Соединения. Вложенные запросы. Индексы. Транзакции. Views. функции

Distinct Limit

Получение уникальных значений Distinct

- Если из таблицы выбирать не все столбцы, то может происходить дублирование
- Пусть есть таблица заказов **orders**

id	orderDate	customerId
1	2012-01-01	1
2	2013-04-12	2
3	2012-03-05	2

- Мы хотим получить покупателей, у которых есть заказы
- Как это сделать?

Получение уникальных значений Distinct

id	orderDate	customerId
1	2012-01-01	1
2	2013-04-12	2
3	2012-03-05	2

- SELECT customerId FROM orders
- Что мы получим?

customerId
1
2
2

- А хотим так:

customerId
1
2

Получение уникальных значений Distinct

- SELECT DISTINCT customerId
  FROM orders;
- Слово DISTINCT выкидывает из результата полностью дублирующиеся строки

customerId
1
2

id	orderDate	customerId
1	2012-01-01	1
2	2013-04-12	2
3	2012-03-05	2

Порционность результатов Limit

- Не всегда нам нужны все результаты, которые удовлетворяют запросу
- Допустим, у нас есть новостной сайт и на главной странице мы показываем новости
- Как правило, на странице показывают не все новости, а только несколько последних
- Остальные грузятся только по требованию
- Это большая экономия по скорости поиска и по памяти

Limit

- Есть 2 варианта – если передать в LIMIT 1 аргумент или 2 аргумента

- С одним аргументом LIMIT выдает указанное количество строк от начала результата (или меньше, если в результате меньше строк)
- SELECT * FROM cities
  LIMIT 5;

- Так выдадутся 5 произвольных строк

Limit и сортировка

- SELECT * FROM cities
  LIMIT 5;

- Так выдадутся 5 произвольных строк
- По-хорошему, LIMIT следует всегда использовать совместно с сортировкой, иначе порядок не гарантируется, могут быть выбраны любые строки

- SELECT * FROM cities
  ORDER BY name
  LIMIT 5;

Limit с двумя аргументами

- Когда в LIMIT два аргумента, то первый означает сколько строк в результате надо пропустить от начала, а второй – сколько строк взять от этой позиции
- SELECT * FROM cities
  LIMIT 5, 10;
- Так выдадутся 10 произвольных строк, причем с номерами от 6 до 15
- Правильно всегда использовать сортировку
- SELECT * FROM cities
  ORDER BY name
  LIMIT 5, 10;

Задача «Distinct, limit»

- Из таблицы стран получить:
  - Названия континентов, на которых есть страны с площадью не менее 950000
  - Получить первые 5 стран с наибольшей площадью
  - Получить страны, которые занимают 6-10 места по площади включительно

