

WITH RANK_BILLS AS (

SELECT C.NAME,

SQL

БЕЗ СЛЁЗ

(SELECT SUM(COST) FROM BILLCONTENT WHERE B.BID = B.BID) AS BILLSUM,

(SELECT SUM(PAID) FROM BILLCONTENT WHERE B.BID = B.BID) AS BILLPAY,

ROW_NUMBER() OVER (PARTITION BY C.CID ORDER BY CASE

WHEN '2021-01-01'::DATE BETWEEN RP.SINCE

AND RP.UPTO THEN RP.UPTO

ELSE B.PAYDATE

END DESC) AS RN

FROM CLIENTS C

LEFT JOIN BILLS B ON C.CID = B.CID

LEFT JOIN BILLCONTENT BC ON B.BID = BC.BID

LEFT JOIN RETAILPACKS RP ON B.CID = RP.CID

WHERE BC.PRODUCT = 'ПРОДУКТ'

AND '2021-01-01'::DATE BETWEEN RP.SINCE

ID IN (1, 2))

)

SELECT RB.NAME,

RB.INN,

RB.NUM,

RB.BDATE,

RB.PAYDATE,

RB.BILLSUM,

RB.BILLPAY

FROM RANK_BILLS RB

WHERE RB.RN = 1;

САННИКОВ А.А.



Санников А.А.

SQL без слёз, 2024. – 320 с.

ISBN 978-5-6051474-0-4

В книге «SQL без слёз» рассматриваются основы языка SQL на примере СУБД PostgreSQL.

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения автора.

Информация, содержащаяся в книге, полностью проверена и достоверна. Но тем не менее, имея в виду человеческий фактор, в книге возможны орфографические или синтаксические ошибки. Автор книги не несёт ответственности за возможные ошибки, связанные с использованием книги.

© Санников А.А., 2024

Содержание

Введение.....	12
Программное обеспечение	13
Тестовые данные для работы	14
Теоретические аспекты.....	19
База данных.....	19
Таблица	19
СУБД	20
Что такое СУБД?	20
Виды СУБД	22
Язык SQL	23
Типы данных.....	24
Числовые типы данных	24
Символьные типы данных	26
Денежные типы данных	26
Двоичные типы данных	26
Логические типы данных	27
Временные типы данных	27
Объекты базы данных.....	28
Схема базы данных.....	29
Порядок выполнения SQL-запроса	29
Типы команд в SQL	31
Команды DDL.....	31
Команды DML	31
Команды DCL.....	31
Команды TCL	32
Основные моменты	33
Комментарии	33
Разделитель команд (;).....	33

Отображение структуры таблицы	34
Зарезервированные слова	35
Регистр в операторах и функциях	36
Оператор CREATE TABLE	37
Классический способ создания таблицы	37
Создание таблицы при помощи оператора SELECT	39
1 способ	39
2 способ	40
Оператор ALTER TABLE	42
Добавление столбца в таблицу	42
Изменение типа данных столбца в таблице	43
Изменение имени столбца в таблице	44
Удаление столбца в таблице	45
Изменение имени таблицы	47
Первичные и внешние ключи	48
Первичные ключи (Primary key)	48
Создание Primary key при помощи CREATE TABLE	48
Создание Primary key при помощи ALTER TABLE	50
Проверка существования Primary key	51
Суть работы Primary key	52
Удаление Primary key	53
Внешние ключи (Foreign key)	54
Создание Foreign key при помощи CREATE TABLE	55
Создание Foreign key при помощи ALTER TABLE	57
Проверка существования Foreign key	59
Суть работы Foreign key	60
Удаление Foreign key	61
Каскадное удаление Foreign key (ON DELETE CASCADE)	62
Установка для столбцов Foreign key значения NULL (ON DELETE SET NULL)	66

Ограничения	71
Виды ограничений.....	71
Ограничения первичного ключа (Primary key).....	71
Ограничения внешнего ключа (Foreign key).....	72
Ограничения уникальности (Unique)	72
Ограничения NOT NULL.....	75
Ограничения проверки значения (Check).....	77
Оператор COMMENT	81
Просмотр комментариев к таблицам и столбцам.....	81
Добавление комментария для таблицы.....	83
Добавление комментария для столбца	84
Удаление комментариев у таблицы и колонок	85
Обработка транзакций	87
Оператор BEGIN.....	87
Оператор COMMIT	87
Оператор ROLLBACK.....	89
Оператор SAVEPOINT	91
Оператор SET TRANSACTION	94
Оператор SELECT	98
Вывод всех столбцов из таблицы	98
Вывод одного столбца из таблицы.....	98
Вывод нескольких столбцов из таблицы.....	99
Вывод уникальных строк (DISTINCT)	100
Ограничение результатов запроса.....	103
Оператор FROM	105
Оператор AS (псевдонимы)	106
Оператор (конкатенация)	110
Оператор WHERE (фильтрация данных).....	112
Арифметические операторы.....	113

Сложение (+)	113
Вычитание (-)	114
Умножение (*)	115
Деление (/)	116
Остаток от деления (%)	117
Операторы сравнения	119
Оператор =	119
Оператор <> и !=	120
Оператор < и <=	121
Оператор > и >=	122
Оператор IN	124
Оператор BETWEEN	126
Проверка значений NULL	128
Оператор IS NULL	128
Оператор IS NOT NULL	129
Оператор LIKE	130
Метасимвол %	130
Метасимвол _	131
Экранирование символов	132
Логические операторы	135
Оператор AND	135
Оператор OR	136
Оператор NOT	138
Сортировка записей (ORDER BY)	140
Направление сортировки	140
Сортировка по одному столбцу	140
Сортировка по нескольким столбцам	141
Сортировка по номеру столбца	142
Подзапросы (SUBQUERIES)	145

Подзапросы в блоке FROM.....	145
Подзапросы в блоке WHERE.....	146
Подзапросы в блоке SELECT	148
Оператор EXISTS и подзапросы	150
Оператор INSERT.....	152
Добавление одной полной строки.....	152
Добавление части строки.....	154
Добавление нескольких строк	155
Добавление результатов запроса (INSERT SELECT)	156
Оператор UPDATE	159
Обновление определенного столбца в таблице	159
Обновление нескольких столбцов в таблице	160
Оператор DELETE.....	162
Удаление всех строк	162
Удаление определенных строк.....	163
Оператор TRUNCATE TABLE	164
Оператор DROP	165
Оператор MERGE.....	166
Агрегатные функции.....	171
Функция SUM().....	172
Функция COUNT().....	173
Функция AVG()	174
Функция MAX().....	174
Функция MIN().....	175
Группировка данных (GROUP BY).....	177
Сложные группировки	180
Оператор ROLLUP	180
Оператор CUBE.....	184
Оператор GROUPING SETS	187

Функция GROUPING.....	190
Фильтрация групп (HAVING).....	193
Комбинированные запросы.....	196
Оператор UNION и UNION ALL.....	196
Оператор UNION	196
Оператор UNION ALL	198
Сортировка результатов выборки	199
Оператор EXCEPT	201
Оператор INTERSECT.....	204
Предикаты ANY (SOME) / ALL	207
Предикат ANY (SOME).....	207
Предикат ALL.....	208
Объединение таблиц (JOIN).....	210
Классический способ объединения.....	210
Внутреннее объединение (JOIN/INNER JOIN).....	212
Левое внешнее соединение (LEFT OUTER JOIN)	214
Правое внешнее соединение (RIGHT OUTER JOIN).....	216
Полное внешнее соединение (FULL OUTER JOIN).....	218
Перекры́стное соединение (CROSS JOIN)	221
Самообъединение (SELF JOIN).....	224
Естественное объединение (NATURAL JOIN).....	226
Обобщённое табличное выражение (WITH).....	230
Функции преобразования	235
Функция TO_NUMBER.....	235
Функция TO_CHAR.....	236
Функция TO_DATE	237
Математические функции	239
Функция ABS.....	239
Функция CEIL.....	240

Функция FLOOR.....	241
Функция MOD.....	242
Функция POWER	243
Функция ROUND.....	244
Функция SIGN	245
Функция SQRT	246
Функция TRUNC	247
Работа с датой и временем	249
Функция NOW	249
Функция CURRENT_DATE	249
Функция CURRENT_TIME	250
Функция EXTRACT	250
Функция AGE	251
Символьные/строчные функции.....	253
Функция ASCII.....	253
Функция CHR	254
Функция LENGTH	254
Функции LOWER / UPPER / INITCAP.....	255
Функции TRIM / RTRIM / LTRIM	257
Функция SUBSTR	258
Функция REPLACE	259
Условные выражения.....	261
Оператор CASE.....	261
Простая команда CASE.....	261
Поисковая команда CASE.....	263
Функция COALESCE.....	266
Функция NULLIF.....	267
Функция GREATEST.....	268
Функция LEAST.....	269

Оконные функции	270
Что такое оконные функции?	270
Синтаксис оконных функций	271
OVER()	272
PARTITION BY	274
ORDER BY	275
ROWS и RANGE	277
Виды оконных функций	278
Агрегатные функции	278
Ранжирующие функции	286
Функции смещения	294
Аналитические функции	303
Представления (VIEW)	308
Что такое представления?	308
Создание представления	309
Обновление представления	312
Удаление представления	312
Управление доступом к данным	313
Для чего нужны привилегии?	313
Кто выдаёт права доступа к объектам?	313
Виды привилегий	313
Объектные привилегии	313
Системные привилегии	314
Информация о пользователях, ролях и привилегиях?	315
Оператор GRANT	315
Аргументы ALL и PUBLIC	316
Передача привилегий с использованием GRANT OPTION	316
Оператор REVOKE	317
Оператор DENY	317

Заключение.....	319
Полезные ресурсы	320

Введение

В современных информационных технологиях знание языка [SQL](#) стало уже неотъемлемым профессиональным навыком. SQL — универсальный язык структурированных запросов, который применяется для управления данными в реляционной базе данных.

Основы языка SQL поддерживаются почти всеми современными [СУБД](#) (системами управления базами данных), такими как Oracle, PostgreSQL, MySQL, Microsoft SQL Server и т.д. Это позволяет разработчикам и аналитикам применять свои навыки в различных проектах и быть востребованными специалистами на рынке труда.

Цель книги

Научить любого человека работать с реляционными базами данных и получать из них необходимую информацию посредством выполнения SQL-запросов.

В книге не будут рассматриваться такие темы, как оптимизация SQL-запросов, проектирование, нормализация и администрирование баз данных. И хотя это очень важные темы, но большинству людей, которые только начинают изучение SQL, нужно сначала освоить основы, а затем переходить к более сложным темам. Поэтому в книге будут рассмотрены основы языка SQL на примере СУБД PostgreSQL.

Программное обеспечение

Для работы с базами данных PostgreSQL использовалось бесплатное программное обеспечение DBeaver, которое работает на базе операционных систем Windows, Linux и MacOS.

Пиктограммы, используемые в книге

Пиктограммы, расположенные на полях книги указывают на важную информацию.



Данную информацию не стоит игнорировать, она может понадобиться вам в будущем.

Программное обеспечение

Для изучения языка SQL вам потребуется установить на свой компьютер специальное программное обеспечение или вы можете воспользоваться готовым решением в виде онлайн-сервисов, которые позволяют запускать SQL код.

Система управления базами данных

PostgreSQL — это объектно-реляционная система управления базами данных (СУБД), которая предоставляет мощные функции и расширенные возможности для хранения и обработки данных. PostgreSQL является одной из самых популярных и мощных СУБД с открытым исходным кодом.

Официальный сайт: <https://postgresql.org>

Клиент для работы с базой данных

DBeaver — это бесплатная и универсальная программа для работы с базами данных, которая поддерживает подключение к PostgreSQL с помощью драйвера JDBC.

Официальный сайт: <https://dbeaver.io>

Онлайн-сервис

SQLiteOnline.com — это бесплатный онлайн-инструмент для работы с такими базами данных, как: SQLite, MariaDB, MS SQL и PostgreSQL.

Официальный сайт: <https://sqliteonline.com>

Тестовые данные для работы

Для изучения SQL потребуются тестовые данные. Для этого необходимо скопировать соответствующий код и выполнить в своей системе управления базами данных (СУБД).

-- 1 этап: Создаём таблицы

```
DROP TABLE IF EXISTS employees; -- Таблица с сотрудниками
```

```
CREATE TABLE employees (  
    id INT4 NOT NULL, -- Идентификатор сотрудника  
    last_name VARCHAR(100) NOT NULL, -- Фамилия  
    first_name VARCHAR(100) NOT NULL, -- Имя  
    gender VARCHAR(15) NOT NULL, -- Пол  
    birthday DATE NOT NULL, -- Дата рождения  
    email VARCHAR(100) NULL, -- Электронный адрес  
    id_department INT4 NULL, -- Идентификатор отдела  
    id_boss INT4 NULL, -- Идентификатор руководителя  
    salary INT4 NULL -- Заработная плата  
);
```

```
DROP TABLE IF EXISTS departments; -- Таблица с отделами
```

```
CREATE TABLE departments (  
    id INT4 NOT NULL, -- Идентификатор отдела  
    name_department VARCHAR(100) NULL -- Наименование отдела  
);
```

-- 2 этап: Добавление комментариев к таблицам и их столбцам

```
COMMENT ON TABLE employees IS 'Список сотрудников';  
COMMENT ON COLUMN employees.id IS 'Идентификатор сотрудника';  
COMMENT ON COLUMN employees.last_name IS 'Фамилия';  
COMMENT ON COLUMN employees.first_name IS 'Имя';  
COMMENT ON COLUMN employees.gender IS 'Пол';  
COMMENT ON COLUMN employees.birthday IS 'Дата рождения';  
COMMENT ON COLUMN employees.email IS 'Электронный адрес';  
COMMENT ON COLUMN employees.id_department IS 'Идентификатор отдела';
```

```
COMMENT ON COLUMN employees.id_boss IS 'Идентификатор руководителя';
COMMENT ON COLUMN employees.salary IS 'Заработная плата';
```

```
COMMENT ON TABLE departments IS 'Список отделов';
COMMENT ON COLUMN departments.id IS 'Идентификатор отдела';
COMMENT ON COLUMN departments.name_department IS 'Наименование
отдела';
```

-- 3 этап: Добавляем данные в таблицы

-- Таблица с сотрудниками

```
INSERT INTO employees (id, last_name, first_name, gender, birthday,
email, id_department, id_boss, salary)
VALUES
(1, 'Иванов', 'Иван', 'Мужской', '1990-05-15', 'ivanov@mail.ru', 1,
null, 35000),
(2, 'Петров', 'Петр', 'Мужской', '1985-12-20', 'petrov@yandex.ru', 2,
null, 45000),
(3, 'Сидорова', 'Мария', 'Женский', '1992-08-10',
'sidorova@gmail.com', 3, null, 54000),
(4, 'Петров', 'Петр', 'Мужской', '1987-04-25', null, 4, null, 100000),
(5, 'Васильева', 'Екатерина', 'Женский', '1995-03-30',
'vasilieva@yandex.ru', 5, null, 38900),
(6, 'Попов', 'Дмитрий', 'Мужской', '1988-11-05', 'popov@gmail.com', 6,
null, 110000),
(7, 'Соколова', 'Анастасия', 'Женский', '1991-07-12',
'sokolova@mail.ru', 1, 1, 40000),
(8, 'Лебедева', 'Алена', 'Женский', '1994-09-18',
'lebedeva@yandex.ru', 2, 2, 68000),
(9, 'Козлов', 'Артём', 'Мужской', '1986-10-22', 'kozlov@gmail.com', 3,
3, 120000),
(10, 'Новиков', 'Андрей', 'Мужской', '1989-06-08', 'novikov@mail.ru',
4, 4, 42000),
(11, 'Морозова', 'Ольга', 'Женский', '1993-02-14',
'morozova@yandex.ru', 5, 5, 70000),
(12, 'Павлов', 'Сергей', 'Мужской', '1987-12-11', 'pavlov@gmail.com',
6, 6, 130000),
```

(13, 'Григорьева', 'Наталья', 'Женский', '1996-01-25',
'grigoryeva@mail.ru', 1, 1, 60000),
(14, 'Борисов', 'Игорь', 'Мужской', '1985-05-20', 'borisov@yandex.ru',
2, 2, 95000),
(15, 'Максимова', 'Вероника', 'Женский', '1990-08-30',
'maximova@gmail.com', 3, 3, 72000),
(16, 'Кузьмина', 'Елена', 'Женский', '1997-08-15', 'kuzmina@mail.ru',
4, 4, 105000),
(17, 'Егоров', 'Кирилл', 'Мужской', '1986-07-22', null, 5, 5, 62000),
(18, 'Пономарев', 'Денис', 'Мужской', '1989-04-08',
'ponomarev@gmail.com', 6, 6, 108000),
(19, 'Семенова', 'Анна', 'Женский', '1992-02-14', 'semenova@mail.ru',
1, 7, 66000),
(20, 'Федоров', 'Егор', 'Мужской', '1993-09-18', 'f.eg@yandex.ru', 2,
14, 72000),
(21, 'Григорьев', 'Михаил', 'Мужской', '1988-10-25',
'grigoryev@gmail.com', 3, 9, 125000),
(22, 'Миронова', 'Марина', 'Женский', '1995-01-08',
'mironova@mail.ru', 4, 10, 68000),
(23, 'Кудряшов', 'Иван', 'Мужской', '1987-06-20',
'kudryashov@yandex.ru', 5, 11, 80000),
(24, 'Белова', 'Александра', 'Женский', '1986-12-15',
'belova@gmail.com', 6, 12, 115000),
(25, 'Кондратьев', 'Павел', 'Мужской', '1990-12-14', 'prav@mail.ru',
1, 7, 70000),
(26, 'Мельникова', 'Яна', 'Женский', '1993-07-15',
'melnikova@yandex.ru', 2, 8, 80000),
(27, 'Афанасьев', 'Григорий', 'Мужской', '1995-01-08',
'afanasyev@gmail.com', 3, 15, 110000),
(28, 'Макарова', 'Татьяна', 'Женский', '1992-11-18',
'makarova@mail.ru', 4, 10, 65000),
(29, 'Ильина', 'Алёна', 'Женский', '1995-09-30', 'ilina@yandex.ru', 5,
17, 75000),
(30, 'Прокофьев', 'Максим', 'Мужской', '1988-04-05', null, 6, 12,
100000),
(31, 'Семенов', 'Артём', 'Мужской', '1987-08-12', 'semenov@mail.ru',
1, 13, 70000),

(32, 'Карпова', 'Анна', 'Женский', '1990-06-18', 'karp@yandex.ru', 2, 14, 75000),
(33, 'Фомин', 'Дмитрий', 'Мужской', '1994-02-14', 'fomin@gmail.com', 3, 9, 90000),
(34, 'Денисова', 'Вероника', 'Женский', '1997-11-25', 'denisova@mail.ru', 4, 16, 62000),
(35, 'Кузнецова', 'Екатерина', 'Женский', '1991-07-22', 'kuznetsova@yandex.ru', 5, 17, 72000),
(36, 'Макаров', 'Алексей', 'Мужской', '1986-10-08', 'makarov@gmail.com', 6, 18, 115000),
(37, 'Миронов', 'Игорь', 'Мужской', '1988-07-15', 'mironov@mail.ru', 1, 13, 72000),
(38, 'Ковалева', 'Елена', 'Женский', '1995-12-30', 'kovaleva@yandex.ru', 2, 8, 80000),
(39, 'Кудряшова', 'Анастасия', 'Женский', '1990-09-18', 'kudryashova@gmail.com', 3, 15, 100000),
(40, 'Иванова', 'Ольга', 'Женский', '1987-10-25', 'ivanova@mail.ru', 4, 16, 67000),
(41, 'Попова', 'Екатерина', 'Женский', '1993-01-08', 'popova@yandex.ru', 5, 11, 73000),
(42, 'Лебедев', 'Сергей', 'Мужской', '1996-07-14', 'lebedev@gmail.com', 6, 18, 120000),
(43, 'Кузнецов', 'Игорь', 'Мужской', '1985-11-20', null, 1, 7, 75000),
(44, 'Соколов', 'Михаил', 'Мужской', '1989-08-30', 'sokolov@yandex.ru', 2, 14, 85000),
(45, 'Пономарева', 'Максим', 'Женский', '1984-05-28', 'ponomareva@gmail.com', 3, 9, 108000),
(46, 'Семенов', 'Иван', 'Мужской', '1992-06-15', 'semenovi@mail.ru', 4, 10, 64000),
(47, 'Григорьев', 'Павел', 'Мужской', '1987-12-04', 'grigoryev@mail.ru', 5, 11, 77000),
(48, 'Борисова', 'Анна', 'Женский', '1990-10-18', 'borisova@yandex.ru', 6, 18, 95000),
(49, 'Карпов', 'Денис', 'Мужской', '1995-01-20', 'karpov@gmail.com', 1, 13, 68000),
(50, 'Максимов', 'Максим', 'Мужской', '1988-07-28', 'maximov@mail.ru', 2, 14, 102000);

```
-- Таблица с отделами
INSERT INTO departments (id, name_department)
VALUES
(1, 'Отдел маркетинга'),
(2, 'Отдел финансов'),
(3, 'Отдел разработки'),
(4, 'Отдел кадров'),
(5, 'Отдел логистики'),
(6, 'Отдел качества'),
(7, 'Отдел взыскания'),
(8, 'Отдел безопасности'),
(9, 'Отдел поддержки');
```

-- 4 этап: Проверка данных в таблицах

```
SELECT * FROM employees;
SELECT * FROM departments;
```

Теоретические аспекты

База данных

База данных (Database) — это упорядоченный набор структурированной информации или данных, которые обычно хранятся в электронном виде в компьютерной системе. База данных обычно управляется системой управления базами данных ([СУБД](#)). Данные вместе с СУБД, а также приложения, которые с ними связаны, называются системой баз данных, или, для краткости, просто базой данных.

