ozon{tech

SQL

План занятия

Backing services

В манифесте The Twelve Factor App вводится понятие сторонней службы. Она не является другим микросервисом, доступна по сети, и приложение без нее работает. Базы данных тоже являются такой сторонней службой.

Сервис при этом нигде в коде не должен делать различий в том, с какой БД он работает: продовой, тестовой и т. п. Единственное, чем управляется подключение, — это URL или похожая на него строка DSN (data source name).

Пример DSN:

`postgres://postgres:@localhost:5432/test?sslmode=disable`.

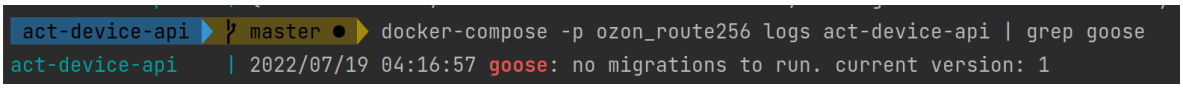
Адрес и настройки подключения хранятся в сервисах конфигурации (etcd/vault) или конфигурациях приложения.

Миграции

Инстансом базы данных (размещением на физических или виртуальных хостах) управляет SRE, часто вместе с разработчиком сервиса. Тем, как организована БД внутри, ее сущностями — только разработчик сервиса. Схема данных и сами данные в сервисах Ozon хранятся как SQL-код, обычно в том же репозитории, что и сам сервис.

В тестовом device-api можно найти миграцию в файле migrations/00001_fill_tables.sql, а код запуска агента goose, который управляет миграцией, — в cmd/grpc-server/main.go. Миграция будет применяться на старте сервиса. Если goose увидит, что она уже применена, код не будет выполняться — сервис будет стартовать дальше. Если в тестах будет необходима новая миграция, ее SQL нужно будет поместить в файл migrations/00002_<имя миграции>.sql. Как видите, имя файла содержит порядковый номер, чтобы goose смог этот порядок поддерживать и при выполнении скриптов.

В логах сервиса можно найти записи про успешные или неуспешные миграции на новую версию DB.



RDB & SQL

RDB

До появления идеи (и реализации) реляционной базы данных разработчику нужно было четко понимать, где и как хранятся данные, что происходит при их чтении с носителя, как обновить данные без потерь.

Эдгар Кодд более 50 лет назад, в 1970 году, в своей статье A Relational Model of Data for Large Shared Data Banks предложил новую модель хранения данных. В ней и последующих работах он вводит понятие нормальных форм (1NF) — необходимых условий для хранения данных.

Перед тем как разобраться с ними, нам нужно сначала понять, что такое отношение (relation) — сущность, которая дала название реляционным базам данных.

Представим, что когда-то не было нашего тестового проекта device-api. Администратор вручную собирал списки устройств на руках у сотрудников и вел эти списки в своем черном блокноте в клеточку.

Типы данных или домены — те поля, которые он учитывает в своих записях (пользователь, OS на устройстве, дата, когда запись появилась в блокноте). Если мы возьмем все возможные значения этих доменов (очень много дат, несколько OS и всех сотрудников) и выпишем в очень длинный список всех возможных их комбинаций, получим декартово произведение. Как мы понимаем, в нем мало смысла: одновременно у всех сотрудников не могут быть все устройства компании.

Администратор пишет себе в блокнот реальное положение дел, которое является подмножеством этого декартового произведения. Оно называется отношением, и его мы хотим хранить в базе данных.

Теперь мы готовы дать определения некоторых нормальных форм. Работая с базой данных настоящего сервиса, вы заметите продиктованные ими принципы хранения.

1NF: все строки уникальны, в столбце везде один и тот же тип данных, в ячейке атомарное значение.

2NF: находится в 1NF и есть ключ, от которого зависят остальные столбцы. Здесь и далее предполагается, что отношение, которое удовлетворяет очередной нормальной форме, удовлетворяет и всем предыдущим тоже. Если вы когда-либо видели лук, поймете, о чем я.