Соединения таблиц

Соединения таблиц

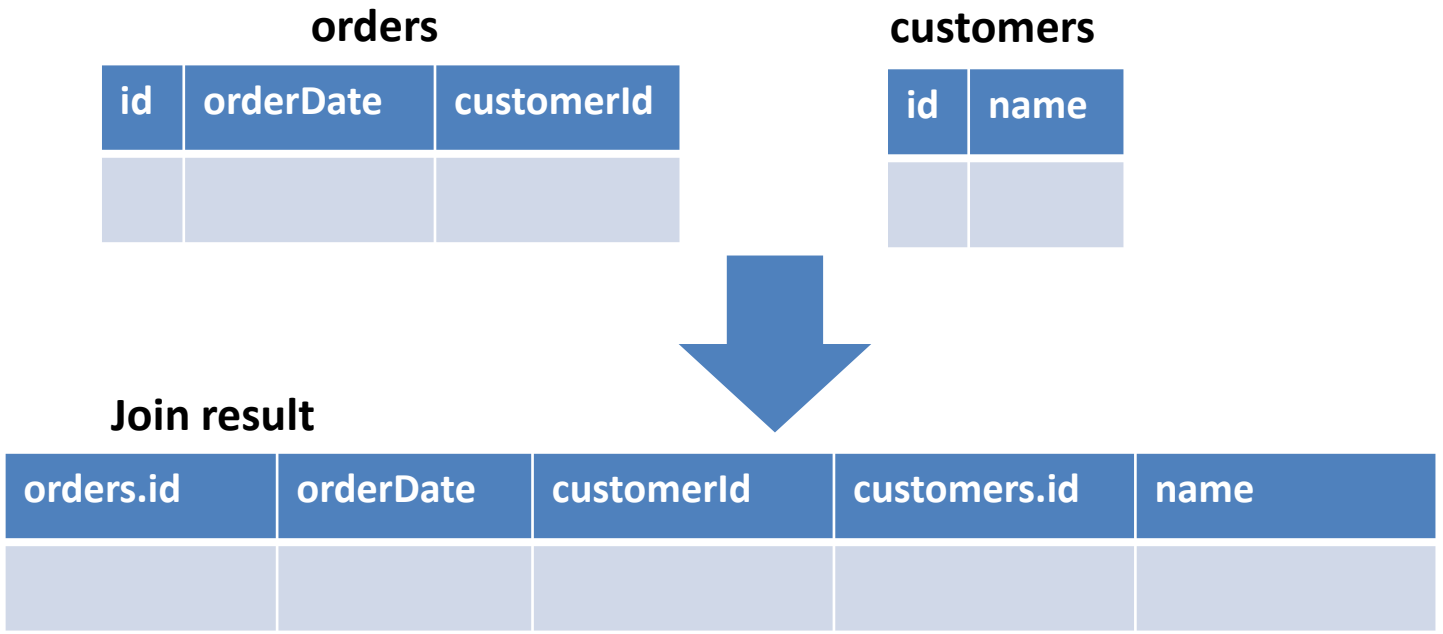
- Обычно запросы затрагивают не одну, а сразу несколько таблиц
- Например, надо найти покупателей, которые чаще всего покупали некоторый продукт
- Покупатели, продукты и заказы хранятся в разных таблицах, и, чтобы получить нужную информацию, придется задействовать несколько таблиц
- Это делается при помощи соединений таблиц (**join**)

Виды join’ов

- Есть несколько видов соединений:
  - **CROSS JOIN** – декартово произведение (все комбинации)
  - **INNER JOIN** – декартово произведение, но с фильтрацией по некоторому условию
  - **OUTER JOIN**
    - **LEFT JOIN**
    - **RIGHT JOIN**
    - **FULL JOIN**

JOIN’ы

- При JOIN’ах мы соединяем таблицы, и у нас в результате получается таблица, в которой есть столбцы из обеих таблиц
- При этом перед каждым именем столбца добавляется префикс – имя таблицы. Например, **orders.id**
- И можно просто выбрать нужные нам столбцы



CROSS JOIN

- Берутся все строки первой таблицы и все строки второй таблицы и соединяются между собой всеми возможными способами
- SELECT * FROM orders, customers
- Или другой синтаксис:
- SELECT * FROM orders
  CROSS JOIN customers

- Число получившихся строк = M * N, где M – число строк в первой таблице, N – число строк второй таблицы
- CROSS JOIN нужен очень редко, т.к. обычно надо комбинировать только нужные строки, а не все со всеми

CROSS JOIN

orders

id	orderDate	customerId
1	2012-01-01	1
2	2013-04-12	1
3	2012-03-05	2

customers

id	name
1	Иван
2	Петр

```
SELECT *
FROM orders,
customers;
```

Join result

orders.id	orderDate	customerId	customers.id	name
1	2012-01-01	1	1	Иван
1	2012-01-01	1	2	Петр
2	2013-04-12	1	1	Иван
2	2013-04-12	1	2	Петр
3	2012-03-05	2	1	Иван
3	2012-03-05	2	2	Петр