Существует два основных типа баз данных:

- реляционные (базы данных, в которых данные хранятся в виде таблиц, и связаны между собой сквозными параметрами);
- нереляционные (базы данных, в которых не используется табличная схема строк и столбцов, то есть данные не хранятся в виде таблиц).

Данные в наиболее распространенных типах современных баз данных обычно хранятся в [таблицах](#), так как данными в таблицах легко управлять.

Таблица

Таблица (Table) — это набор данных, который состоит из строк и столбцов.

Ниже на рисунке показана структура типовой таблицы:

- зеленым цветом выделена строка (одна запись, которая содержит информацию, относящуюся к определенному объекту или элементу);
- синим цветом выделен столбец или атрибут (значение, которое хранится для каждой строки);
- красным цветом выделено поле (значение, которое находится на пересечении строки и столбца).

id	last_name	first_name	birthday
1	Иванов	Иван	1990-05-15
2	Петров	Петр	1985-12-20
3	Сидорова	Мария	1992-08-10
4	Петров	Петр	1987-04-25

СУБД

Что такое СУБД?

СУБД (Database Management System, DBMS) — это система управления базами данных, которая позволяет создавать новые базы данных и манипулировать их данными ([получение](#), [добавление](#), [обновление](#) и [удаление](#)). Также СУБД предоставляет средства для администрирования, обеспечивает безопасность, надежность и целостность данных в базе данных.

СУБД можно представить, как посредника между базой данных и пользователем.



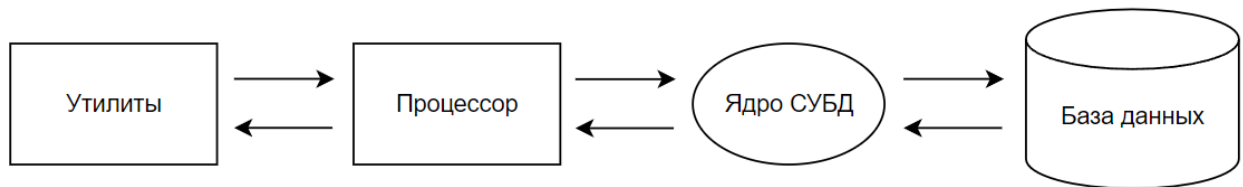
СУБД состоит из набора инструментов, каждый из которых выполняет определенные действия с базой данных. И чтобы все эти инструменты работали синхронно и правильно функционировали, у СУБД должна быть хорошая архитектура.

Главные элементы СУБД:

- утилиты и программные средства (необходимы для того, чтобы пользователь базы данных мог писать запросы, а администратор базы данных осуществлять настройку базы);
- процессор или компилятор (выполняет преобразование SQL-запросов от пользователя в машинный код, а затем возвращает полученный результат в форме, которая понятна пользователю);

- ядро (отвечает за работу всей системы, и через него проходят все процессы, связанные с обработкой данных и их хранения);
- база данных (это место, в котором хранится упорядоченный набор структурированной информации или данных).

Абстрактное представление архитектуры СУБД показано на рисунке ниже.



Основные функции, которыми должна обладать СУБД:

- поддержка языка [SQL](#) (универсальный язык структурированных запросов);
- управление данными во внешней памяти (физические устройства или среда для хранения данных);
- управление данными в оперативной памяти с использованием дискового кэша (промежуточный буфер с быстрым доступом, содержащий информацию, которая может быть запрошена с наибольшей степенью вероятности);
- журналирование изменений (сохранение информации, которая потребуется для восстановления базы данных в предыдущее согласованное состояние в случае логического или физического отказа);
- резервное копирование (процесс создания копии данных на внешнем носителе, который предназначен для восстановления данных в случае их повреждения или полного разрушения);
- восстановление базы данных после сбоев (функция, которая возвращает базу данных в актуальное и консистентное состояние).

Основные требования, которые предъявляются к СУБД:

- разделение данных и программ (данные должны быть отделены от кода программ, и СУБД должна допускать возможность для независимого изменения структуры данных и кода программ);
- высокоуровневый язык запросов (СУБД должна предоставлять универсальное средство для обработки данных, которое не включено в состав какого-либо приложения);

- целостность (СУБД должна предотвращать запись данных, которые нарушают установленные ограничения);
- согласованность (СУБД должна предотвращать появление некорректных данных при параллельной или псевдопараллельной работе нескольких приложений);
- отказоустойчивость (СУБД не должна допускать таких ситуаций, как потеря данных, даже в тех случаях, когда произошёл отказ оборудования);
- защита и разграничение прав доступа (СУБД должна предотвращать несанкционированный доступ к данным и предоставлять пользователю доступ только к тем данным, которые соответствуют правам этого пользователя).

Виды СУБД

Существует очень много видов СУБД, которые отличаются друг от друга своим типом и параметрами. Чтобы в них было проще ориентироваться, рассмотрим их по следующим классификациям:

- распределенность;
- хранение и обработка данных;
- язык запросов;
- структура и организация данных.

Распределенность

Современные системы управления базами данных могут быть одновременно, как локальными, так и распределенными.

- локальные СУБД (всё содержимое базы данных хранится на одном сервере);
- распределенные СУБД (содержимое базы данных частично хранится на разных серверах).

Хранение и обработка данных

По способу хранения и обработки данных, выделяют следующие типы СУБД:

- клиент-серверные (СУБД и база данных находятся на одном сервере, к которому пользователи обращаются с запросами. Получить доступ к СУБД можно с любого компьютера);
- файл-серверные (база данных находится на одном сервере, а СУБД находится на устройствах, с которых отправляются запросы к базе данных. Для получения данных, у пользователя должна быть установлена СУБД);

- встраиваемые (это локальные СУБД, которые являются отдельным модулем для управления данными внутри приложений. Этот тип СУБД в основном реализован в виде дополнительных библиотек для языков программирования).

Язык запросов

Для формирования запросов к базе данных, СУБД могут использовать структурированный и неструктурированный язык запросов:

- SQL (универсальный язык структурированных запросов к базам данных);
- NoSQL (язык для запросов к базе данных основан на другом языке программирования, как например Python).

Структура и организация данных

СУБД также разделяются по способу предоставления информации внутри базы данных:

- реляционные (при использовании такого вида СУБД, данные хранятся в виде таблиц и связаны друг с другом через сквозные параметры и ограничения. Такой подход обеспечивает строгую структуру таблицы и построчное хранение данных);
- документно-ориентированная (в отличие от реляционных СУБД, данный вид СУБД хранит информацию в виде документов, а не в виде таблиц, строк и колонок);
- графовые (в такой СУБД все элементы имеют взаимосвязи в виде графа, и где у каждого узла есть множество связей с другими графами);
- колоночные (в таких СУБД данные хранятся последовательно, в виде одной колонки, и предполагается, что на такой же позиции в каждой колонке будут храниться значения, которые относятся к одной строке).

Язык SQL

SQL (Structured Query Language) — это универсальный язык структурированных запросов, который применяется для управления данными в реляционной базе данных.

Используя только язык SQL, нельзя написать полноценную программу или приложение, так как SQL предназначен для взаимодействия с базой данных. Логика приложения нужно будет написать на другом языке, как например: PL/SQL, Python, Java, Delphi, C++ и т.д.

Особенности языка SQL:

- это язык запросов, а не язык программирования (язык SQL используется в качестве дополнения к языкам программирования, чтобы взаимодействовать с базой данных);
- простая и понятная структура (структура языка SQL достаточно проста и начинающим разработчикам будет легко разобраться в синтаксисе языка);
- SQL не относится к числу запатентованных языков, используемых поставщиками СУБД (почти все современные СУБД поддерживают язык SQL, поэтому знание SQL позволит взаимодействовать с любой базой данных).

Типы данных

Каждая [СУБД](#) имеет свой набор типов данных, которые она поддерживает. Типы данных определяют формат, размер и способы обработки различных типов информации в базе данных. Они могут отличаться по своей структуре, ограничениям и специфическим функциональным возможностям.

В этой книге рассматриваются основные типы данных PostgreSQL.

Числовые типы данных

Официальное руководство по числовым типам данных: [PostgreSQL](#).

Тип	Псевдоним	Описание	Диапазон / Размерность
<code>smallint</code>	<code>int2</code>	Целое число малого диапазона.	От -32768 до +32767. Занимает 2 байта.
<code>integer</code>	<code>int</code> , <code>int4</code>	Целое число.	От -2147483648 до +2147483647. Занимает 4 байта.
<code>bigint</code>	<code>int8</code>	Целое число большого диапазона.	От -9223372036854775808 до 9223372036854775807. Занимает 8 байтов.

Продолжение таблицы с числовыми типами данных.

Тип	Псевдоним	Описание	Диапазон / Размерность
<code>numeric(p,s)</code>	<code>decimal(p,s)</code>	Точное число с заданной точностью.	До 131072 цифр до десятичной точки. До 16383 цифр после запятой.
<code>real</code>	<code>float4</code>	Число одинарной точности с плавающей запятой.	Точность 6 десятичных цифр. Занимает 4 байта.
<code>double precision</code>	<code>float8</code>	Число двойной точности с плавающей запятой.	Точность 15 десятичных цифр. Занимает 8 байт.
<code>smallserial</code>	<code>serial2</code>	Малое автоинкрементное целое число.	От 1 до 32767. Занимает 2 байта.
<code>serial</code>	<code>serial4</code>	Автоинкрементное целое число.	От 1 до 2147483647. Занимает 4 байта.
<code>bigserial</code>	<code>serial8</code>	Большое автоинкрементное число	От 1 до 9223372036854775807. Занимает 8 байт.

Символьные типы данных

Официальное руководство по символьным типам данных: [PostgreSQL](#).

Тип	Псевдоним	Описание	Диапазон / Размерность
<code>character (n)</code>	<code>char (n)</code>	Строка с фиксированным количеством символов.	Строка всегда имеет строго заданный размер <code>n</code> .
<code>character varying (n)</code>	<code>varchar (n)</code>	Строка с переменной длиной.	Отграничивается количеством символов <code>n</code> .
<code>text</code>	Нет	Текст произвольной длины.	Нет ограничений.

Денежные типы данных

Официальное руководство по денежным типам данных: [PostgreSQL](#).

Тип	Псевдоним	Описание	Диапазон / Размерность
<code>money</code>	Нет	Сумма валюты	От -92233720368547758.08 до +92233720368547758.07. Занимает 8 байт.

Двоичные типы данных

Официальное руководство по двоичным типам данных: [PostgreSQL](#).

Тип	Псевдоним	Описание	Диапазон / Размерность
<code>bytea</code>	Нет	Двоичная строка переменной длины.	1 или 4 байта плюс фактическая двоичная строк.

Логические типы данных

Официальное руководство по логическим типам данных: [PostgreSQL](#).

Тип	Псевдоним	Описание	Диапазон / Размерность
boolean	bool	Логическое значение.	true или false. Занимает 1 байт.

Временные типы данных

Официальное руководство по временным типам данных: [PostgreSQL](#).

Тип	Псевдоним	Описание	Диапазон / Размерность
timestamp [(p)] [without time zone]	Нет	Дата и время без часового пояса.	От - 4713 г до н.э., до 294276 г н.э. Занимает 8 байт.
timestamp [(p)] with time zone	Нет	Дата и время с часовым поясом.	От - 4713 г до н.э., до 294276 г н.э. Занимает 8 байт.
date	Нет	Дата (без времени суток).	От 4713 г. до н.э. до 5874897 г н.э. Занимает 4 байта.
time [(p)] [without time zone]	Нет	Время суток (без даты).	От 00:00:00 до 24:00:00. Занимает 8 байт.

Продолжение таблицы с временными типами данных.

Тип	Псевдоним	Описание	Диапазон / Размерность
<code>time [(p)]</code> <code>with time zone</code>	Нет	Время суток (без даты), с часовым поясом.	От 00:00:00+1459 до 24:00:00-1459. Занимает 12 байт.
<code>interval</code> <code>[fields]</code> <code>[(p)]</code>	Нет	Временной интервал.	От -178000000 лет до 178000000 лет. Занимает 16 байт.

Объекты базы данных

В базе данных существуют не только таблицы, но и другие объекты, с которыми вам придётся столкнуться в реальной жизни. Рассмотрим несколько типов объектов:

- таблицы (`TABLE` — это объект базы данных, состоящий из строк и столбцов, в которых хранится информация);
- индексы (`INDEX` — это специальный объект базы данных, который предназначен для повышения производительности работы с данными в таблице);
- ограничения (`CONSTRAINT` - применяются в базе данных для того, чтобы предотвратить вставку некорректных данных в таблицы базы данных);
- представления (`VIEW` — это объект базы данных, который представляет из себя результат выполнения запроса к данным в таблицах базы данных);
- триггеры (`TRIGGERS` — это хранимые процедуры, они содержат в себе код SQL, и запускаются при возникновении какого-либо события, которое относится к указанной таблице);
- процедуры (`PROCEDURE` — это именованная подпрограмма, которая состоит из последовательности SQL-команд, задача которой выполнить указанное действие);
- функции (`FUNCTION` — это именованная подпрограмма, которая состоит из последовательности SQL-команд, задача которой выполнить указанное действие и вернуть определенное значение).

Схема базы данных

Схема (Schema) — это логическая область, которая в большинстве случаев ассоциируется с именем учетной записи и объединяет несколько объектов базы данных. Другими словами, схема выполняет логическую группировку объектов в базе данных.



Имя схемы можно не указывать в том случае, когда в базе данных существует только одна схема или вы обращаетесь к объектам базы данных, которые относятся к вашей схеме.

Имя схемы указывается всегда перед названием объекта базы данных. Например, когда нужно получить данные из таблицы `employees`, которая находится в схеме `hr`, то нужно написать запрос следующим образом.

```
SELECT список_столбцов  
FROM hr.employees;
```

Ниже на рисунке продемонстрирован пример с использованием схемы базы данных. В левой части изображения приведён список таблиц, и не совсем понятно, кто владелец той или иной таблицы. А в правой части, приведен тот же список таблиц, но уже с указанием схемы, и теперь стало понятно, кто является владельцем таблицы.

Таблицы без указания схемы

<code>cars</code>
<code>music</code>
<code>images</code>
<code>apps</code>

Таблицы с указанием схемы

<code>ivanov.cars</code>
<code>petrov.music</code>
<code>ivanov.images</code>
<code>sidorov.apps</code>

Порядок выполнения SQL-запроса

Порядок выполнения SQL запроса — это фактическая последовательность, в которой механизм [СУБД](#) обрабатывает различные конструкции SQL-запроса. Говоря другими

словами, порядок выполнения SQL-запроса это не то же самое, что порядок конструкций (блоков) в запросе. Придерживаясь определенного порядка выполнения запроса, ядро СУБД может свести к минимуму дисковые операции по вводу/выводу данных.

Возьмем следующий SQL-запрос и посмотрим, как он будет выполняться:

```
SELECT d.name_department,  
       COUNT(e.id) AS cnt_employees,  
       SUM(e.salary) AS sum_salary  
FROM employees e  
JOIN departments d  
ON d.id = e.id_department  
WHERE e.gender = 'Мужской'  
GROUP BY d.name_department  
HAVING sum(e.salary) >= 205000  
ORDER BY d.name_department ASC  
LIMIT 3;
```

Порядок выполнения запроса:

1. **Блок FROM:** определяется список таблиц, которые будут задействованы в запросе. В данном случае это `employees` и `departments`;
2. **Блок JOIN:** выполняется операция объединения таблиц по условию в блоке `ON`. В данном случае это условие `d.id = e.id_department`, которое связывает две таблицы по идентификатору отдела;
3. **Блок WHERE:** применяется условие фильтрации к объединенным таблицам. В данном случае это условие по отбору сотрудников мужского пола `e.gender = 'Мужской'`;
4. **Блок GROUP BY:** выполняется группировка строк по названию отдела, это столбец `d.name_department`;
5. **Блок HAVING:** выполняется фильтрация групп по условию `sum(e.salary) >= 205000`, оно означает, что сумма заработной платы всех сотрудников отдела должна быть больше или равна 205 000 рублей;
6. **Блок SELECT:** выводятся столбцы и агрегатные функции из каждой группы. В данном случае это столбец `d.name_department`, и агрегатные функции `COUNT(e.id) AS cnt_employees` и `SUM(e.salary) AS sum_salary`;
7. **Блок ORDER BY:** сортировка полученного результирующего набора по столбцу `d.name_department` в порядке возрастания `ASC`;

8. **Оператор LIMIT:** выполняется ограничение результирующего набора, то есть выводятся не все результаты запроса, а только указанное количество строк. В данном случае на экран выводится только 3 строки.

Типы команд в SQL

Команды в SQL разделяются на несколько основных групп: DDL, DML, DCL и TCL.

Команды DDL

Команды DDL (Data Definition Language) — позволяют пользователю манипулировать объектами в базе данных, под манипуляцией понимается создание и удаление объектов. Список команд DDL:

- [CREATE TABLE;](#)
- [ALTER TABLE;](#)
- [TRUNCATE TABLE;](#)
- [DROP.](#)

Команды DML

Команды DML (Data Manipulation Language) — позволяют пользователю выполнять действия над данными, то есть манипулировать ими. Список команд DML:

- [SELECT;](#)
- [INSERT;](#)
- [UPDATE;](#)
- [DELETE;](#)
- [MERGE.](#)

Команды DCL

Команды DCL (Data Control Language) — позволяют установить необходимые права доступа для объектов базы данных, с которыми будут работать пользователи. Список команд DCL:

- [GRANT;](#)
- [REVOKE;](#)
- [DENY.](#)

Команды TCL

Команды TCL (Transaction Control Language) — необходимы для управления транзакциями. Список основных команд TCL:

- [BEGIN;](#)
- [COMMIT;](#)
- [ROLLBACK;](#)
- [SAVEPOINT;](#)
- [SET TRANSACTION.](#)

Основные моменты

Комментарии

Комментарии очень полезный инструмент, который позволяет добавлять описание к важным функциональным участкам кода, а также отключать участки кода при отладке или тестировании.

Существует два вида комментариев:

- однострочные (применяются тогда, когда нужно закомментировать одну строку);
- многострочные (применяются в том случае, когда нужно закомментировать несколько строк).

