://www.highload.ru/2017/abstracts/2930.html

Спасибо
за внимание!