CROSS JOIN

- При использовании JOIN’ов обычно берут не все столбцы, а только нужные:
- SELECT orders.id, orders.orderDate, customers.name
  FROM orders, customers

orders

id	orderDate	customerId
1	2012-01-01	1
2	2013-04-12	1
3	2012-03-05	2

customers

id	name
1	Иван
2	Петр

result

id	orderDate	name
1	2012-01-01	Иван
1	2012-01-01	Петр
2	2013-04-12	Иван
2	2013-04-12	Петр
3	2012-03-05	Иван
3	2012-03-05	Петр

INNER JOIN

- Используется гораздо чаще, является привычным нам соединением
- SELECT orders.id, orders.orderDate, customers.name
  FROM orders
  INNER JOIN customers
  ON customers.id = orders.customerId

orders

id	orderDate	customerId
1	2012-01-01	1
2	2013-04-12	1
3	2012-03-05	2

customers

id	name
1	Иван
2	Петр

result

id	orderDate	name
1	2012-01-01	Иван
2	2013-04-12	Иван
3	2012-03-05	Петр

Строка одной таблицы соединяется не с каждой строкой из другой таблицы, а только с теми, где выполняется равенство

INNER JOIN

- По факту INNER JOIN – это CROSS JOIN с фильтрацией по условию – оставляем только строки, для которых выполняется условие из ON
- SELECT orders.id, orders.orderDate, customers.name
  FROM orders
  INNER JOIN customers
  ON customers.id = orders.customerId;

orders

id	orderDate	customerId
1	2012-01-01	1
2	2013-04-12	1
3	2012-03-05	2

customers

id	name
1	Иван
2	Петр

result

id	orderDate	name
1	2012-01-01	Иван
2	2013-04-12	Иван
3	2012-03-05	Петр

INNER JOIN

orders

id	orderDate	customerId
1	2012-01-01	1
2	2013-04-12	1
3	2012-03-05	2

id	name
1	Иван
2	Петр

customers

Результатом будут только выделенные строки

Join result

orders.id	orderDate	customerId	customers.id	name
1	**2012-01-01**	**1**	**1**	**Иван**
1	2012-01-01	1	2	Петр
2	**2013-04-12**	**1**	**1**	**Иван**
2	2013-04-12	1	2	Петр
3	2012-03-05	2	1	Иван
3	**2012-03-05**	**2**	**2**	**Петр**

OUTER JOIN

- OUTER JOIN - то же самое, что и INNER JOIN, но по-разному идет работа если в другой таблице не нашлось соответствие
- INNER JOIN откидывает строки, если им не нашлось соответствие в другой таблице

customers

id	name
1	Иван
2	Петр

orders

id	orderDate	customerId
1	2012-01-01	1
2	2013-04-12	1

Inner join result

name	id	orderDate
Иван	1	2012-01-01
Иван	2	2013-04-12

- В результате нет строки с покупателем Петр

LEFT JOIN

- В OUTER JOIN – если не нашлось соответствие, то строка в результат попадет, а все значения из этой таблицы будут NULL
- LEFT JOIN – берет все строки первой таблицы, ищет соответствие во второй, и, если что, ставит NULL
- SELECT customers.name, orders.id, orders.orderDate
  FROM customers
  LEFT JOIN orders
  ON customers.id = orders.customerId

customers

id	name
1	Иван
2	Петр

orders

id	orderDate	customerId
1	2012-01-01	1
2	2013-04-12	1

left join result

name	id	orderDate
Иван	1	2012-01-01
Иван	2	2013-04-12
Петр	**NULL**	**NULL**

RIGHT, FULL JOIN

- RIGHT JOIN – то же самое, что LEFT JOIN, только там наоборот берутся все записи из второй таблицы, а если не нашлось соответствие в левой, в полях левой таблицы ставится NULL
- FULL JOIN – берутся все записи всех таблиц, если не нашлось соответствие, то ставится NULL
- Это более редкие виды JOIN’ов