Синтаксис однострочных комментариев:

Однострочные комментарий начинаются с двух символов тире (--), и после этих символов должен идти текст комментария или код, который нужно закомментировать.

```
-- Получаем все данные из таблицы employees  
SELECT *  
FROM employees;
```

Синтаксис многострочных комментариев:

Многострочные комментарии начинаются с символов /* и заканчиваются символами */, а между этими символами должен располагаться текст комментария или код.

```
/* Получаем все данные  
   из таблицы employees */  
SELECT *  
FROM employees;
```

Разделитель команд (;)

SQL код представляет собой последовательность команд, которые определяются логически, а не физически. Другими словами, их границы определяются не физическим завершением строки кода, а специальным символом завершения - точка с запятой (;).

Поэтому каждый SQL-запрос и каждая исполняемая команда должна завершаться символом - точка с запятой (;), чтобы ядро [СУБД](#) понимало, где заканчивается текущая команда и начинается новая.

Если выполнить следующие два запроса вместе, то будет получена ошибка, так как ядро СУБД не понимает, где завершилась первая команда и начинается вторая.

```
select * from employees
select * from departments
```

А теперь проставляем символ для разделения команд и снова пробуем выполнить эти запросы. Теперь ядро СУБД понимает, где завершилась первая команда и начинается вторая, поэтому никаких ошибок получено не будет.

```
select * from employees;
select * from departments;
```

Отображение структуры таблицы

Можно вывести на экран структуру любой таблицы при помощи графического интерфейса СУБД или при помощи SQL-запроса, который выводит информацию о структуре таблицы из системного каталога информационной схемы.



К сожалению, для отображения структуры таблицы в разных СУБД используются разные команды, например в Oracle и MySQL используется команда `DESCRIBE имя_таблицы;`

SQL-запрос:

```
SELECT table_schema,
       table_name,
       column_name,
       data_type,
       character_maximum_length,
       is_nullable
FROM information_schema.columns
WHERE table_schema = 'имя_схемы'
      AND table_name = 'имя_таблицы';
```

Практический пример: необходимо вывести структуру таблицы `employees`.

```
SELECT table_schema,  
       table_name,  
       column_name,  
       data_type,  
       character_maximum_length,  
       is_nullable  
FROM information_schema.columns  
WHERE table_schema = 'public'  
       AND table_name = 'employees';
```

Выполняем запрос и получаем следующий результат.

	ABC table_schema ▼	ABC table_name ▼	ABC column_name ▼	ABC data_type ▼	123 character_maximum_length ▼	ABC is_nullable ▼
Таблица	1 public	employees	salary	integer	[NULL]	YES
	2 public	employees	birthday	date	[NULL]	NO
	3 public	employees	id_department	integer	[NULL]	YES
	4 public	employees	id_boss	integer	[NULL]	YES
Текст	5 public	employees	id	integer	[NULL]	NO
	6 public	employees	last_name	character varying	100	NO
	7 public	employees	first_name	character varying	100	NO
	8 public	employees	gender	character varying	15	NO
Запись	9 public	employees	email	character varying	100	YES

Зарезервированные слова

В PostgreSQL существует список зарезервированных ключевых слов, которые нельзя использовать в качестве названий таблиц, столбцов, функций, процедур и т.д.



Под «нельзя использовать» подразумевается, что нельзя присваивать объекту базы данных имя, состоящее из одного зарезервированного слова, но если использовать их в комбинации с другими словами, то проблем не возникнет.

Полный список можно найти на страницах официального руководства [PostgreSQL](#).

ALL	CREATE	FOREIGN	LIKE	RETURNING
ANALYSE	CROSS	FREEZE	LIMIT	RIGHT
ANALYZE	CURRENT_DATE	FROM	LOCALTIME	SELECT

AND	CURRENT_ROLE	FULL	LOCALTIMESTAMP	SESSION_USER
ANY	CURRENT_TIME	GRANT	NATURAL	SIMILAR
ARRAY	CURRENT_TIMESTAMP	GROUP	NOT	SOME
AS	CURRENT_USER	HAVING	NOTNULL	SYMMETRIC
ASC	DEFAULT	ILIKE	NULL	TABLE
AUTHORIZATION	DEFERRABLE	IN	OFFSET	THEN
BETWEEN	DESC	INITIALLY	ON	TO
BINARY	DISTINCT	INNER	ONLY	TRAILING
BOTH	DO	INTERSECT	OR	TRUE
CASE	ELSE	INTO	ORDER	UNION
CAST	END	IS	OUTER	UNIQUE
CHECK	EXCEPT	ISNULL	OVERLAPS	USER
COLLATE	FALSE	JOIN	PLACING	USING
COLUMN	FETCH	LEADING	PRIMARY	VERBOSE
CONSTRAINT	FOR	LEFT	REFERENCES	WHEN

Регистр в операторах и функциях

В SQL, регистр в имени операторов и функций не имеет значения при выполнении запросов. Это означает, что можно использовать как прописные, так и строчные буквы при написании операторов и функций.

Например, операторы `SELECT`, `select` и `SeLeCT` будут интерпретированы одинаково и выполняться согласно правилам SQL. Регистронезависимость позволяет разработчикам писать SQL-код в том стиле, который они предпочитают. Но лучше придерживаться общепринятых правил и писать операторы и функции в верхнем регистре.

Оператор CREATE TABLE

Любая [таблица](#) в базе данных должна создаваться по определенным правилам, давайте рассмотрим самые основные:

- создаваемая таблица должна иметь уникальное имя, в рамках создаваемой [схемы](#) (логическая область, которая ассоциируется с именем учетной записи и объединяет несколько [объектов базы данных](#));
- у таблицы всегда определено количество столбцов (колонок), и оно варьируется в диапазоне, от 0 до 1024;
- имя столбца (колоники) должно быть уникальным, но уже в пределах создаваемой таблицы (но имя столбца не должно совпадать со списком [зарезервированных слов](#));
- каждому столбцу должен быть присвоен [тип данных](#);
- в создаваемой таблице может быть неограниченное количество строк (в большинстве случаев оно определяется дисковым пространством);
- для данных в таблице можно создавать ограничения.

В таблице могут быть определены [первичные ключи](#), [внешние ключи](#) и установлены дополнительные [ограничения](#).

Классический способ создания таблицы

Создание таблицы классическим способом начинается с оператора `CREATE TABLE`, а затем перечисляются необходимые столбцы, типы данных и устанавливаются ограничения для столбцов таблицы.



В этой главе не будут рассматриваться такие темы, как [первичные ключи](#), [внешние ключи](#) и [ограничения](#).

Синтаксис:

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `column_name` – имя столбца в таблице;
- `data_type` – тип данных для столбца;

- [NOT NULL | NULL] – ограничение, которое отвечает за то, может ли значение столбца быть NULL или нет.

```
CREATE TABLE schema.table_name (
    [column_name_1] [data_type] [NOT NULL | NULL],
    ...
    [column_name_n] [data_type] [NOT NULL | NULL]
);
```

Практический пример: необходимо создать таблицу `list_users`, в которой будет храниться информация о сотрудниках, а именно:

Имя столбца	Тип данных	Описание столбца
id	integer	Идентификатор пользователя
first_name	varchar(30)	Имя
last_name	varchar(40)	Фамилия
gender	varchar(10)	Пол
age	integer	Возраст
email	varchar(80)	Электронный адрес

Код для создания таблицы `list_users`:

```
CREATE TABLE list_users (
    id integer,
    first_name varchar(30),
    last_name varchar(40),
    gender varchar(10),
    age integer,
    email varchar(80)
);
```

Выполняем запрос и получаем информационное сообщение, что таблица `list_users` успешно создана. В дальнейшем, таблица `list_users` использоваться не будет, поэтому удалим её при помощи оператора [DROP](#).

```
DROP TABLE IF EXISTS list_users;
```

Создание таблицы при помощи оператора SELECT

Создать таблицу можно при помощи оператора [SELECT](#), это удобно, когда нужно создать временную таблицу. Создать таблицу при помощи оператора `SELECT` можно двумя способами.

1 способ

Можно скопировать только структуру таблицы (без переноса данных). Чтобы это осуществить, необходимо добавить условие `1=0` в блок `WHERE`.

Синтаксис:

- `schema` – наименование схемы, в которой находится объект;
- `new_table_name` – имя таблицы;
- `column_list` – список столбцов в таблице;
- `table_name` – таблица, из которой будут получены данные.

```
CREATE TABLE schema.new_table_name AS
SELECT column_list
FROM schema.table_name
WHERE 1=0;
```

Практический пример: необходимо скопировать структуру таблицы `employees` в новую таблицу `employees_temp` с тремя столбцами `id`, `last_name` и `first_name`.

Пояснение к SQL-запросу:

- в блоке `CREATE TABLE` указываем имя будущей таблицы;
- в блоке `SELECT` перечисляем список нужных столбцов;
- в блоке `FROM` указываем имя таблицы, из которой будут получены данные;
- в блоке `WHERE` указываем условие `1=0`, чтобы скопировать только структуру таблицы.

```
CREATE TABLE employees_temp AS
SELECT id, last_name, first_name
FROM employees
WHERE 1=0;
```

Таблица `employees_temp` создана, теперь нужно вывести данные таблицы на экран.

```
SELECT *  
FROM employees_temp;
```

Обратите внимание, что данных в таблице `employees_temp` нет, но структура таблицы создана на основании таблицы `employees`.

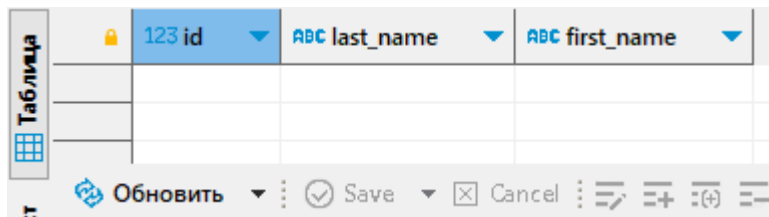


Таблица	123 id	ABC last_name	ABC first_name
ст			

В дальнейшем, таблица `employees_temp` использоваться не будет. Поэтому удаляем её при помощи оператора [DROP](#).

```
DROP TABLE IF EXISTS employees_temp;
```

2 способ

Можно скопировать структуру таблицы и её данные, и добавлять условие `1=0` блок `WHERE` уже НЕ нужно.



При создании таблицы способом `CREATE TABLE AS` обратите внимание, что столбец не может хранить значение `NULL`.

Синтаксис:

- `schema` – наименование схемы, в которой находится объект;
- `new_table_name` – имя таблицы;
- `column_list` – список столбцов в таблице;
- `table_name` – таблица, из которой будут получены данные.

```
CREATE TABLE schema.new_table_name AS  
SELECT column_list  
FROM schema.table_name;
```


Практический пример: необходимо скопировать данные и структуру таблицы `employees` в новую таблицу `employees_temp` с тремя столбцами `id`, `last_name` и `first_name`.

Пояснение к SQL-запросу:

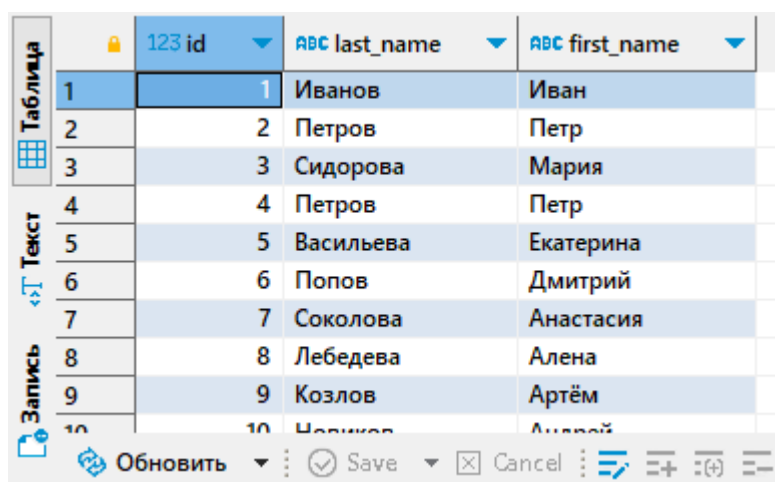
- в блоке `CREATE TABLE` указываем имя будущей таблицы;
- в блоке `SELECT` перечисляем список нужных столбцов;
- в блоке `FROM` указываем имя таблицы, из которой будут получены данные.

```
CREATE TABLE employees_temp AS  
SELECT id, last_name, first_name  
FROM employees;
```

Таблица `employees_temp` создана, теперь нужно вывести данные таблицы на экран.

```
SELECT *  
FROM employees_temp;
```

Выполняем запрос и получаем список сотрудников. Обратите внимание, что результат запроса на изображении показан не полностью.



	123 id	ABC last_name	ABC first_name
1	1	Иванов	Иван
2	2	Петров	Петр
3	3	Сидорова	Мария
4	4	Петров	Петр
5	5	Васильева	Екатерина
6	6	Попов	Дмитрий
7	7	Соколова	Анастасия
8	8	Лебедева	Алена
9	9	Козлов	Артём
10	10	Михайлов	Андрей

В дальнейшем, таблица `employees_temp` использоваться не будет. Поэтому удаляем её при помощи оператора [`DROP`](#).

```
DROP TABLE IF EXISTS employees_temp;
```

Оператор ALTER TABLE

При помощи оператора `ALTER TABLE` можно добавлять, удалять, модифицировать и переименовывать столбцы и таблицы, а также работать с [первичными](#) и [внешними](#) ключами.



По завершению изучения данной главы, восстановите данные в таблицах `employees` и `departments`. Для этого необходимо выполнить весь код из главы – [Тестовые данные для работы](#).

Добавление столбца в таблицу

Чтобы добавить новый столбец в таблицу, нужно использовать оператор `ALTER TABLE` с дополнительным параметром `ADD COLUMN`.

Синтаксис:

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `column_name` – имя нового столбца;
- `data_type` – тип данных нового столбца.

```
-- Добавление одного столбца в таблицу
ALTER TABLE schema.table_name
ADD COLUMN column_name data_type;
```

```
-- Добавление нескольких столбцов в таблицу
ALTER TABLE имя_схемы.имя_таблицы
ADD COLUMN column_name_1 data_type,
..
ADD COLUMN column_name_n data_type;
```

Практический пример: в таблицу `employees` необходимо добавить два дополнительных столбца:

Имя столбца	Тип данных
address	varchar(500)
zip_code	integer

Пояснение к SQL-запросу:

- в блоке `ALTER TABLE` указываем имя таблицы, в которую нужно добавить дополнительные столбцы;
- в блоке `ADD COLUMN` указываем имена столбцов и их тип данных.

```
ALTER TABLE employees
ADD COLUMN address varchar(500),
ADD COLUMN zip_code integer;
```

Выводим [структуру таблицы](#) `employees` на экран и видим, что столбцы `address` и `zip_code` были успешно добавлены.

	ABC table_schema	ABC table_name	ABC column_name	ABC data_type	123 character_maximum_length	ABC is_nullable
1	public	employees	id	integer	[NULL]	NO
2	public	employees	birthday	date	[NULL]	NO
3	public	employees	id_department	integer	[NULL]	YES
4	public	employees	id_boss	integer	[NULL]	YES
5	public	employees	salary	integer	[NULL]	YES
6	public	employees	zip_code	integer	[NULL]	YES
7	public	employees	last_name	character varying	100	NO
8	public	employees	first_name	character varying	100	NO
9	public	employees	gender	character varying	15	NO
10	public	employees	address	character varying	500	YES
11	public	employees	email	character varying	100	YES

Изменение типа данных столбца в таблице

Чтобы изменить тип данных столбца в таблице, нужно использовать оператор `ALTER TABLE` с дополнительным параметром `ALTER COLUMN`.



Тип данных столбца будет изменен в том случае, когда каждое значение в столбце можно будет преобразовать в новый тип данных.

Синтаксис:

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `column_name` – имя столбца, для которого будет установлен новый тип данных;
- `data_type` – новый тип данных для столбца.

```
ALTER TABLE schema.table_name
ALTER COLUMN column_name TYPE data_type;
```

Практический пример: в таблице `employees` необходимо обновить тип данных у столбца `email`:

Имя столбца	Старый тип данных	Новый тип данных
email	varchar(100)	text

Пояснение к SQL-запросу:

- в блоке `ALTER TABLE` указываем имя таблицы, в которой нужно обновить тип данных у столбца;
- в блоке `ALTER COLUMN` указываем имя столбца и его новый тип данных.

```
ALTER TABLE employees
ALTER COLUMN email TYPE text;
```

Выводим [структуру таблицы](#) `employees` на экран и видим, что у столбца `email` был обновлен тип данных.

	ABC table_schema	ABC table_name	ABC column_name	ABC data_type	123 character_maximum_length	ABC is_nullable
1	public	employees	id	integer	[NULL]	NO
2	public	employees	birthday	date	[NULL]	NO
3	public	employees	id_department	integer	[NULL]	YES
4	public	employees	id_boss	integer	[NULL]	YES
5	public	employees	salary	integer	[NULL]	YES
6	public	employees	zip_code	integer	[NULL]	YES
7	public	employees	last_name	character varying	100	NO
8	public	employees	first_name	character varying	100	NO
9	public	employees	gender	character varying	15	NO
10	public	employees	address	character varying	500	YES
11	public	employees	email	text	[NULL]	YES

Изменение имени столбца в таблице

Изменить имя столбца в таблице можно при помощи оператора `ALTER TABLE` с добавлением дополнительного параметра `RENAME COLUMN`.

Синтаксис:

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `old_name` – старое имя столбца;
- `new_name` – новое имя столбца.

```
ALTER TABLE schema.table_name  
RENAME COLUMN old_name TO new_name;
```

Практический пример: в таблице `employees` необходимо изменить имя столбца `email` на `email_address`.

Пояснение к SQL-запросу:

- в блоке `ALTER TABLE` указываем имя таблицы, в которой нужно обновить имя столбца;
- в блоке `RENAME COLUMN` указываем сначала старое название столбца, а затем новое.

```
ALTER TABLE employees  
RENAME COLUMN email TO email_address;
```

Выводим [структуру таблицы](#) `employees` на экран и видим, что у столбца `email` было изменено название на `email_address`.

	ABC table_schema	ABC table_name	ABC column_name	ABC data_type	123 character_maximum_length	ABC is_nullable
1	public	employees	id	integer	[NULL]	NO
2	public	employees	birthday	date	[NULL]	NO
3	public	employees	id_department	integer	[NULL]	YES
4	public	employees	id_boss	integer	[NULL]	YES
5	public	employees	salary	integer	[NULL]	YES
6	public	employees	zip_code	integer	[NULL]	YES
7	public	employees	last_name	character varying	100	NO
8	public	employees	first_name	character varying	100	NO
9	public	employees	gender	character varying	15	NO
10	public	employees	address	character varying	500	YES
11	public	employees	email_address	text	[NULL]	YES

Обновить Save Cancel Экспорт данных ... 200 11 11 строк п

Удаление столбца в таблице

Чтобы удалить столбец в таблице, нужно использовать оператор `ALTER TABLE` с дополнительным параметром `DROP COLUMN`.

Синтаксис:

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `column_name` – имя столбца.

-- Удаление одного столбца в таблице

```
ALTER TABLE schema.table_name  
DROP COLUMN column_name;
```

-- Удаление нескольких столбцов в таблице

```
ALTER TABLE schema.table_name  
DROP COLUMN column_name_1,  
..  
DROP COLUMN column_name_n;
```

Практический пример: в таблице `employees` необходимо удалить столбцы `address` и `zip_code`.

Пояснение к SQL-запросу:

- в блоке `ALTER TABLE` указываем имя таблицы, в которой нужно удалить столбцы;
- в блоке `DROP COLUMN` указываем имена столбцов для удаления.

```
ALTER TABLE employees  
DROP COLUMN address,  
DROP COLUMN zip_code;
```

Выводим [структуру таблицы](#) `employees` на экран и видим, что столбцы `address` и `zip_code` были удалены.