3NF: 2NF и столбцы не зависят друг от друга.

За 3NF непредсказуемо идет форма Бойса-Кодда, но на ней формализация не заканчивается. Всего нормальных форм восемь, не буду их описывать, поскольку нам понадобится еще больше терминологии. Лучше потратим это время на SQL.

SQL

Еще одна идея, описанная четыре года спустя, в 1974 году, Дональдом Чемберленом и Раймондом Бойсом в труде SEQUEL: A structured English query language. Авторы

предложили простой язык, описывающий операции над табличными данными, и понятный не только разработчику.

Вместе с подходом RDB SQL реализуется большинством современных движков баз данных, при этом у SQL есть своя стандартизация, которая гарантирует, что различия между синтаксисом будут минимальными (но нет).

В первом стандарте SQL-86 было много хорошо известных каждому терминов: SELECT, FROM, WHERE, GROUP BY, INSERT, UPDATE, DELETE, CREATE TABLE… но не было JOIN. Предполагалось, что в FROM может быть перечислено несколько таблиц, дальше из них собиралось декартово произведение — и нужные нам строки фильтровались условиями в WHERE.

Затем в SQL-89 появилось описание primary- и foreign-ключей и DEFAULT, а в SQL-92 — привычный JOIN. Можно смело сказать, что SQL-92 — это и есть тот базовый SQL, который должен знать каждый.

Стандарт развивается и сейчас: после череды SQL:1999, SQL:2003, SQL:2006, SQL:2008, SQL:2011, SQL:2016 (в том числе описывает поддержку JSON) последний на сегодня стандарт — это SQL:2019.

Теперь мы хорошо понимаем, что реляционные базы и SQL тесно связаны, потому что они описывают манипуляции над табличными данными, но это вовсе не синонимы.

SQL-101

Сейчас, благодаря стандартизации RDB, мы можем использовать любой из десятка клиентов, чтобы посмотреть, что хранится в базе данных нашего device-api.

Для ежедневной работы мы используем встроенные в IDE клиенты и https://redash.io/. Наши аналитики используют также Jupiter Notebooks.

Примеры, которые есть в презентации и в https://gitlab.ozon.dev/-/snippets/7, я выполняю в pgAdmin4.

Конфигурацию для подключения можно подсмотреть в конфигурации нашего сервиса config.yml.

Команды SQL условно подразделяются на подмножества.

- DDL — Data Definition Language (определения таблиц и других объектов в БД).
- DML — Data Manipulation Language (пишем и читаем данные).
- TCL — Transaction Control Language (управляем транзакциями).
- DCL — Data Control Language (управляем доступом к данным).

Посмотрим на их примеры.

Мы настроили подключение к базе данных и увидели таблицы в device-api в схеме public. Сгенерируем первый запрос: правой кнопкой на таблице devices > View/Edit Data > First 100 rows.

```
SELECT * FROM public.devices
ORDER BY id ASC LIMIT 100

```

SELECT — один из терминов DML. В запросе указан ORDER BY, без него вы получите несортированный набор строк, и LIMIT (обязательно используйте его при работе с продовыми БД).

Перед именем таблицы указан public — имя схемы по умолчанию. Схемы похожи на xml namespaces и на любые другие namespace. В прикладном использовании вы можете создавать отдельные схемы, чтобы помещать туда тестовые триггеры, функции, таблицы и пр. и избегать конфликтов с такими же объектами в схеме данных приложения. Пример использования есть в миграции migrations/00001_fill_tables.sql.

Следующий пример:

```
SELECT *
FROM DEVICES D
JOIN DEVICES_EVENTS DE ON D.ID = DE.DEVICE_ID
WHERE 1 = 1
    AND D.REMOVED
ORDER BY D.ID ASC
LIMIT 100
```

Он тоже содержит термины из DML. JOIN — объединения таблиц по условию, они были нормализованы, и нам нужно восстановить запись. В ON — условие для JOIN, их может быть несколько. Во внешних источниках вы можете прочитать про типа JOIN — LEFT/RIGHT/INNER/CROSS (да, декартово произведение).