Пример RIGHT JOIN

- Возьмем пример про LEFT JOIN, просто поменяем порядок таблиц и вид JOIN'а
- ```
  SELECT customers.name, orders.id, orders.orderDate
  FROM orders
  RIGHT JOIN customers
      ON customers.id = orders.customerId
  ```

orders

id	orderDate	customerId
1	2012-01-01	1
2	2013-04-12	1

customers

id	name
1	Иван
2	Петр

Right join result

name	id	orderDate
Иван	1	2012-01-01
Иван	2	2013-04-12
Петр	**NULL**	**NULL**
  ```

Пример FULL JOIN

- SELECT a1, a2, b1, b2
  FROM a
  FULL JOIN b
    ON a1 = b1

a

a1	a2
1	a
2	b
10	c

b

b1	b2
1	aa
3	bb

Full join result

a1	a2	b1	b2
1	a	1	aa
2	b	NULL	NULL
10	c	NULL	NULL
NULL	NULL	3	bb

- Тут есть все строки каждой из таблиц. И если в одной из таблиц нет соответствия, в ее полях ставится NULL

Алиасы для таблиц

- При использовании join'ов таблицам тоже можно давать алиасы при помощи слова AS, это удобно для краткости
- ```
  SELECT o.id, o.orderDate, c.name
  FROM orders AS o
  INNER JOIN customers AS c
      ON c.id = o.customerId;
  ```
- Также как для полей, слово AS не обязательно, тут результат тот же самый:
- ```
  SELECT o.id, o.orderDate, c.name
  FROM orders o
  INNER JOIN customers c
      ON c.id = o.customerId;
  ```

Необязательность префикса для поля

- При обращении к полям не обязательно указывать имя или алиас таблицы
- Если поле есть только в одной из таблиц, то SQL сам поймет какое поле имелось в виду
- А вот если в нескольких таблицах есть поле с одинаковым именем, то тогда нужно обязательно указывать таблицу или ее алиас
- ```
  SELECT o.id, orderDate, name
  FROM orders AS o
  INNER JOIN customers AS c
      ON c.id = customerId;
  ```
- Рекомендую всегда ставить префикс, так понятнее, хоть и несколько длиннее
  ```

JOIN по нескольким таблицам

- Можно делать join по нескольким таблицам
-
```
SELECT m.name, c.name, d.name
FROM movies AS m
INNER JOIN categories AS c
    ON c.id = m.categoryId
INNER JOIN directors AS d
    ON d.id = m.directorId;
```

id	name	categoryId	directorId
1	Avatar	1	1
2	Star Wars	1	2

id	name
1	Adventure
2	Comedy

id	name
1	Cameron
2	Lucas

m.name	c.name	d.name
Avatar	Adventure	Cameron
Star Wars	Adventure	Lucas

Итого по JOIN’ам

- Самый используемый – INNER JOIN
- Иногда используют LEFT JOIN
- Остальные используются редко

Задача «Join»

- Попробовать CROSS JOIN между таблицами городов и стран
- При помощи INNER JOIN: вывести код страны, название страны и название города-столицы
- Сделайте запрос, который выводит имя города, его численность, а также код и название его страны
- Для каждого континента выведите количество городов из этого континента. Учтите, что в Антарктиде городов нет, но надо чтобы все равно вывелся 0
- Выведите количество официальных языков для каждой страны в порядке убывания количества этих языков

Задача «Left Join»

- Сделайте БД из 2 таблиц - таблица заказов и таблица клиентов
- У клиента есть имя, у заказа – дата, id клиента и сумма
- Заполните эти таблицы данными. И сделайте, чтобы у одного из клиентов не было заказов
- Посчитайте сколько каждый клиент сделал заказов. Причем для того, кто ничего не заказал, должно вывестись 0