	ABC table_schema	ABC table_name	ABC column_name	ABC data_type	123 character_maximum_length	ABC is_nullable
Таблица	public	employees	salary	integer	[NULL]	YES
	public	employees	birthday	date	[NULL]	NO
	public	employees	id_department	integer	[NULL]	YES
	public	employees	id_boss	integer	[NULL]	YES
Текст	public	employees	id	integer	[NULL]	NO
	public	employees	last_name	character varying	100	NO
	public	employees	first_name	character varying	100	NO
Запись	public	employees	gender	character varying	15	NO
	public	employees	email_address	text	[NULL]	YES

Обновить Save Cancel Экспорт данных ... 200 9 9 строк по

Изменение имени таблицы

Чтобы изменить имя таблицы, нужно использовать оператор `ALTER TABLE` с дополнительным параметром `RENAME TO`.

Синтаксис:

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – старое имя таблицы;
- `new_table_name` – новое имя таблицы.

```
ALTER TABLE schema.table_name  
RENAME TO new_table_name;
```

Практический пример: необходимо изменить имя таблицы `employees` на `list_users`.

Пояснение к SQL-запросу:

- в блоке `ALTER TABLE` указываем имя таблицы, у которой нужно изменить имя;
- в блоке `RENAME TO` указываем новое имя таблицы.

```
ALTER TABLE employees  
RENAME TO list_users;
```

Имя таблицы было успешно обновлено и теперь, чтобы получить данные, нужно обращаться к таблице `list_users`.

В дальнейшем, таблица `list_users` использоваться не будет, поэтому удалим её при помощи оператора [DROP](#).

```
DROP TABLE IF EXISTS list_users;
```

Первичные и внешние ключи

Первичные и внешние ключи в SQL — это очень важная тема, так что будьте внимательны при изучении.



После изучения данной главы, выполните удаление всех созданных таблиц при помощи оператора [DROP](#).

Первичные ключи (Primary key)

Первичным ключом (Primary key, PK) в таблице обозначается столбец, который содержит набор уникальных значений, по которым можно однозначно идентифицировать строку в рамках таблицы. Первичный ключ не может содержать пустые значения, поскольку всегда имеет ограничения [NOT NULL](#).

В качестве первичного ключа может выступать не только один столбец, их может быть два и более. Первичные ключи необходимы для поддержания целостности базы данных.

Создать первичный ключ можно двумя способами:

- в момент создания таблицы при помощи оператора [CREATE TABLE](#);
- после создания таблицы при помощи оператора [ALTER TABLE](#).

Создание Primary key при помощи CREATE TABLE

Для создания первичного ключа при помощи оператора `CREATE TABLE` необходимо использовать ключевое слово `CONSTRAINT`.

Синтаксис:

- `schema` — наименование схемы, в которой находится объект;
- `table_name` — имя таблицы;
- `[NOT NULL | NULL]` — ограничение, которое отвечает за то, может ли значение столбца быть `NULL` или нет;
- `name_pk` — имя будущего первичного ключа;

- `column_list` – столбец или список столбцов через запятую, по которым будет создан первичный ключ.

```
CREATE TABLE schema.table_name (
    [column_name_1] [data_type] [NOT NULL | NULL],
    ...
    [column_name_n] [data_type] [NOT NULL | NULL],
    CONSTRAINT name_pk PRIMARY KEY(column_list)
);
```

Практический пример: необходимо создать таблицу `list_users` с первичным ключом `pk_users_id` по столбцу `id`.

Структура таблицы `list_users`:

Имя столбца	Тип данных	Ограничения	Primary key
<code>id</code>	<code>integer</code>	<code>NOT NULL</code>	PK
<code>last_name</code>	<code>varchar(30)</code>		
<code>first_name</code>	<code>varchar(30)</code>		

Пояснение к SQL-запросу:

- в блоке `CREATE TABLE` указываем имя будущей таблицы;
- в блоке `CONSTRAINT` указываем имя первичного ключа и имя столбца, по которому он будет создан.

```
CREATE TABLE list_users (
    id INTEGER NOT NULL,
    last_name VARCHAR(30),
    first_name VARCHAR(30),
    CONSTRAINT pk_users_id PRIMARY KEY(id)
);
```

После выполнения запроса будет создана таблица `list_users` с первичным ключом `pk_users_id` по столбцу `id`.

Удаляем таблицу `list_users` при помощи оператора [DROP](#), так как в дальнейшем она будет создана заново.

```
DROP TABLE IF EXISTS list_users;
```

Создание Primary key при помощи ALTER TABLE

Чтобы создать первичный ключ в таблице, нужно использовать оператор `ALTER TABLE` с дополнительным параметром `ADD CONSTRAINT`.

Синтаксис:

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `name_pk` – имя первичного ключа;
- `column_list` – столбец или список столбцов через запятую, по которым будет создан первичный ключ.

```
ALTER TABLE schema.table_name  
ADD CONSTRAINT name_pk PRIMARY KEY(column_list);
```

Практический пример: необходимо сначала создать таблицу `list_users`, а затем создать первичный ключ `pk_users_id` по столбцу `id` при помощи оператора `ALTER TABLE`.

Структура таблицы `list_users`:

Имя столбца	Тип данных	Ограничения	Primary key
<code>id</code>	<code>integer</code>	<code>NOT NULL</code>	PK
<code>last_name</code>	<code>varchar(30)</code>		
<code>first_name</code>	<code>varchar(30)</code>		

Код для создания таблицы `list_users`:

```
CREATE TABLE list_users (  
    id INTEGER NOT NULL,  
    last_name VARCHAR(30),  
    first_name VARCHAR(30)
```

);

Создаем первичный ключ в таблице `list_users`.

Пояснение к SQL-запросу:

- в блоке `ALTER TABLE` указываем имя таблицы, в которой нужно создать первичный ключ;
- в блоке `ADD CONSTRAINT` указываем имя первичного ключа `pk_users_id` и столбец `id` по которому будет создан ключ.

```
ALTER TABLE list_users  
ADD CONSTRAINT pk_users_id PRIMARY KEY(id);
```

После выполнения запроса в таблице `list_users` будет создан первичный ключ `pk_users_id` по столбцу `id`.

Проверка существования Primary key

Проверить существование первичного ключа в таблице можно при помощи графического интерфейса СУБД или SQL-запроса, который выводит информацию из информационной схемы.

SQL-запрос:

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

```
SELECT t.constraint_schema,  
       t.table_name,  
       t.constraint_name  
FROM information_schema.table_constraints t  
WHERE t.table_name = 'table_name'  
      AND t.table_schema = 'schema'  
      AND t.constraint_type = 'PRIMARY KEY';
```

Практический пример: необходимо проверить существование первичного ключа в таблице `list_users`.

Пояснение к SQL-запросу:

- в блоке WHERE для столбца table_schema указываем значение public, а для table_name значение list_users. Обратите внимание, что имя схемы может отличаться.

```
SELECT t.constraint_schema,
       t.table_name,
       t.constraint_name
FROM information_schema.table_constraints t
WHERE t.table_name = 'list_users'
      AND t.table_schema = 'public'
      AND t.constraint_type = 'PRIMARY KEY';
```

Выполняем запрос и получаем результат, в котором видно, что в таблице list_users есть первичный ключ pk_users_id.

Таблица	ABC constraint_schema	ABC table_name	ABC constraint_name
1	public	list_users	pk_users_id

Суть работы Primary key

Чтобы продемонстрировать работу первичного ключа (Primary key), воспользуемся ранее созданной таблицей list_users.

Заполняем таблицу list_users данными при помощи оператора [INSERT](#):

```
INSERT INTO list_users(id, last_name, first_name)
VALUES (1, 'Иванов', 'Иван'),
       (2, 'Сидоров', 'Виктор'),
       (3, 'Карпов', 'Дмитрий');
```

Выводим данные таблицы list_users на экран.

```
SELECT *
FROM list_users;
```

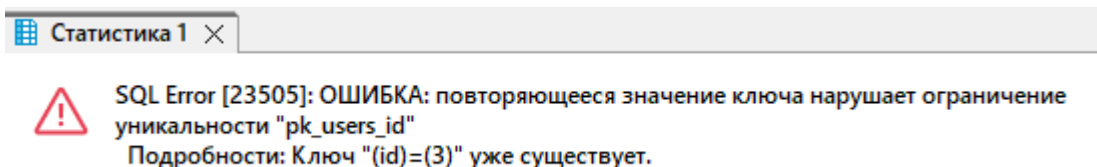
Обратите внимание, что значения в столбце id уникальные и не повторяются.

Таблица		123 id	ABC last_name	ABC first_name
	1	1	Иванов	Иван
	2	2	Сидоров	Виктор
	3	3	Карпов	Дмитрий
Текст				
Обновить Save Cancel				

А теперь попробуем добавить ещё одну запись в таблицу `list_users`, и укажем в столбце `id` уже существующее значение.

```
INSERT INTO list_users(id, last_name, first_name)
VALUES (3, 'Дунаев', 'Максим');
```

После выполнения оператора `INSERT INTO`, будет получена ошибка (повторяющееся значение ключа нарушает ограничение уникальности ключа `pk_users_id`) и запись добавлена не будет.



Это означает, что при добавлении записи в таблицу `list_users`, была нарушена уникальность первичного ключа `pk_users_id`. Другими словами, мы пытаемся добавить запись в таблицу со значением, которое было уже определено, как уникальное.

Удаление Primary key

Чтобы удалить первичный ключ в таблице, нужно использовать оператор `ALTER TABLE` с дополнительным параметром `DROP CONSTRAINT`.

Синтаксис:

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `name_pk` – имя первичного ключа.

```
ALTER TABLE schema.table_name
DROP CONSTRAINT name_pk;
```

Практический пример: необходимо удалить первичный ключ `pk_users_id` из таблицы `list_users`.

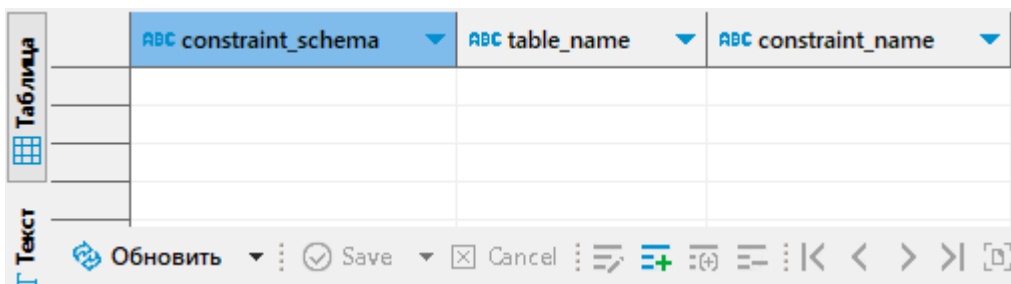
Пояснение к SQL-запросу:

- в блоке `ALTER TABLE` указываем имя таблицы, в которой нужно удалить первичный ключ;
- в блоке `DROP CONSTRAINT` указываем имя первичного ключа;

```
ALTER TABLE list_users  
DROP CONSTRAINT pk_users_id;
```

Выполняем запрос, а затем проверяем уже известным нам [способом](#) действительно ли был удален первичный ключ из таблицы `list_users`.

Запрос вернул результат, в котором нет ни одной строки, а это означает, что первичный ключ был удален.



В дальнейшем, таблица `list_users` использоваться не будет, поэтому удалим её при помощи оператора [DROP](#).

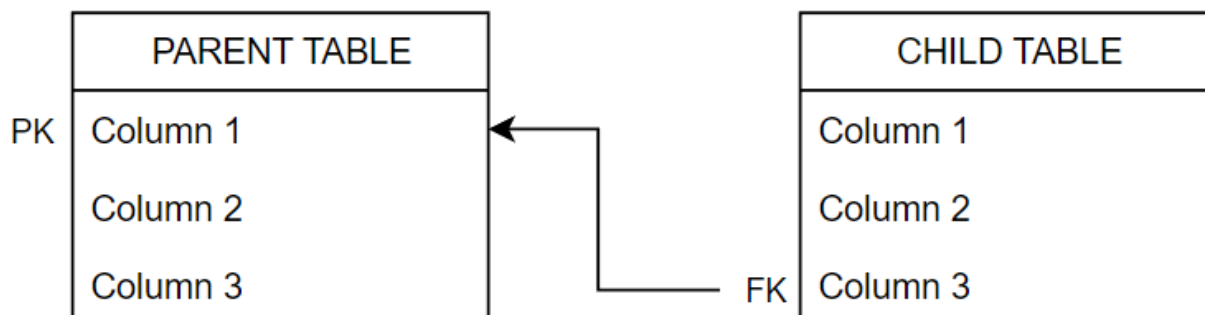
```
DROP TABLE IF EXISTS list_users;
```

Внешние ключи (Foreign key)

Внешним ключом (Foreign key, FK) в таблице обозначаются столбцы, которые позволяют установить связь с другой таблицей. Другими словами, в этих столбцах есть данные, благодаря которым можно осуществить установить связь с данными в других таблицах.

Внешний ключ обеспечивает целостность ваших данных в базе данных. Также `Foreign key` означает, что значения одной таблицы должны в обязательном порядке появиться в другой таблице.

Таблица, на которую ссылаются, называется - родительской таблицей (parent table), а таблица с внешним ключом, называется - дочерней таблицей (child table). Внешний ключ в дочерней таблице, обычно ссылается на [первичный ключ](#) (primary key) в родительской таблице.



Создать внешний ключ можно двумя способами:

- в момент создания таблицы при помощи оператора [CREATE TABLE](#);
- после создания таблицы при помощи оператора [ALTER TABLE](#).

Создание Foreign key при помощи CREATE TABLE

Для создания внешнего ключа при помощи оператора `CREATE TABLE` необходимо использовать ключевое слово `CONSTRAINT`.

Синтаксис:

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `column_name` – имя столбца;
- `data_type` – тип данных столбца;
- `[NOT NULL | NULL]` – ограничение, которое отвечает за то, может ли значение столбца быть `NULL` или нет;
- `name_fk` – имя будущего внешнего ключа;
- `column_name_fk` – имя столбца, для которого создается внешний ключ;
- `ref_table_name` – имя внешней таблицы;
- `ref_column_name` – имя столбца внешней таблицы;
- `ON DELETE CASCADE` – при удалении родительской строки все дочерние строки в дочерней таблице будут автоматически удалены. Данный параметр будет рассмотрен в разделе - [Каскадное удаление Foreign key](#);

- ON DELETE SET NULL – при удалении родительской строки, значение в столбце дочерней таблицы, ссылающемся на родительский столбец, будет установлено в NULL. Данный параметр будет рассмотрен в разделе - [Установка для столбца Foreign key значения NULL](#).

```
CREATE TABLE schema.table_name (
    [column_name_1] [data_type] [NOT NULL | NULL],
    ...
    [column_name_n] [data_type] [NOT NULL | NULL],
    CONSTRAINT name_fk
    FOREIGN KEY (column_name_fk)
    REFERENCES ref_table_name(ref_column_name)
    [ON DELETE CASCADE | ON DELETE SET NULL];
);
```

Практический пример: необходимо создать две таблицы `transport_company` и `product`, а затем связать их между собой при помощи первичного и внешнего ключа.

Структура таблицы `transport_company` (транспортные компании):

Имя столбца	Тип данных	Ограничения	Primary key
id	integer	NOT NULL	PK
name	varchar(100)	NOT NULL	

Код для создания таблицы `transport_company`:

```
CREATE TABLE transport_company (
    id INTEGER NOT NULL,
    name VARCHAR(100) NOT NULL,
    CONSTRAINT transport_company_pk PRIMARY KEY(id)
);
```

Структура таблицы `product` (список ожидаемых товаров):

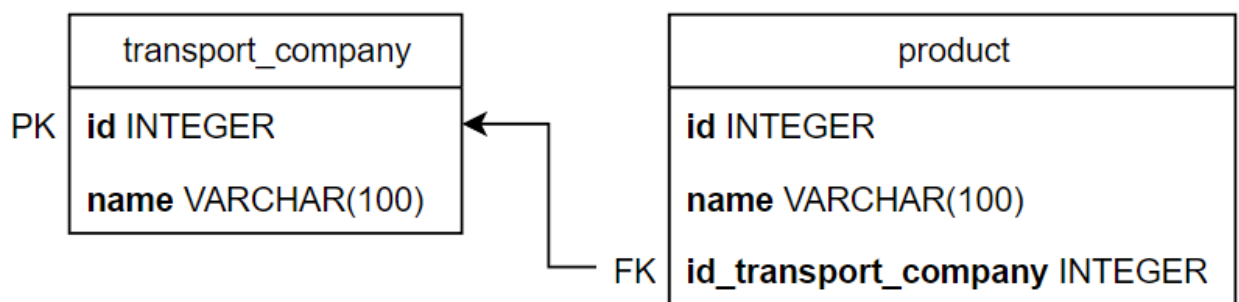
Имя столбца	Тип данных	Ограничения	Foreign key
id	integer	NOT NULL	

name	varchar(100)	NOT NULL	
id_transport_company	integer	NOT NULL	FK

Код для создания таблицы product:

```
CREATE TABLE product (
    id INTEGER NOT NULL,
    name VARCHAR(100) NOT NULL,
    id_transport_company INTEGER NOT NULL,
    CONSTRAINT transport_company_fk
    FOREIGN KEY(id_transport_company)
    REFERENCES transport_company(id)
);
```

Для наглядного представления связи таблиц transport_company и product, на рисунке ниже представлена схема.



Удаляем таблицы transport_company и product при помощи оператора [DROP](#), так как в дальнейшем они будут созданы заново.

```
DROP TABLE IF EXISTS product;
DROP TABLE IF EXISTS transport_company;
```

Создание Foreign key при помощи ALTER TABLE

Чтобы создать внешний ключ в таблице, нужно использовать оператор ALTER TABLE с дополнительным параметром ADD CONSTRAINT.

Синтаксис:

- schema – наименование схемы, в которой находится объект;

- `table_name` – имя таблицы;
- `name_fk` – имя будущего внешнего ключа;
- `column_name_fk` – имя столбца, для которого создается внешний ключ;
- `ref_table_name` – имя внешней таблицы;
- `ref_column_name` – имя столбца внешней таблицы;
- `ON DELETE CASCADE` – при удалении родительской строки все дочерние строки в дочерней таблице будут автоматически удалены. Данный параметр будет рассмотрен в разделе - [Каскадное удаление Foreign key](#);
- `ON DELETE SET NULL` – при удалении родительской строки, значение в столбце дочерней таблицы, ссылающемся на родительский столбец, будет установлено в NULL. Данный параметр будет рассмотрен в разделе - [Установка для столбца Foreign key значения NULL](#).

```
ALTER TABLE schema.table_name
ADD CONSTRAINT name_fk
FOREIGN KEY (column_name_fk)
REFERENCES ref_table_name(ref_column_name)
[ON DELETE CASCADE | ON DELETE SET NULL];
```

Практический пример: необходимо создать две таблицы `transport_company` и `product`, а затем связать их между собой при помощи первичного и внешнего ключа. При создании внешнего ключа нужно использовать оператор `ALTER TABLE`.

Структура таблицы `transport_company` (транспортные компании):

Имя столбца	Тип данных	Ограничения	Primary key
id	integer	NOT NULL	PK
name	varchar(100)	NOT NULL	

Код для создания таблицы `transport_company`:

```
CREATE TABLE transport_company (
    id INTEGER NOT NULL,
    name VARCHAR(100) NOT NULL,
    CONSTRAINT transport_company_pk PRIMARY KEY(id)
);
```

Структура таблицы `product` (список ожидаемых товаров):

Имя столбца	Тип данных	Ограничения
<code>id</code>	<code>integer</code>	NOT NULL
<code>name</code>	<code>varchar(100)</code>	NOT NULL
<code>id_transport_company</code>	<code>integer</code>	NOT NULL

Код для создания таблицы `product` (таблица создается без внешнего ключа):

```
CREATE TABLE product (  
    id INTEGER NOT NULL,  
    name VARCHAR(100) NOT NULL,  
    id_transport_company INTEGER NOT NULL  
);
```

При помощи оператора `ALTER TABLE` создаём внешний ключ `transport_company_fk` для таблицы `product`, который будет ссылаться на столбец `id` таблицы `transport_company`.

```
ALTER TABLE product  
ADD CONSTRAINT transport_company_fk  
FOREIGN KEY (id_transport_company)  
REFERENCES transport_company(id);
```

Проверка существования Foreign key

Проверить существование внешнего ключа в таблице можно при помощи графического интерфейса СУБД или SQL-запроса, который выводит информацию из информационной схемы.

