

Что такое данные, информация, база данных? Что «под капотом» БД?

Данные – это информация в формализованном виде, т.е. пригодном для интерпретации, обработки, передачи. **Информация** – это структурированные данные.

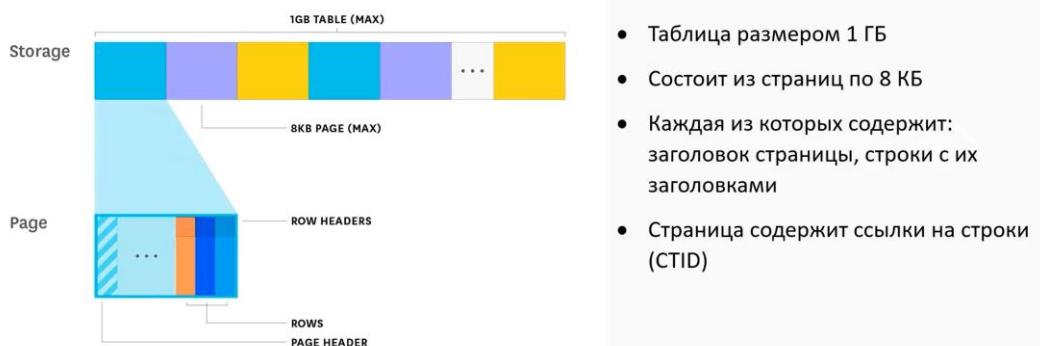
База данных — это набор данных, хранящиеся в структурированном виде.

Система управления базами данных СУБД — это совокупность языковых и программных средств, которая осуществляет доступ к данным, позволяет их создавать, менять и удалять, обеспечивает безопасность данных.



Под капотом СУБД: файлы, лежащие в каталогах.

И разные базы по-разному их интерпретируют.



Семь из десяти самых популярных СУБД — реляционные (связанные, relation/связь)
Это Oracle, MySQL, Microsoft SQL Server, PostgreSQL, IBM Db2, Microsoft Access, SQLite.

Есть также: **MongoDB** – документ-ориентированная СУБД; **Redis** - хранилище по типу «ключ-значение»; **Elasticsearch** - поисковой движок



DBeaver – бесплатный **многоплатформенный** инструмент работы с базами данных для разработчиков, администраторов баз данных, аналитиков и всех, кому необходимо работать с базами данных.

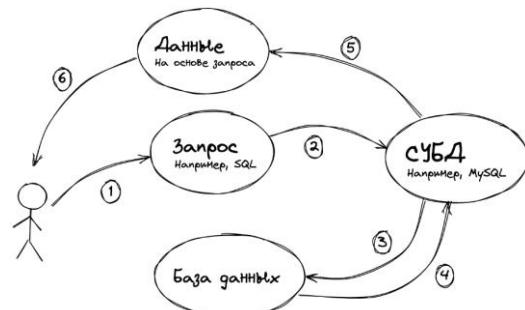
Не нужно каждый раз устанавливать софт определённой БД (Workbench, PGAdmin и т.д. =)

Поддерживает ВСЕ популярные базы данных: MySQL, PostgreSQL, SQLite, Oracle, DB2, SQL Server, Sybase, MS Access, Teradata, Firebird, Apache Hive, Phoenix, Presto и др. <https://dbeaver.io/>

Что такое SQL?

Язык программирования структурированных запросов **Structured Query Language, SQL**.

Внешние программы формируют запрос к СУБД на языке SQL



СУБД — это система, позволяющая создавать базы данных и манипулировать сведениями из них.

Что такое SQL-Сервер и как он работает?

Взаимодействие с СУБД происходит по клиент-серверному принципу.

Некая внешняя программа посыпает запрос в виде операторов и команд на языке SQL, СУБД его обрабатывает и высыпает ответ.

Для упрощения примем, что SQL Сервер = СУБД.

Виды СУБД

- 1) **Иерархические** (древовидная структура, например файл структура)
 - 2) **Сетевые** (каждого узла может быть более одного родителя)
 - 3) **Объектно-ориентированные** (данные организованы в виде классов, объектов (принципы ООП))
 - 4) **Реляционные** (данные организованы в виде таблиц) - самые распространённые
- Реляционными называются базы данных, в основе построения которых лежит реляционная модель.

Данные в реляционных структурах организованы в виде набора таблиц, называемых отношениями.

Каждая таблица состоит из столбцов и строк.

Каждая строка таблицы представляет собой набор связанных значений, относящихся к одному объекту/сущности.

Каждая строка в таблице может быть помечена уникальным идентификатором (первичным ключом), а строки из нескольких таблиц могут быть связаны с помощью внешних ключей.

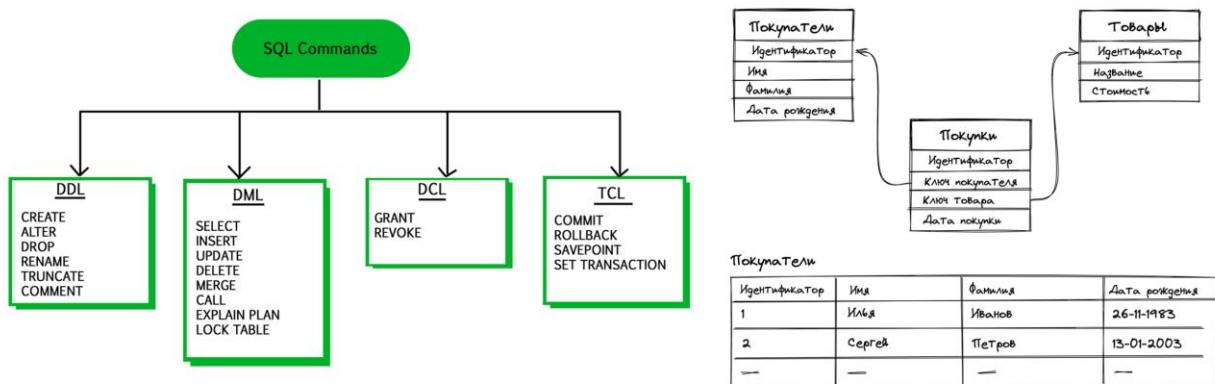
Особенности реляционных БД

- 1) Модель данных в реляционных БД определена заранее и является строго типизированной
- 2) Данные хранятся в таблицах, состоящих из столбцов и строк
- 3) На пересечении каждого столбца и строкки допускается только одно значение
- 4) Каждый столбец проименован и имеет определенный тип, которому следуют значения со всех строк столбца
- 5) Столбцы располагаются в определённом порядке, который определяется при создании таблицы
- 6) В таблице может не быть ни одной строчки, но обязательно должен быть хотя бы один столбец
- 7) Запросы к базе данных возвращают результат в виде таблиц

РЕЛЯЦИОННЫЕ БАЗЫ ДАННЫХ



Основные SQL операторы и пример реальная базы данных (связи между таблицами):



Язык запросов SQL

SQL — язык структурированных запросов (SQL, Structured Query Language), который используется в качестве эффективного способа сохранения данных, поиска их частей, обновления, извлечения и удаления из БД

Основные SQL операторы

- 1) **DDL, Data Definition Language** Создание структуры БД и ее объектов
- 2) **DML, Data Manipulation Language** Взаимодействие с данными: вставка, удаление, изменение и чтение;
- 3) **TCL, Transaction Control Language** Управление транзакциями;
- 4) **DCL, Data Control Language** Управление правами доступа к данным и структурам БД

SQL-запросы можно условно разделить на две группы:

1. Получение данных — к ним относится оператор SELECT
2. Изменение данных — к ним относятся операторы INSERT, UPDATE и DELETE

Для стандартизации работы с SQL-серверами взаимодействие с БД можно выполнять через единую точку — **JDBC (Java DataBase Connectivity)**. Реализация пакета `java.sql` для работы с СУБД
Производители всех популярных SQL-серверов выпускают для них драйверы JDBC.

Что такое **Query**? Это запрос.

Java Persistence Query Language, JPQL — платформенно-независимый объектно-ориентированный язык запросов, являющийся частью спецификации **JPA** или Java Persistence API. JPQL используется для написания запросов к сущностям, хранящимся в реальной базе данных.

Как передать в объект Query параметры?

HQL (Hibernate Query Language) запрос всегда начинается с получения объекта Query из Session вызовом метода `createQuery()`, в который передаётся текст запроса.

Какие бывают связи таблиц SQL? (будет еще про это в теме Hibernate)

Многие ко многим <code>@ManyToMany</code>	Каждому работнику соответствует одна и более должностей. Каждой должности соответствует один и более работников.
Один ко многим <code>@OneToMany</code>	Вариант: связь ОБЯЗАТЕЛЬНА (телефон принадлежит только одному пользователю). А пользователю могут принадлежать 1 и более телефонов (многие)
	Вариант: связь НЕ обязательна (на Земле живут все люди. Каждый человек живет на Земле. Но планета может существовать без людей)
Один к одному <code>@OneToOne</code>	Вариант: связь ОБЯЗАТЕЛЬНА (у загранпаспорта обязательно есть только один владелец). В этом случае, это обязательная связь)
	Вариант: связь НЕ обязательна (наличие загранпаспорта необязательно – его может и не быть у гражданина. Это необязательная связь)

Синтаксис, форматирование: **ctrl Alt + L**

Ключевые слова пишутся в **верхнем регистре**, остальные в нижнем, с **нижним подчёркиванием** в качестве разделителя. Точка с запятой в конце.

```
CREATE DATABASE company_repository;
CREATE SCHEMA company_storage;
```

1. Что такое DDL? Какие операции в него входят? Рассказать про них.

Data Definition Language, DDL – создание структуры базы данных и ее объектов.

Это краткое название языка определения данных, который имеет дело со схемами БД и описаниями того, как данные должны храниться в базе данных.

Операции	Описание	Запрос
CREATE	<p>Создание базы данных и ее объектов, таких как: таблица, индекс, представления, процедура хранения, функция и триггеры.</p> <p>Если нужно создать таблицу в пакете (SCHEMA) example:</p> <pre>CREATE TABLE example.employees (column1 datatype, column2 datatype,);</pre> <p>DATABASE (БД) → SCHEMA (пакет) → TABLE (таблица) <u>По умолчанию</u> таблица будет создана в <u>пакете public</u></p>	<pre>CREATE DATABASE IF NOT EXISTS; имя_базы_данных; CREATE SCHEMA example; CREATE TABLE table_name (column1 datatype, column2 datatype, column3 datatype, ...);</pre>
ALTER (альтернатива)	<p>Изменение структуры существующей базы данных:</p> <ul style="list-style-type: none"> • добавить, удалить столбец → • изменить тип данных столбца (ниже) <pre>ALTER TABLE table_name ADD column_name datatype; ALTER TABLE Customers ADD Email varchar(255); ALTER TABLE table_name DROP COLUMN column_name;</pre>	
DROP	Удаление таблицы из базы данных	ОСТОРОЖНО!!! Удалил таблицу целиком: <pre>DROP TABLE table_name;</pre>
TRUNCATE	Удаление <u>всех</u> записей из таблицы и мест, выделенных для этих записей, НО НЕ самой таблицы из БД.	<pre>TRUNCATE TABLE table_name;</pre>

2. Что такое DML? Какие операции в него входят? Рассказать про них.

Data Manipulation Language, DML – это операторы манипуляции данными.

Операции	Описание	Запрос
SELECT	<p>Выбирает данные, удовлетворяющие заданным условиям</p>	<pre>SELECT member_id, member_name, status FROM FamilyMembers;</pre> <p>Выведите поля (такие-то) из таблицы (такой-то).</p> <pre>SELECT * FROM Company;</pre> <p>(вывести все столбцы)</p> <pre>SELECT DISTINCT name FROM Passenger;</pre> <p>(только уникальные)</p>
INSERT	<p>Добавляет новые записи в таблицу.</p> <pre>CREATE TABLE company_storage.company (id INT, name VARCHAR(128), date DATE); INSERT INTO company(id, name, date) VALUES (1, 'Google', '2001-01-01');</pre>	<pre>INSERT INTO table_name (column1, column2, column3, ...) VALUES (value1, value2, value3, ...); INSERT INTO table_name VALUES (value1, value2, value3, ...);</pre> <p>В круглых скобках: 1) тип данных (колонки, розовый), после ключевого слова VALUES 2) указываем значения (зелёный цвет)</p>

UPDATE	Изменяет существующие данные	<pre>--Обновление UPDATE Goods SET ProductDescription = 'Товар NEW', Price = 400 WHERE Category = 1;</pre> <table border="1"> <thead> <tr> <th>ProductId</th><th>Category</th><th>ProductName</th><th>ProductDescription</th><th>Price</th></tr> </thead> <tbody> <tr> <td>1</td><td>1</td><td>Системный блок</td><td>Товар NEW</td><td>400,00</td></tr> <tr> <td>2</td><td>1</td><td>Монитор</td><td>Товар NEW</td><td>400,00</td></tr> <tr> <td>3</td><td>2</td><td>Смартфон</td><td>Товар</td><td>100,00</td></tr> </tbody> </table>	ProductId	Category	ProductName	ProductDescription	Price	1	1	Системный блок	Товар NEW	400,00	2	1	Монитор	Товар NEW	400,00	3	2	Смартфон	Товар	100,00
ProductId	Category	ProductName	ProductDescription	Price																		
1	1	Системный блок	Товар NEW	400,00																		
2	1	Монитор	Товар NEW	400,00																		
3	2	Смартфон	Товар	100,00																		
DELETE	Удаляет данные	<pre>DELETE FROM имя_таблицы [WHERE условие_отбора_записей];</pre> <pre>DELETE Reservations, Rooms FROM Reservations JOIN Rooms ON Reservations.room_id = Rooms.id WHERE Rooms.has_kitchen = false;</pre>																				

3. Что такое TCL? Какие операции в него входят? Рассказать про них.

Transaction Control Language, TCL – это операторы управления транзакциями.

Операции	Описание	Запрос
COMMIT	применяет транзакцию	<pre>-- Start a new transaction BEGIN TRANSACTION -- SQL Statements INSERT INTO Product VALUES(116, 'Headphone', 2000, 30) UPDATE Product SET Price = 450 WHERE Product_id = 113 -- Commit changes COMMIT TRANSACTION</pre>
ROLLBACK	откатывает все изменения, сделанные в контексте текущей транзакции	<pre>-- Start a new transaction BEGIN TRANSACTION -- SQL Statements UPDATE Product SET Price = 5000 WHERE Product_id = 114 DELETE FROM Product WHERE Product_id = 116 --Undo Changes ROLLBACK TRANSACTION</pre>
SAVEPOINT	разбивает транзакцию на логические точки сохранения, чтобы не откатывать всю транзакцию. правильный СИНТАКСИС: ROLLBACK TO <SAVEPOINT-NAME>	<pre>DROP TABLE student; -- IMPLICIT COMMIT (DROP is a DDL) CREATE TABLE Student(StudID NUMBER(6)); -- IMPLICIT COMMIT (CREATE TABLE is DDL) INSERT INTO Student VALUES (123); SAVEPOINT ABC; -- SAVEPOINT DELETE Student; SELECT * FROM STUDENT; -- still nothing. ROLLBACK TO ABC; -- rolling back TO SAVEPOINT ABC; SELECT * FROM STUDENT; -- STUDENT IS BACK!</pre> <p style="text-align: right;">WHAT WE WANT</p>

4. Что такое DCL? Какие операции в него входят? Рассказать про них.

Data Control Language, DCL (happens-before) – операторы определения доступа к данным.

Операции	Описание	Запрос
GRANT	предоставляет пользователю (группе) разрешения на определенные операции с объектом	<p>Предоставление разрешения SELECT на таблицу без использования фразы OBJECT</p> <pre>GRANT SELECT ON Person.Address TO RosaQdM; GO</pre> <p>Предоставление учетной записи домена разрешения SELECT на таблицу</p> <pre>GRANT SELECT ON Person.Address TO [Adventureworks2012\RosaQdM]; GO</pre>

REVOKE	отзывает ранее выданные разрешения	Следующий пример отменяет разрешение SELECT у пользователя RosaQdM для таблицы Person.Address в базе данных AdventureWorks2012. <code>USE AdventureWorks2012; REVOKE SELECT ON OBJECT::Person.Address FROM RosaQdM; GO</code>
DENY	задает запрет, имеющий приоритет над разрешением	<code>DENY permission [,...n] } ON SCHEMA :: schema_name TO database_principal [,...n] [CASCADE] [AS denying_principal]</code> Разрешение на список свойств поиска DENY (Transact-SQL) <code>DENY permission [,...n] ON SEARCH PROPERTY LIST :: search_property_list_name TO database_principal [,...n] [CASCADE] [AS denying_principal]</code>

5. Нюансы работы с NULL в SQL. Как проверить поле на NULL?

NULL – это специальное значение (псевдозначение), которое может быть записано в поле таблицы базы данных. NULL соответствует понятию «пустое поле», то есть «поле, не содержащее никакого значения». Введено для того, чтобы различать в полях БД пустые значения и отсутствующие значения.

NULL означает отсутствие, неизвестность информации. Значение NULL не является значением в полном смысле слова: по определению оно означает отсутствие значения и не принадлежит ни одному типу данных.

Поэтому NULL не равно ни логическому значению FALSE, ни пустой строке, ни нулю. Операторы сравнения служат для сравнения двух выражений, их результатом может являться ИСТИНА (1), ЛОЖЬ (0) и NULL.

Результат сравнения с NULL является NULL. Исключением является оператор эквивалентности.

Оператор эквивалентность аналогичен оператору равенства, с одним лишь исключением: в отличие от него, оператор эквивалентности вернет ИСТИНУ при сравнении `NULL <=> NULL`.

IS [NOT] NULL — позволяет узнать равно ли проверяемое значение NULL.