Кроме этого, тут есть немного синтаксического сахара — D, DE — алиасы к именам таблиц. Наверняка в таблицах найдутся одинаковые имена колонок, чтобы исключить неоднозначность, их можно адресовать с именем таблицы (DEVICES.id) или с именем алиаса (D.id).

В D.REMOVED хранится bool, и true/false можно опустить.

1=1 используется, чтобы можно было перечислять все условия WHERE … AND с новых строк, комментировать отдельные AND-условия. Обычно SQL-сервер довольно умен, чтобы это условие не изменяло план запроса (о плане речь чуть позже).

Запрос выше можно переписать несколькими способами.

```
WITH REMOVED AS
    (SELECT *
        FROM DEVICES D
        WHERE D.REMOVED )
SELECT *
FROM REMOVED R
JOIN DEVICES_EVENTS DE ON R.ID = DE.DEVICE_ID

```

WITH удобен для отладки. Можно выделить и выполнить только часть в скобках. В некоторых запросах он может быть эффективнее, чем просто JOIN.

```
SELECT *
FROM DEVICES_EVENTS DE
WHERE DE.DEVICE_ID IN
        (SELECT ID
            FROM DEVICES D
            WHERE D.REMOVED)
```

Можем использовать и подзапросы в теле основного запроса.

Все запросы выше используют SELECT *. Не делайте этого! На продовых данных и тем более в коде выбирайте только нужные вам столбцы. Это позволит избежать хрупкости тестов при изменениях схемы. Без перечисления столбцов можно изменить порядок колонок, будут лишние чтения.

После такого предупреждения дальше будем использовать только имена колонок.

Найдем пользователя, у которого на руках больше о телефона:

```
SELECT USER_ID,
    COUNT(1) AS CC
FROM DEVICES D
WHERE NOT D.REMOVED
    AND PLATFORM in ('Android',
                     'Ios')
GROUP BY USER_ID
ORDER BY USER_ID DESC
HAVING COUNT(1) > 1
```

HAVING не знает про алиасы колонок (CC), но позволяет справиться с задачей, когда надо посчитать условия для каких-то агрегирующих функций.

В выборках часто приходится работать с диапазонами дат и значений, для этого есть операторы =, <, > и более человеческое BETWEEN.

```sql
SELECT PLATFORM,
    UPDATED_AT, *
FROM DEVICES D
WHERE UPDATED_AT BETWEEN NOW() - interval '1 month' AND NOW()
    OR (UPDATED_AT > '2022-04-19'::TIMESTAMP
                    AND UPDATED_AT < '2022-05-19'::TIMESTAMP)
ORDER BY UPDATED_AT DESC
```

Тут есть и пример встроенной функции NOW(), возвращающей тайм-штамп времени выполнения запроса, приведения строки к типу данных (::TIMESTAMP) и Postgres-специфичного interval, который очень удобен при работе с датами.

В Postgres поддержаны типы данных JSON и JSONB. Кажется, им не место в реляционной БД! Но реальные потребности бизнеса и разработки диктуют свои условия. Когда-то для хранения json понадобился бы NoSQL, сейчас его можно хранить и работать с ним вместе с данными реляционной БД.

```sql
SELECT PAYLOAD - >> 'platform'
FROM DEVICES_EVENTS
```

Такой тип данным нам понадобится, чтобы хранить данные разных форматов в одной колонке (вспомните про 1NF). Возможно, форматы наших данных часто меняются или нам нужны логи изменений данных.