SQL-запрос:

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

```
SELECT t.constraint_schema,  
       t.table_name,  
       t.constraint_name
```

```
FROM information_schema.table_constraints t
WHERE t.table_name = 'table_name'
      AND t.table_schema = 'schema'
      AND t.constraint_type = 'FOREIGN KEY';
```

Практический пример: необходимо проверить существование внешнего ключа в таблице product.

Пояснение к SQL-запросу:

- в блоке WHERE для столбца table_schema указываем значение public, а для table_name значение product. Обратите внимание, что имя схемы может отличаться.

```
SELECT t.constraint_schema,
       t.table_name,
       t.constraint_name
FROM information_schema.table_constraints t
WHERE t.table_name = 'product'
      AND t.table_schema = 'public'
      AND t.constraint_type = 'FOREIGN KEY';
```

Выполняем запрос и получаем результат, в котором видно, что в таблице product есть внешний ключ transport_company_fk.

Таблица	ABC constraint_schema	ABC table_name	ABC constraint_name
	1	public	product
			transport_company_fk

Суть работы Foreign key

Чтобы продемонстрировать работу внешнего ключа (Foreign key), воспользуемся ранее созданными таблицами transport_company и product.

Заполняем таблицу transport_company данными при помощи оператора [INSERT](#):

```
INSERT INTO transport_company(id, name)
VALUES (1, 'ТК Ястреб'),
       (2, 'ТК Молния'),
```

```
(3, 'ТК Зебра');
```

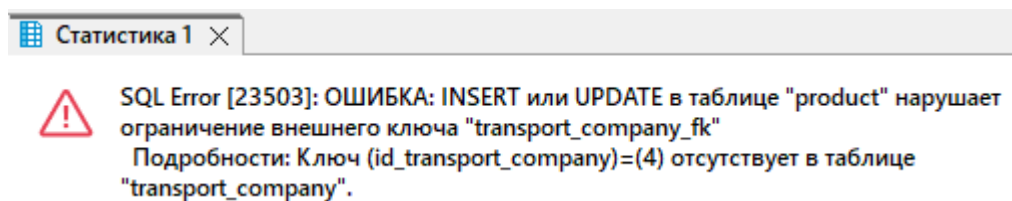
Теперь заполняем данными таблицу `product`. Обратите внимание, что в столбце `id_transport_company` указываются только существующие идентификаторы транспортных компаний, которые находятся в таблице `transport_company`.

```
INSERT INTO product (id, name, id_transport_company)
VALUES (1000, 'Двигатель 4B10', 1),
      (1001, 'Двигатель 4B11', 2),
      (1002, 'Двигатель 4B12', 3);
```

А теперь попробуем добавить ещё одну запись в таблицу `product`, и укажем в поле `id_transport_company` несуществующий идентификатор транспортной компании (его нет в таблице `transport_company`).

```
INSERT INTO product(id, name, id_transport_company)
VALUES (1003, 'Двигатель 4G63', 4);
```

После выполнения оператора `INSERT INTO`, будет получена ошибка (оператор `INSERT` или `UPDATE` в таблице `product` нарушает ограничение внешнего ключа `transport_company_fk`) и запись добавлена не будет.



Это означает, что при добавлении записи в таблицу `product`, внешний ключ `transport_company_fk` таблицы `product`, ссылается на столбец `id` таблицы `transport_company` и не находит в ней транспортную компанию с идентификатором 4.

Удаление Foreign key

Чтобы удалить внешний ключ в таблице, нужно использовать оператор `ALTER TABLE` с дополнительным параметром `DROP CONSTRAINT`.

Синтаксис:

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;

- name_fk – имя внешнего ключа.

```
ALTER TABLE schema.table_name
DROP CONSTRAINT name_fk;
```

Практический пример: необходимо удалить внешний ключ transport_company_fk из таблицы product.

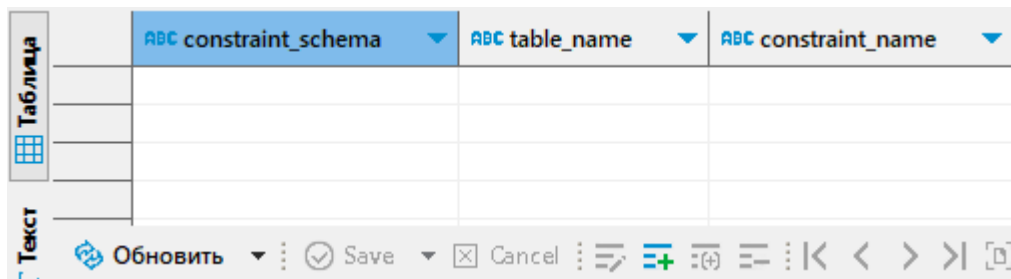
Пояснение к SQL-запросу:

- в блоке ALTER TABLE указываем имя таблицы, в которой нужно удалить внешний ключ;
- в блоке DROP CONSTRAINT указываем имя внешнего ключа;

```
ALTER TABLE product
DROP CONSTRAINT transport_company_fk;
```

Выполняем запрос, а затем проверяем уже известным нам [способом](#) действительно ли был удален внешний ключ из таблицы product.

Запрос вернул результат, в котором нет ни одной строки, а это означает, что внешний ключ был удален.



Удаляем таблицы transport_company и product при помощи оператора [DROP](#), так как в дальнейшем они будут созданы заново.

```
DROP TABLE IF EXISTS product;
DROP TABLE IF EXISTS transport_company;
```

Каскадное удаление Foreign key (ON DELETE CASCADE)

Каскадное удаление внешних ключей (Foreign keys with cascade delete) означает, что при удалении записей в родительской таблице, будут также удалены записи из дочерней таблицы, которые связаны внешним ключом с родительской таблицей.

Синтаксис:

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `column_name` – имя столбца;
- `data_type` – тип данных столбца;
- `[NOT NULL | NULL]` – ограничение, которое отвечает за то, может ли значение столбца быть `NULL` или нет;
- `name_fk` – имя будущего внешнего ключа;
- `column_name_fk` – имя столбца, для которого создается внешний ключ;
- `ref_table_name` – имя внешней таблицы;
- `ref_column_name` – имя столбца внешней таблицы;
- `ON DELETE CASCADE` – при удалении родительской строки все дочерние строки в дочерней таблице будут автоматически удалены.

```
-- Создание FK при помощи оператора CREATE TABLE
CREATE TABLE schema.table_name (
    [column_name_1] [data_type] [NOT NULL | NULL],
    ...
    [column_name_n] [data_type] [NOT NULL | NULL],
    CONSTRAINT name_fk
    FOREIGN KEY (column_name_fk)
    REFERENCES ref_table_name(ref_column_name)
    ON DELETE CASCADE;
);
```

```
-- Создание FK при помощи оператора ALTER TABLE
ALTER TABLE schema.table_name
ADD CONSTRAINT name_fk
FOREIGN KEY (column_name_fk)
REFERENCES ref_table_name(ref_column_name)
ON DELETE CASCADE;
```

Практический пример: необходимо создать две таблицы `transport_company` и `product`, а затем заполнить их данными.

Структура таблицы `transport_company` (транспортные компании):

Имя столбца	Тип данных	Ограничения	Primary key
id	integer	NOT NULL	PK
name	varchar(100)	NOT NULL	

Код для создания таблицы `transport_company`:

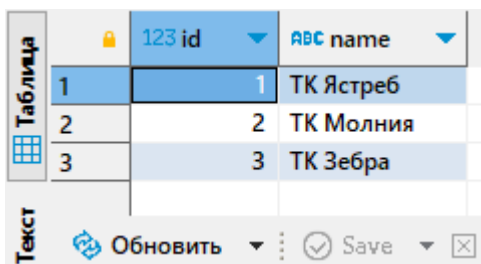
```
CREATE TABLE transport_company (
    id INTEGER NOT NULL,
    name VARCHAR(100) NOT NULL,
    CONSTRAINT transport_company_pk PRIMARY KEY(id)
);
```

Добавляем данные в таблицу `transport_company` при помощи оператора [INSERT](#):

```
INSERT INTO transport_company(id, name)
VALUES (1, 'ТК Ястреб'),
       (2, 'ТК Молния'),
       (3, 'ТК Зебра');
```

Выполняем запрос к таблице `transport_company`, после добавления данных.

```
SELECT *
FROM transport_company;
```



	123 id	ABC name
1	1	ТК Ястреб
2	2	ТК Молния
3	3	ТК Зебра

Структура таблицы `product` (список ожидаемых товаров):

Имя столбца	Тип данных	Ограничения	Foreign key
id	integer	NOT NULL	
name	varchar(100)	NOT NULL	

id_transport_company	integer	NOT NULL	FK
----------------------	---------	----------	----

Код для создания таблицы `product` (для внешнего ключа указываем параметр `ON DELETE CASCADE`):

```
CREATE TABLE product (
  id INTEGER NOT NULL,
  name VARCHAR(100) NOT NULL,
  id_transport_company INTEGER NOT NULL,
  CONSTRAINT transport_company_fk
  FOREIGN KEY(id_transport_company)
  REFERENCES transport_company(id)
  ON DELETE CASCADE
);
```

Добавляем данные в таблицу `product` при помощи оператора [INSERT](#):

```
INSERT INTO product(id, name, id_transport_company)
VALUES (1000, 'Двигатель 4B10', 1),
      (1001, 'Двигатель 4B11', 2),
      (1002, 'Двигатель 4B12', 3),
      (1003, 'Бампер передний', 2),
      (1004, 'Бампер задний', 2),
      (1005, 'Блок климат-контроля', 2);
```

Выполняем запрос к таблице `product`, после добавления данных. Обратите внимание, что от транспортной компании с идентификатором 2, ожидается самое большое количество доставок.

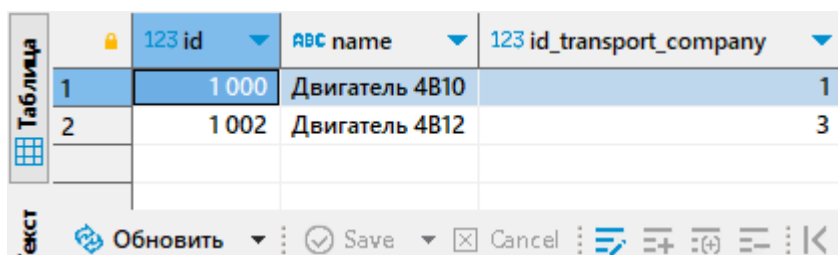
```
SELECT *
FROM product;
```

	123 id	ABC name	123 id_transport_company
1	1 000	Двигатель 4B10	1
2	1 001	Двигатель 4B11	2
3	1 002	Двигатель 4B12	3
4	1 003	Бампер передний	2
5	1 004	Бампер задний	2
6	1 005	Блок климат-контроля	2

Для наглядной работы параметра `ON DELETE CASCADE`, выполняем удаление транспортной компании с идентификатором 2 в таблице `transport_company` при помощи оператора [DELETE](#).

```
DELETE FROM transport_company  
WHERE id = 2;
```

После удаления транспортной компании, выполняем запрос к таблице `product` и проверяем, было ли выполнено удаление всех записей, которые ссылались на транспортную компанию 2. Как видим, все записи успешно удалены.



	123 id	ABC name	123 id_transport_company
1	1 000	Двигатель 4B10	1
2	1 002	Двигатель 4B12	3

Удаляем таблицы `transport_company` и `product` при помощи оператора [DROP](#), так как в дальнейшем они будут созданы заново.

```
DROP TABLE IF EXISTS product;  
DROP TABLE IF EXISTS transport_company;
```

Установка для столбцов Foreign key значения NULL (ON DELETE SET NULL)

Установка для столбцов внешних ключей значения NULL (Foreign keys with "set null on delete") означает, что при удалении записей в родительской таблице, записи в дочерней таблице удалены не будут, им будут присвоены значения NULL.

Синтаксис:

- `schema` – наименование схемы, в которой находится объект;
- `имя_таблицы` – имя таблицы;
- `column_name` – имя столбца;
- `data_type` – тип данных столбца;
- `[NOT NULL | NULL]` – ограничение, которое отвечает за то, может ли значение столбца быть NULL или нет;
- `name_fk` – имя будущего внешнего ключа;
- `column_name_fk` – имя столбца, для которого создается внешний ключ;

- `ref_table_name` – имя внешней таблицы;
- `ref_column_name` – имя столбца внешней таблицы;
- `ON DELETE SET NULL` – при удалении родительской строки, значение в столбце дочерней таблицы, ссылающемся на родительский столбец, будет установлено в `NULL`.

```
-- Создание FK при помощи оператора CREATE TABLE
CREATE TABLE schema.table_name (
    [column_name_1] [data_type] [NOT NULL | NULL],
    ...
    [column_name_n] [data_type] [NOT NULL | NULL],
    CONSTRAINT name_fk
    FOREIGN KEY (column_name_fk)
    REFERENCES ref_table_name(ref_column_name)
    ON DELETE SET NULL;
);
```

```
-- Создание FK при помощи оператора ALTER TABLE
ALTER TABLE schema.table_name
ADD CONSTRAINT name_fk
FOREIGN KEY (column_name_fk)
REFERENCES ref_table_name(ref_column_name)
ON DELETE SET NULL;
```

Практический пример: необходимо создать две таблицы `transport_company` и `product`, а затем заполнить их данными.

Структура таблицы `transport_company` (транспортные компании):

Имя столбца	Тип данных	Ограничения	Primary key
id	integer	NOT NULL	PK
name	varchar(100)	NOT NULL	

Код для создания таблицы `transport_company`:

```
CREATE TABLE transport_company (
    id INTEGER NOT NULL,
```

```

name VARCHAR(100) NOT NULL,
CONSTRAINT transport_company_pk PRIMARY KEY(id)
);

```

Добавляем данные в таблицу `transport_company` при помощи оператора [INSERT](#):

```

INSERT INTO transport_company(id, name)
VALUES (1, 'ТК Ястреб'),
       (2, 'ТК Молния'),
       (3, 'ТК Зебра');

```

Выполняем запрос к таблице `transport_company`, после добавления данных.

```

SELECT *
FROM transport_company;

```

	123 id	ABC name
1	1	ТК Ястреб
2	2	ТК Молния
3	3	ТК Зебра

Структура таблицы `product` (список ожидаемых товаров):

Имя столбца	Тип данных	Ограничения	Foreign key
id	integer	NOT NULL	
name	varchar(100)	NOT NULL	
id_transport_company	integer	NULL	FK

Код для создания таблицы `product` (для внешнего ключа указываем параметр `ON DELETE SET NULL`):

```

CREATE TABLE product (
  id INTEGER NOT NULL,
  name VARCHAR(100) NOT NULL,
  id_transport_company INTEGER NULL,
  CONSTRAINT transport_company_fk

```

```

FOREIGN KEY(id_transport_company)
REFERENCES transport_company(id)
ON DELETE SET NULL
);

```

Добавляем данные в таблицу `product` при помощи оператора [INSERT](#):

```

INSERT INTO product(id, name, id_transport_company)
VALUES (1000, 'Двигатель 4B10', 1),
      (1001, 'Двигатель 4B11', 2),
      (1002, 'Двигатель 4B12', 3),
      (1003, 'Бампер передний', 2),
      (1004, 'Бампер задний', 2),
      (1005, 'Блок климат-контроля', 2);

```

Выполняем запрос к таблице `product`, после добавления данных. Обратите внимание, что от транспортной компании с идентификатором 2, ожидается самое большое количество доставок.

```

SELECT *
FROM product;

```

	123 id	ABC name	123 id_transport_company
1	1 000	Двигатель 4B10	1
2	1 001	Двигатель 4B11	2
3	1 002	Двигатель 4B12	3
4	1 003	Бампер передний	2
5	1 004	Бампер задний	2
6	1 005	Блок климат-контроля	2

Для наглядной работы параметра `ON DELETE SET NULL`, выполняем удаление транспортной компании с идентификатором 2 в таблице `transport_company` при помощи оператора [DELETE](#).

```

DELETE FROM transport_company
WHERE id = 2;

```

После удаления транспортной компании, выполняем запрос к таблице `product` и проверяем, было ли установлено значение `NULL` для всех записей, которые ссылались на транспортную компанию 2. Как видим, все записи успешно получили значение `NULL`.

	123 id	ABC name	123 id_transport_company
1	1 000	Двигатель 4B10	1
2	1 002	Двигатель 4B12	3
3	1 001	Двигатель 4B11	[NULL]
4	1 003	Бампер передний	[NULL]
5	1 004	Бампер задний	[NULL]
6	1 005	Блок климат-контроля	[NULL]

Удаляем таблицы `transport_company` и `product` при помощи оператора [DROP](#), так как в дальнейшем они использоваться не будут.

```
DROP TABLE IF EXISTS product;
```

```
DROP TABLE IF EXISTS transport_company;
```

Ограничения

Ограничения (Constraint) — это синтаксические конструкции для столбцов таблицы, они предназначены для поддержания ссылочной целостности данных или для добавления корректных данных в таблицу согласно принятой бизнес-логике. То есть ограничения не позволяют вставить в поле (ячейку) таблицы данные, которые не соответствуют заданному ограничению.



После изучения данной главы, выполните удаление всех созданных таблиц при помощи оператора [DROP](#).

Виды ограничений

В PostgreSQL существует несколько видов ограничений, они приведены в таблице ниже:

Ограничение	Группа	Название
PRIMARY KEY	Ограничение целостности.	Ограничения первичного ключа.
FOREIGN KEY	Ограничение целостности.	Ограничения внешнего ключа.
UNIQUE	Ограничение целостности.	Ограничения уникальности.
CHECK	Ограничение проверки.	Ограничения проверки значения.
NOT NULL	Ограничения NOT NULL	Ограничения пустых значений.

Ограничения первичного ключа (Primary key)

Первичный ключ (Primary key) — очень важный тип ограничения, который используется для того, чтобы значения указанного столбца были идентифицированы уникальным образом.

Для получения дополнительной информации, смотрите раздел - [Первичные ключи](#).

Ограничения внешнего ключа (Foreign key)

Внешний ключ (`Foreign key`) — это столбец или сочетание нескольких столбцов в таблице, при помощи которых можно установить связь между данными в двух таблицах для последующего осуществления контроля данных.

Для получения дополнительной информации, смотрите раздел - [Внешние ключи](#).

Ограничения уникальности (Unique)

Ограничение `UNIQUE` гарантирует, что все строки в таблице будут строго уникальными. Это означает, что в таблице одно значение не появится два раза.

Синтаксис:

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `column_name` – имя столбца;
- `data_type` – тип данных столбца;
- `[NOT NULL | NULL]` – ограничение, которое отвечает за то, может ли значение столбца быть `NULL` или нет;
- `name_u` – имя будущего ограничения `unique`;
- `column_list` – столбец или список столбцов через запятую, по которым будет создано ограничение.

```
-- Создание ограничения при помощи CREATE TABLE
CREATE TABLE schema.table_name (
    column_name_1 [data_type] [NOT NULL | NULL],
    ...
    column_name_n [data_type] [NOT NULL | NULL],
    CONSTRAINT name_u UNIQUE(column_list)
);
```

```
-- Создание ограничения при помощи ALTER TABLE
ALTER TABLE schema.table_name
ADD CONSTRAINT name_u UNIQUE(column_list);
```

```
-- Удаление ограничения
ALTER TABLE schema.table_name
```



```
DROP CONSTRAINT name_u;
```

Проверка существования ограничения UNIQUE:

Проверить существование ограничения UNIQUE в таблице можно при помощи графического интерфейса СУБД или SQL-запроса, который выводит информацию из информационной схемы.