Для примера выведем всех членов семьи, у которых статус в семье не равен NULL:

```
SELECT * FROM FamilyMembers  
WHERE status IS NOT NULL;
```

Выражение **NULL != NULL** не будет истинным, ведь **нельзя однозначно сравнивать** одну неизвестность с другой. Кстати, ложным это выражение тоже не будет, потому что при вычислении условий Oracle не ограничивается состояниями ИСТИНА и ЛОЖЬ. Из-за наличия элемента неопределенности в виде NULLа существует еще одно состояние — НЕИЗВЕСТНО.

Попробуем выбрать все записи, которые входят в набор (1, 2, NULL):

```
select * from t where a in(1,2,null); -- вернёт [1,2]
```

```
-- Predicate Information:  
-- filter("A"=1 OR "A"=2 OR "A"=TO_NUMBER(NULL))
```

Как видим, строка с NULLом не выбралась. Произошло это из-за того, что вычисление предиката "A"=TO_NUMBER(NULL) вернуло состояние НЕИЗВЕСТНО. Для того, чтобы включить NULLы в результат запроса, придётся указать это явно:

```
select * from t where a in(1,2) or a is null; -- вернёт [1,2,NULL]
```

```
-- Predicate Information:  
-- filter("A" IS NULL OR "A"=1 OR "A"=2)
```

6. Виды Join'ов (виды связывания таблиц).

Часто приходится делать **выборку из нескольких таблиц**, каким-то образом объединяя их.

Общая структура многотабличного запроса →

```
SELECT поля_таблицы
FROM таблица_1
[[INNER] | [[LEFT | RIGHT | FULL][OUTER]] JOIN таблица_2
    ON условие_соединения
[[INNER] | [[LEFT | RIGHT | FULL][OUTER]] JOIN таблица_n
    ON условие_соединения]
```

В большинстве случаев условием соединения является равенство столбцов таблиц (таблица_1.поле = таблица_2.поле), однако точно так же можно использовать и другие операторы сравнения.

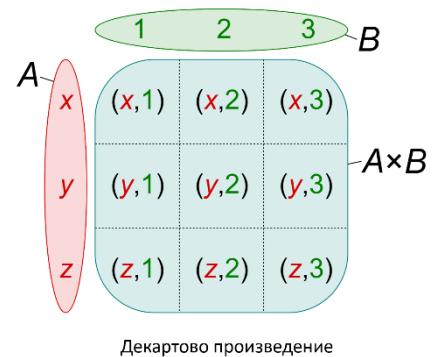
Соединение бывает внутренним INNER или внешним OUTER.

При этом внешнее соединение делится на левое (LEFT), правое (RIGHT) и полное (FULL).

INNER JOIN

По умолчанию, если не указаны какие-либо параметры, JOIN выполняется как INNER JOIN, то есть как внутреннее (перекрёстное) соединение таблиц.

Внутреннее соединение — это соединение двух таблиц, при котором каждая запись из первой таблицы соединяется с каждой записью второй таблицы, создавая тем самым все возможные комбинации записей обеих таблиц (декартово произведение) →



Например, объединим таблицы покупок **Payments** и членов семьи **FamilyMembers** таким образом, чтобы дополнить каждую покупку данными о том, кто её совершил.

Для того, чтобы решить поставленную задачу выполним запрос, который объединяет поля строки из одной таблицы с полями другой, если выполняется условие, что покупатель товара **family_member** совпадает с идентификатором члена семьи **member_id**:

```
SELECT *
FROM Payments
JOIN FamilyMembers ON family_member = member_id;
```

Данные в таблице Payments:

payment_id	date	family_member	good	amount	unit_price
1	2005-02-12 00:00:00	1	1	1	2000
2	2005-03-23 00:00:00	2	1	1	2100
3	2005-05-14 00:00:00	3	4	5	20
4	2005-07-22 00:00:00	4	5	1	350

Данные в таблице FamilyMembers:

member_id	status	member_name	birthday
1	father	Headley Quincey	1960-05-13 00:00:00
2	mother	Flavia Quincey	1963-02-16 00:00:00
3	son	Andie Quincey	1983-06-05 00:00:00
4	daughter	Lela Quincey	1985-06-07 00:00:00

В результате вы можете видеть, что каждая строка из таблицы `Payments` дополнилась данными о члене семьи, который совершил покупку. Обратите внимание на поля `family_member` и `member_id` — они одинаковы, что и было отражено в запросе.

payment_id	date	family_member	good	amount	unit_price	member_id
1	2005-02-12 00:00:00	1	1	1	2000	1
2	2005-03-23 00:00:00	2	1	1	2100	2
3	2005-05-14 00:00:00	3	4	5	20	3
4	2005-07-22 00:00:00	4	5	1	350	4
5	2005-07-26 00:00:00	4	7	2	150	4
6	2005-02-20 00:00:00	5	6	1	100	5

Для внутреннего соединения таблиц также можно использовать оператор `WHERE`.

Например, вышеприведённый запрос, написанный с помощью `INNER JOIN`, будет выглядеть так →

```
SELECT *
FROM Payments, FamilyMembers
WHERE family_member = member_id;
```

books			
id	book_title	author	year_of_writing
1	Eugene Onegin	Pushkin	1831
2	Kashtanka	Chekhov	1887
3	Three Musketeers	Duma	1844
4	Dar	Nabokov	1938
5	War and Peace	Tolstoy	1867
6	Captains daughter	Pushkin	1836
7	Luzhin Defense	Nabokov	1930
8	Three sisters	Chekhov	1900
9	Count of Monte Cristo	Duma	1844

employees			
id	first_name	last_name	salary
1	Alex	Pushkin	5000
2	Lev	Tolstoy	3000
3	Anton	Chekhov	2000
4	Alex	Pushkin	5000
5	Alex	Duma	1000
6	Lev	Tolstoy	3000
7	Anton	Chekhov	2000
8	Alex	Pushkin	5000
9	Alex	Duma	1000
10	Lev	Tolstoy	3000
11	Anton	Chekhov	2000
12	Vladimir	Nabokov	6000

```
SELECT DISTINCT b.book_title,
   b.author || ' ' || employees.first_name fio
FROM books AS b
INNER JOIN public.employees
  ON b.author = employees.last_name;
```

book_title	fio
Captains daughter	Pushkin Alex
Count of Monte Cristo	Duma Alex
Eugene Onegin	Pushkin Alex
Three Musketeers	Duma Alex
War and Peace	Tolstoy Lev
Luzhin Defense	Nabokov Vladimir
Kashtanka	Chekhov Anton
Three sisters	Chekhov Anton
Dar	Nabokov Vladimir

OUTER JOIN

Внешнее соединение может быть трёх типов: левое (LEFT), правое (RIGHT) и полное (FULL). По умолчанию оно является **полным**.

Главным отличием внешнего соединения от внутреннего является то, что оно **обязательно возвращает все строки** одной (LEFT, RIGHT) или двух таблиц (FULL).

Внешнее левое соединение (LEFT OUTER JOIN)

Соединение, которое возвращает все значения из левой таблицы, соединённые с соответствующими значениями из правой таблицы если они удовлетворяют условию соединения, или заменяет их на NULL в обратном случае.

Для примера получим из базы данных расписание звонков объединённых с соответствующими занятиями в расписании занятий:

Данные в таблице `Timepair` (расписание звонков):

id	start_pair	end_pair
1	08:30:00	09:15:00
2	09:20:00	10:05:00
3	10:15:00	11:00:00
4	11:05:00	11:50:00

Данные в таблице `Schedule` (расписание занятий):

id	date	class	number_pair	teacher	subject	classroom
1	2019-09-01	9	1	11	1	47
4	2019-09-02	9	1	4	3	13
7	2019-09-03	9	1	5	6	36
10	2019-09-04	9	1	9	9	39

```
SELECT *
FROM Timepair
LEFT JOIN Schedule ON Schedule.number_pair = Timepair.id;
```

В выборку попали все строки из левой таблицы, дополненные данными о занятиях. Примечательно, что в конце таблицы есть строки с полями, заполненными `NULL`. Это те строки, для которых не нашлось соответствующих занятий, однако они присутствуют в левой таблице, поэтому тоже были выведены.

timepair.id	start_pair	end_pair	schedule.id	date	class	number_pair	teacher
1	08:30:00	09:15:00	4	2019-09-02	9	1	4
1	08:30:00	09:15:00	7	2019-09-03	9	1	5
1	08:30:00	09:15:00	16	2019-08-30	9	1	2
1	08:30:00	09:15:00	10	2019-09-04	9	1	9
1	08:30:00	09:15:00	1	2019-09-01	9	1	11
1	08:30:00	09:15:00	13	2019-09-05	9	1	3

Внешнее правое соединение (RIGHT OUTER JOIN)

Соединение, которое возвращает все значения из правой таблицы, соединённые с соответствующими значениями из левой таблицы если они удовлетворяют условию соединения, или заменяет их на `NULL` в обратном случае.

```
SELECT DISTINCT book_title,
                author
FROM books
    LEFT JOIN public.employees
        ON books.author = employees.last_name;
```

book_title	author
1 Encyclopedia	<null>
2 Captains daughter	Pushkin
3 Eugene Onegin	Pushkin
4 War and Peace	Tolstoy
5 Count of Monte Cristo	Duma
6 Three Musketeers	Duma
7 Nose	Gogol
8 Kashtanka	Chekhov
9 Three sisters	Chekhov
10 Dar	Nabokov
11 Luzhin Defense	Nabokov

```
SELECT DISTINCT book_title,
                author
FROM books
    RIGHT JOIN public.employees
        ON books.author = employees.last_name;
```

book_title	author
1 Captains daughter	Pushkin
2 Eugene Onegin	Pushkin
3 War and Peace	Tolstoy
4 Count of Monte Cristo	Duma
5 Three Musketeers	Duma
6 Nose	Gogol
7 Kashtanka	Chekhov
8 Three sisters	Chekhov
9 Dar	Nabokov
10 Luzhin Defense	Nabokov

Выборка формируется по уникальным книгам. Слева в результирующий набор попала Энциклопедия без автора. Аналогичную выборку получим при использовании Full Join. А справа, т.к. используем RIGHT JOIN, то книга без автора в выборку не попадает.

Внешнее полное соединение (FULL OUTER JOIN)

Соединение, которое выполняет внутреннее соединение записей и дополняет их левым внешним соединением и правым внешним соединением.

Алгоритм работы полного соединения:

- Формируется таблица на основе внутреннего соединения (INNER JOIN).
- В таблицу добавляются значения не вошедшие в результат формирования из левой таблицы (LEFT OUTER JOIN).
- В таблицу добавляются значения не вошедшие в результат формирования из правой таблицы (RIGHT OUTER JOIN).

Соединение FULL JOIN реализовано НЕ во всех СУБД.

Например, в MySQL оно отсутствует, однако его можно очень просто эмулировать.

<https://sql-academy.org/ru/guide/multi-table-request-join>

Базовые запросы для разных вариантов объединения таблиц (запрос Join и схема):

<p>Получение всех данных из левой таблицы, соединённых с соответствующими данными из правой:</p> <pre>SELECT поля_таблиц FROM левая_таблица LEFT JOIN правая_таблица ON правая_таблица.ключ = левая_таблица.ключ</pre>	<p>Получение данных, относящихся только к левой таблице:</p> <pre>SELECT поля_таблиц FROM левая_таблица LEFT JOIN правая_таблица ON правая_таблица.ключ = левая_таблица.ключ WHERE правая_таблица.ключ IS NULL</pre>
<p>Получение всех данных из правой таблицы, соединённых с соответствующими данными из левой:</p> <pre>SELECT поля_таблиц FROM левая_таблица RIGHT JOIN правая_таблица ON правая_таблица.ключ = левая_таблица.ключ</pre>	<p>Получение данных, относящихся только к правой таблице:</p> <pre>SELECT поля_таблиц FROM левая_таблица RIGHT JOIN правая_таблица ON правая_таблица.ключ = левая_таблица.ключ WHERE левая_таблица.ключ IS NULL</pre>
<p>Получение данных, относящихся как к левой, так и к правой таблице:</p> <pre>SELECT поля_таблиц FROM левая_таблица INNER JOIN правая_таблица ON правая_таблица.ключ = левая_таблица.ключ</pre>	<p>Получение данных, относящихся к левой и правой таблицам одновременно (обратное INNER JOIN):</p> <pre>SELECT поля_таблиц FROM левая_таблица FULL OUTER JOIN правая_таблица ON правая_таблица.ключ = левая_таблица.ключ WHERE левая_таблица.ключ IS NULL OR правая_таблица.ключ IS NULL</pre>
<p>Получение всех данных, относящихся к левой и правой таблицам, а также их внутреннему соединению:</p> <pre>SELECT поля_таблиц FROM левая_таблица FULL OUTER JOIN правая_таблица ON правая_таблица.ключ = левая_таблица.ключ</pre>	

7. Что лучше использовать join или подзапросы? Почему?

Обычно лучше использовать JOIN, поскольку в большинстве случаев он более понятен и лучше оптимизируется СУБД (но 100% этого гарантировать нельзя).

Так же JOIN имеет заметное преимущество над подзапросами в случае, когда список выбора SELECT содержит столбцы более, чем из одной таблицы.

Подзапросы лучше использовать в случаях, когда нужно вычислять агрегатные значения и использовать их для сравнений во внешних запросах.

Пример подзапроса (вложенный запрос + математическая операция):

```
SELECT *,
       (select avg(salary) from employees) avg,
       (select max(salary) from employees) max
FROM public.employees;
```

	id	first_name	last_name	salary	avg	max
1	1	Alex	Pushkin	5000	2909.090909090909	5000
2	2	Lev	Tolstoy	3000	2909.090909090909	5000
3	4	Alex	Pushkin	5000	2909.090909090909	5000
4	5	Alex	Duma	1000	2909.090909090909	5000

```
select *,  
       (select max(salary) from employee) - salary diff  
  from employee;
```

1	6	Ivan	Sidorov	1	500	1500
2	7	Ivan	Ivanov	2	1000	1000
3	8	Arni	Paramonov	2	<null>	<null>
4	9	Petr	Petrov	3	2000	0

Alias (псевдоним) — это имя, назначенное источнику данных в запросе при использовании выражения в качестве источника данных или для упрощения ввода и прочтения инструкции SQL.

Это м.б. полезно, если имя источника данных слишком длинное или его трудно вводить.

```
select * from (select * from (VALUES (1, 'Google', '2001-01-01'),  
                                         (2, 'Apple', '2002-10-29'),  
                                         (3, 'Facebook', '1995-09-13')) t) y  
SELECT id, first_name, last_name  
FROM public.employees AS empl  
WHERE last_name LIKE '%о%';
```

Применение [\[править | править код\]](#)

Псевдонимы позволяют:

- задавать таблицам или столбцам другие имена:
- COLUMN ALIASES используются для упрощения чтения столбцов в результирующем наборе.
- TABLE ALIASES используются для сокращения SQL-кода, чтобы упростить его чтение или когда вы выполняете самостоятельное соединение (то есть: перечисление одной и той же таблицы более одного раза).
- дать имя полю, у которого до этого вообще не было имени. В результате будет поле с именем Num, которое содержит одну строку со значением 1.

```
Select 1 As Num
```

- использовать одну и ту же таблицу в операторе Select много раз.
- при использовании не указывать AS. Например, общепринятым является использование таких псевдонимов, как «рі» для таблиц, называемых «price_information».
- облегчить администрирование большого количества серверов, так как они задаются администратором SQL-сервера, и для каждого экземпляра может быть задано любое количество псевдонимов.

8. Что делает UNION?

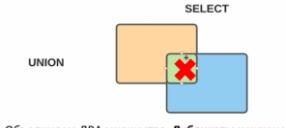
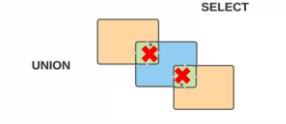
В языке SQL ключевое слово **UNION** применяется для **объединения результатов** двух SQL-запросов в единую таблицу, состоящую из схожих записей.

Оба запроса должны возвращать одинаковое число столбцов и совместимые типы данных в соответствующих столбцах.

UNION ALL выборка **будет включать дубликаты**.

Необходимо отметить, что UNION сам по себе не гарантирует порядок записей. Записи из второго запроса могут оказаться в начале, в конце или вообще перемешаться с записями из первого запроса.

В случаях, когда требуется определенный порядок, необходимо использовать ORDER BY.

С удалением дублей:	<code>SELECT * FROM имя_таблицы1 WHERE условие UNION SELECT * FROM имя_таблицы2 WHERE условие</code>	 Объединяя два множества, дубликаты исключены.
Без удаления дублей:	<code>SELECT * FROM имя_таблицы1 WHERE условие UNION ALL SELECT * FROM имя_таблицы2 WHERE условие</code>	 Объединяя два множества, дубликаты возможны.
Можно объединять не две таблицы, а три или более:	<code>SELECT * FROM имя_таблицы1 WHERE условие UNION SELECT * FROM имя_таблицы2 WHERE условие UNION SELECT * FROM имя_таблицы3 WHERE условие UNION SELECT * FROM имя_таблицы4 WHERE условие</code>	 Объединяя два, три и т.д. множеств, дублей нет.

9. Чем WHERE отличается от HAVING?

Ответа про то, что используются в разных частях запроса недостаточно...

Условный оператор WHERE применяют в ситуациях, когда требуется сделать **выборку по определенному условию** (часто используется).

```
SELECT поля_таблиц FROM список_таблиц
WHERE условия_на_ограничения_строк
[логический_оператор другое_условия_на_ограничения_строк];
```

Для этого в операторе **SELECT** существует параметр **WHERE**, после которого следует условие для ограничения строк. Если запись удовлетворяет этому условию, то попадает в результат, иначе отбрасывается.

Выведем все полёты, которые были совершены на самолёте «Boeing», но, при этом, вылет был не из Лондона:

```
SELECT * FROM Trip
WHERE plane = 'Boeing' AND NOT town_from = 'London';
```

Оператор **HAVING** используется для **фильтрации строк по значениям агрегатных функций**.

```
SELECT [константы, агрегатные_функции, поля_группировки]
FROM имя_таблицы
GROUP BY поля_группировки
HAVING условие_на_ограничение_строк_после_группировки
```

Общая структура запроса →

Отличие HAVING от WHERE

WHERE = ВЫБОРКА + ГРУППИРОВКА, то есть сначала выбираются записи по условию, а затем могут быть сгруппированы, отсортированы и т.д.