Если мы зайдем во вкладку SQL в свойствах таблицы devices, увидим код:

```sql
-- Table: public.devices_events

-- DROP TABLE IF EXISTS public.devices_events;

CREATE TABLE IF NOT EXISTS public.devices_events
(
    id integer NOT NULL DEFAULT nextval('devices_events_id_seq'::regclass),
    device_id bigint,
    type smallint NOT NULL,
    status smallint NOT NULL,
    payload jsonb,
    created_at timestamp without time zone NOT NULL DEFAULT now(),
    updated_at timestamp without time zone NOT NULL DEFAULT now(),
    CONSTRAINT devices_events_pkey PRIMARY KEY (id),
    CONSTRAINT devices_events_device_id_fkey FOREIGN KEY (device_id)
        REFERENCES public.devices (id) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)

TABLESPACE pg_default;

ALTER TABLE IF EXISTS public.devices_events
    OWNER to docker;
```

Он сгенерирован автоматически и может отличаться от того, которым создавалась таблица в миграции. Выполняемые тут команды относятся к подмножествам DDL (CREATE) и DCL (ALTER). Стоит обратить внимание, что на некоторых колонках есть DEFAULT — значит, их можно не перечислять при вставке записи в БД, в них запишется значение из DEFAULT.

Если вы обращаетесь к БД в коде тестов (что нельзя назвать рекомендованной практикой), вы наверняка будете использовать параметризованные запросы. Формат собираемого запроса зависит от вашей клиентской библиотеки, возможно, вы

используете ORM или собираете его f-строками. Однако параметризацию в Postgres можно встретить и в обычном коде.

```
PREPARE platform (text) AS
SELECT USER_ID,
    PLATFORM
FROM DEVICES
WHERE PLATFORM = $1;

EXECUTE platform('Android');

```

PREPARE рекомендуется для часто выполняемых запросов. Код, выполняемый в PREPARE, прочитается и распарсится однократно, будет храниться на серверной стороне, дожидаясь выполнения.

Транзакции

Наша БД не только реляционная (и немного NoSQL из-за jsonb) — она еще и транзакционная. Транзакция — множество операций, которое переводит БД из одного согласованного (то есть отвечающего бизнес-логике) состояния в другое. Обязательно вместе с упоминанием транзакции рассмотрим ее свойства ACID.

Atomicity — Атомарность. Транзакция или выполнена или нет

Consistency — Согласованность. Ее упомянули в определении транзакции.

Isolation — Изолированность. Отсутствие влияние одной транзакции на другую. Тут разработчикам могут понадобиться блокировки, примеры проблем и борьбы с ними можно найти в блоге ozontech на habr. Это очень интересная тема, она порождает и интересные подходы в тестировании, например: https://github.com/allaboutapps/integresql#approach-2a-isolation-by-transactions

Durability — Надежность. Даже если после подтверждения транзакции отключили питание в ЦОД, данные уже сохранены.

Посмотрим простой пример с Read Comitted на двух коннектах к БД:

```
BEGIN;
INSERT INTO devices (platform, user_id, entered_at)
VALUES ('Ubuntu', 1, now())
SELECT * FROM devices WHERE platform = 'Ubuntu'
COMMIT;
```

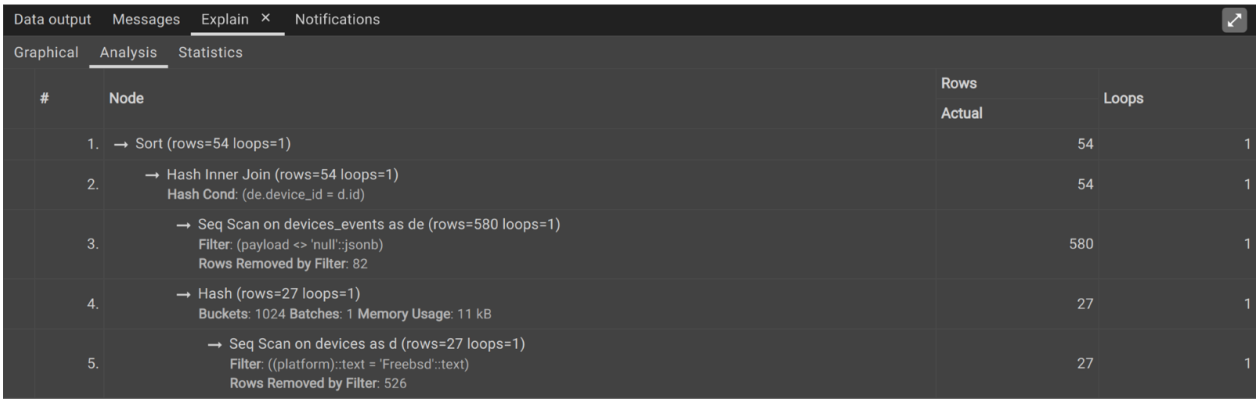
Если посмотреть во втором соединении данные в БД, мы не увидим записей с platform = ‘Ubuntu’, пока не будут выполнен COMMIT (ой, опять использовал SELECT *).