SQL-запрос:

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – таблица, для которой нужно проверить существование ограничения UNIQUE.

```
SELECT t.table_schema,  
       t.table_name,  
       t.constraint_name,  
       t.constraint_type  
FROM information_schema.table_constraints t  
WHERE t.table_schema = 'schema'  
      AND t.table_name = 'table_name'  
      AND constraint_type = 'UNIQUE';
```

Практический пример: необходимо создать таблицу `structure_company`, в которой будет храниться информация о структуре компании.

Структура таблицы `structure_company`:

Имя столбца	Тип данных	Ограничения
id	integer	NOT NULL
department	varchar(100)	NOT NULL, UNIQUE
city	varchar(100)	NOT NULL, UNIQUE

Код для создания таблицы `structure_company`:

```
CREATE TABLE structure_company (  
    id INTEGER NOT NULL,  
    department VARCHAR(100) NOT NULL,
```

```
city VARCHAR(100) NOT NULL,  
CONSTRAINT u_department UNIQUE(department, city)  
);
```

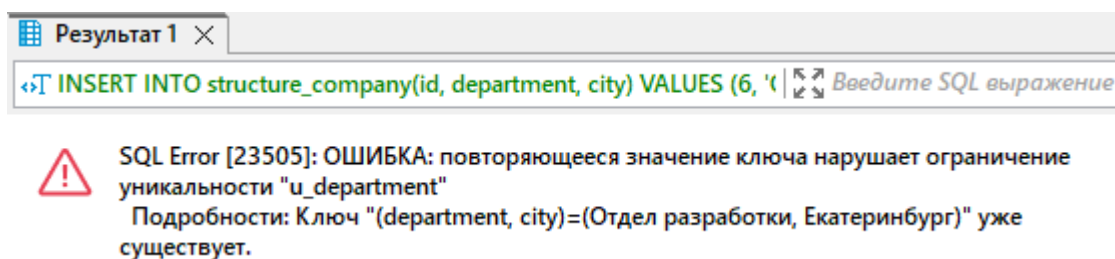
После создания таблицы `structure_company` добавляем в неё данные при помощи оператора [INSERT](#):

```
INSERT INTO structure_company(id, department, city)  
VALUES (1, 'Отдел бухгалтерии', 'Екатеринбург'),  
       (2, 'Отдел аналитики', 'Екатеринбург'),  
       (3, 'Отдел разработки', 'Екатеринбург'),  
       (4, 'Отдел бухгалтерии', 'Москва'),  
       (5, 'Отдел аналитики', 'Москва');
```

Теперь для демонстрации работы ограничения `UNIQUE`, попробуем добавить в таблицу строку с повторяющимися значениями `department` и `city` (такая комбинация столбцов уже существует, и имеет идентификатор в таблице 3).

```
INSERT INTO structure_company(id, department, city)  
VALUES (6, 'Отдел разработки', 'Екатеринбург');
```

После выполнения оператора `INSERT INTO`, будет получена ошибка (повторяющееся значение нарушает ограничение уникальности `u_department`) и запись добавлена не будет. Другими словами, мы пытаемся добавить значения, комбинация которых уже существует в таблице.



Удаляем таблицу `structure_company` при помощи оператора [DROP](#), так как в дальнейшем она использоваться не будет.

```
DROP TABLE IF EXISTS structure_company;
```

Ограничения NOT NULL

Ограничение `NOT NULL` предназначено для того, чтобы контролировать процесс вставки пустых значений в поле (ячейку) таблицы, то есть, значения, которые вносятся в поле таблицы должны быть не `NULL`.

Синтаксис:

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `column_name` – имя столбца;
- `data_type` – тип данных столбца;
- `[NOT NULL | NULL]` – ограничение, которое отвечает за то, может ли значение столбца быть `NULL` или нет.

```
-- Создание ограничения при помощи CREATE TABLE
CREATE TABLE schema.table_name (
    column_name_1 [data_type] [NOT NULL | NULL],
    ...
    column_name_n [data_type] [NOT NULL | NULL]
);
```

```
-- Создание ограничения при помощи ALTER TABLE
ALTER TABLE schema.table_name
ALTER COLUMN column_name SET NOT NULL;
```

```
-- Удаление ограничения
ALTER TABLE schema.table_name
ALTER COLUMN column_name DROP NOT NULL;
```

Проверка существования ограничения NOT NULL:

Проверить существование ограничения `NOT NULL` в таблице можно при помощи графического интерфейса СУБД или SQL-запроса, который выводит информацию из информационной схемы.

SQL-запрос:

- `schema` – наименование схемы, в которой находится объект;

- `table_name` – таблица, для которой нужно проверить существование ограничения NOT NULL.

```
SELECT t.table_schema,
       t.table_name,
       t.data_type,
       t.is_nullable
FROM information_schema.columns t
WHERE t.table_schema = 'schema'
      AND t.table_name = 'table_name'
      AND t.is_nullable = 'NO';
```

Практический пример: необходимо создать таблицу `list_users`, в которой будет храниться информация о сотрудниках.

Структура таблицы `list_users`:

Имя столбца	Тип данных	Ограничения
id	integer	NOT NULL
last_name	varchar(100)	NOT NULL
first_name	varchar(100)	NOT NULL

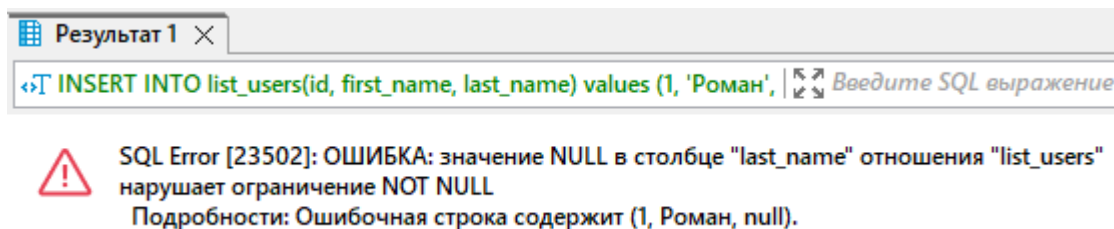
Код для создания таблицы `list_users`:

```
CREATE TABLE list_users (
  id INTEGER NOT NULL,
  first_name VARCHAR(100) NOT NULL,
  last_name VARCHAR(100) NOT NULL
);
```

Для демонстрации работы ограничения NOT NULL, добавим в таблицу `list_users` данные, в которых допущена ошибка (для поля `last_name` указано пустое значение).

```
INSERT INTO list_users(id, first_name, last_name)
values (1, 'Роман', null);
```

После выполнения оператора `INSERT INTO`, будет получена ошибка (значение `NULL` в столбце `last_name` нарушает ограничение `NOT NULL`) и запись добавлена не будет. Другими словами, мы пытаемся добавить пустое значение в столбец `last_name`, для которого установлено ограничение `NOT NULL`.



Удаляем таблицу `list_users` при помощи оператора [DROP](#), так как в дальнейшем она будет создана заново.

```
DROP TABLE IF EXISTS list_users;
```

Ограничения проверки значения (Check)

Ограничение `CHECK` применяется для того, чтобы осуществлять проверку значений столбца на соответствие установленным параметрам (параметры устанавливаются разработчиком).

Ограничение `CHECK` представляет собой выражение, которое возвращает значение `TRUE` или `FALSE` для каждой строки в таблице. Если выражение возвращает `TRUE`, то значение считается допустимым и строка будет добавлена в таблицу, в противном случае будет вызвано исключение при попытке вставить или обновить недопустимые значения. Также ограничение может содержать сложные логические выражения, включающие [операторы сравнения](#), [логические операторы](#) и функции.

При использовании ограничения `CHECK`, стоит учитывать несколько основных моментов:

- проверка значений должна ссылаться только на столбцы в этой таблице;
- проверка значений не может содержать [SQL-подзапрос](#).

Синтаксис:

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `column_name` – имя столбца;

- `data_type` – тип данных столбца;
- `[NOT NULL | NULL]` – ограничение, которое отвечает за то, может ли значение столбца быть NULL или нет;
- `name_c` – имя будущего ограничения CHECK;
- `condition_check` – условия проверки столбца.

```
-- Создание ограничения при помощи CREATE TABLE
CREATE TABLE schema.table_name (
    column_name_1 [data_type] [NOT NULL | NULL],
    ...
    column_name_n [data_type] [NOT NULL | NULL],
    CONSTRAINT name_c CHECK(condition_check)
);
```

```
-- Создание ограничения при помощи ALTER TABLE
ALTER TABLE schema.table_name
ADD CONSTRAINT name_c CHECK(condition_check);
```

```
-- Удаление ограничения
ALTER TABLE schema.table_name
DROP CONSTRAINT name_c;
```

Проверка существования ограничения CHECK:

Проверить существование ограничения CHECK в таблице можно при помощи графического интерфейса СУБД или SQL-запроса, который выводит информацию из информационной схемы.

SQL-запрос:

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – таблица, для которой нужно проверить существование ограничения CHECK.

```
SELECT t.table_schema,
       t.table_name,
       t.constraint_name
FROM information_schema.table_constraints t
WHERE t.table_schema = 'schema'
```

```
AND t.table_name = 'table_name'
AND constraint_type = 'CHECK';
```

Практический пример: необходимо создать таблицу `list_users`, в которой будет храниться информация о сотрудниках.

Структура таблицы `list_users`:

Имя столбца	Тип данных	Ограничения
id	integer	NOT NULL
last_name	varchar(100)	NOT NULL
age	integer	NOT NULL, CHECK

Код для создания таблицы `list_users`:

- для столбца `age` задается ограничение `chk_age` с условием, что возраст не должен быть меньше 18 лет и не больше 30 лет.

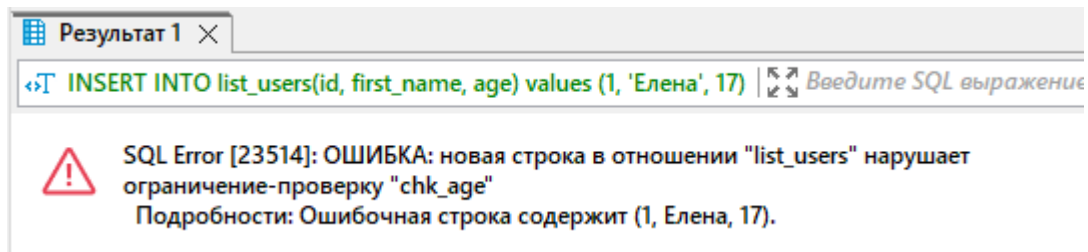
```
CREATE TABLE list_users (
    id INTEGER NOT NULL,
    first_name VARCHAR(100) NOT NULL,
    age INTEGER NOT NULL,
    CONSTRAINT chk_age CHECK (age >= 18 AND age <= 30)
);
```

Для демонстрации работы ограничения `CHECK`, попробуем добавить в таблицу `list_users` сотрудников с возрастом 17 и 35 лет.

```
-- Возраст сотрудника 17 лет
INSERT INTO list_users(id, first_name, age)
values (1, 'Елена', 17);
```

```
-- Возраст сотрудника 35 лет
INSERT INTO list_users(id, first_name, age)
values (2, 'Денис', 35);
```

После выполнения оператора `INSERT INTO`, и в том, и в другом случае, будет получена ошибка (добавляемая строка нарушает ограничение проверки `chk_age`) и запись добавлена не будет. Другими словами, мы пытаемся добавить запись в таблицу, которая не соответствует установленному ограничению столбца `age`.



Удаляем таблицу `list_users` при помощи оператора [DROP](#), так как в дальнейшем она использоваться не будет.

```
DROP TABLE IF EXISTS list_users;
```


Оператор COMMENT

При помощи оператора `COMMENT` можно добавить поясняющий комментарий к таблице и её столбцам.



По завершению изучения данной главы, восстановите данные в таблицах `employees` и `departments`. Для этого необходимо выполнить весь код из главы – [Тестовые данные для работы](#).

Просмотр комментариев к таблицам и столбцам

Посмотреть комментарии к таблицам и их столбцам можно при помощи SQL-запроса или средствами графического интерфейса вашей СУБД. Рассмотрим способ получения комментариев при помощи SQL-запроса.

SQL-запрос:

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – таблица, для которой нужно вывести описание.

```
-- Выводит комментарий к таблице
SELECT t.table_name,
       obj_description(c.oid) AS table_comment
FROM information_schema.tables t
INNER JOIN pg_class c
ON t.table_name = c.relname
WHERE t.table_schema = 'schema'
      AND t.table_name = 'table_name';

-- Выводит комментарий к столбцам таблицы
SELECT t.table_name,
       t.column_name,
       col_description(c.oid, t.ordinal_position::int) AS
column_comment
FROM information_schema.columns t
INNER JOIN pg_class c
```

```
ON t.table_name = c.relname
WHERE t.table_schema = 'schema'
      AND t.table_name = 'table_name';
```

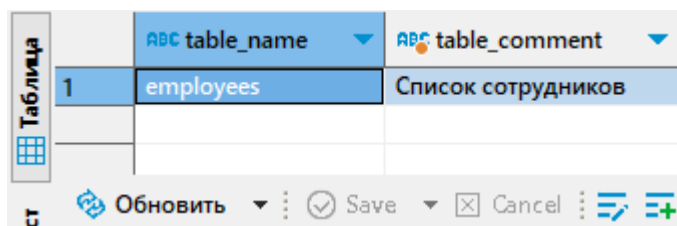
Практический пример 1: необходимо вывести комментарий к таблице `employees`.

Пояснение к SQL-запросу:

- в блоке `WHERE` указываем для `table_schema` значение `public` и для `table_name` значение `employees`. Обратите внимание, что имя схемы может отличаться.

```
SELECT t.table_name,
       obj_description(c.oid) AS table_comment
FROM information_schema.tables t
INNER JOIN pg_class c
ON t.table_name = c.relname
WHERE t.table_schema = 'public'
      AND t.table_name = 'employees';
```

Выполняем запрос и получаем комментарий к таблице `employees`.



Практический пример 2: необходимо вывести комментарии к столбцам таблицы `employees`.

Пояснение к SQL-запросу:

- в блоке `WHERE` указываем для `table_schema` значение `public` и для `table_name` значение `employees`. Обратите внимание, что имя схемы может отличаться.

```
SELECT t.table_name,
       t.column_name,
       col_description(c.oid, t.ordinal_position::int) AS
column_comment
FROM information_schema.columns t
INNER JOIN pg_class c
```

```
ON t.table_name = c.relname
WHERE t.table_schema = 'public'
      AND t.table_name = 'employees';
```

Выполняем запрос и получаем комментарии к столбцам таблицы `employees`.

	ABC table_name ▼	ABC column_name ▼	ABC column_comment ▼
1	employees	id	Идентификатор сотрудника
2	employees	last_name	Фамилия
3	employees	first_name	Имя
4	employees	gender	Пол
5	employees	birthday	дата рождения
6	employees	email	Электронный адрес
7	employees	id_department	Идентификатор отдела
8	employees	id_boss	Идентификатор руководителя
9	employees	salary	Заработная плата

Добавление комментария для таблицы

Оператор `COMMENT ON TABLE` позволяет добавить комментарий к таблице. Комментарий должен быть осмысленным и давать представление о том, какие данные хранятся в таблице.

Синтаксис:

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `comment_table` – текст комментария.

```
COMMENT ON TABLE schema.table_name
IS 'comment_table';
```

Практический пример: необходимо добавить новый комментарий к таблице `employees`, текст нового комментария `'Это новый комментарий к таблице'`.

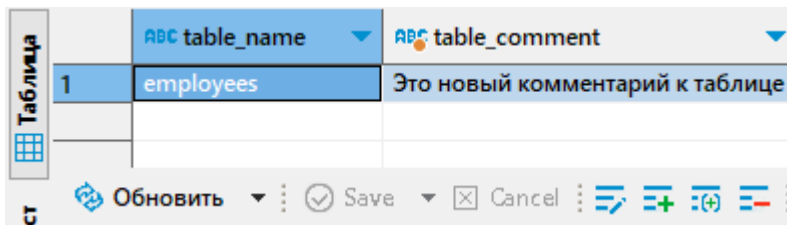
Пояснение к SQL-запросу:

- в блоке `COMMENT ON TABLE` указываем имя схемы `public` и имя таблицы `employees`, для которой нужно обновить комментарий. Обратите внимание, что имя схемы может отличаться;

- в блоке `IS` указываем новый комментарий для таблицы.

```
COMMENT ON TABLE public.employees
IS 'Это новый комментарий к таблице';
```

Выводим новый комментарий к таблице `employees` уже известным нам способом.



Добавление комментария для столбца

Оператор `COMMENT ON COLUMN` позволяет добавить комментарий к столбцу таблицы. Комментарий должен описывать смысл и назначение столбца, чтобы упростить понимание структуры данных.

Синтаксис:

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `column_name` – имя столбца;
- `comment_column` – текст комментария.

```
COMMENT ON COLUMN schema.table_name.column_name
IS 'comment_column';
```

Практический пример: необходимо добавить новые комментарии к столбцам таблицы `employees`:

- столбец `last_name`, комментарий – Фамилия сотрудника;
- столбец `first_name`, комментарий – Имя сотрудника.

Пояснение к SQL-запросу:

- в блоке `COMMENT ON COLUMN` указываем имя схемы `public`, имя таблицы `employees` и имена столбцов, для которых нужно обновить комментарии. Обратите внимание, что имя схемы может отличаться;
- в блоке `IS` указываем новые комментарии для столбцов.

```
COMMENT ON COLUMN public.employees.last_name IS 'Фамилия сотрудника';
COMMENT ON COLUMN public.employees.first_name IS 'Имя сотрудника';
```

Выводим комментарии к столбцам таблицы `employees` уже известным нам способом, и видим, что комментарии для столбцов `last_name` и `first_name` были обновлены.

	ABC table_name ▼	ABC column_name ▼	ABC column_comment ▼
1	employees	id	Идентификатор сотрудника
2	employees	last_name	Фамилия сотрудника
3	employees	first_name	Имя сотрудника
4	employees	gender	Пол
5	employees	birthday	дата рождения
6	employees	email	Электронный адрес
7	employees	id_department	Идентификатор отдела
8	employees	id_boss	Идентификатор руководителя
9	employees	salary	Заработная плата

Удаление комментариев у таблицы и колонок

Чтобы удалить ранее добавленные комментарии к таблице или к столбцам в таблице, достаточно оставить пустое значение комментария.

Синтаксис:

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `column_name` – имя столбца.

```
-- Удаление комментария у таблицы
COMMENT ON TABLE schema.table_name IS '';
```

```
-- Удаление комментария у столбца таблицы
COMMENT ON COLUMN schema.table_name.column_name IS '';
```

Практический пример: необходимо удалить комментарии для столбцов `last_name`, `first_name` и `gender` таблицы `employees`.

Пояснение к SQL-запросу:

- в блоке `COMMENT ON COLUMN` указываем имя схемы `public`, имя таблицы `employees` и имена столбцов, для которых нужно удалить комментарии. Обратите внимание, что имя схемы может отличаться;
- в блоке `IS` указываем пустое значение для столбцов.

```
COMMENT ON COLUMN public.employees.last_name IS '';
COMMENT ON COLUMN public.employees.first_name IS '';
COMMENT ON COLUMN public.employees.gender IS '';
```

Выводим комментарии к столбцам таблицы `employees` уже известным нам способом, и видим, что комментарии для столбцов `last_name`, `first_name` и `gender` были удалены.

	ABC table_name ▼	ABC column_name ▼	ABC column_comment ▼
1	employees	id	Идентификатор сотрудника
2	employees	last_name	[NULL]
3	employees	first_name	[NULL]
4	employees	gender	[NULL]
5	employees	birthday	дата рождения
6	employees	email	Электронный адрес
7	employees	id_department	Идентификатор отдела
8	employees	id_boss	Идентификатор руководителя
9	employees	salary	Заработная плата

Обновить Save Cancel

Обработка транзакций

Транзакция — это результат работы одной команды или набора команд, которые завершаются как одно целое. Изменения, которые были внесены в базу данных в ходе выполнения SQL-команд, либо фиксируются на постоянной основе, либо отменяются.