HAVING = ГРУППИРОВКА + ВЫБОРКА, т.е. сначала группируются записи, а затем выбираются по условию, при этом, в отличие от **WHERE**, в нём можно использовать **значения агрегатных функций**.

Пример использования:

выведем общую сумму, потраченную на покупки, для каждого члена семьи, где общая сумма покупки меньше, чем 5000 рублей:

```
SELECT family_member, SUM(unit_price * amount) AS sum
FROM Payments GROUP BY family_member
HAVING sum < 5000;
```

10. Что такое GROUP BY?

Иногда требуется узнать информацию не о самих объектах, а об определенных группах, которые они образуют. Для этого используется оператор **GROUP BY** и **агрегатные функции**.

Общая структура запроса →

```
SELECT [константы, агрегатные_функции, поля_группировки]
FROM имя_таблицы
GROUP BY поля_группировки;
```

Пример использования:

выведем общую сумму, потраченную на покупки, для каждого члена семьи, где общая сумма покупки **менее** 5000 руб.:

```
SELECT family_member, SUM(unit_price * amount) AS sum
FROM Payments GROUP BY family_member
HAVING sum < 5000;
```

При выполнении запроса происходит группировка по полю **family_member** и суммирование общей суммы, потраченной на покупки каждым из членов семьи.

Ниже представлен набор данных, находящихся в таблице Payments:

family_member	unit_price	amount
5	250	1
3	2200	1
2	66000	1
1	8	5
1	7	5
2	8	3

Как видно, образовались группы записей, объединённых одним **family_member**. После этого мы можем внутри каждой из этих групп применить формулу суммы, которая умножит количество товара на его стоимость, а потом просуммирует все получившиеся значения:

family_member	SUM(unit_price * amount)
1	2504
2	74194
3	3600
4	650
5	1060

Следует иметь в виду, что для **GROUP BY** все значения **NULL** трактуются как равные, т.е. при группировке по полю, содержащему **NULL**-значения, все такие строки попадут в одну группу.

11. Что такое ORDER BY?

При выполнении **SELECT** запроса, строки по умолчанию возвращаются в неопределенном порядке.

Фактический порядок строк в этом случае зависит от плана соединения и сканирования, а также от порядка расположения данных на диске, поэтому полагаться на него нельзя. Для упорядочивания записей используется конструкция **ORDER BY**.

Общая структура запроса **ORDER BY**

```
SELECT поля_таблиц FROM список_таблиц
ORDER BY столбец_1 [ASC | DESC][, столбец_n [ASC | DESC]];
```

В этой структуре запроса необязательные параметры указаны в квадратных скобках:

- **DESC** — сортировка по убыванию
- **ASC** (по умолчанию) — сортировка по возрастанию

```
SELECT DISTINCT id,
               first_name,
               last_name,
               salary
      FROM employees
     WHERE salary > 1000
    ORDER BY salary ASC
;
```

DESCending переводится, как: нисходящий, убывающий, падающий.

ASCending переводится, как: восходящий, поднимающийся возрастающий.

Сортировка по нескольким столбцам

Для сортировки результатов по двум или более столбцам их следует указывать через запятую.

```
SELECT поля_таблиц FROM список_таблиц
ORDER BY столбец_1 [ASC | DESC], столбец_2 [ASC | DESC];
```

Данные будут сортироваться по первому столбцу, но, в случае если попадаются несколько записей с совпадающими значениями в первом столбце, то они сортируются по второму столбцу. Количество столбцов, по которым можно отсортировать, не ограничено.

Правило сортировки применяется только к тому столбцу, за которым оно следует →

ORDER BY столбец_1, столбец_2 DESC

не то же самое, что

ORDER BY столбец_1 DESC, столбец_2 DESC

Примеры использования:

Выведем названия авиакомпаний в алфавитном порядке из таблицы Company. Сортировка строковых данных осуществляется в лексикографическом (алфавитном) порядке.

```
SELECT name FROM Company
ORDER BY name;
```

Выведем всю информацию о полетах, отсортированную по времени вылета самолета в порядке возрастания и по времени прилета в аэропорт в порядке убывания, из таблицы Trip.

```
SELECT * FROM Trip
ORDER BY time_out, time_in DESC;
```

В данном примере в начале отсортируется информация по времени вылета. Затем там, где время вылета совпадает, отсортируется по времени прилёта.

```
✓ SELECT DISTINCT id,
                  first_name,
                  last_name,
                  salary
            FROM employees
        ORDER BY salary
       LIMIT 5
;
```

	id	first_name	last_name	salary
1	5	Alex	Duma	1000
2	9	Alex	Duma	1000
3	3	Anton	Chekhov	2000
4	7	Anton	Chekhov	2000
5	11	Anton	Chekhov	2000

```
SELECT eml.first_name, air.email
  FROM test_db.public.employees AS eml,
       airline_db.public.passengers AS air;
```

← выборка по нескольким базам данных

12. Что такое DISTINCT?

DISTINCT используется для исключения повторяющихся строк из результата.

Иногда возникают ситуации, в которых нужно получить **только уникальные записи**. Для этого вы можете использовать DISTINCT.

Например, выведем список городов без повторений, в которые летали самолеты →

```
SELECT DISTINCT town_to FROM Trip;
```

Справа применяем ключевое слово DISTINCT, получаем выборку по уникальным именам.

```
insert into public.employees(first_name, last_name, salary)
values ('Alex', 'Pushkin', 5000),
       ('Alex', 'Duma', 1000),
       ('Lev', 'Tolstoy', 3000),
       ('Anton', 'Chekhov', 2000)
;
```

	id	first_name	last_name	salary
1	Alex	Pushkin	5000	
2	Lev	Tolstoy	3000	
3	Anton	Chekhov	2000	
4	Alex	Pushkin	5000	
5	Alex	Duma	1000	
6	Lev	Tolstoy	3000	
7	Anton	Chekhov	2000	

```
SELECT DISTINCT
    first_name
FROM employees;
```

	first_name
1	Anton
2	Lev
3	Alex

12. Что такое LIMIT?

Устанавливаем ограничение на максимальное количество записей, которые хотим получить в результирующем наборе. Например, вывести уникальные имена, но не более двух →

```
SELECT DISTINCT
    first_name
FROM employees
LIMIT 2;
```

	first_name
1	Anton
2	Lev

Вывести все уникальные фамилии из списка.

Пропустить две (OFFSET 2), оставить в результате две записи (LIMIT 2).

Первые две уникальные записи: Пушкин и Дюма пропущены, Чехов и Толстой остаются.

```
SELECT DISTINCT
    last_name
FROM employees
;
```

	last_name
1	Pushkin
2	Duma
3	Chekhov
4	Tolstoy

```
SELECT DISTINCT
    last_name
FROM employees
LIMIT 2
OFFSET 2
;
```

	last_name
1	Chekhov
2	Tolstoy

Важно!

Когда мы делаем выборку SELECT FROM, то порядок НЕ гарантирован и может меняться в зависимости от вставки или индекса. Поэтому **сначала нужно сделать сортировку** (например, по идентификатору), а потом добавлять ограничение LIMIT и OFFSET.

Для сортировки используем ORDER BY (см. выше).

13. Что такое EXISTS?

EXISTS берет подзапрос, как аргумент, и оценивает его как TRUE, если подзапрос возвращает какие-либо записи и FALSE, если нет.

14. Расскажите про операторы IN, BETWEEN, LIKE.

Ключевое слово **LIKE** используем, когда нужно сделать выборку по полю (тут по имени).
 Можем указать любое поле и даже только одну букву → **WHERE first_name LIKE 'A%**
 LIKE чувствителен к регистру ('alex' в выборку не попадёт).

<pre>SELECT DISTINCT id, first_name, last_name, salary FROM employees WHERE first_name LIKE 'Alex' ORDER BY salary ASC ;</pre>	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: center;">id</th> <th style="text-align: center;">first_name</th> <th style="text-align: center;">last_name</th> <th style="text-align: center;">salary</th> </tr> </thead> <tbody> <tr><td style="text-align: center;">1</td><td style="text-align: center;">5</td><td style="text-align: center;">Alex</td><td style="text-align: center;">Duma</td></tr> <tr><td style="text-align: center;">2</td><td style="text-align: center;">9</td><td style="text-align: center;">Alex</td><td style="text-align: center;">Duma</td></tr> <tr><td style="text-align: center;">3</td><td style="text-align: center;">1</td><td style="text-align: center;">Alex</td><td style="text-align: center;">Pushkin</td></tr> <tr><td style="text-align: center;">4</td><td style="text-align: center;">4</td><td style="text-align: center;">Alex</td><td style="text-align: center;">Pushkin</td></tr> <tr><td style="text-align: center;">5</td><td style="text-align: center;">8</td><td style="text-align: center;">Alex</td><td style="text-align: center;">Pushkin</td></tr> </tbody> </table>	id	first_name	last_name	salary	1	5	Alex	Duma	2	9	Alex	Duma	3	1	Alex	Pushkin	4	4	Alex	Pushkin	5	8	Alex	Pushkin
id	first_name	last_name	salary																						
1	5	Alex	Duma																						
2	9	Alex	Duma																						
3	1	Alex	Pushkin																						
4	4	Alex	Pushkin																						
5	8	Alex	Pushkin																						

Если в результирующий набор должны попасть данные в заданном диапазоне (например, зарплата сотрудников), то используем ключевое слово **BETWEEN** (удобен также для дат).

<pre>SELECT DISTINCT id, first_name, last_name, salary FROM employees WHERE salary BETWEEN 2000 AND 3000 ORDER BY salary ASC ;</pre>	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: center;">id</th> <th style="text-align: center;">first_name</th> <th style="text-align: center;">last_name</th> <th style="text-align: center;">salary</th> </tr> </thead> <tbody> <tr><td style="text-align: center;">1</td><td style="text-align: center;">3</td><td style="text-align: center;">Anton</td><td style="text-align: center;">Chekhov</td></tr> <tr><td style="text-align: center;">2</td><td style="text-align: center;">7</td><td style="text-align: center;">Anton</td><td style="text-align: center;">Chekhov</td></tr> <tr><td style="text-align: center;">3</td><td style="text-align: center;">11</td><td style="text-align: center;">Anton</td><td style="text-align: center;">Chekhov</td></tr> <tr><td style="text-align: center;">4</td><td style="text-align: center;">2</td><td style="text-align: center;">Lev</td><td style="text-align: center;">Tolstoy</td></tr> <tr><td style="text-align: center;">5</td><td style="text-align: center;">6</td><td style="text-align: center;">Lev</td><td style="text-align: center;">Tolstoy</td></tr> <tr><td style="text-align: center;">6</td><td style="text-align: center;">10</td><td style="text-align: center;">Lev</td><td style="text-align: center;">Tolstoy</td></tr> </tbody> </table>	id	first_name	last_name	salary	1	3	Anton	Chekhov	2	7	Anton	Chekhov	3	11	Anton	Chekhov	4	2	Lev	Tolstoy	5	6	Lev	Tolstoy	6	10	Lev	Tolstoy
id	first_name	last_name	salary																										
1	3	Anton	Chekhov																										
2	7	Anton	Chekhov																										
3	11	Anton	Chekhov																										
4	2	Lev	Tolstoy																										
5	6	Lev	Tolstoy																										
6	10	Lev	Tolstoy																										

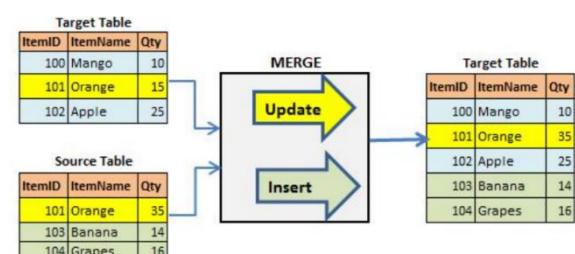
Если в выборку должны попасть результаты с конкретными данными, то используем **IN**:

<pre>SELECT DISTINCT id, first_name, last_name, salary FROM employees WHERE salary IN (1000, 5000) ORDER BY salary ASC ;</pre>	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: center;">id</th> <th style="text-align: center;">first_name</th> <th style="text-align: center;">last_name</th> <th style="text-align: center;">salary</th> </tr> </thead> <tbody> <tr><td style="text-align: center;">1</td><td style="text-align: center;">5</td><td style="text-align: center;">Alex</td><td style="text-align: center;">Duma</td></tr> <tr><td style="text-align: center;">2</td><td style="text-align: center;">9</td><td style="text-align: center;">Alex</td><td style="text-align: center;">Duma</td></tr> <tr><td style="text-align: center;">3</td><td style="text-align: center;">1</td><td style="text-align: center;">Alex</td><td style="text-align: center;">Pushkin</td></tr> <tr><td style="text-align: center;">4</td><td style="text-align: center;">4</td><td style="text-align: center;">Alex</td><td style="text-align: center;">Pushkin</td></tr> <tr><td style="text-align: center;">5</td><td style="text-align: center;">8</td><td style="text-align: center;">Alex</td><td style="text-align: center;">Pushkin</td></tr> </tbody> </table>	id	first_name	last_name	salary	1	5	Alex	Duma	2	9	Alex	Duma	3	1	Alex	Pushkin	4	4	Alex	Pushkin	5	8	Alex	Pushkin
id	first_name	last_name	salary																						
1	5	Alex	Duma																						
2	9	Alex	Duma																						
3	1	Alex	Pushkin																						
4	4	Alex	Pushkin																						
5	8	Alex	Pushkin																						

15. Что делает оператор MERGE? Какие у него есть ограничения?

MERGE позволяет осуществить **СЛИЯНИЕ** данных одной таблицы с данными другой таблицы.

При слиянии таблиц **проверяется условие**, и если оно истинно, то выполняется UPDATE, а если нет - INSERT. При этом изменять поля таблицы в секции UPDATE, по которым идет связывание двух таблиц, нельзя.



```

MERGE dbo.TestTable AS T_Base --Целевая таблица
USING dbo.TestTableDop AS T_Source --Таблица источник
ON (T_Base.ProductId = T_Source.ProductId) --Условие объединения
WHEN MATCHED THEN --Если истина (UPDATE)
    UPDATE SET ProductName = T_Source.ProductName, Summa = T_Source.Summa
WHEN NOT MATCHED THEN --Если НЕ истина (INSERT)
    INSERT (ProductId, ProductName, Summa)
        VALUES (T_Source.ProductId, T_Source.ProductName, T_Source.Summa)
--Посмотрим, что мы сделали
OUTPUT $action AS [Операция], Inserted.ProductId,
        Inserted.ProductName AS ProductNameNEW,
        Inserted.Summa AS SummaNEW,
        Deleted.ProductName AS ProductNameOLD,
        Deleted.Summa AS SummaOLD; --Не забываем про точку с запятой
--Итоговый результат
SELECT * FROM dbo.TestTable
SELECT * FROM dbo.TestTableDop

```

Результаты

Операция	ProductId	ProductNameNEW	SummaNEW	ProductNameOLD	SummaOLD
1	1	Компьютер	500,00	Компьютер	0,00
2	2	Принтер	300,00	Принтер	0,00
3	4	Монитор	400,00	NULL	NULL

Product

ProductId	ProductName	Summa
1	Компьютер	500,00
2	Принтер	300,00
3	Монитор	0,00
4	Монитор	400,00

Product

ProductId	ProductName	Summa
1	Компьютер	500,00
2	Принтер	300,00
3	Монитор	400,00

16. Какие агрегатные/агрегирующие функции вы знаете?

Агрегатная функция вычисляет единственное значение (получаем ОДНУ строку), обрабатывая множество строк. Агрегатные функции (иногда называют агрегирующие) есть во всех СУБД.

Агрегатные функции, вычисляющие по столбцу для набора строк:

- sum (сумму)
- avg (среднее)
- max (максимум)
- min (минимум)
- count (количество), есть также вариант со звёздочкой: `SELECT count(*) FROM employees;`
- concat (конкатенация, например, объединение имени и фамилии в одном столбце)

Не агрегатные математические функции (их много) – применяют действие ко всем значениям в столбце:

- upper/lower (перевод в верхний/нижний регистр)
- now (текущая дата и т.д.)

SELECT sum(salary)
FROM employees
;

Output sum(salary):numeric

1	32000
---	-------

SELECT avg(salary)
FROM employees
;

Output avg(salary):numeric

1	2909.0909090909090909
---	-----------------------

SELECT max(salary)
FROM employees
;

Output max(salary):integer

1	5000
---	------

SELECT min(salary)
FROM employees
;

Output min(salary):integer

1	1000
---	------

-- Количество строк
SELECT count(salary)
FROM employees
;

Output count(salary):bigint

1	11
---	----

SELECT upper(first_name)
FROM employees;

Output upper(first_name):text

1	ALEX
2	LEV
3	ALEX
4	ALEX
5	LEV

SELECT lower(first_name)
FROM employees;

Output lower(first_name):text

1	alex
2	lev
3	alex
4	alex
5	lev

SELECT upper(first_name),
concat(first_name, ' ', last_name) fio
FROM employees;

-- Функция, аналогичная concat (конкатенация)
SELECT upper(first_name),
first_name || ' ' || last_name fio,
now()
FROM employees;

Output Result 14

1	ALEX	Alex Pushkin	2022-12-05 06:42:18.700015 +00:00
2	LEV	Lev Tolstoy	2022-12-05 06:42:18.700015 +00:00
3	ALEX	Alex Pushkin	2022-12-05 06:42:18.700015 +00:00
4	ALEX	Alex Duma	2022-12-05 06:42:18.700015 +00:00
5	LEV	Lev Tolstoy	2022-12-05 06:42:18.700015 +00:00

SELECT last_name, avg(salary)
FROM public.employees GROUP BY last_name;

Output last_name avg

1	Pushkin	5000
2	Duma	1000
3	Nabokov	6000
4	Chekhov	2000
5	Tolstoy	3000

Важно понимать, как соотносятся агрегатные функции и SQL-предложения WHERE и HAVING.