Разбор задачи про LEFT JOIN, часть 1

- ```
  SELECT c.id, c.name, COUNT(o.id) AS ordersCount
  FROM customers AS c
  LEFT JOIN orders AS o
      ON c.id = o.customerId
  GROUP BY c.id, c.name;
  ```
- Чтобы понять сколько заказов у каждого клиента, нам надо сгруппировать по клиенту, и взять COUNT
- Тут нужен именно LEFT JOIN, а не INNER JOIN, т.к. мы хотим вывести всех клиентов. Если сделать INNER, то клиенты без заказов отбросятся, т.к. нет соответствия в таблице заказов
- В полях из правой таблицы, если нет соответствия, будет NULL. И помним, что COUNT выдает количество не NULL значений. Поэтому для клиентов без заказов COUNT будет 0
- Тут COUNT можно считать по любому не NULL полю правой таблицы, результат будет тот же
  ```

Разбор задачи про LEFT JOIN, часть 2

- ```
  SELECT c.id, c.name, COUNT(o.id) AS ordersCount
  FROM customers AS c
  LEFT JOIN orders AS o
      ON c.id = o.customerId
  GROUP BY c.id, c.name;
  ```
- Группировать надо именно по **c.id**, а не **o.customerId**. Потому что **o.customerId** будет NULL, если для клиента не нашлось заказов. Получится что все клиенты без заказов попадут в одну группу, и COUNT посчитается по ним общий, что неверно
- Тут группируем по обоим полям – **c.id**, **c.name**
- По идее, нам нужен только **c.name**, но могут быть клиенты с одинаковым именем, поэтому еще группируем и по **c.id**, чтобы такие клиенты учитывались отдельно
- И помним, что по стандарту, в SELECT можно просто так выводить только столбцы, которые указаны в GROUP BY
  ```

Вложенные запросы

Вложенные запросы (подзапросы)

- Запросы можно вкладывать друг в друга
- В некоторых случаях без этого не удастся решить задачу
- Например, для каждой страны нужно найти город с максимальной численностью населения

- ```
  SELECT countryCode, name, population
  FROM city AS c
  WHERE population = (SELECT MAX(population)
                      FROM city
                      WHERE countryCode = c.countryCode);
  ```
- Это пример, когда подзапрос выдает 1 значение. Такой подзапрос можно вставлять везде, где требуется одно значение
  ```

Вложенные запросы (подзапросы)

- Для каждой страны нужно найти город с максимальной численностью населения
- ```
  SELECT countryCode, name, population
  FROM city AS c
  WHERE population = (SELECT MAX(population)
                      FROM city
                      WHERE countryCode = c.countryCode);
  ```
- Тут мы задали связь между подзапросом и внешним запросом через алиас «**c**»
- Поэтому для каждой строки внешнего запроса подзапрос выполнится отдельно, используя **c.countryCode**
- Это работает достаточно медленно, но позволяет решать сложные задачи
  ```

Вложенные запросы

- Может быть подзапрос, который выдает набор однотипных данных
- Он может быть полезен если мы используем IN внутри WHERE

- ```
  SELECT * FROM city AS c
  WHERE countryCode IN (SELECT code
                        FROM country
                        WHERE continent = 'Asia');
  ```
- В этом случае подзапрос может быть заменен на JOIN с последующей фильтрацией, но так бывает не всегда
- Этот подзапрос работает быстро, т.к. нет связи с внешним запросом – его можно выполнять хоть отдельно
  ```

Вложенные запросы

- Есть идеи как заменить?
- ```
  SELECT * FROM city AS c
  WHERE countryCode IN (SELECT code
                        FROM country
                        WHERE continent = 'Asia');
  ```
- ```
  SELECT c.* FROM city AS c
  INNER JOIN country AS ct
     ON ct.code = c.countryCode
  WHERE ct.continent = 'Asia';
  ```

Вложенные запросы

- Может быть подзапрос, который выдает таблицу
- Его можно использовать в FROM

- ```
  SELECT * FROM (SELECT * FROM city
                 WHERE countryCode = 'RUS') AS c
  WHERE population >= 100000;
  ```

- В этом случае подзапрос может быть заменен на WHERE, но так бывает не всегда
  ```

Практика

- Для каждой страны найти город с минимальным количеством населения

Продвинутые вещи

Индексы

Индексы

- Допустим, в нашей программе мы часто делаем запрос к таблице заказов **orders** по полю **price**
- На больших данных этот запрос будет тормозить, потому что строки БД не упорядочены по полю **price**. СУБД придется просматривать всю таблицу целиком