Оператор BEGIN

Оператор `BEGIN` запускает блок транзакции, и означает что все операторы, которые следуют после него должны закончиться явным оператором [COMMIT](#) или [ROLLBACK](#).

В блоке `BEGIN` операторы выполняются быстрее, так как для запуска или фиксации транзакции производится много операций, которые создают нагрузку на центральный процессор и жесткие диски.

Синтаксис:

```
-- Запуск транзакции
BEGIN TRANSACTION;
-- Тело транзакции
SQL_запрос;
-- Завершение транзакции
[COMMIT | ROLLBACK];
```

Оператор COMMIT

Оператор `COMMIT` производит фиксацию всех изменений, которые внесены в базу данных в ходе сессии текущей транзакции. После того, как оператор `COMMIT` будет выполнен, все изменения становятся видимыми для других сессий или пользователей.



Оператор `COMMIT` снимает все блокировки таблиц, которые установлены во время текущей сессии. Также оператор `COMMIT` удаляет все точки сохранения, установленные после выполнения последней команды `ROLLBACK` или `COMMIT`.

Если у вас используется режим транзакций `AUTO COMMIT`, то можно не указывать оператор `COMMIT` после команд [DML](#).

После того, как все изменения будут зафиксированы при помощи оператора `COMMIT`, выполнить их откат будет невозможно.

Синтаксис:

Оператор `COMMIT` выполняется всегда после команд DML.

```
SQL_запрос;  
COMMIT;
```

Практический пример: необходимо создать таблицу `list_users` и добавить в неё новую запись, а затем зафиксировать изменения при помощи оператора `COMMIT`.

Структура таблицы `list_users`:

Имя столбца	Тип данных	Ограничения
id	integer	NOT NULL
first_name	varchar(50)	NOT NULL
age	integer	NOT NULL

Код для создания таблицы `list_users`:

```
CREATE TABLE list_users (  
    id INTEGER NOT NULL,  
    first_name VARCHAR(50) NOT NULL,  
    age INTEGER  
);
```

Добавляем данные в таблицу при помощи оператора [INSERT](#):

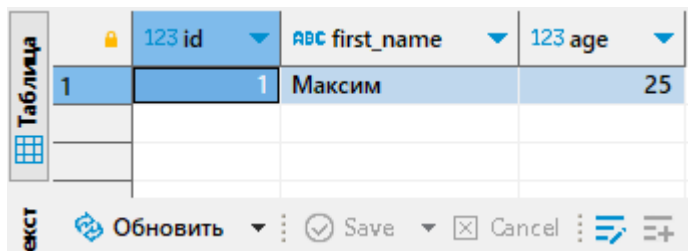
```
-- Запуск транзакции  
BEGIN TRANSACTION;  
  
-- Добавление данных в таблицу  
INSERT INTO list_users(id, first_name, age)  
VALUES (1, 'Максим', 25);
```

Фиксируем изменения в таблице.

```
COMMIT;
```


После фиксации изменений выводим данные таблицы `list_users` и видим, что запись была успешно добавлена в таблицу.

```
SELECT *  
FROM list_users;
```



	123 id	ABC first_name	123 age
1	1	Максим	25

Удаляем таблицу `list_users` при помощи оператора [DROP](#), так как в дальнейшем она использоваться не будет.

```
DROP TABLE IF EXISTS list_users;
```

Оператор ROLLBACK

Оператор `ROLLBACK` позволяет выполнить отмену внесённых изменений в базу данных текущей транзакцией. Другими словами, можно откатить свои действия на предыдущий этап (например, вы случайно удалили данные из таблицы, а оператор `ROLLBACK` позволит вернуть ваши данные в то состояние, в котором они были до удаления).

Синтаксис:

Существует две разновидности команды `ROLLBACK`:

- команда без параметров (отменяет все изменения, которые были выполнены в ходе текущей транзакции);
- команда с дополнительной секцией `TO SAVEPOINT`, она указывает до какой точки изменения необходимо произвести откат (отменяет все внесенные изменения и снимает все блокировки с таблиц, которые установлены после заданной точки сохранения). Более подробно о точках сохранения смотрите раздел - [Оператор SAVEPOINT](#).

Когда происходит откат до указанной точки сохранения, все установленные после неё точки сохранения стираются, но указанная точка остаётся. Это означает, что вы можете возобновить транзакцию с этой точки, и, если появится необходимость вернуться к этой точке снова вы сможете сделать это без особого труда.

```
-- Полный откат изменений
ROLLBACK;

-- Возврат к точке сохранения
ROLLBACK TO SAVEPOINT имя_точки_сохранения;
```

Практический пример: рассмотрим такой случай, когда в таблицу `list_users` была добавлена строка с неправильными данными, но транзакция ещё не была зафиксирована оператором `COMMIT`. Нужно выполнить откат внесенных изменений при помощи оператора `ROLLBACK`.

Структура таблицы `list_users`:

Имя столбца	Тип данных	Ограничения
id	integer	NOT NULL
first_name	varchar(50)	NOT NULL
age	integer	NOT NULL

Код для создания таблицы `list_users`:

```
CREATE TABLE list_users (
    id INTEGER NOT NULL,
    first_name VARCHAR(50) NOT NULL,
    age INTEGER
);
```

Добавляем данные в таблицу при помощи оператора [INSERT](#):

```
-- Запуск транзакции
BEGIN TRANSACTION;

-- Добавление данных в таблицу
INSERT INTO list_users(id, first_name, age)
VALUES (1, 'Ольга', -30);
```

Выводим данные таблицы на экран и видим, что у сотрудника указан неправильный возраст.

```
SEELCT *
```

```
FROM list_users;
```

Таблица	123 id	ABC first_name	123 age
1	1	Ольга	-30

экст Обновить Save Cancel

Выполняем откат внесенных изменений.

```
ROLLBACK;
```

После выполнения отката, повторно выводим данные таблицы `list_users` и видим, что запись была успешно удалена из таблицы.

```
SELECT *  
FROM list_users;
```

list_users 1 X			
SELECT * FROM list_users Введите SQL выражение чтобы			
Таблица	123 id	ABC first_name	123 age

экст Обновить Save Cancel

Удаляем таблицу `list_users` при помощи оператора [DROP](#), так как в дальнейшем она использоваться не будет.

```
DROP TABLE IF EXISTS list_users;
```

Оператор SAVEPOINT

Оператор `SAVEPOINT` позволяет установить точку сохранения в транзакции, к которой можно будет обратиться по необходимости и выполнить откат действий. Когда выполняется откат на какую-то конкретную точку сохранения, то отменяются все изменения и удаляются все блокировки после этой точки, но сохраняются изменения и блокировки, которые были до этой точки сохранения.

Синтаксис:

Имя точки сохранения (`name_savepoint`) — это необъявленный идентификатор, который должен соответствовать общим правилам формирования идентификаторов PostgreSQL:

- длина идентификатора должна быть не более 63 символов;
- имя идентификатора должно начинаться с буквы или символа подчеркивания (`_`).
- имя идентификатора должно состоять только из букв (в любом регистре), цифр и символа подчеркивания (`_`);
- имя идентификатора не должно содержать пробелов или специальных символов, таких как `!, @, #, $, %, ^, &, *, (,), -, +, =, {, }, [,], |, \, :, ;, ", ', <, >, ,, ., ?, /, ~, ``
- имя идентификатора не должно совпадать с [зарезервированными словами](#) PostgreSQL.

```
-- Создание точки сохранения
SAVEPOINT name_savepoint;

-- Возвращение к точке сохранения
ROLLBACK TO SAVEPOINT name_savepoint;
```

Практический пример: рассмотрим такой случай, когда в таблице `list_users` уже хранятся данные сотрудников и в неё необходимо добавить новые данные несколькими частями с использованием точек сохранения `SAVEPOINT`.

Структура таблицы `list_users`:

Имя столбца	Тип данных	Ограничения
<code>id</code>	<code>integer</code>	<code>NOT NULL</code>
<code>full_name</code>	<code>varchar(50)</code>	<code>NOT NULL</code>
<code>name_department</code>	<code>Varchar(50)</code>	<code>NOT NULL</code>

Код для создания таблицы `list_users`:

```
CREATE TABLE list_users (
    id INTEGER NOT NULL,
    full_name VARCHAR(50) NOT NULL,
    name_department VARCHAR(50) NOT NULL
```

```
);
```

Добавляем запись в таблицу `list_users` при помощи оператора [INSERT](#) и фиксируем изменения оператором [COMMIT](#).

```
-- Запуск транзакции
BEGIN TRANSACTION;

-- Добавление данных в таблицу
INSERT INTO list_users(id, full_name, name_department)
VALUES (1, 'Богданов Р.М', 'Отдел безопасности');

-- Фиксация изменений
COMMIT;
```

Теперь добавляем в таблицу данные несколькими частями и после добавления каждой части создаем точку сохранения.

```
-- Запуск транзакции
BEGIN TRANSACTION;

-- 1 часть данных
INSERT INTO list_users(id, full_name, name_department)
VALUES (2, 'Суходольская В.И', 'Отдел бухгалтерии'),
       (3, 'Иванова К.В', 'Отдел бухгалтерии');
SAVEPOINT add_part_users_1;

-- 2 часть данных
INSERT INTO list_users(id, full_name, name_department)
VALUES (4, 'Сидоров П.К', 'Отдел тестирования'),
       (5, 'Кузнецов И.О', 'Отдел тестирования');
SAVEPOINT add_part_users_2;

-- 3 часть данных
INSERT INTO list_users(id, full_name, name_department)
VALUES (6, 'Лосев С.А', 'Отдел маркетинга'),
       (7, 'Орлова О.Ю', 'Отдел маркетинга');
SAVEPOINT add_part_users_3;
```

После добавления данных, выводим содержимое таблицы на экран. Обратите внимание, что все данные были успешно добавлены в таблицу, но они пока не зафиксированы.

```
SELECT *
FROM list_users;
```

	123 id	ABC full_name	ABC name_department
1	1	Богданов Р.М	Отдел безопасности
2	2	Суходольская В.И	Отдел бухгалтерии
3	3	Иванова К.В	Отдел бухгалтерии
4	4	Сидоров П.К	Отдел тестирования
5	5	Кузнецов И.О	Отдел тестирования
6	6	Лосев С.А	Отдел маркетинга
7	7	Орлова О.Ю	Отдел маркетинга

А теперь представим, что необходимо выполнить откат изменений и сохранить только тех сотрудников, которые были добавлены на 1 этапе. Для этого необходимо воспользоваться оператором `ROLLBACK` и указать точку для отката.

```
ROLLBACK TO SAVEPOINT add_part_users_1;
```

После выполнения оператора `ROLLBACK`, снова выводим все записи из таблицы и видим, что в таблице остались только те сотрудники, которые были добавлены на 1 этапе.

Сейчас остаётся только зафиксировать внесенные изменения в таблицу при помощи оператора `COMMIT`.

	123 id	ABC full_name	ABC name_department
1	1	Богданов Р.М	Отдел безопасности
2	2	Суходольская В.И	Отдел бухгалтерии
3	3	Иванова К.В	Отдел бухгалтерии

Удаляем таблицу `list_users` при помощи оператора [DROP](#), так как в дальнейшем она использоваться не будет.

```
DROP TABLE IF EXISTS list_users;
```

Оператор SET TRANSACTION

Оператор `SET TRANSACTION` устанавливает характеристики только для текущей транзакции и на последующие транзакции не влияет. Перед оператором `SET`

TRANSACTION должен идти обязательно оператор [BEGIN](#), который начинает транзакцию, если он не указан, то СУБД выдаст ошибку и завершит свою работу.



Если необходимо установить характеристики для всех транзакций в рамках одного сеанса, то нужно использовать оператор `SET SESSION CHARACTERISTICS AS TRANSACTION` и указать уровень изоляции транзакции.

Синтаксис:

- `BEGIN TRANSACTION` - оператор, который запускает транзакцию;
- `ISOLATION LEVEL` – уровень изоляции текущей транзакции:
 - `SERIALIZABLE` - уровень изоляции, который предоставляет возможность пользователям, видеть только те данные в таблицах, которые были на момент начала его сессии;
 - `REPEATABLE READ` - уровень изоляции, который гарантирует, что транзакция, которая повторно читает данные из таблицы в разные временные интервалы, каждый раз будет видеть одни и те же значения, которые были на момент запуска транзакции.;
 - `READ COMMITTED` - оператор видит только те строки, которые были зафиксированы до начала его выполнения. Этот уровень устанавливается по умолчанию. Важный момент, в PostgreSQL уровень `READ UNCOMMITTED` обрабатывается как `READ COMMITTED`.
- `READ WRITE` - устанавливает текущую транзакцию как операцию чтения и записи данных в таблицу;
- `READ ONLY` - устанавливает текущую транзакцию доступной «только для чтения». В транзакциях такого типа всем запросам доступны лишь те изменения, которые были зафиксированы до начала транзакции.

```
-- Запуск транзакции
```

```
BEGIN TRANSACTION;
```

```
-- Установка характеристики транзакции
```

```
SET TRANSACTION
```

```
-- Уровень изоляции транзакции
```

```
ISOLATION LEVEL [SERIALIZABLE | REPEATABLE READ | READ COMMITTED |  
                 READ UNCOMMITTED]
```

```
-- режим работы транзакции
```

```

[READ WRITE | READ ONLY];
-- Тело транзакции
SQL_запрос;

```

Практический пример: необходимо создать таблицу `list_users` и добавить в нее одну запись, но при добавлении записи нужно установить уровень изоляции для транзакции и режим доступа при помощи оператора `SET TRANSACTION`.

Структура таблицы `list_users`:

Имя столбца	Тип данных	Ограничения
id	integer	NOT NULL
first_name	varchar(50)	NOT NULL
age	integer	NOT NULL

Код для создания таблицы `list_users`:

```

CREATE TABLE list_users (
    id INTEGER NOT NULL,
    first_name VARCHAR(50) NOT NULL,
    age INTEGER NOT NULL
);

```

Запускаем транзакцию при помощи оператора [BEGIN TRANSACTION](#).

```
BEGIN TRANSACTION;
```

Устанавливаем уровень изоляции для транзакции `READ COMMITED` и режим доступа `READ WRITE`.

```

SET TRANSACTION
    ISOLATION LEVEL READ COMMITTED
    READ WRITE;

```

Добавляем данные в таблицу при помощи оператора [INSERT](#).

```

INSERT INTO list_users(id, first_name, age)
VALUES (1, 'Мария', 25);

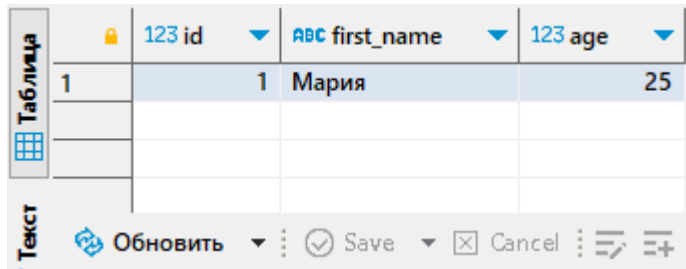
```


Фиксируем изменения при помощи оператора [COMMIT](#).

```
COMMIT;
```

После добавления данных, выводим содержимое таблицы на экран. Обратите внимание, что все данные были успешно добавлены в таблицу.

```
SELECT *  
FROM list_users;
```



The screenshot shows a database table viewer interface. On the left, there are two tabs: 'Таблица' (Table) and 'Текст' (Text), with 'Таблица' selected. The table has three columns: 'id' (with a lock icon and value 123), 'first_name' (with an 'ABC' icon and value 123), and 'age' (with a value 123). The first row of data shows 'id' 1, 'first_name' Мария, and 'age' 25. Below the table, there is a toolbar with buttons: 'Обновить' (Refresh), 'Save', and 'Cancel', along with some icons for table manipulation.

	123 id	ABC first_name	123 age
1	1	Мария	25

Удаляем таблицу `list_users` при помощи оператора [DROP](#), так как в дальнейшем она использоваться не будет.

```
DROP TABLE IF EXISTS list_users;
```

Оператор SELECT

Оператор `SELECT` позволяет получить данные из указанной таблицы, и вывести их на экран.

Вывод всех столбцов из таблицы

Для вывода всех столбцов из таблицы, необходимо указать символ `*` (звёздочка) сразу после оператора `SELECT`.

Синтаксис:

- `*` – выводит все столбцы из таблицы;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

```
SELECT *  
FROM schema.table_name;
```

Практический пример: из таблицы `employees` необходимо вывести все столбцы.

```
SELECT *  
FROM employees;
```

Результат выполнения запроса. Обратите внимание, что результат запроса на изображении показан не полностью.

	123 id	ABC last_name	ABC first_name	ABC gender	birthday	ABC email	123 id_department	123 id_boss	123 salary
1	1	Иванов	Иван	Мужской	1990-05-15	ivanov@mail.ru	1	[NULL]	35 000
2	2	Петров	Петр	Мужской	1985-12-20	petrov@yandex.ru	2	[NULL]	45 000
3	3	Сидорова	Мария	Женский	1992-08-10	sidorova@gmail.com	3	[NULL]	54 000
4	4	Петров	Петр	Мужской	1987-04-25	[NULL]	4	[NULL]	100 000
5	5	Васильева	Екатерина	Женский	1995-03-30	vasilieva@yandex.ru	5	[NULL]	38 900
6	6	Попов	Дмитрий	Мужской	1988-11-05	popov@gmail.com	6	[NULL]	110 000
7	7	Соколова	Анастасия	Женский	1991-07-12	sokolova@mail.ru	1	1	40 000
8	8	Лебедева	Алена	Женский	1994-09-18	lebedeva@yandex.ru	2	2	68 000
9	9	Козлов	Артём	Мужской	1986-10-22	kozlov@gmail.com	3	3	120 000
10	10	Новиков	Андрей	Мужской	1989-06-08	novikov@mail.ru	4	4	42 000

Вывод одного столбца из таблицы

Для вывода одного столбца из таблицы, необходимо указать его имя сразу после оператора `SELECT`.

Синтаксис:

- `column_name` – имя столбца;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

```
SELECT column_name
FROM schema.table_name;
```

Практический пример: из таблицы `employees` необходимо вывести столбец `first_name`.

```
SELECT first_name
FROM employees;
```

Результат выполнения запроса. Обратите внимание, что результат запроса на изображении показан не полностью.

Таблица	ABC first_name
1	Иван
2	Петр
3	Мария
4	Петр
5	Екатерина
6	Дмитрий
7	Анастасия
8	Алена
9	Артём
10	Андрей

Вывод нескольких столбцов из таблицы

Для вывода нескольких столбцов из таблицы, необходимо указать их имена через запятую сразу после оператора `SELECT`.

Синтаксис:

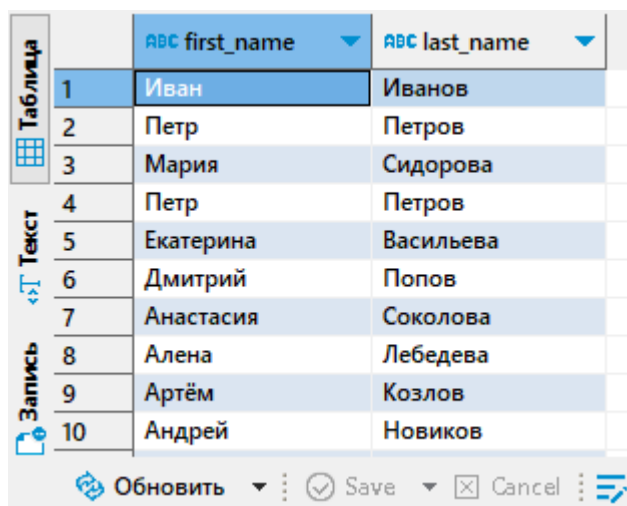
- `column_name` – имя столбца;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

```
SELECT column_name_1, ... ,column_name_n
FROM schema.table_name;
```

Практический пример: из таблицы `employees` необходимо вывести столбец `first_name` и `last_name`.

```
SELECT first_name, last_name
FROM employees;
```

Результат выполнения запроса. Обратите внимание, что результат запроса на изображении показан не полностью.