Основное отличие WHERE от HAVING заключается в том, что WHERE сначала выбирает строки, а затем группирует их и вычисляет агрегатные функции (таким образом, она отбирает строки для вычисления агрегатов), тогда как HAVING отбирает строки групп после группировки и вычисления агрегатных функций. Следовательно...

WHERE НЕ должно содержать агрегатных функций; не имеет смысла использовать агрегатные функции для определения строк для вычисления агрегатных функций.

HAVING всегда СОДЕРЖИТ агрегатные функции!

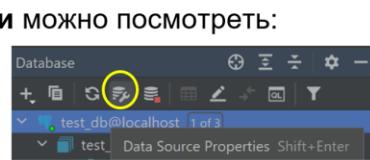
Строго говоря, вы можете написать предложение HAVING, не используя агрегаты, но это редко бывает полезно. То же самое условие может работать более эффективно на стадии WHERE.)

Все доступные агрегатные функции можно посмотреть:

Database, иконка с гаечным ключом

Data Source Properties.

Далее pg_catalog (для постгреса),
папка routines.



pg_catalog
tables 62
views 67
routines 2867
F abbrev (cidr): text
F abbrev (inet): text
F abs (bigint): bigint
F abs (double precision): double
F abs (integer): integer
F abs (numeric): numeric

- * smallint * bigint → bigint
- * smallint * integer → integer
- * smallint * money → money

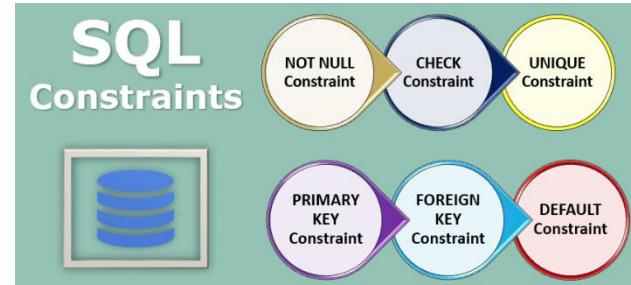
Математические операции
и ожидаемый результат в папке operators

17. Что такое ограничения (constraints)? Какие вы знаете?

Ограничения SQL — это правила, применяемые к столбцам данных таблицы.

Они используются, чтобы ограничить типы данных, которые могут храниться в таблице. Ограничения могут применяться либо на уровне столбцов, либо на уровне таблицы.

Это обеспечивает точность и надежность данных в базе.



NOT NULL – возможность вставки пустых значений в колонку таблицы.

CHECK – ограничение на диапазон значений (год для даты, например)

UNIQUE – значение в этой колонке должно быть уникальным.

PRIMARY KEY (pkey) колонка содержит уникальное NOT NULL значение (часто это ID).

Может быть только одним в таблице! В отличие от UNIQUE, например (ограничение для столбца, но не для таблицы).

FOREIGN KEY (fkey) защищает от действий, которые могут нарушить связи между таблицами. FOREIGN KEY в одной таблице указывает на PRIMARY KEY в другой.

Поэтому данное ограничение нацелено на то, чтобы не было записей FOREIGN KEY, которым не отвечают записи PRIMARY KEY.

```
salary integer
employee_pkey (id)
employee_first_name_last_name_key (first_name, last_name)
employee_company_id_fkey (company_id) → company (id)
```

Есть также неявные ограничения, например по количеству символов для VARCHAR (аналог строки, последовательность символов в одинарных кавычках). Указывается в круглых скобках. Один символ char = 1 байт памяти, соответственно 128 – это и количество символов, и количество памяти.

```

CREATE TABLE company_storage.company
(
    id INT PRIMARY KEY ,
    name VARCHAR(128) UNIQUE NOT NULL ,
    date DATE NOT NULL CHECK (date > '1995-01-01' AND date < '2020-01-01')
    -- NOT NULL
    -- UNIQUE
    -- CHECK
    -- PRIMARY KEY == UNIQUE NOT NULL!
);

```

При попытке «обойти ограничение» (указан год, меньше, чем 1995), вылетает ошибка:

```

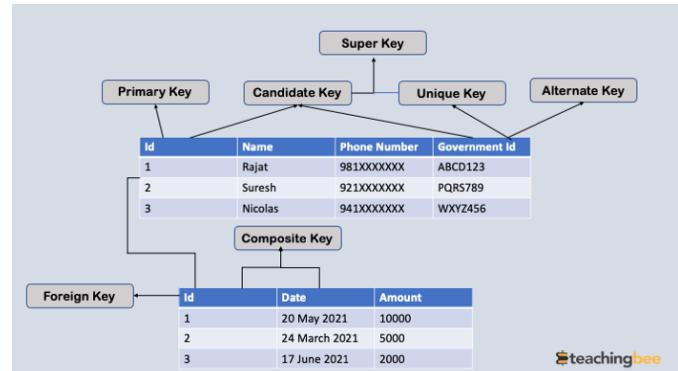
21 ① INSERT INTO company(id, name, date)
22   VALUES (1, 'Google', '2001-01-01'),
23           (2, 'Apple', '2002-10-29'),
24           (3, 'Facebook', '1993-09-13');
25
26   -- DROP TABLE public.company;
27
[23514] ERROR: new row for relation "company" violates check constraint
"company_date_check"
Detail: Failing row contains (3, Facebook, 1993-09-13).

```

18. Что такое суррогатные ключи?

Суррогатный ключ — понятие теории реляционных баз данных.

Это дополнительное служебное поле, добавленное к уже имеющимся информационным полям таблицы, единственное предназначение которого — служить первичным ключом Primary Key.



Дайте определение терминам «простой», «составной» (composite), «потенциальный» (candidate) и «альтернативный» (alternate) ключ.

Простой ключ состоит из одного атрибута (поля). Составной - из двух и более.

Потенциальный ключ - простой или составной ключ, который уникально идентифицирует каждую запись набора данных. При этом потенциальный ключ должен обладать критерием неизбыточности: при удалении любого из полей набор полей перестает уникально идентифицировать запись.

Из множества всех потенциальных ключей набора данных выбирают первичный ключ, все остальные ключи называют альтернативными.

Что такое «первичный ключ» (primary key)? Каковы критерии его выбора?

Первичный ключ (primary key) в реляционной модели данных один из потенциальных ключей отношения, выбранный в качестве основного ключа (ключа по умолчанию).

Если в отношении имеется единственный потенциальный ключ, он является и первичным ключом. Если потенциальных ключей несколько, один из них выбирается в качестве первичного, а другие называются «альтернативными».

В качестве первичного обычно выбирается тот из потенциальных ключей, который наиболее удобен. Поэтому в качестве первичного ключа, как правило, выбирают тот, который имеет наименьший размер (физического хранения) и/или включает наименьшее количество атрибутов. Другой критерий выбора первичного ключа — сохранение его уникальности со временем. Поэтому в качестве первичного ключа стараются выбирать такой потенциальный ключ, который с наибольшей вероятностью никогда не утратит уникальность.

Что такое «внешний ключ» (foreign key)?

Внешний ключ (foreign key) — подмножество атрибутов некоторого отношения А, значения которых должны совпадать со значениями некоторого потенциального ключа некоторого отношения В. Виды отношений таблиц и моделирование процессов UML.

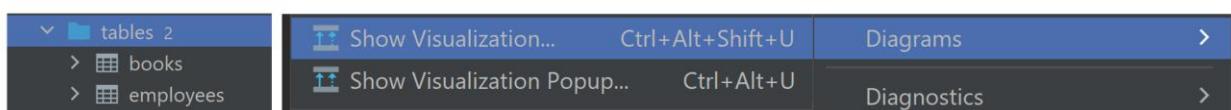
Один ко многим		
Employee		
1	1 Ivan	Ivanov
2	2 Svetlana	Svetikova
3	3 Petr	Petrov

Один к одному		
Employee		
1	1 Ivan	Ivanov
2	2 Svetlana	Svetikova
3	3 Petr	Petrov

Многие ко многим		
Employee		
1	1 Ivan	Ivanov
2	2 Svetlana	Svetikova
3	3 Petr	Petrov



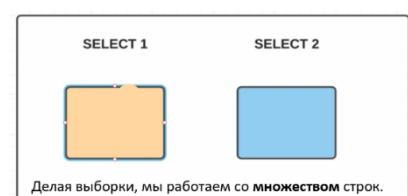
Как посмотреть графическое отображение БД в Идее или DBeaver? Выбираем таблицы (или БД), кликаем правой кнопкой мышки и в выпадающем списке выбираем Diagrams.

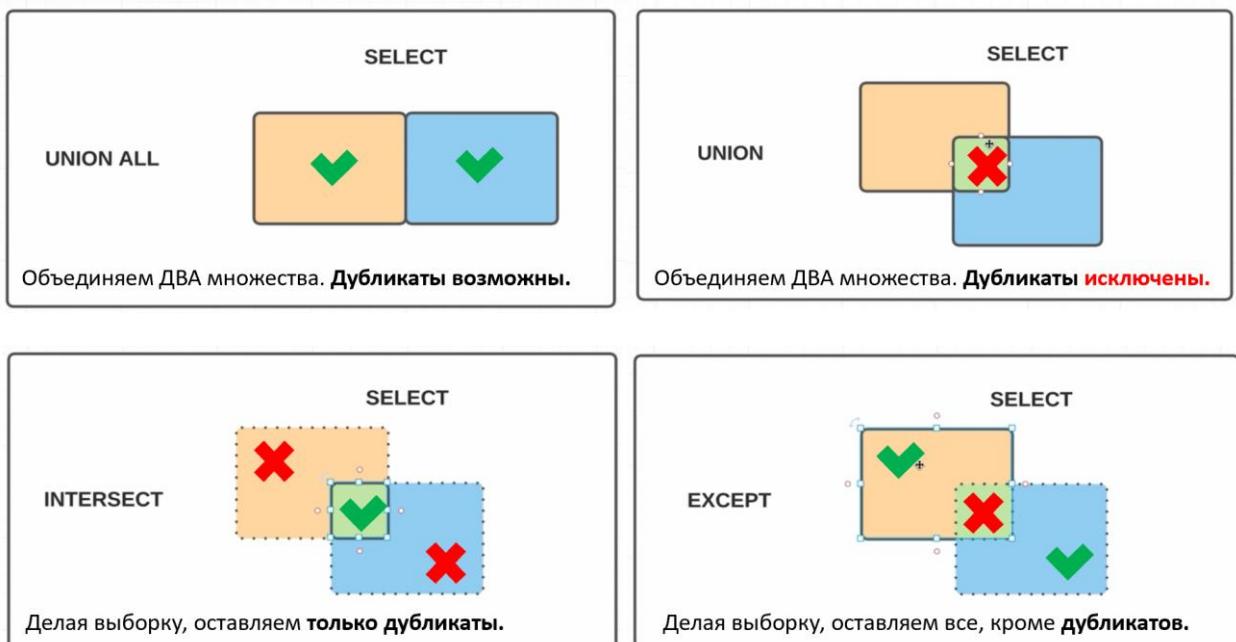


Теория множеств

Ограничения для операций со множествами:

- кол-во полей/столбцов в строках должны совпадать
- тип данных в колонках должны совпадать





Объединения (UNION и UNION ALL), пересечения (INTERSECT) и исключения (EXCEPT):

values (1, 2), (3, 4), (5, 6), (7, 8) union all values (1, 2), (3, 4), (5, 6), (7, 8)	values (1, 2), (3, 4), (5, 6), (7, 8) union values (1, 2), (3, 4), (5, 6), (7, 8)	values (1, 2), (3, 4), (5, 6), (7, 18) intersect values (1, 2), (3, 4), (5, 6), (7, 8)	values (1, '2'), ('3', '4'), ('5', '6'), ('7', '8') except values (1, '2'), ('2', '4'), ('5', '6'), ('7', '9')																																																																		
<table border="1"> <thead> <tr> <th></th> <th>column1</th> <th>column2</th> </tr> </thead> <tbody> <tr><td>1</td><td>1</td><td>2</td></tr> <tr><td>2</td><td>3</td><td>4</td></tr> <tr><td>3</td><td>5</td><td>6</td></tr> <tr><td>4</td><td>7</td><td>8</td></tr> <tr><td>5</td><td>1</td><td>2</td></tr> <tr><td>6</td><td>3</td><td>4</td></tr> <tr><td>7</td><td>5</td><td>6</td></tr> <tr><td>8</td><td>7</td><td>8</td></tr> </tbody> </table>		column1	column2	1	1	2	2	3	4	3	5	6	4	7	8	5	1	2	6	3	4	7	5	6	8	7	8	<table border="1"> <thead> <tr> <th></th> <th>column1</th> <th>column2</th> </tr> </thead> <tbody> <tr><td>1</td><td>1</td><td>6</td></tr> <tr><td>2</td><td>1</td><td>2</td></tr> <tr><td>3</td><td>7</td><td>8</td></tr> <tr><td>4</td><td>3</td><td>4</td></tr> </tbody> </table>		column1	column2	1	1	6	2	1	2	3	7	8	4	3	4	<table border="1"> <thead> <tr> <th></th> <th>column1</th> <th>column2</th> </tr> </thead> <tbody> <tr><td>1</td><td>5</td><td>6</td></tr> <tr><td>2</td><td>1</td><td>2</td></tr> <tr><td>3</td><td>7</td><td>8</td></tr> <tr><td>4</td><td>3</td><td>4</td></tr> </tbody> </table>		column1	column2	1	5	6	2	1	2	3	7	8	4	3	4	<table border="1"> <thead> <tr> <th></th> <th>column1</th> <th>column2</th> </tr> </thead> <tbody> <tr><td>1</td><td>7</td><td>8</td></tr> <tr><td>2</td><td>3</td><td>4</td></tr> </tbody> </table>		column1	column2	1	7	8	2	3	4
	column1	column2																																																																			
1	1	2																																																																			
2	3	4																																																																			
3	5	6																																																																			
4	7	8																																																																			
5	1	2																																																																			
6	3	4																																																																			
7	5	6																																																																			
8	7	8																																																																			
	column1	column2																																																																			
1	1	6																																																																			
2	1	2																																																																			
3	7	8																																																																			
4	3	4																																																																			
	column1	column2																																																																			
1	5	6																																																																			
2	1	2																																																																			
3	7	8																																																																			
4	3	4																																																																			
	column1	column2																																																																			
1	7	8																																																																			
2	3	4																																																																			

*Какой вид структуры данных (какое дерево) используется при работе с базами данных?

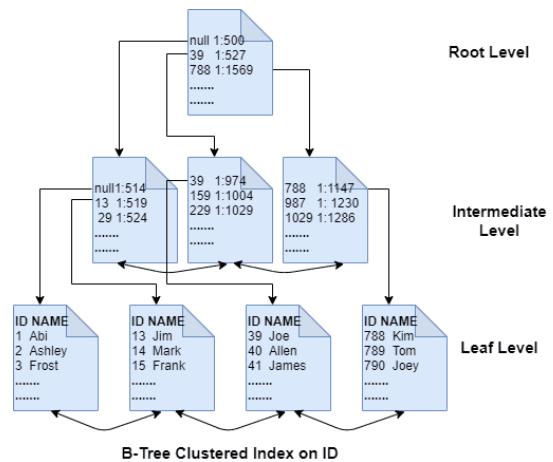
Как под капотом SQL организован поиск и извлечение нужных данных?

Таблица – это файл на жёстком диске, где в бинарном виде представлены данные, отражённые в таблицах.

Индексы помогают обеспечивать формирование быстрой выборки данных из БД (один индекс = один файл).

Структура данных **B-Tree** используется по умолчанию для быстрого поиска данных по уровням (не путать с бинарным деревом).

Семейство **B-Tree** индексов — это наиболее часто используемый тип данных (индексов), организованных как сбалансированное дерево, упорядоченных ключей и созданы для эффективной работы с дисковой памятью. Они поддерживаются практически всеми СУБД как реляционными, так не реляционными, и практически для всех типов данных.



Отличие B-Tree от бинарного дерева: в бинарном дереве поиска каждый узел содержит лишь одно значение (ключ) и не более 2-х потомков.

Но существует **особый вид** дерева поиска, называемый **B-Tree** (Би-дерево).
Здесь узел содержит больше одного значения и больше 2-х потомков.

Справка: Деревья представляют собой структуры данных, в которых реализованы операции над динамическими множествами. Из таких операций хотелось бы выделить — поиск элемента, поиск минимального (максимального) элемента, вставка, удаление, переход к родителю, переход к ребенку.

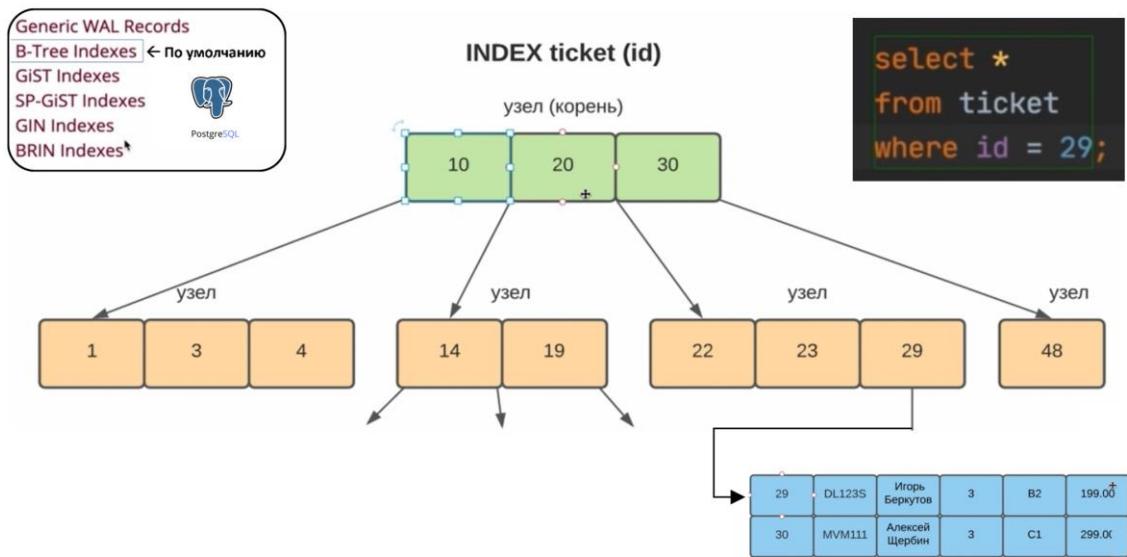
Таким образом, **дерево может использоваться и как обыкновенный словарь, и как очередь с приоритетами**.