- Чтобы ускорить поиск, используются **индексы**
- Индекс формируется из значений одного или нескольких столбцов, и позволяет быстрее искать по запросам по этим столбцам
- Индекс упорядочен по указанным столбцам и содержит указатели на строки таблицы (аналогии – бинарный поиск и предметный указатель в книге)

Индексы

- Для использования индексов пользователю ничего не требуется
- Нужно просто создать индекс (потом его можно удалить)
- На работу БД это никак не повлияет (кроме скорости работы), потому что это дополнительные данные – если они есть, то СУБД их использует. Если нет, то нет

- Добавление индекса ускоряет поиск, но замедляет вставку, удаление и изменение строк
- Потому что при этом кроме изменения самих строк, нужно менять и индекс, а это дополнительные затраты

Пример создания индекса

- Добавляем индекс к столбцу **create_date** таблицы **request**:
- CREATE INDEX ix_create_date
  ON request (create_date);

- Удалить индекс можно так:
- DROP INDEX ix_create_date
  ON request;

Индексы

- Для каждой СУБД есть средства профилирования, построения плана выполнения запросов, и оптимизаторы
- Оптимизатор сам может порекомендовать создать определенные индексы, чтобы ускорить нужные нам запросы

- Дополнительная ссылка про индексы:
- http://ruhighload.com/post/%D0%A0%D0%B0%D0%B1%D0%BE%D1%82%D0%B0+%D1%81+%D0%B8%D0%BD%D0%B4%D0%B5%D0%BA%D1%81%D0%B0%D0%BC%D0%B8+%D0%B2+MySQL

Views

View

- Допустим, у нас уже есть большая и сложная база с большим числом таблиц, сложными связями между ними, в таблицах много столбцов и т.д.
- И у нас есть некоторая задача, в которой используется только часть таблиц и часть столбцов
- Неудобно каждый раз делать какие-то хитрые запросы с фильтрацией, JOIN’ами и т.д.
- Но и данные дублировать не хочется
- Для этих целей можно использовать виртуальные таблицы (**view**)

View

- **View** – это просто некоторый запрос, который сохранен как объект БД
- То есть просто взяли запрос, который выдает таблицу, и дали ему имя
- И теперь к этой таблице можно обращаться из любых других запросов
- Принцип инкапсуляции в чистом виде

View

- Допустим, для нашей задачи нам надо только страны из двух регионов и надо знать названия столиц
- Сделаем для этой задачи удобную **view**
- USE world;
- CREATE VIEW AvailableCountries
  AS SELECT c.code, c.name AS countryName, ct.name AS cityName
  FROM country AS c
  INNER JOIN city AS ct
  ON ct.id = c.capital
  WHERE region IN (‘Southern Europe’, ‘South America’);

- SELECT * FROM world.AvailableCountries;

View

- По факту, для **view** никакой таблицы не создается, и когда мы используем ее в других запросах, то просто подставляется реализация **view**, как будто мы бы прописали ее сами внутри запроса

Транзакции

Транзакции

- Одним из важнейших достоинств БД является поддержка **транзакций**
- **Транзакция** – это последовательность операций с БД, которая представляет собой логическую единицу работы с данными
- Транзакция может быть выполнена либо целиком и успешно, либо не выполнена вообще