Планировщик запросов

Однажды ваш код может перестать работать так хорошо, как раньше. Обычно это связано с ростом размера БД, но не только. Используйте в таких ситуациях планировщик запросов.

```
EXPLAIN ANALYSE
SELECT *
FROM DEVICES_EVENTS DE
JOIN DEVICES D ON D.ID = DE.DEVICE_ID
WHERE DE.PAYLOAD != 'null'
    AND D.PLATFORM = 'Freebsd'
ORDER BY DE.ID DESC,
    D.ID ASC
```

EXPLAIN ANALYSE выполнит запрос (без ANALYSE только построит примерный план выполнения) и визуализирует условные стоимости составляющих запрос операций.



Data output | Messages | Explain × | Notifications

Graphical | Analysis | Statistics

#	Node	Rows Actual	Loops
1.	→ Sort (rows=54 loops=1)	54	1
2.	→ Hash Inner Join (rows=54 loops=1) **Hash Cond**: (de.device_id = d.id)	54	1
3.	→ Seq Scan on devices_events as de (rows=580 loops=1) **Filter**: (payload <> 'null'::jsonb) **Rows Removed by Filter**: 82	580	1
4.	→ Hash (rows=27 loops=1) **Buckets**: 1024 **Batches**: 1 **Memory Usage**: 11 kB	27	1
5.	→ Seq Scan on devices as d (rows=27 loops=1) **Filter**: ((platform)::text = 'Freebsd'::text) **Rows Removed by Filter**: 526	27	1

Конкретно в этом случае стоит обратить внимание на seq scan, в котором участвуют все записи из таблицы.

Часто с медленными запросами нам помогают индексы, но их создают и сопровождают обычно разработчики.

Разное

После раздела SQL101 все равно остались вещи, связанные с эксплуатацией БД, о которых нельзя не сказать.

Во-первых, это реплики. Баз данных несколько, обычно их количество кратно числу ЦОДов. Нужны они для отказоустойчивости, увеличения быстродействия чтения. Сервис на Golang использует клиентскую балансировку, поддержанную в библиотеках платформы. Это код, способный обновлять знание о том, куда подключаться, используя etcd (сервис хранения конфигураций) и управляющий пулом соединений. Понимание того, что чтение и запись происходят на разных инстансах, помогает в отладке тестов: иногда только что созданные данные не видны при чтении. И дело не в транзакции.

Во-вторых, это такие средства масштабирования БД, как шардирование и партицирование. БД или отдельные таблицы бывают огромными, тогда одну таблицу разделяют на несколько по какому-то алгоритму — это партицирование (вертикальное шардирование). Если таблицы разнести и на разные инстансы, получится горизонтальное шардирование, или просто шардирование. На практике разносится не одна, а несколько таблиц, в совокупности называемые секцией. Алгоритм, по которому выбирается разбиение на партиции, всегда зависит от проекта и его данных, задача при этом одна — получить равномерную нагрузку на шардах. Тесты, работающие с БД, тоже должны знать этот алгоритм для понимания, откуда получать данные.

В конце отметим, что Postgres — не единственная БД, которую использует Ozon. В ряде наших проектов используется MSSQL, которые синтаксически отличается от Postgres. Хороший справочник с примерами тоже можно найти в блоге ozontech. Для сбора статистики мы активно используем Clickhouse.

В статье “One Size Fits All”: An Idea Whose Time Has Come and Gone хорошо раскрывается идея: под разные задачи нужны разные БД, тогда костылей будет меньше, а код — лучше.