Основные операции в деревьях выполняются за время, пропорциональное его высоте. Сбалансированные деревья минимизируют свою высоту (к примеру, высота бинарного сбалансированного дерева с n узлами равна $\log n$).

Разделяют уровни B-дерева:

- корневой узел (м.б. из нескольких сегментов) + ссылки на узлы промежуточного ур.
- промежуточные узлы (количество узлов +1 к кол-ву сегментов корневого) + ссылки
- узлы-листья (ссылки на следующий уровень отсутствуют)

Поиск индекса начинается от корневого узла. Меньшее значение сегмента всегда левее. Ищем, например, 29. Сравниваем попарно значения, начиная с левого корневого сегмента. Значение сегмента меньше? Тогда идем вправо на этом же уровне. Значение больше? Тогда переходим на уровень ниже и продолжаем двигаться слева направо по сегментам узла и т.д.



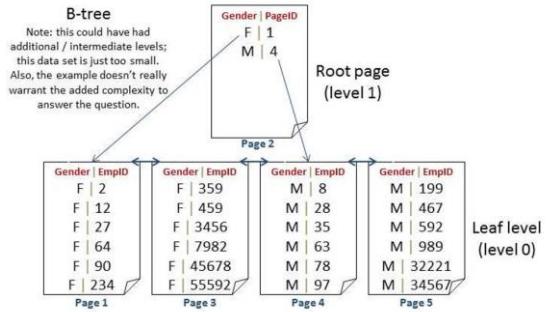
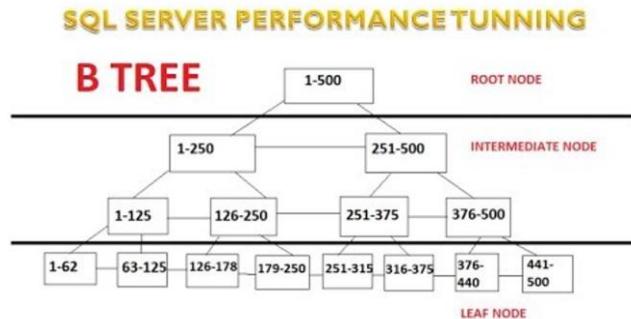
Найдя нужный индекс (29), переходим к нужному файлу **по ссылке** (1 индекс = 1 файл).

Операция поиска выполняется за время $O(t \log t)$, где t – минимальная степень. Важно здесь, что дисковых операций мы совершаляем всего лишь $O(\log t)$!

<https://habr.com/ru/post/114154/>

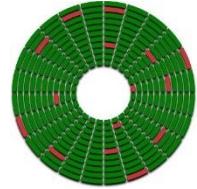
Ссылка может быть представлена смещением количества байт от начала файла таблицы, чтобы найти искомую строчку (запись в таблице индекса).

Индекс (29), являющийся представлением таблицы, называется **кластерным индексом**.



Почему так? Количество элементов в одном узле хранятся в одном сегменте на жёстком диске (см. рис.) и нужно обеспечить более-менее равномерное распределение памяти, а также наименьшее возможное количество обращений к памяти диска для оптимальной работы с БД.

За один раз компьютер считывает в оперативную память все данные из одного сегмента на жёстком диске.



*Как оценить сложность выполняемого запроса?

Стоимостной оптимизатор делает расчёт на основании двух показателей:

- **page_cost**, измеряется в единицах стоимости 1.0 – сколько страниц нужно считать для выполнения одного запроса (input-output).
- **cpu_cost** – как много операций сделал процессор для выполнения запроса. Например, прошлись по строке, считали информацию одной записи в таблице. Стоимость будет 0,01.

В PostgreSQL есть специальная таблица **pg_class**, которая хранит статистические данные, на основании которых базируется работа стоимостного оптимизатора.

```
select reltuples,
       relkind,
       relpages
  from pg_class
 where relname = 'ticket';
```

*Как узнать, сколько места занимает таблица в памяти?

В приведённом примере таблица имеет 6 полей.

```
-- 8 + 6 + 28 + 8 + 2 + 8 = 60

select avg(bit_length(passenger_no) / 8),
       avg(bit_length(passenger_name) / 8),
       avg(bit_length(seat_no) / 8)
  from ticket;
```

Для целочисленных типов данных в поле берём фиксированное значение (8 байт для BIGINT).

Символьные типы определяем через запрос →

Потом суммируем все поля одной записи и получаем результат одной строки таблицы в байтах. Умножив на кол-во записей в таблице, можно оценить весь объём данных.

Full Scan – это очень дорогая операция (перебор всех строк и полей).
Индексы позволяют уменьшить стоимость запросов.

*Методы сканирования таблиц:

- Индексное (index scan)
- Исключительно индексное сканирование (index only scan)
- Сканирование по битовой карте (bitmap scan)
- Последовательное сканирование (sequential scan)

```
-- index only scan
-- index scan
-- bitmap scan (index scan, heap scan)

-- 0 1 0 0 1 0 1 1 0 0 0 ... 636 times
-- 2 5 7 8

-- 0 1 0 0 1 0 1 1 0 0 0 ... 636 times
-- 0 0 0 1 1 0 0 1 0 0 0 ... 636 times

-- 0 1 0 1 1 0 1 0 0 0 ... 636 times OR
-- 0 0 0 0 1 0 0 1 0 0 0 ... 636 times AND
```

19. Что такое индексы? Какие они бывают?

Индексы – это специальные структуры в базах данных, которые позволяют ускорить поиск и сортировку по определённому полю или набору полей в таблице, а также обеспечивают уникальность данных.

Подробно и доступно тут → <https://youtu.be/lAWQNcAEiKw?t=932>

- Индекс – объект БД, который можно создать и удалить
- Позволяет искать значения без полного перебора
- Оптимизация выборки «небольшого» числа записей
- «Небольшое» число – число относительное кол-ва записей
- По PRIMARY KEY и UNIQUE столбцам индекс создаётся автоматически
- Индексы не бесплатны

Когда мы создаём первичный ключ (чаще всего это уникальный идентификатор id), то **автоматически формируется индекс**.

Совокупность этих данных/индексов – это и есть **структура B-Tree** (по умолчанию), которая создаёт отдельный файл, где находятся все ключи.

```
SELECT amname FROM pg_am;
```

- B-tree (сбалансированное дерево)
- Хеш-индекс
- GiST (обобщённое дерево поиска)
- GIN (обобщённый обратный)
- SP-GiST (GiST с двоичным разбиением пространства)
- BRIN (блочно-диапазонный)

Balanced Tree

- Создаётся по умолчанию (`CREATE INDEX index_name ON table_name (column_name)`)
- Поддерживает операции:
`<`, `>`, `<=`, `>=`, `=`
- Поддерживает LIKE 'abc%' (но не '%abc')
- Индексирует NULL
- Сложность поиска $O(\log N)$

Hash

- `CREATE INDEX index_name ON table_name USING HASH (column_name);`
- Поддерживает только операцию “=”
- Не отражается в журнале предзаписи (WAL)
- Не рекомендуется к применению (в общем и целом)
- Сложность поиска $O(1)$

Индексы бывают **кластерные и не кластерные** (еще называют кластеризованные и нет).

Если по ссылке с индекса лежит отдельная таблица (отдельный файл), то такой индекс называется **кластерным**.

Кластерный индекс хранит реальные строки/записи данных в листьях индекса. Важной характеристикой кластерного индекса является то, что **все значения отсортированы** в определенном порядке либо возрастания, либо убывания. Таким образом, таблица или представление может иметь только один кластерный индекс. В дополнение следует отметить, что данные в таблице хранятся в отсортированном виде только в случае, если у этой таблицы создан кластерный индекс. Таблица, не имеющая кластерного индекса, называется **кучей**.

Не кластерный индекс, созданный для такой таблицы, содержит только указатели ссылки на записи таблицы (а не саму таблицу).

	MySQL	PostgreSQL	MS SQL	Oracle
B-Tree index	Есть	Есть	Есть	Есть
Spatial indexes Поддерживаемые пространственные индексы	R-Tree с квадратичным разбиением	Rtree_GiST (используется линейное разбиение)	4-х уровневый Grid-based spatial index (отдельные для географических и геодезических данных)	R-Tree с квадратичным разбиением; Quadtree
Hash index	Только в таблицах типа Memory	Есть	Нет	Нет
Bitmap index	Нет	Есть	Нет	Есть
Reverse index	Нет	Нет	Нет	Есть
Inverted index	Есть	Есть	Есть	Есть
Partial index	Нет	Есть	Есть	Нет
Function based index	Нет	Есть	Есть	Есть

Как и зачем создавать индекс?

Для того, чтобы добавить индекс, необходимо использовать команду **CREATE INDEX**, что позволит указать имя индекса и определить таблицу и колонку или индекс колонки и определить, используется ли индекс по возрастанию или по убыванию.

Создание некластеризованного индекса в таблице или представлении:

```
CREATE INDEX index1 ON schema1.table1 (column1);
```

Создание кластеризованного индекса в таблице и использование имени, состоящего из трех элементов, для таблицы:

```
CREATE CLUSTERED INDEX index1 ON database1.schema1.table1 (column1);
```

Создание некластеризованного индекса с ограничением уникальности и указание порядка сортировки:

```
CREATE UNIQUE INDEX index1 ON schema1.table1 (column1 DESC, column2 ASC, column3 DESC);
```

Индексы – это решение многих проблем с производительностью, но слишком много индексов на таблицах будет влиять на производительность операторов INSERT, UPDATE и DELETE. Это **связано с обновлением индексов**, когда вы добавляете (INSERT), изменяете (UPDATE) или удаляете (DELETE) данные.

Зачем создавать индекс?

Оптимизация! Кратное улучшение времени поиска за счёт применения индекса:

 До применения индекса: Query returned successfully in 21 secs 523 msec.

```
8 EXPLAIN
9 SELECT *
10 FROM perf_test
11 WHERE LOWER(annotation) LIKE('ab%');
12
13 CREATE INDEX idx_perf_test_annotation_lower ON perf_test(LOWER(annotation));
```

 После применения индекса =)

Data Output Explain Messages Notifications

Successfully run. Total query runtime: 348 msec. 39073 rows affected.

*Что такое селективность?

Селективность – это отношение количества уникальных элементов к кол-ву записей, т.е. строчек в таблице. В примере ниже селективность плохая, меньше 20%. Оптимально – 1.

Селективность влияет на скорость поиска, поэтому **необходимо определять порядок индексов**.

Важно также понимать, что DML функции (обновление, изменение, удаление данных) обновляет также все индексы, поэтому **быть внимательным при создании индексов**.

```
select count(distinct flight_id) from ticket;
select count(*) from ticket;
-- 9 / 55
-- 55 / 55 = 1 I
```



*Анализ выполнения запросов (если есть проблемы с производительностью)

EXPLAIN

- Если есть проблема с производительностью – надо понять откуда «растут ноги»
- EXPLAIN query позволяет посмотреть на план выполнения запроса
- EXPLAIN ANALYZE query прогоняет запрос, показывает план и реальность

```
15 EXPLAIN
16 SELECT *
17 FROM perf_test
18 WHERE id = 3700000
19
20 CREATE INDEX idx_perf_test_id ON perf_test(id);
```

Data Output Explain Messages Notifications

QUERY PLAN

text	Bitmap Heap Scan on perf_test (cost=939.93..117842.15 rows=50000 width=...
1	Recheck Cond: (id = 3700000)
2	-> Bitmap Index Scan on idx_perf_test_id (cost=0.00..927.43 rows=50000 w...
3	Index Cond: (id = 3700000) -> Bitmap Index Scan on idx_perf_test_id (cost=0.00..927.43 rows=50000 width=0)
4	

Запрос `explain analyze` показывает план выполнения запроса: что ожидали и что получили. Можно это использовать, чтобы проанализировать, каким образом выполняется запрос + получить статистические данные.

```
explain analyze
select *
from test1
where number1 < 1000;
```

QUERY PLAN

1	Bitmap Heap Scan on test1 (cost=0.95..795.75 rows=5504 width=17) (actual time=0.955..3.144 rows=5621 loops=1)
2	Recheck Cond: (number1 < 1000)
3	Heap Blocks: exact=596
4	-> Bitmap Index Scan on test1_number1_idx (cost=0.00..89.57 rows=5504 width=0) (actual time=0.814..0.815 rows=5621 loops=1)
5	Index Cond: (number1 < 1000)
6	Planning Time: 0.102 ms
7	Execution Time: 3.844 ms

Планируемое время – сколько времени планировщик потратил на построение плана.

Реальное время выполнения запроса.

Cost – стоимость; actual – время на выполнение запроса; rows – количество строк, которое вернулось.

Heap Blocks – как много элементов построено в последовательности (нули и единицы), то есть сколько памяти выделено для считывания данных из таблицы.

Например, проанализируем, как происходит связывание Join с помощью планового выполнения запросов?

ТРИ варианта связывания таблиц (и как это устроено): nested loop, hash join, merge join.

- Nested Loop** работает на маленькой выборке. Считывает одну таблицу, чтобы обратиться по индексу второй.
- Hash Join** работает на больших выборках. Считывает обе таблицы полностью, а потом на основе второй таблицы создаёт хэш-таблицу (операция считывания которой происходит за константное время O(1) – очень быстро).

- **Merge Join** работает на основании отсортированных ключей. Сканирует обе таблицы, сортирует, а затем быстро сравнивает элементы попарно.

Способ связывания выбирает планировщик на основании статистических и кэшированных данных (после предыдущих запросов).

```
explain analyze
select *
from test1
join test2 t
  on test1.id = t.test1_id
limit 100;
```

Здесь **Nested Loop** занимался связыванием двух таблиц (снизу-вверх: в первой поиск по индексу p_key, во второй использовался full scan, время выполнения соответствующее).

Nested Loop используется, когда записей не очень много, а тут есть ограничение **limit**.

Как устроено? Идёт обычным циклом по просканированной таблице `test_2`, а дальше Index Condition связывает индекс из таблицы `test_2` с индексом первой таблицы (`id=t.test_1_id`); `loops=100` – это значит, что 100 раз выполняли это действие по 1 строке за указанные времена и стоимость для одной записи:

```
explain analyze
select *
from test1 t1
join test2 t2
  on t1.id = t2.test1_id;
```

Тут используется **Hash Join**. Он также состоит из двух частей. Сканирует первую таблицу, а на основании просканированной второй таблицы создаёт хэш-таблицу.

Сначала происходит full scan одной таблицы, но в случае второй мы уже используем не индекс. Элемент Hash полностью просканировал таблицу и на основании полного считывания информации создал структуру данных Hash Table (хэш таблицу). Поиск по ней происходит за константное время $O(1)$. Можно увидеть, сколько бакетов создано.

Batches: 2, а значение >1 означает, что оперативной памяти не хватило и часть инфы была сохранена на диск. То, что попало в Memory заняло около 3 мб.

Hash Condition (синяя заливка строки) показывает, что данные довольно быстро получены. Условия связывания указаны в скобках.

```
explain analyze
select *
from test1 t1
join (select * from test2 order by test1_id) t2
  on t1.id = t2.test1_id;
```

Также быстрый вариант **Merge Join**. Состоит из трёх основных частей, использует **отсортированную** последовательность.

Для сканирования таблицы `test_1` использовали индекс p_key. Для таблицы `test_2` сначала использовали full scan, далее отсортировали по ключу `test_1_id`

Указан метод сортировки и количество выделенной памяти около 3 мб. Далее Merge Join обходит обычным циклом обе отсортированных последовательности (как если бы циклом for мы обходили два отсортированных массива и попарно сравнивали значения).

20. Чем TRUNCATE отличается от DELETE?

DELETE - оператор DML, удаляет записи из таблицы, которые удовлетворяют критерию WHERE при этом задействуются триггеры, ограничения и т.д. **РОЛЛБЕК**

TRUNCATE - DDL оператор, удаляет таблицу и создает ее заново. Причем если на эту таблицу есть ссылки FOREIGN KEY или таблица используется в репликации, то пересоздать такую таблицу не получится).

21. Что такое хранимые процедуры? Для чего они нужны?

Хранимые процедуры позволяют повысить производительность, расширяют возможности программирования и поддерживают функции безопасности данных. В большинстве СУБД при первом запуске хранимой процедуры она компилируется (выполняется синтаксический анализ и генерируется план доступа к данным) и в дальнейшем её обработка осуществляется быстрее.

Хранимая процедура — объект базы данных, представляющий собой набор SQL-инструкций, который хранится на сервере. Хранимые процедуры очень похожи на обычные процедуры языков высокого уровня, у них могут быть входные и выходные параметры и локальные переменные, в них могут производиться числовые вычисления и операции над символьными данными, результаты которых могут присваиваться переменным и параметрам. В хранимых процедурах могут выполняться стандартные операции с базами данных (как DDL, так и DML). Кроме того, в хранимых процедурах возможны циклы и ветвления, то есть в них могут использоваться инструкции управления процессом исполнения.

Хранимые процедуры позволяют повысить производительность, расширяют возможности программирования и поддерживают функции безопасности данных. В большинстве СУБД при первом запуске хранимой процедуры она компилируется (выполняется синтаксический анализ и генерируется план доступа к данным) и в дальнейшем её обработка осуществляется быстрее.

22. Что такое представления (VIEW)? Для чего они нужны?

Представление, View – виртуальная таблица, представляющая данные одной или более таблиц альтернативным образом. **Зачем?** Чтобы делать выборки (операция SELECT).