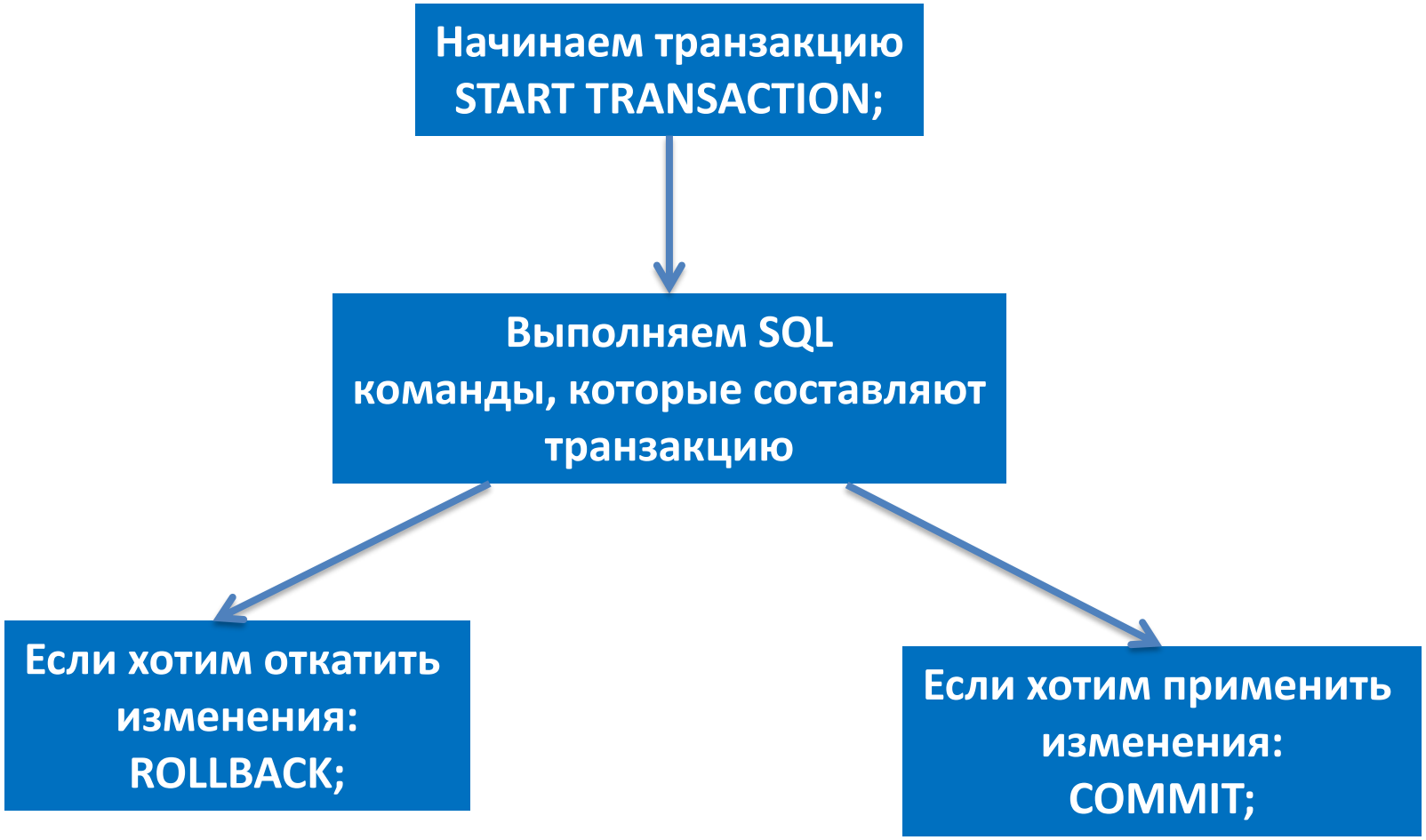
Пример транзакции

- Классический пример транзакции – перевод денег между двумя счетами
  1. Начать транзакцию
  2. Снять со счета 1 сумму 1000 рублей
  3. Увеличить баланс счета 2 на сумму 1000 рублей
  4. Завершить транзакцию
- Тут недопустимо, чтобы деньги снялись с одного счета, но не пришли на второй. Или чтобы пришли на второй счет, но не снялись с первого
- Если что-то идет не так, то все изменения откатываются, и деньги остаются на счете-отправителе

Свойства ACID

- Транзакция должна удовлетворять 4 свойствам, которые называют **ACID** (по первым буквам названий):
  - **Atomicity** (атомарность) – транзакция должна быть выполнена целиком, либо не выполнена целиком
  - **Consistency** (согласованность) – транзакция переводит данные из одного корректного состояния в другое корректное состояние
  - **Isolation** (изолированность) – параллельные транзакции не должны оказывать влияние на результат текущей транзакции
  - **Durability** (долговечность) – если транзакция завершена, то изменения должны оставаться сохраненными даже если будут низкоуровневые проблемы, например, с оборудованием

Работа с транзакциями в MySQL



Работа с транзакциями в MySQL

- START TRANSACTION;

```
UPDATE account
SET money = money – 1000
WHERE id = 1;

UPDATE account
SET money = money + 1000
WHERE id = 2;

COMMIT;
```

Программные средства
Функции, процедуры, триггеры

Функции, процедуры и триггеры

- В MySQL и других СУБД также есть и программные средства
- Можно создавать свои **функции**, **хранимые процедуры** и **триггеры**
- Они будут храниться в БД и их можно будет использовать

- **Функция** – некоторый код, который можно вызывать, например, из SELECT
- Функции не должны ничего менять, т.е. вызывать INSERT, UPDATE, DELETE

- В каждой СУБД синтаксис программных средств разный

Функции, процедуры и триггеры

- **Хранимая процедура** – то же самое что и функция, но имеет другой синтаксис вызова – через CALL или EXECUTE
- И процедуры могут что-то менять – вызывать INSERT, UPDATE, DELETE

- **Триггер** – код, который будет срабатывать при некотором событии. Например, при добавлении строки в таблицу
- Триггеры можно использовать для автозаполнения некоторых полей или проверки корректности данных, или оповещений о событиях
- http://www.zoonman.ru/library/mysql_sr_and_t.htm

Заполнение тестовыми данными

- Часто бывает нужно сгенерировать некоторые тестовые данные
- Для этого можно написать хранимую процедуру, в которой использовать **переменные** и **циклы**
- Допустим, у нас такая таблица, и надо вставить в нее 100000 тестовых строк
- ```
  CREATE TABLE your_table
  (
     id INT AUTO_INCREMENT PRIMARY KEY,
     your_field INT NOT NULL,
     name VARCHAR(50) NOT NULL
  );
  ```
  ```

Заполнение тестовыми данными

- Напишем хранимую процедуру для заполнения таблицы:
-
```
DELIMITER $$
CREATE PROCEDURE prepare_data()
BEGIN
   DECLARE i INT DEFAULT 1;
   WHILE i <= 100000 DO
      INSERT INTO your_table (your_field, name)
      VALUES (i, CONCAT('Строка ', i));
      SET i = i + 1;
   END WHILE;
END$$
DELIMITER ;
```

Можете сами доработать, чтобы заполнялось много таблиц, и не только одно поле

- Ее можно вызвать так: `CALL prepare_data();`
- Потом процедуру можно удалить: `DROP PROCEDURE prepare_data;`

Про DELIMETER

- В предыдущем примере перед созданием процедуры была вызвана следующая команда: DELIMITER $$
- А после создания: DELIMITER ;
- При этом код самой процедуры заканчивался так: END$$

- Как помним ; используется в качестве разделителя команд
- Но при создании процедуры будет ошибка - ; из кода процедуры будет считаться завершением объявления процедуры

- Чтобы все работало, надо временно заменить символ разделителя команд – для этого используется команда DELIMETER
- В ней мы указываем новую последовательность для разделителя - $$. Потом мы используем ее при завершении процедуры. А потом возвращаем обычный разделитель - ;

Задача «Тестовые данные»

- Напишите скрипт, который заполнит БД из задачи Shop категориями и товарами
- Пусть создастся 100 категорий и 5000 товаров
- Пусть товары привязываются к категориям случайным образом. И случайным образом пусть выставляется цена
- Используйте функцию RAND
- https://www.w3schools.com/sql/func_mysql_rand.asp
- https://dev.mysql.com/doc/refman/8.0/en/mathematical-functions.html#function_rand

- Задачу присылать вместе со скриптом по созданию самой БД из задачи Shop