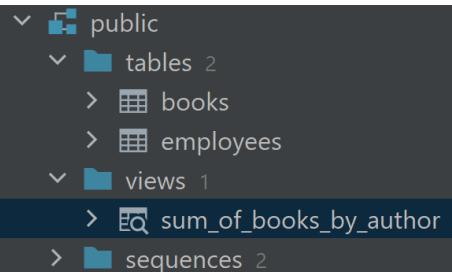
В действительности представление – всего лишь результат выполнения оператора SELECT, который хранится в структуре памяти, напоминающей SQL таблицу. Они работают в запросах и операторах DML точно также как и основные таблицы, но не содержат никаких собственных данных.

Представления значительно расширяют возможности управления данными.

Это способ дать публичный доступ к некоторой (но не всей) информации в таблице.

Как создать VIEW? К команде SQL добавить одну строку и появится папка views, где будут храниться типичные запросы.

```
CREATE VIEW sum_of_books_by_author AS
SELECT author,
       count(books.id)
FROM public.books
GROUP BY books.author;
```

**Как использовать?**

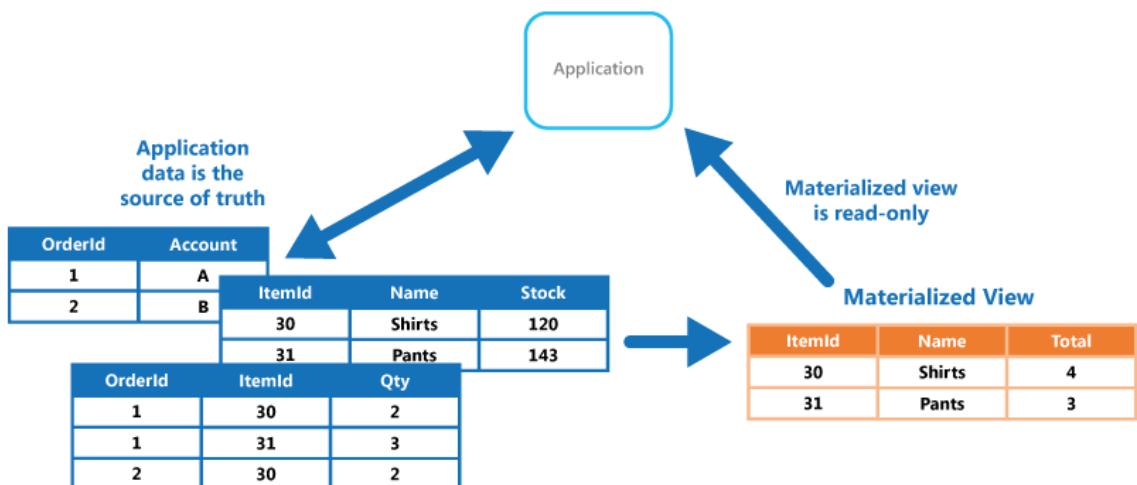
```
SELECT * FROM sum_of_books_by_author;
```

The screenshot shows a SQL query in a code editor: 'select * from employee_view where name = 'Facebook''. To its right is a table viewer displaying results. The table has columns: name, last_name, salary, max, min. The data shows two rows for Facebook employees: Petrov and Sidorov, with salaries 2000 and 1650 respectively.

Что такое **Materialized view** и чем отличается от просто View?

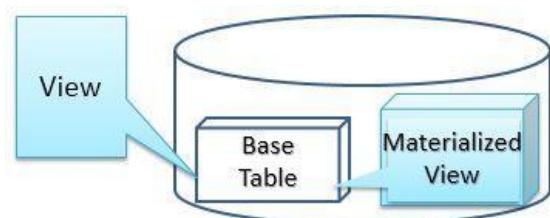
Материализованное представление — физический объект базы данных, содержащий результат выполнения запроса. **Только для чтения!**

Материализованные представления позволяют многократно ускорить выполнение запросов, обращающихся к большому количеству записей, позволяя за секунды выполнять запросы к терабайтам данных.



Полезны, если нам нужно запретить пользователю доступ к некоторым атрибутам таблицы и разрешить доступ к другим атрибутам.

Например, сотрудник может искать имя, адрес, должность, возраст и другие факторы в таблице сотрудников, но он не должен иметь права просматривать или получать доступ к зарплате других сотрудников.

**Отличия:**

	View	Materialized View
Что это?	Виртуальная таблица, созданная в результате сохранённого запроса	физическая копия, картинка или снимок базовой таблицы на момент сохранения
Где хранятся?	не хранятся физически на диске	хранятся на диске

Обновления	обновляется, так как запрос, создающий представление, выполняется каждый раз при использовании VIEW	обновляется вручную или путем применения к нему триггеров (события JOB для обновления данных. Например, регулярная выгрузка статистических данных).
Реакция на запрос	Медленнее, т.к. формируется в ответ на сохраненный запрос	Реагирует быстрее, т.к. предварительно сформировано
Место в памяти	это просто отображение, поэтому ему не требуется место в памяти	использует пространство памяти, хранящееся на диске

23. Что такое временные таблицы? Для чего они нужны?

Временная таблица – это объект базы данных, который хранится и управляется системой базы данных на временной основе. Они могут быть локальными или глобальными.

Используется для сохранения результатов вызова хранимой процедуры, уменьшение числа строк при соединениях, агрегирование данных из различных источников или как замена курсоров и параметризованных представлений.

24. Что такое транзакции? Расскажите про принципы ACID.

Транзакция – это единица работы в рамках соединения с базой данных.

Имеет ДВА состояния:

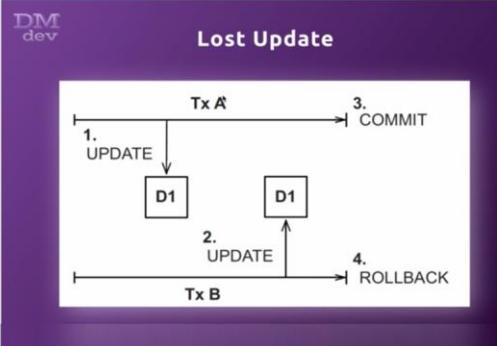
- **Выполняется** полностью – **commit**
- **Откатывается** полностью – **rollback**

ACID – это набор свойств, которыми должна обладать каждая транзакция БД.



Transaction isolation issues

- **Lost Update** - потерянное обновление
- **Dirty Read** - “грязное” чтение
- **Non Repeatable Read** - неповторяющееся чтение
- **Phantom Read** - фантомное чтение



Lost Update – при одновременном изменении одного блока данных разными транзакциями теряются все изменения, кроме последнего;

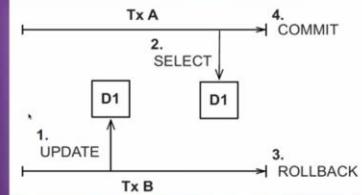
DM dev

Dirty Read

Происходит, когда первая транзакция читает изменения, сделанные другой транзакцией, но эти изменения еще не были ей закомичены. После чего вторая транзакция откатывает эти изменения, а первая продолжает работу с «грязными» данными

DM dev

Dirty Read



```
flight_repository=# select * from aircraft;
id | model
---+---
1 | Боинг 777-300
2 | Боинг 737-300
3 | Аэробус A320-200
4 | Суперджет-100
(4 rows)
```

```
flight_repository=# begin;
BEGIN
flight_repository=# update aircraft
flight_repository=# set model = model || '-1'
flight_repository=# where id = 1;
UPDATE 1
flight_repository=# select * from aircraft
flight_repository=# where id = 1;
id | model
---+---
1 | Боинг 777-300-1
(1 row)
flight_repository=#
```

```
flight_repository=# begin;
BEGIN
flight_repository=# select * from aircraft where id = 1;
id | model
---+---
1 | Боинг 777-300
(1 row)
flight_repository=# commit;
COMMIT
flight_repository# show transaction isolation level;
transaction_isolation
-----+
read committed
(1 row)
flight_repository#
```

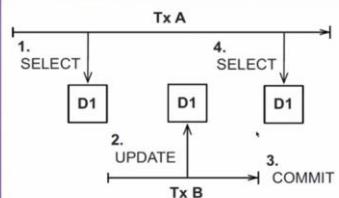
DM dev

Non Repeatable Read

Происходит, когда первая транзакция читает одни и те же данные дважды, но после первого прочтения вторая транзакция изменяет (update) эти же данные и делает коммит, вследствие чего вторая выборка в первой транзакции вернет другой результат

DM dev

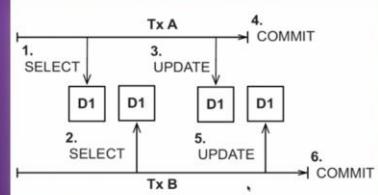
Non Repeatable Read



DM dev

Non Repeatable Read

Существует особый случай Non Repeatable Read называемый **last commit wins**, когда обе транзакции читают одни и те же данные, но первая успевает изменить и закомитить их раньше, чем произойдет изменение и коммит во второй транзакции, вследствие чего изменения первой транзакции теряются



```

flight_repository=# begin;
BEGIN
flight_repository=# select * from aircraft where id = 1;
id | model
-----+
1 | Boeing 777-300
(1 row)

flight_repository=# update aircraft
set model = model || '-1'
where id = 1;
UPDATE 1
flight_repository=# select * from aircraft where id = 1;
id | model
-----+
1 | Boeing 777-300-1
(1 row)

flight_repository=# commit; ←
COMMIT
flight_repository=#

```

```

flight_repository=# begin;
BEGIN
flight_repository=# select * from aircraft where id = 1;
id | model
-----+
1 | Boeing 777-300
(1 row)

flight_repository=# select * from aircraft where id = 1;
id | model
-----+
1 | Boeing 777-300-1
(1 row)

flight_repository=#

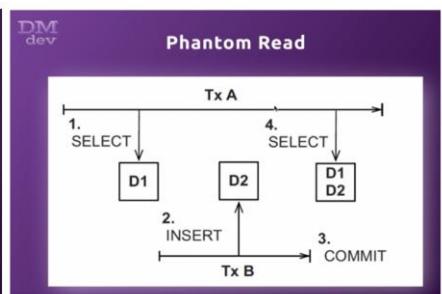
```

После **commit** получили **другие данные в рамках одной транзакции!**

DM dev

Phantom Read

Происходит, когда первая транзакция читает одни и те же данные дважды, но после первого прочтения вторая транзакция добавляет новые строки или удаляет старые и делает коммит, вследствие чего вторая выборка в первой транзакции вернет другой результат (разное количество записей)



```

flight_repository=# begin;
BEGIN
flight_repository=# select count(*) from aircraft;
count
-----
4
(1 row)

flight_repository=# begin;
BEGIN
flight_repository=# select count(*) from aircraft;
count
-----
4
(1 row)

flight_repository=# insert into aircraft(model) VALUES ('test');
INSERT 0 1
flight_repository=# commit; ←
COMMIT
flight_repository=#

```

flight_repository=# commit; ←

```

flight_repository=#

```

25. Расскажите про уровни изолированности транзакций.

Уровень изолированности транзакций — условное значение, определяющее, в какой мере в результате выполнения логически параллельных транзакций в СУБД допускается получение несогласованных данных.

Уровень изоляции	Фантомное чтение	Неповторяющееся чтение	«Грязное» чтение	Потерянное обновление ^[3]
SERIALIZABLE	✓	✓	✓	✓
REPEATABLE READ	✗	✓	✓	✓
READ COMMITTED	✗	✗	✓	✓
READ UNCOMMITTED	✗	✗	✗	✓

В порядке роста уровня изолированности транзакций и надежности работы с данными:

Чтение неподтверждённых данных ✖✖✖♥ (read uncommitted, dirty read) — чтение незафиксированных изменений как своей транзакции, так и параллельных транзакций. Нет гарантии, что данные, измененные другими транзакциями, не будут в любой момент изменены в результате их отката, поэтому такое чтение является потенциальным источником ошибок. Невозможны потерянные изменения, возможны неповторяемое чтение и фантомы.

Типичный способ реализации данного уровня изоляции — блокировка данных на время выполнения команды изменения, что гарантирует, что команды изменения одних и тех же строк, запущенные параллельно, фактически выполняются последовательно, и ни одно из изменений не потерянется.

Транзакции, выполняющие только чтение, при данном уровне изоляции никогда не блокируются.

Чтение подтвержденных данных ✖✖♥♥ (read committed) — чтение всех изменений своей транзакции и зафиксированных изменений параллельных транзакций. Потерянные изменения и грязное чтение не допускается, возможны неповторяемое чтение и фантомы

```

flight_repository=# select * from aircraft;
+----+-----+
| id | model |
+----+-----+
| 1  | Боинг 777-300 |
| 2  | Боинг 737-300 |
| 3  | Аэробус A320-200 |
| 4  | Суперджет-100 |
+----+-----+
(4 rows)

flight_repository=# begin;
BEGIN
flight_repository==# update aircraft
flight_repository==# set model = model || '-1'
flight_repository==# where id = 1;
UPDATE 1
flight_repository==# select * from aircraft
flight_repository==# where id = 1;
+----+-----+
| id | model |
+----+-----+
| 1  | Боинг 777-300-1 |
+----+-----+
(1 row)

flight_repository==# commit;
COMMIT
flight_repository# show transaction isolation level;
transaction_isolation
+-----+
| read committed |
+-----+
(1 row)

flight_repository# 

```

Блокирование читаемых и изменяемых данных.

Заключается в том, что пишущая транзакция блокирует изменяемые данные для читающих транзакций, работающих на уровне read committed или более высоком, до своего завершения, препятствуя, таким образом, «грязному» чтению, а данные, блокируемые читающей транзакцией, освобождаются сразу после завершения операции SELECT (таким образом, ситуация «неповторяющегося чтения» может возникать на данном уровне изоляции).

Сохранение нескольких версий параллельно изменяемых строк.

При каждом изменении строки СУБД создаёт новую версию этой строки, с которой продолжает работать изменившая данные транзакция, в то время как любой другой «читающей» транзакции возвращается последняя зафиксированная версия. Преимущество такого подхода в том, что он обеспечивает большую скорость, так как предотвращает блокировки.

Повторяемость чтения ✖♥♥♥ (repeatable read, snapshot) — чтение всех изменений своей транзакции, любые изменения, внесенные параллельными транзакциями после начала своей, недоступны. Потерянные изменения, грязное и неповторяемое чтение невозможны, возможны фантомы.

```

flight_repository# begin transaction isolation level repeatable read;
BEGIN
flight_repository==# select * from aircraft where id = 1;
+----+-----+
| id | model |
+----+-----+
| 1  | Боинг 777-300-1 |
+----+-----+
(1 row)

flight_repository==# select * from aircraft where id = 1;
+----+-----+
| id | model |
+----+-----+
| 1  | Боинг 777-300-1 |
+----+-----+
(1 row)

flight_repository==# update aircraft
set model = model || '-1'
where id = 1;
UPDATE 1
flight_repository==# select * from aircraft where id = 1;
+----+-----+
| id | model |
+----+-----+
| 1  | Боинг 777-300-1-1 |
+----+-----+
(1 row)

flight_repository==# commit;
COMMIT
flight_repository# 

```

Блокировки в разделяющем режиме применяются ко всем данным, считываемым любой инструкцией транзакции, и сохраняются до её завершения. Это запрещает другим транзакциям изменять строки, которые были считаны незавершённой транзакцией. Пользоваться данным и более высокими уровнями транзакций без необходимости обычно не рекомендуется.

```

flight_repository=# begin transaction isolation level repeatable read;
BEGIN
flight_repository==# select * from aircraft where id = 1;
id | model
---+
1 | Boeing 777-300-1-1
(1 row)

flight_repository==# update aircraft
set model = model || '-1'
where id = 1;
UPDATE 1
flight_repository==# select * from aircraft where id = 1;
id | model
---+
1 | Boeing 777-300-1-1-1
(1 row)

flight_repository==# commit;
COMMIT
flight_repository=#

```

1 | Boeing 777-300-1-1

1; ERROR: could not serialize access due to concurrent update

```

flight_repository=# begin transaction isolation level repeatable read;
BEGIN
flight_repository==# select * from aircraft where id = 1;
id | model
---+
1 | Boeing 777-300-1-1
(1 row)

flight_repository==# update aircraft
odel = model || '-2'
1;
ERROR: could not serialize access due to concurrent update
flight_repository=#!#

```

flight_repository=#!# rollback;

ROLLBACK

flight_repository=#!#

Упорядочиваемость (Serializable) — результат параллельного выполнения стерилизуемой транзакции с другими транзакциями должен быть логически эквивалентен результату их какого-либо последовательного выполнения. Проблемы синхронизации не возникают.

```

flight_repository# begin;
BEGIN
flight_repository==# select count(*) from aircraft;
count
---+
4
(1 row)

flight_repository==# insert into aircraft(model) VALUES ('test');
INSERT 0 1
flight_repository==# commit;
COMMIT
flight_repository# begin;
BEGIN
flight_repository==# insert into aircraft(model) VALUES ('test1');
INSERT 0 1
flight_repository==# commit;
COMMIT
flight_repository# select count(*) from aircraft;
count
---+
6
(1 row)
flight_repository=#

```

flight_repository# begin transaction isolation level serializable;

BEGIN

flight_repository==# select count(*) from aircraft;

count

---+

5

(1 row)

flight_repository==# select count(*) from aircraft;

count

---+

5

(1 row)

flight_repository==# select count(*) from aircraft;

count

---+

5

(1 row)

flight_repository==#

Самый высокий уровень изолированности; транзакции полностью изолируются друг от друга. Результат выполнения нескольких параллельных транзакций должен быть таким, как если бы они выполнялись последовательно. Только на этом уровне параллельные транзакции не подвержены эффекту «фантомного чтения».

26. Что такое нормализация и де- нормализация? Расскажите про 3 нормальные формы?

Нормализация – процесс удаления избыточных данных. Это метод проектирования БД, который позволяет привести базу к минимальной избыточности.

ОШИБКА!!! Не говорите, что нормализация – это процесс приведения к нормальной форме (масло масленое).

Избыточность данных – это ситуация, когда одни и те же данные хранятся в базе в нескольких местах (таблицах). Именно это и приводит к аномалиям.

Нормализация нужна для:

- устранения аномалий
- повышения производительности
- повышения удобства управления данными

Базовые принципы реляционной теории:

- порядок строк не имеет значения
- порядок столбцов не имеет значения

<https://youtu.be/zqQxWdTpSIA>

Пример аномалии (данные избыточны и это проблема).

Решение: вынести данные о материале в отдельную таблицу (справа), появляется связь.

Идентификатор предмета	Наименование предмета	Материал
1	Стул	Металл
2	Стол	Массив дерева
3	Кровать	ЛДСП
4	Шкаф	Массив дерева
5	Комод	ЛДСП

Идентификатор предмета	Наименование предмета	Материал
1	Стул	Металл
2	Стол	Натуральное дерево
3	Кровать	ЛДСП
4	Шкаф	Массив дерева
5	Комод	ЛДСП
6	Тумба	Дерево

Идентификатор предмета	Наименование предмета	Идентификатор материала
1	Стул	2
2	Стол	1
3	Кровать	3
4	Шкаф	1
5	Комод	3

Идентификатор материала	Материал
1	Массив дерева
2	Металл
3	ЛДСП

Процесс нормализации:
Следуя определённым правилам и соблюдая определённые требования, практикуем таблицы в базе данных. Правила группируются в наборы, и если база соответствует определённому набору правил, то она находится в определённой нормальной форме.

Нормальная форма базы данных – это набор правил и критериев, которым должна отвечать БД.

Процесс нормализации – это последовательный процесс приведения базы данных к эталонному виду (переход от одной нормальной формы к следующей).

Полный порядок нормальных форм:

- 1) Ненормализованная форма или нулевая нормальная форма (UNF);
- 2) Первая нормальная форма (1NF);
- 3) Вторая нормальная форма (2NF);
- 4) Третья нормальная форма (3NF);
- 5) Нормальная форма Бойса-Кодда (BCNF);
- 6) Четвёртая нормальная форма (4NF);
- 7) Пятая нормальная форма (5NF);
- 8) Доменно-ключевая нормальная форма (DKNF);
- 9) Шестая нормальная форма (6NF).

Статья на хабре, откуда взяты примеры таблиц: <https://habr.com/ru/post/254773/>

Первая нормальная форма 1НФ

- В таблице НЕ должно быть дублирующих строк
- В каждой ячейке таблицы хранится атомарное значение (одно НЕ составное)
- В столбце хранятся данные одного типа
- Отсутствуют массивы и списки в любом виде

Фирма	Модели
BMW	M5, X5M, M1
Nissan	GT-R

Нарушение нормализации 1НФ происходит в моделях BMW, т.к. в одной ячейке содержится список из 3 элементов: M5, X5M, M1, т.е. он не является атомарным.

Преобразуем таблицу к 1НФ:

Фирма	Модели
BMW	M5
BMW	X5M
BMW	M1
Nissan	GT-R

Вторая нормальная форма 2НФ

- Таблица должна находиться в 1НФ
- Таблица должна иметь ключ
- Все неключевые столбцы должны зависеть от полного ключа (в случае, если ключ составной)

Пример: человек и его паспорт. По серии и номеру (составной ключ) можно определить уникального человека.

Отношение находится во 2НФ, если оно находится в 1НФ и каждый не ключевой атрибут неприводимо зависит от Первичного Ключа (ПК).

Неприводимость означает, что в составе потенциального ключа отсутствует меньшее подмножество атрибутов, от которого можно также вывести данную функциональную зависимость. Например, дана такая таблица с составным ключом (модель + фирма):

Модель	Фирма	Цена	Скидка
M5	BMW	5500000	5%
X5M	BMW	6000000	5%
M1	BMW	2500000	5%
GT-R	Nissan	5000000	10%

Таблица находится в первой нормальной форме, но не во второй. Цена машины зависит от модели и фирмы (составной ключ). Скидка зависит ТОЛЬКО от фирмы, то есть зависимость от первичного ключа неполная.

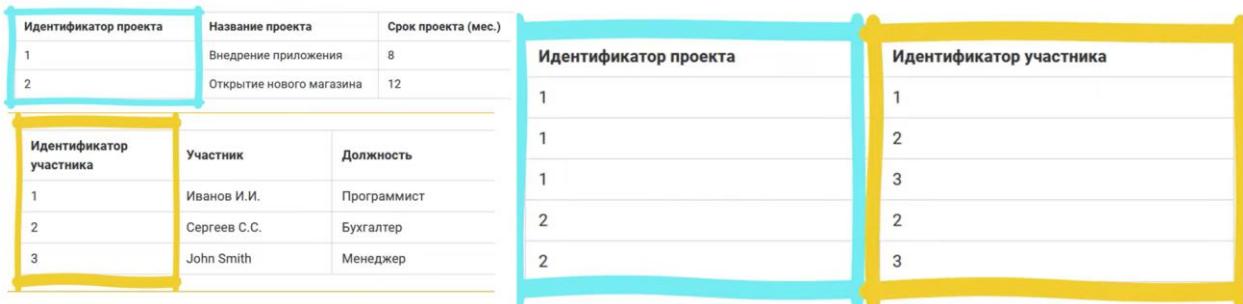
Исправляется это путем декомпозиции на два отношения, в которых **не ключевые атрибуты зависят от первичного ключа ПК.**

<u>Модель</u>	<u>Фирма</u>	Цена
M5	BMW	5500000
X5M	BMW	6000000
M1	BMW	2500000
GT-R	Nissan	5000000



<u>Фирма</u>	Скидка
BMW	5%
Nissan	10%

Декомпозиция: процесс разбиения одного отношения (таблицы) на несколько:



Третья нормальная форма ЗНФ

- Таблица должна находиться во 2НФ
- В таблицах отсутствует транзитивная зависимость

Отношение находится в ЗНФ, когда находится во 2НФ и каждый не ключевой атрибут не транзитивно зависит от первичного ключа. Проще говоря, второе правило требует выносить все не ключевые поля, содержимое которых может относиться к нескольким записям таблицы в отдельные таблицы. Таблица находится во 2НФ, но не в ЗНФ:

<u>Модель</u> (первичный ключ)	Магазин	TRANZIT	Телефон
BMW	Риал-авто		87-33-98
Audi	Риал-авто		87-33-98
Nissan	Нект-Авто		94-54-12

В отношении атрибут «Модель» является первичным ключом. Личных телефонов у автомобилей нет, и телефон зависит исключительно от магазина.

Таким образом, в отношении существуют следующие функциональные зависимости:

Модель → Магазин, Магазин → Телефон, Модель → Телефон. Зависимость **Модель** → **Телефон** является транзитивной, следовательно, отношение не находится в ЗНФ.

В результате разделения исходного отношения получаются **два отношения**, находящиеся в ЗНФ →

Магазин → Телефон
Модель → Магазин

Магазин	Телефон
Риал-авто	87-33-98
Некст-Авто	94-54-12

Модель	Магазин
BMW	Риал-авто
Audi	Риал-авто
Nissan	Некст-Авто

В процессе проектирования БД нужно найти баланс между отсутствием аномалий и приемлемой производительностью.

Полностью нормализованная БД – плохая БД.

Хорошая БД – база, которая достаточно нормализована, чтобы не создавать аномалии для пользователей, и одновременно имеет хорошую производительность.

Денормализация БД

Денормализация базы данных — это **процесс осознанного приведения базы данных к виду**, в котором она не будет соответствовать правилам нормализации. Обычно это необходимо для повышения производительности и скорости извлечения данных, за счет увеличения избыточности данных.

При денормализации важно сохранить баланс между повышением скорости работы базы и увеличением риска появления противоречивых данных, между облегчением жизни программистам, пишущим Select'ы, и усложнением задачи тех, кто обеспечивает наполнение базы и обновление данных. Поэтому проводить денормализацию базы надо очень аккуратно, очень выборочно, только там, где без этого никак не обойтись.

Если заранее нельзя подсчитать плюсы и минусы денормализации, то изначально необходимо реализовать модель с нормализованными таблицами, и лишь затем, для оптимизации проблемных запросов проводить денормализацию.

Оправдан ли будет переход?

Определить требования (чего хотим достичь) -> определить требования к данным (что нужно соблюдать) -> найти минимальный шаг, удовлетворяющий эти требования -> подсчитать затраты на реализацию -> реализовать.

Используемые термины

Атрибут — свойство некоторой сущности. Часто называется полем таблицы.

Домен атрибута — множество допустимых значений, которые может принимать атрибут.

Кортеж — конечное множество взаимосвязанных допустимых значений атрибутов, которые вместе описывают некоторую сущность (строка таблицы).

Отношение — конечное множество кортежей (таблица).

Схема отношения — конечное множество атрибутов, определяющих некоторую сущность. Иными словами, это структура таблицы, состоящей из конкретного набора полей.

Проекция — отношение, полученное из заданного путём удаления и (или) перестановки некоторых атрибутов.

Функциональная зависимость между атрибутами (множествами атрибутов) X и Y означает, что для любого допустимого набора кортежей в данном отношении: если два кортежа совпадают по значению X, то они совпадают по значению Y. Например, если значение атрибута «Название компании» — Canonical Ltd, то значением атрибута «Штаб-квартира» в таком кортеже всегда будет Millbank Tower, London, United Kingdom. Обозначение: $\{X\} \rightarrow \{Y\}$.

Нормальная форма — требование, предъявляемое к структуре таблиц в теории реляционных баз данных для устранения из базы избыточных функциональных зависимостей между атрибутами (полями таблиц).

Метод нормальных форм (НФ) состоит в сборе информации о объектах решения задачи в рамках одного отношения и последующей декомпозиции этого отношения на несколько взаимосвязанных отношений на основе процедур нормализации отношений.

Цель нормализации: исключить избыточное дублирование данных, которое является причиной аномалий, возникших при добавлении, редактировании и удалении кортежей(строк таблицы).

Аномалией называется такая ситуация в таблице БД, которая приводит к противоречию в БД либо существенно усложняет обработку БД. Причиной является излишнее дублирование данных в таблице, которое вызывается наличием функциональных зависимостей от не ключевых атрибутов.

Аномалии-модификации проявляются в том, что изменение одних данных может повлечь просмотр всей таблицы и соответствующее изменение некоторых записей таблицы.

Аномалии-удаления — при удалении какого либо кортежа из таблицы может пропасть информация, которая не связана на прямую с удаляемой записью.

Аномалии-добавления возникают, когда информацию в таблицу нельзя поместить, пока она не полная, либо вставка записи требует дополнительного просмотра таблицы.

27. Что такое TIMESTAMP?

Для работы с датой и временем в MySQL есть несколько типов данных:

DATE, TIME, DATETIME и TIMESTAMP.

Тип	Описание	Диапазон значений	Размер
DATE	Хранит значения даты в виде ГГГГ-ММ-ДД. Например, 2022-12-05	от 1000-01-01 до 9999-12-31	3 байта
TIME	Хранит значения времени в формате ЧЧ:ММ:СС. (или в формате ЧЧЧ:ММ:СС для значений с большим количеством часов). Например, 800:50:50	от -838:59:59 до 838:59:59	3 байта
DATETIME	Хранит значение даты и времени в виде ГГГГ-ММ-ДД ЧЧ:ММ:СС. Например, 2022-12-05 10:37:22	от 1000-01-01 00:00:00 до 9999-12-31 23:59:59	8 байт
TIMESTAMP	Хранит значение даты и времени в виде ГГГГ-ММ-ДД ЧЧ:ММ:СС. Например, 2022-12-05 10:37:22	от 1970-01-01 00:00:01 до 2038-01-19 03:14:07	4 байта

Отличие TIMESTAMP и DATETIME

Типы данных **DATETIME** и **TIMESTAMP** в MySQL похожи друг на друга, так как оба направлены на хранение даты и времени. Но между ними есть ряд существенных отличий, определяющих какой из этих типов данных, когда лучше использовать.

DATETIME

Хранит значения в диапазоне от 1000-01-01 00:00:00 до 9999-12-31 23:59:59 и при этом занимает 8 байт. Этот тип данных не зависит от временной зоны, установленной в MySQL. Он всегда отображается ровно в таком виде, в котором был установлен и в котором храниться в базе данных. То есть при изменении часового пояса, отображение времени не изменится.

```
CREATE TABLE datetime_table (datetime_field DATETIME);
SET @session.time_zone="+00:00"; -- сбрасываем часовой пояс в MySQL
INSERT INTO datetime_table VALUES("2022-06-16 16:37:23");
SET @session.time_zone="+03:00"; -- меняем часовой пояс в MySQL
SELECT * FROM datetime_table;
```

datetime_field

2022-06-16 16:37:23

TIMESTAMP

Хранит сколько прошло секунд с 1970-01-01 00:00:00 по нулевому часовому поясу и занимает 4 байта. При выборках отображается с учетом текущего часового пояса. Часовой пояс можно задать в настройках операционной системы, где работает MySQL, в глобальных настройках MySQL или в конкретной сессии. В базе данных при создании записи с типом **TIMESTAMP** значение сохраняется по нулевому часовому поясу.

```
CREATE TABLE timestamp_table (timestamp_field TIMESTAMP);
SET @session.time_zone="+00:00"; -- сбрасываем часовой пояс в MySQL
INSERT INTO timestamp_table VALUES("2022-06-16 16:37:23");
SET @session.time_zone="+03:00"; -- меняем часовой пояс в MySQL
SELECT * FROM timestamp_table;
```

timestamp_field

2022-06-16 19:37:23

Также стоит помнить о существенном ограничении **TIMESTAMP** в диапазоне возможных значений от 1970-01-01 00:00:01 до 2038-01-19 03:14:07, что ограничивает его применении. Так, **данный тип данных не подойдет для хранения дат рождения пользователей.**

Способ задания значений:

Значения **DATETIME**, **DATE** и **TIMESTAMP** могут быть заданы одним из следующих способов:

- Как строка в формате YYYY-MM-DD HH:MM:SS или в формате YY-MM-DD HH:MM:SS для указания даты и времени
- Как строка в формате YYYY-MM-DD или в формате YY-MM-DD для указания только даты

При указании даты допускается использовать любой знак пунктуации в качестве разделительного между частями разделов даты или времени. Также возможно задавать дату вообще без разделительного знака, слитно.

```
CREATE TABLE date_table (datetime TIMESTAMP);
INSERT INTO date_table VALUES("2022-06-16 16:37:23");
INSERT INTO date_table VALUES("22.05.31 8+15+04");
INSERT INTO date_table VALUES("2014/02/22 16*37*22");
INSERT INTO date_table VALUES("20220616163723");
INSERT INTO date_table VALUES("2021-02-12");
SELECT * FROM date_table;
```

datetime
2022-06-16 16:37:23
2022-05-31 08:15:04
2014-02-22 16:37:22
2022-06-16 16:37:23
2021-02-12 00:00:00

28. Расскажи про шардирование баз данных

При большом количестве данных запросы начинают долго выполняться, и сервер перестаёт справляться с нагрузкой. Одно из решений для оптимизации — это масштабирование базы данных. Например, **шардинг или репликация** (копирование).

Шардинг бывает вертикальным (партиционирование для 1 экз. БД) и **горизонтальным**.

Допустим, есть большая таблица пользователей. Партиционирование — это когда одну большую таблицу разделяют на много маленьких по какому-либо принципу.

Единственное отличие горизонтального масштабирования от вертикального в том, что горизонтальное будет разносить данные по разным инстансам в других базах.

Т.е. только записи с category_id=1 будут попадать в эту таблицу.

На базовую таблицу необходимо добавить правило. Когда мы будем работать с таблицей news, вставка на запись с category_id = 1 должна попасть именно в партицию news_1. Правило называем как хотим.

```
01. CREATE TABLE news (
02.     id bigint not null,
03.     category_id int not null,
04.     author character varying not null,
05.     rate int not null,
06.     title character varying
07. )
```

```
01. CREATE RULE news_insert_to_1 AS ON INSERT TO news
02. WHERE (category_id = 1)
03. DO INSTEAD INSERT INTO news_1 VALUES (NEW.*)
```

29. Как сделать запрос из двух баз?

Допустим, у нас есть две таблицы: с товарами (есть поле `owner_id`, отвечающего за `id` владельца товара) и с пользователями (есть поле `id`).

Мы хотим одним **SQL-запросом** получить все записи, причём чтобы в каждой была информация о пользователе и его одном товаре. В следующей записи была информация о том же пользователе и следующем его товаре. Когда товары этого пользователя закончатся, то переходить к следующему пользователю.

Таким образом, мы должны соединить две таблицы и получить результат, в котором каждая запись содержит информацию о пользователе и об одном его товаре.

```
SELECT * FROM users, products WHERE users.id = products.owner_id
```

Алгоритм здесь уже несложный: берётся первая запись из таблицы `users`. Далее берётся её `id` и анализируются все записи из таблицы `products`, добавляя в результат те, у которых `owner_id` равен `id` из таблицы `users`. Таким образом, на первой итерации собираются все товары у первого пользователя. На второй итерации собираются все товары у второго пользователя и так далее.

30. Что такое триггер?

Триггер (trigger) — это хранимая процедура особого типа, исполнение которой обусловлено действием по модификации данных: добавлением, удалением или изменением данных в заданной таблице реляционной базы данных.

Триггер запускается сервером автоматически и все производимые им модификации данных рассматриваются как выполняемые в транзакции, в которой выполнено действие, вызвавшее срабатывание триггера.

Момент запуска триггера определяется с помощью ключевых слов BEFORE (триггер запускается до выполнения связанного с ним события) или AFTER (после события).

Доп. вопросы:

Что такое sql-injection (SQL инъекции)?

Внедрение SQL кода (англ. SQL injection) — один из распространённых способов взлома сайтов и программ, работающих с базами данных, основанный на внедрении в запрос произвольного SQL-кода.

SQL инъекция означает ввод/вставку SQL-кода в запрос с помощью введенных пользователем данных. Это может произойти в любых приложениях, использующих реляционные базы данных, такие как Oracle, MySQL, PostgreSQL и SQL Server.

Составление SQL запроса вручную

```
Statement statement = connection.createStatement();
String SQL = "INSERT INTO Person VALUES(" + 1 + "," + person.getName() +
            "," + person.getAge() + "," + person.getEmail() + ")";
statement.executeUpdate(SQL);
```

- 1) Неудобно
- 2) Легко допустить ошибку
- 3) Угроза SQL инъекции

SQL инъекция

Один из самых распространённых способов взлома сайтов и программ, работающих с базами данных

```
String SQL = "INSERT INTO Person VALUES(" + 1 + "," + person.getName() +
            "," + person.getAge() + "," + person.getEmail() + ")";
```

Так как строки из HTML формы напрямую конкatenируются в SQL запросе, злоумышленник может подобрать такую строку, которая нанесет вред

Если в качестве email'a в форме мы введем строку:
`test@mail.ru'); DROP TABLE Person; --`

```
INSERT INTO Person VALUES(1, 'test', 15, 'test@mail.ru'); DROP TABLE Person; --';
```

SQL инъекции

Бывают разные

UserId:

Then, the SQL statement will look like this:

```
SELECT * FROM Users WHERE UserId = 105 OR 1=1;
```

PreparedStatement

То, что должно использоваться в JDBC API для всех запросов, где содержатся данные, полученные от пользователя

```
PreparedStatement preparedStatement =
        connection.prepareStatement(sql: "INSERT INTO Person VALUES(1, ?, ?, ?)");
preparedStatement.setString(parameterIndex: 1, person.getName());
preparedStatement.setInt(parameterIndex: 2, person.getAge());
preparedStatement.setString(parameterIndex: 3, person.getEmail());
preparedStatement.executeUpdate();
```

SQL запрос компилируется один раз и не может быть изменен.
Данные от пользователя могут быть вставлены лишь в указанные места запроса и не могут изменить сам SQL запрос.

Н. Алишев (Spring Framework. Урок 26: SQL инъекции. PreparedStatement. JDBC API) <https://youtu.be/Y2sRuCUpJ78>

Как выбрать между Statement, PreparedStatement и CallableStatement?

Statement – SQL-выражение, подготовленное к выполнению в рамках определенной JDBC-сессии. Выполняется методом `execute` для обычного выражения, `executeUpdate` для модифицирующего, `executeBatch` для пакетного. Когда ожидаемый размер результата больше `Integer.MAX_VALUE`, используются версии методов `executeLarge*`.

После выполнения, экземпляр `Statement` владеет `ResultSet`-ом, и другими данными о результате выполнения, такими как количество обновленных записей и сгенерированные ключи.

PreparedStatement – пред-скомпилированная версия `Statement`, его наследник. Эффективнее выполняет одно и то же выражение множество раз. Входные параметры объявляются в SQL-выражении символом `?`, следом сеттерами задаются их типы и значения. Делегирует `обязанность скомпилировать` введенные пользователем параметры базе данных.

CallableStatement – наследник `PreparedStatement` для вызова `хранимых процедур`. Кроме входных параметров, позволяет регистрировать `выходные`.

Экземпляры всех трех типов создаются методами интерфейса `Connection`.

Какие классы вовлечены в соединение с базой данных?

DriverManager управляет всеми JDBC-драйверами в приложении. Представляет набор статических методов. Лениво загружает [системным класс-лоадером](#) доступные *Пред-сконфигурированные* драйверы:

- По списку полных имен классов из проперти `jdbc.drivers`;
- Через Service Provider Interface ([SPI](#)).

Менеджер занимается созданием экземпляра **Connection** – ключевого класса при работе с базой данных. Альтернативный менеджеру (и даже рекомендуемый) способ соединения с источником данных – **ConnectionBuilder**. Билдер получают из `javax.sql.DataSource` – формально это часть Java EE, так что здесь не будем подробно на нем останавливаться.

Driver – главный класс реализации JDBC-драйвера. Когда загружается класслоадером, *сам регистрирует себя* в [DriverManager](#). Так что кроме предсконфигурированных драйверов, дополнительные можно загрузить просто вызвав `Class.forName`.

Можно явно создавать `Connection` через драйвер, минуя менеджера и билдер. Драйвер предоставляет информацию о возможных/требуемых для своей работы свойствах в виде массива **DriverPropertyInfo**.

DriverAction – дополнительный интерфейс, который должен реализовывать `Driver`, если хочет получать уведомления о *раз-регистрации* [DriverManager](#)-ом.

Что можно делать с классом Connection?

Итак, в результате [соединения JDBC драйвера](#) создается объект `Connection` – сессия работы с базой данных. Это главный класс при работе с JDBC. Основная роль этого класса – исполнение [SQL-выражений](#) (`Statement`) и получение их результатов в виде `ResultSet`.

`Connection` предоставляет в виде класса `DatabaseMetaData` мета-информацию о базе данных в целом: таблицы, поддерживаемая грамматика SQL, хранимые процедуры, возможности этого соединения, и т.д..

В коннекшне задается множество настройки самого соединения. Это [уровень изоляции транзакций](#), режим авто-коммита, ключи [шардирования](#), и многое другое. Маппинг типов данных SQL в Java-типы задается здесь же, свойством `typeMap`.

Помимо выполнения выражений, `Connection` предоставляет средства для управления транзакциями. Его методами можно создать Savepoint, откатиться к нему, закоммитить транзакцию когда авто-коммит отключен.

Какая разница между `@ElementCollection`, `@OneToOne` и `@ManyToMany`?

Все эти аннотации – часть JPA (Java Persistence API).

С их использованием мы регулярно сталкиваемся в реализациях JPA, таких как Hibernate.

Когда в базу данных сохраняется сущность, в которой есть **поле-коллекция**, это поле обязано быть помеченным одной из аннотаций.

`@OneToOne` и `@ManyToMany` хранят вложенные объекты как отдельные полноценные сущности – для них действуют все те же требования, которые JPA выдвигает для всех `@Entity` классов. Каждая из аннотаций отвечает за свое [отношение](#).

@ElementCollection создает коллекцию встраиваемых классов. Применять её можно только на коллекции, тип элементов которых помечен **@Embeddable**, или входит в список стандартных встраиваемых классов (обертки примитивов, строки, даты, и т.д.).

На уровне хранения в реляционной базе, для **@ElementCollection** будет также создана **отдельная таблица**. Технически она будет находиться в отношении one-to-many.

Но из Java кода коллекция будет выглядеть встроенной: **её элементом не нужно иметь собственные id, ими нельзя манипулировать отдельно от основной сущности.**

Единственное, чем такая коллекция отличается от встроенного поля-примитива – её можно загружать лениво (включено по умолчанию).

***SQL или NoSQL — вот в чём вопрос** (No SQL или Not Only SQL) – посмотреть перед собеседованиями эту статью и далее видео → <https://habr.com/ru/company/rvds/blog/324936/>

В мире технологий баз данных существует два основных направления: SQL и NoSQL, реляционные и нереляционные базы данных. Различия между ними заключаются в том, как они спроектированы, какие типы данных поддерживают, как хранят информацию.

Реляционные БД сгруппированы в таблицах, формат которых задан на этапе проектирования хранилища.

Нереляционные БД устроены иначе. То, что в реляционной БД будет разбито на несколько взаимосвязанных таблиц, в нереляционной может храниться в виде целостной сущности. Нереляционные базы лучше поддаются масштабированию.

Возможности, которые стали причиной популярности таких NoSQL баз данных, как MongoDB, CouchDB, Cassandra, HBase:

1. **Хранение больших объёмов неструктурированной информации.**

База данных NoSQL не накладывает ограничений на типы хранимых данных. Более того, при необходимости в процессе работы можно добавлять новые типы данных.

2. **Использование облачных вычислений и хранилищ.**

Облачные хранилища — отличное решение, но они требуют, чтобы данные можно было легко распределить между несколькими серверами для обеспечения масштабирования. Использование, для тестирования и разработки, локального оборудования, а затем перенос системы в облако, где она и работает — это именно то, для чего созданы NoSQL базы данных.

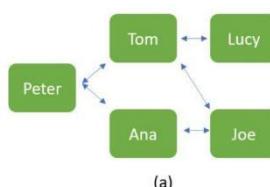
3. **Быстрая разработка.**

Если вы разрабатываете систему, используя agile-методы, применение реляционной БД способно замедлить работу. NoSQL базы данных не нуждаются в том же объёме подготовительных действий, которые обычно нужны для реляционных баз.

Тут показана база данных, содержащая сведения о взаимоотношениях людей.

Вариант **a** — это безсхемная структура, построенная в виде графа, характерная для NoSQL-решений.

Вариант **b** показывает, как те же данные можно представить в структурированном виде, типичном для SQL.



NODE	FRIENDS
Peter	{Tom, Ana}
Tom	{Joe, Lucy, Peter}
Ana	{Joe, Lucy, Peter}
Lucy	{Tom}
Joe	{Tom, Ana}

(b)

Безсхемность означает, что два документа в структуре данных NoSQL не должны иметь одинаковые поля и могут хранить данные разных типов.

Вот, например, массив объектов, набор полей которых не совпадает.

```
var cars = [  
  { Model: "BMW", Color: "Red", Manufactured: 2016 },  
  { Model: "Mercedes", Type: "Coupe", Color: "Black", Manufactured: "1-1-2017" }  
];
```

И в SQL, и в NoSQL-базах индексы служат одной и той же цели — ускорить и оптимизировать извлечение данных. Но то, как именно они работают — различается из-за разных архитектур баз данных и особенностей хранения информации в базе.

В то время, как SQL-индексы представлены в виде B-деревьев, которые отражают иерархическую структуру реляционных данных, в NoSQL базах данных они указывают на документы, или на части документов, между которыми, в основном, нет никаких отношений.

CRM-приложения являются весьма удачным примером, в котором две системы баз данных выступают не конкурентами, а **существуют в гармонии**, играя каждая свою роль в большой архитектуре управления данными.

Признаки проектов, для которых идеально подойдут SQL-базы:

- Имеются логические требования к данным, которые могут быть определены заранее.
- Очень важна целостность данных.
- Нужна основанная на устоявшихся стандартах, хорошо зарекомендовавшая себя технология, используя которую можно рассчитывать на большой опыт разработчиков и техническую поддержку.

Свойства проектов, для которых подойдёт что-то из сферы NoSQL:

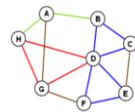
- Требования к данным нечёткие, неопределённые, или развивающиеся с развитием проекта.
- Цель проекта может корректироваться со временем, при этом важна возможность немедленного начала разработки.
- Одни из основных требований к базе данных — скорость обработки данных и масштабируемость.

Всё чаще наблюдается интеграция этих технологий друг в друга.

Например, Microsoft, Oracle и Teradata сейчас предлагают некоторые формы интеграции с Hadoop для подключения аналитических инструментов, основанных на SQL, к миру неструктурированных больших данных.

Дополнительно по теме NoSQL, источник: <https://www.youtube.com/watch?v=JG9Sqnj0xAI&t=112s>

По типам данных:

• key/value store	• document store	• column store	• graph database	• multi-model database
Redis Memcached Riak	MongoDB CouchDB ElasticSearch	HBase Cassandra Vertica	Neo4j 	FoundationDB ArangoDB OrientDB

По способу хранения данных:

in-memory, persistent (in-place updates, snapshots, append-only log).

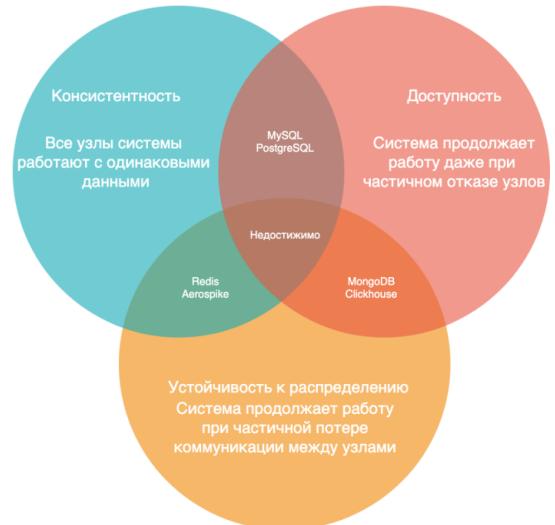
Теорема CAP

Это эвристическое утверждение о том, что в любой реализации распределённых вычислений возможно обеспечить не более двух из трёх следующих свойств:

согласованность данных (англ. consistency) — во всех вычислительных узлах в один момент времени данные не противоречат друг другу;

доступность (англ. availability) — любой запрос к распределённой системе завершается корректным откликом, однако без гарантии, что ответы всех узлов системы совпадают;

устойчивость к разделению (англ. partition tolerance) — расщепление распределённой системы на несколько изолированных секций не приводит к некорректности отклика от каждой из секций.

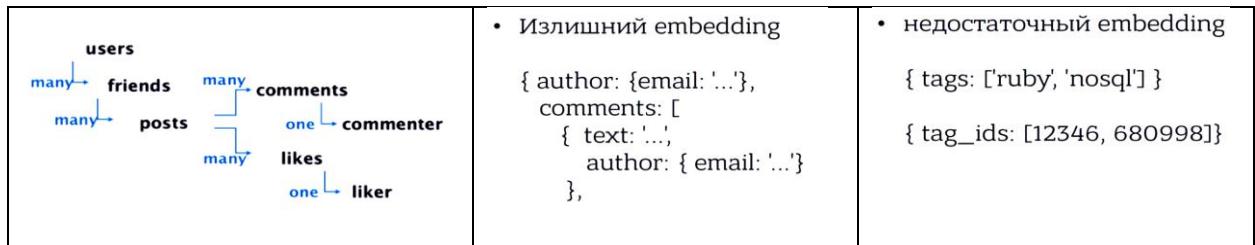


Когда использовать NoSQL БД?

- **Масштабируемость** (линейная масштабируемость, когда путём увеличения ресурсов кластера, мы получаем пропорциональное увеличение характеристик кластера. Круто!)
- **Быстрое прототипирование** (традиционные БД требуют ресурсов на обслуживание, использование не гибкое в меняющихся реалиях бизнес-требований)
- **Высокая доступность** (разносим БД по нескольким дата-центрам)
- **Кэширование**
- **Буферизация** (льётся большой поток данных, который нужно быстро обрабатывать, по потом сохранять результат)ⁱ
- **Очередь заданий** (например, на сайте есть форма регистрации пользователя и во время сеанса не обязательно производить операцию сохранения e-mail в БД. Логичнее поставить это в очередь выполнения заданий).
- **Хранилище бинарников** (например, необходимо хранить фото, тогда можно поднять собственный кластер)
- **Быстрые счётчики**
- **Эффективная оценка кардинальности множеств** (уников) Помощь: алгоритм HyperLogLog.
Проблема: производительность падает пропорционально количеству данных.
- **CMS – система управления контентом.**
- **Полнотекстовый поиск.**

Анти-паттерны: как не нужно делать:

- Ваши данные реляционные (рис. 1)
- Излишний embedding
- Недостаточный embedding
- Неверно выбранный тип данных
- Недостаточно продуманная схема данных



Заключение (в стиле кэп):

- Знайте и изучайте свою предметную область
- Следите за новостями
- Выбирайте БД не только по пресс-релизам (изучайте также недостатки)

Вопросы с реальных собесов:

Какие могут быть проблемы если ты создаешь индекс на таблицу весом 300гб в высоконагруженном проекте?

А как можно улучшить селективность при неизменяемых данных?

ПРАКТИКА

Обязательные задачи SQL <https://habr.com/ru/post/181033/>

Академия SQL (учебник и онлайн тренажёр) <https://sql-academy.org/ru/guide>

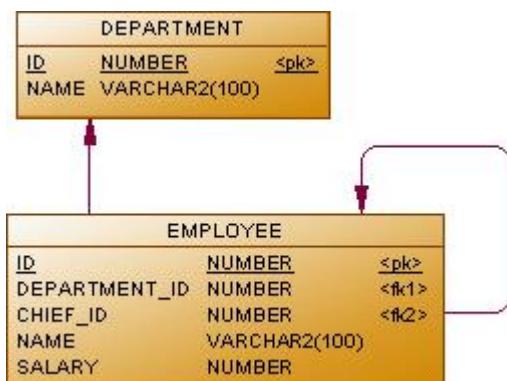
Документация PostgreSQL <https://www.postgresql.org/docs/current/index.html>

Смотрите и практикуйте также курс DMDEV =)

SQL live coding на собеседовании

Кандидату дается условие устно, интервьюер смотрит что он пишет.
Перейти по ссылке и написать два простых запроса.

- 1) Изменить тип данных столбца. В коде поменяю тип данных. Какие операции с БД нужно совершить, чтобы не потерять существующие данные и внести изменения?
- 2) Рассмотрите схему базы данных на рисунке ниже и напишите соответствующие запросы



вопрос	ответ
Вывести список сотрудников, получающих заработную плату большую чем у непосредственного руководителя	<pre>select a.* from employee a, employee b where b.id = a.chief_id and a.salary > b.salary</pre>
Вывести список сотрудников, получающих максимальную заработную плату в своем отделе	<pre>select a.* from employee a where a.salary = (select max(salary) from employee b where b.department_id = a.department_id)</pre>
Вывести список ID отделов, количество сотрудников в которых не превышает 3 человек	<pre>select department_id from employee group by department_id having count(*) <= 3</pre>
Вывести список сотрудников, не имеющих назначенного руководителя, работающего в том-же отделе	<pre>select a.* from employee a left join employee b on (b.id = a.chief_id and b.department_id = a.department_id) where b.id is null</pre>
Найти список ID отделов с максимальной суммарной зарплатой сотрудников	<pre>with sum_salary as (select department_id, sum(salary) salary from employee group by department_id) select department_id from sum_salary a where a.salary = (select max(salary) from sum_salary)</pre>