The screenshot shows a database interface with a table of employee names. The table has two columns: 'first_name' and 'last_name'. The data is as follows:

	first_name	last_name
1	Иван	Иванов
2	Петр	Петров
3	Мария	Сидорова
4	Петр	Петров
5	Екатерина	Васильева
6	Дмитрий	Попов
7	Анастасия	Соколова
8	Алена	Лебедева
9	Артём	Козлов
10	Андрей	Новиков

Вывод уникальных строк (DISTINCT)

Оператор `SELECT` выводит все строки, которые соответствуют критериям отбора. Но бывают такие случаи, когда одна строка или значение повторяется несколько раз. Но как быть, если из таблицы нужно получить только уникальные значения или строки?

Решить этот вопрос помогает оператор `DISTINCT`, он удаляет дубликаты (повторяющиеся значения) из результирующего набора `SELECT`.

Синтаксис:

- `column_name` – имя столбца;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

-- Оператор `DISTINCT` для одного столбца

```
SELECT DISTINCT column_name
FROM schema.table_name;
```

-- Оператор `DISTINCT` для нескольких столбцов

```
SELECT DISTINCT column_name_1, column_name_n
FROM schema.table_name;
```

```
-- Оператор DISTINCT для всех столбцов
SELECT DISTINCT *
FROM schema.table_name;
```

Практический пример 1: из таблицы `employees` необходимо вывести уникальные значения столбца `first_name`.

Сначала выведем из таблицы `employees` значения столбца `first_name` и посмотрим, повторяются ли значения в столбце.

```
SELECT first_name
FROM employees;
```

Сразу можно увидеть, что значения в столбце `first_name` не уникальные, то есть они повторяются. Если полностью посмотреть полученные данные, которые вернул запрос, то могут быть еще повторяющиеся значения.

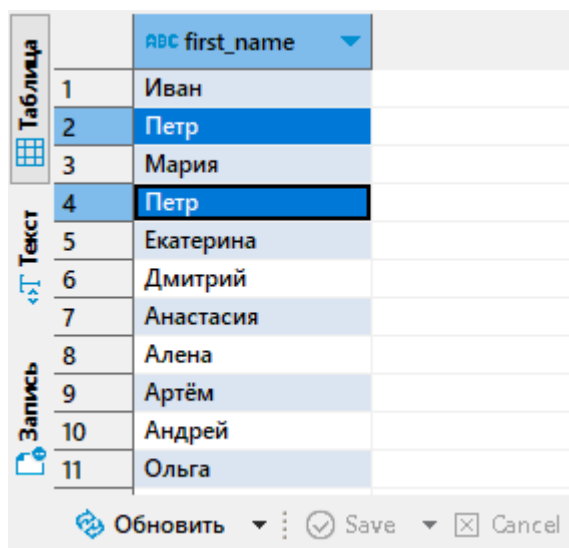


Таблица	first_name
1	Иван
2	Петр
3	Мария
4	Петр
5	Екатерина
6	Дмитрий
7	Анастасия
8	Алена
9	Артём
10	Андрей
11	Ольга

Чтобы убрать повторяющиеся значения в столбце добавляем оператор `DISTINCT` к запросу, который написали ранее.

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

Теперь значения в столбце `first_name` не повторяются. Обратите внимание, что результат запроса на изображении показан не полностью.

Таблица	ABC first_name	
1	Анна	
2	Григорий	
3	Ольга	
4	Петр	
5	Александра	
6	Анастасия	
7	Мария	
8	Яна	
9	Иван	
10	Денис	
11	Сергей	

Обновить Save Cancel

Практический пример 2: из таблицы `employees` необходимо вывести уникальные значения столбцов `first_name` и `last_name`.

Сначала выведем из таблицы `employees` значения столбцов `first_name` и `last_name`, чтобы посмотреть повторяются ли значения.

```
SELECT first_name, last_name
FROM employees;
```

Сразу можно увидеть, что значения в столбцах `first_name` и `last_name` не уникальные, то есть они повторяются. Если полностью посмотреть полученные данные, которые вернул запрос, то могут быть еще повторяющиеся значения.

Таблица	ABC first_name	ABC last_name
1	Иван	Иванов
2	Петр	Петров
3	Мария	Сидорова
4	Петр	Петров
5	Екатерина	Васильева
6	Дмитрий	Попов
7	Анастасия	Соколова
8	Алена	Лебедева
9	Артём	Козлов
10	Андрей	Новиков
11	Ольга	Морозова

Обновить Save Cancel

Чтобы убрать повторяющиеся значения добавляем оператор `DISTINCT` к запросу, который написали ранее.

```
SELECT DISTINCT first_name, last_name
FROM employees;
```

Теперь значения в столбцах `first_name` и `last_name` не повторяются. Обратите внимание, что результат запроса на изображении показан не полностью.

	ABC first_name ▼	ABC last_name ▼
1	Петр	Петров
2	Елена	Ковалева
3	Денис	Карпов
4	Сергей	Павлов
5	Сергей	Лебедев
6	Анна	Борисова
7	Максим	Пономарева
8	Мария	Сидорова
9	Егор	Федоров
10	Андрей	Новиков
11	Михаил	Соколов

Ограничение результатов запроса

Обычно, когда выполняется SQL-запрос, то в конечном результате выводится сразу весь результирующий набор. А как быть если возникает необходимость в том, чтобы ограничить выборку данных и получить желаемое количество строк?

Для таких целей есть специальные операторы, которые ограничивают результат запроса определенным количеством строк. К сожалению, оператор для ограничения строк ведет себя в разных СУБД по-разному и носит другое наименование, например:

- оператор `LIMIT` (используется в PostgreSQL, MySQL, SQLite и т.д.);
- оператор `TOP` (используется в Microsoft SQL Server);
- предикат `ROWNUM` и метод `OFFSET` (используется в Oracle).

Все конструкции для ограничения строк мы рассматривать не будем, поэтому рассмотрим только оператор `LIMIT`, который применяется в PostgreSQL.

Синтаксис:

- `column_list` – список столбцов;
- `schema` – наименование схемы, в которой находится объект;

- table_name – имя таблицы;
- rows_count – количество строк, которые нужно вывести;
- rows_skip_count – сколько строк нужно пропустить перед тем, как вывести нужное количество строк.

```
SELECT column_list
FROM schema.table_name
LIMIT rows_count OFFSET rows_skip_count;
```

Практический пример: из таблицы employees необходимо вывести 7 строк с предварительным смещением в 10 строк.

```
SELECT *
FROM employees
LIMIT 7 OFFSET 10;
```

Выполняем запрос и получаем результат, в котором сначала было выполнено смещение на 10 строк, а затем был осуществлен вывод 7 строк.

	123 id	ABC last_name	ABC first_name	ABC gender	birthday	ABC email	123 id_department	123 id_boss	123 salary
1	11	Морозова	Ольга	Женский	1993-02-14	morozova@yandex.ru	5	5	70 000
2	12	Павлов	Сергей	Мужской	1987-12-11	pavlov@gmail.com	6	6	130 000
3	13	Григорьева	Наталья	Женский	1996-01-25	grigoryeva@mail.ru	1	1	60 000
4	14	Борисов	Игорь	Мужской	1985-05-20	borisov@yandex.ru	2	2	95 000
5	15	Максимова	Вероника	Женский	1990-08-30	maximova@gmail.com	3	3	72 000
6	16	Кузьмина	Елена	Женский	1997-08-15	kuzmina@mail.ru	4	4	105 000
7	17	Егоров	Кирилл	Мужской	1986-07-22	[NULL]	5	5	62 000

Запись 200 7 7 строк получено - 0ms, 2024-01-18

Оператор FROM

Оператор `FROM` является основным и предназначен для указания таблицы, из которой необходимо извлечь данные. Также стоит отметить, что в операторе `FROM` можно указать несколько таблиц или [подзапрос](#) (subquery).

Синтаксис:

- `column_list` – список столбцов;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

-- Одна таблица в операторе FROM

```
SELECT column_list
FROM schema.table_name;
```

-- Несколько таблиц в операторе FROM

```
SELECT column_list
FROM schema.table_name_1, schema.table_name_2;
```

-- Подзапрос в операторе FROM

```
SELECT column_list
FROM (
    sql_подзапрос
);
```

Основные моменты при работе с оператором:

- в операторе `FROM` должно быть указано хотя бы одно имя таблицы или подзапрос;
- если необходимо указать несколько таблиц в операторе `FROM`, то их нужно перечислить через запятую;
- если необходимо объединить несколько таблиц при помощи блока `FROM`, то в обязательном порядке нужно указать условия для объединения этих таблиц, иначе у вас будет [перекрестное соединение таблиц](#) или как его еще называют Декартово произведение (об этом поговорим в другом разделе).

Оператор AS (псевдонимы)

В SQL есть такой оператор, как AS. Он позволяет переопределять и задавать новые имена (псевдонимы или алиасы) для колонок и таблиц. Другими словами, можно сократить длинное наименование таблицы или столбца в удобное для себя наименование и использовать его для последующих вычислений.

Важные моменты при работе с псевдонимами:

- псевдонимы не могут повторяться в рамках одного запроса и подзапроса, то есть их имена должны быть строго уникальными;
- псевдонимы не чувствительны к регистру;
- не рекомендуется использование [зарезервированных системных слов](#) для псевдонима, например: column, drop, table, from и т.д.;
- если псевдоним состоит из одного слова, то его необязательно заключать в кавычки;
- если псевдоним состоит из двух слов, разделенных пробелом, то его нужно заключать в двойные кавычки.

Синтаксис:

- column_name – имя столбца;
- alias_column – псевдоним, который будет установлен для столбца;
- alias_table – псевдоним, который будет установлен для таблицы;
- schema – наименование схемы, в которой находится объект;
- table_name – имя таблицы.

-- Псевдоним для столбца или значения

```
SELECT column_name AS alias_column  
FROM schema.table_name;
```

-- Псевдоним для таблицы

```
SELECT alias_table.column_name_1,  
       alias_table.column_name_n  
FROM schema.table_name AS alias_table;
```

Практический пример 1: есть SQL-запрос, он возвращает два столбца last_name и salary из таблицы employees.

```
SELECT last_name, salary
FROM employees;
```

Обратите внимание на названия столбцов, они выводятся в том виде, в котором хранятся в таблице.

Таблица

Текст

Запись

	ABC last_name	123 salary
1	Иванов	35 000
2	Петров	45 000
3	Сидорова	54 000
4	Петров	100 000
5	Васильева	38 900
6	Попов	110 000
7	Соколова	40 000
8	Лебедева	68 000
9	Козлов	120 000
10	Новиков	42 000

Обновить

Save

Cancel

Теперь рассмотрим такую ситуацию, когда в финальной выборке необходимо изменить названия столбцов:

- название столбца `last_name` заменяем на `family`;
- название столбца `salary` заменяем на `money`.

Присваиваем столбцам псевдонимы при помощи оператора `AS`.

```
SELECT last_name AS family,
       salary AS money
FROM employees;
```

Теперь в финальной выборке названия столбцов выводятся в соответствии с установленными псевдонимами.

		ABC falimy ▼	123 money ▼
1	Иванов	35 000	
2	Петров	45 000	
3	Сидорова	54 000	
4	Петров	100 000	
5	Васильева	38 900	
6	Попов	110 000	
7	Соколова	40 000	
8	Лебедева	68 000	
9	Козлов	120 000	
10	Новиков	42 000	

Обновить Save Cancel

Практический пример 2: есть SQL-запрос, он возвращает три столбца `last_name`, `first_name` и `birthday` из таблицы `employees`.

```
SELECT last_name,
       first_name,
       birthday
FROM employees;
```

Результат выполнения запроса.

	ABC last_name ▼	ABC first_name ▼	🕒 birthday ▼
1	Иванов	Иван	1990-05-15
2	Петров	Петр	1985-12-20
3	Сидорова	Мария	1992-08-10
4	Петров	Петр	1987-04-25
5	Васильева	Екатерина	1995-03-30
6	Попов	Дмитрий	1988-11-05
7	Соколова	Анастасия	1991-07-12
8	Лебедева	Алена	1994-09-18
9	Козлов	Артём	1986-10-22
10	Новиков	Андрей	1989-06-08

Обновить Save Cancel

Рассмотрим случай, когда имя таблицы `employees` использовать по какой-то причине неудобно, тогда в этом случае необходимо установить псевдоним для таблицы при помощи оператора `AS`.

Устанавливаем псевдоним `e` для таблицы `employees`. И теперь, чтобы вывести столбец из таблицы нужно сначала указать псевдоним, затем поставить точку, а дальше указать имя нужного столбца.

```
SELECT e.last_name,  
       e.first_name,  
       e.birthday  
FROM employees AS e;
```

Результат выполнения запроса идентичен предыдущему запросу. Отличие этих двух запросов заключается в том, что в одном мы использовали псевдоним для таблицы, а в другом нет.

Преимущество использования псевдонимов будет ощущаться при [объединении таблиц](#) (об этом поговорим в следующих главах).

Таблица		ABC last_name	ABC first_name	birthday
		Иванов	Иван	1990-05-15
Текст	2	Петров	Петр	1985-12-20
	3	Сидорова	Мария	1992-08-10
	4	Петров	Петр	1987-04-25
	5	Васильева	Екатерина	1995-03-30
	6	Попов	Дмитрий	1988-11-05
	7	Соколова	Анастасия	1991-07-12
	8	Лебедева	Алена	1994-09-18
	9	Козлов	Артём	1986-10-22
	10	Новиков	Андрей	1989-06-08

Обновить Save Cancel

Оператор || (конкатенация)

Оператор || позволяет объединить несколько значений в одно. Объединение строк в одно значение называется – **конкатенация**.

Синтаксис:

- `column_name` – имя столбца;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

```
SELECT column_name_1||column_name_2||column_name_n  
FROM schema.table_name;
```












Практический пример: из таблицы `employees` необходимо вывести два столбца `last_name` и `first_name`, а затем нужно объединить эти столбцы и вывести в дополнительном столбце `full_name`.

Пояснение к SQL-запросу:

- при помощи оператора || объединяем два столбца `last_name` и `first_name`.
Полученному значению присваиваем псевдоним `full_name`.

```
SELECT last_name,  
       first_name,  
       last_name||' '||first_name AS full_name  
FROM employees;
```

Выполняем запрос и получаем список, в котором содержится полное имя сотрудников компании. Обратите внимание, что результат запроса на изображении показан не полностью.

Таблица 		ABC last_name	ABC first_name	ABC full_name
	1	Иванов	Иван	Иванов Иван
	2	Петров	Петр	Петров Петр
	3	Сидорова	Мария	Сидорова Мария
	4	Петров	Петр	Петров Петр
	5	Васильева	Екатерина	Васильева Екатерина
	6	Попов	Дмитрий	Попов Дмитрий
	7	Соколова	Анастасия	Соколова Анастасия
	8	Лебедева	Алена	Лебедева Алена
	9	Козлов	Артём	Козлов Артём
Запись 	10	Новиков	Андрей	Новиков Андрей
<div> Обновить</div> <div> Save</div> <div> Cancel</div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div>				

Оператор WHERE (фильтрация данных)

Фильтрация данных предназначена для исключения данных из финальной выборки, которые не соответствуют условиям отбора.

В таблицах баз данных может храниться различное количество данных, начиная от одной строки и заканчивая несколькими миллионами. Очень редко возникает необходимость в извлечении всех данных таблицы. Намного чаще, требуется извлечь из таблицы какую-то определенную часть. И чтобы извлечь нужную часть данных необходимо написать условия (критерии) для отбора данных.

В SQL-запросах данные фильтруются путем указаний критериев для отбора в блоке `WHERE`. Ключевое слово `WHERE` указывается после блока [FROM](#).

Синтаксис:

- `column_list` – список столбцов;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `condition_filter` – условия для фильтрации.

```
SELECT column_list
FROM schema.table_name
WHERE condition_filter;
```

Как написать условия для фильтрации?

Чтобы написать условия фильтрации нужно использовать комбинации различных операторов, таких как:

- [арифметические операторы](#);
- [операторы сравнения](#);
- [логические операторы](#).

Также, в блоке `WHERE` можно использовать [подзапросы](#) для дополнительной фильтрации результатов, но обо всём этом поговорим в следующих главах, и рассмотрим на практических примерах.

Арифметические операторы

Арифметические операторы позволяют выполнять основные математические операции в блоке [SELECT](#) и [WHERE](#) запроса.

Существуют следующие арифметические операторы:

Оператор	Описание
+	Сложение.
–	Вычитание.
*	Умножение.
/	Деление.
÷	Остаток от деления.

Сложение (+)

Оператор сложения (+) возвращает сумму указанных операндов (значений).

Синтаксис:

- `column_name` – имя столбца или статическое значение, которое может быть задано вручную. Количество значений неограниченно;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

```
SELECT column_name_1 + column_name_2  
FROM schema.table_name;
```

Практический пример: из таблицы `employees` необходимо вывести столбцы `first_name` и `salary`. Дополнительным столбцом вывести сумму столбца `salary` и значения 100, полученному значению присвоить имя `result`.

```
SELECT first_name,
```

```

        salary,
        salary + 100 AS result
FROM employees;

```

Выполняем запрос и получаем следующий результат. Обратите внимание на столбцы `salary` и `result`, в них хранятся разные значения.

	ABC first_name	123 salary	123 result
1	Иван	35 000	35 100
2	Петр	45 000	45 100
3	Мария	54 000	54 100
4	Петр	100 000	100 100
5	Екатерина	38 900	39 000
6	Дмитрий	110 000	110 100
7	Анастасия	40 000	40 100
8	Алена	68 000	68 100
9	Артём	120 000	120 100
10	Андрей	42 000	42 100

Вычитание (-)

Оператор вычитания (-) возвращает разницу указанных операндов (значений).

Синтаксис:

- `column_name` – имя столбца или статическое значение, которое может быть задано вручную. Количество значений неограниченно;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

```

SELECT column_name_1 - column_name_2
FROM schema.table_name;

```

Практический пример: из таблицы `employees` необходимо вывести столбцы `first_name` и `salary`. Дополнительным столбцом вывести разницу столбца `salary` и значения 1500, полученному значению присвоить имя `result`.

```

SELECT first_name,
        salary,
        salary - 1500 AS result
FROM employees;

```

Выполняем запрос и получаем следующий результат. Обратите внимание на столбцы `salary` и `result`, в них хранятся разные значения.

Таблица	ABC first_name	123 salary	123 result
	Иван	35 000	33 500
2	Петр	45 000	43 500
3	Мария	54 000	52 500
4	Петр	100 000	98 500
5	Екатерина	38 900	37 400
6	Дмитрий	110 000	108 500
7	Анастасия	40 000	38 500
8	Алена	68 000	66 500
9	Артём	120 000	118 500
10	Андрей	42 000	40 500

Умножение (*)

Оператор умножения (*) возвращает результат перемножения указанных операндов (значений).

Синтаксис:

- `column_name` – имя столбца или статическое значение, которое может быть задано вручную. Количество значений неограниченно;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

```
SELECT column_name_1 * column_name_2  
FROM schema.table_name;
```

Практический пример: из таблицы `employees` необходимо вывести столбцы `first_name` и `salary`. Дополнительным столбцом вывести результат перемножения столбца `salary` и значения 2, полученному значению присвоить имя `result`.

```
SELECT first_name,  
       salary,  
       salary * 2 AS result  
FROM employees;
```

Выполняем запрос и получаем следующий результат. Обратите внимание на столбцы `salary` и `result`, в них хранятся разные значения.

Таблица		ABC first_name	123 salary	123 result
1		Иван	35 000	70 000
2		Петр	45 000	90 000
3		Мария	54 000	108 000
4		Петр	100 000	200 000
5		Екатерина	38 900	77 800
6		Дмитрий	110 000	220 000
7		Анастасия	40 000	80 000
8		Алена	68 000	136 000
9		Артём	120 000	240 000
10		Андрей	42 000	84 000

Обновить Save Cancel

Деление (/)

Оператор деления (/) возвращает результат деления указанных операндов (значений).

Синтаксис:

- `column_name` – имя столбца или статическое значение, которое может быть задано вручную. Количество значений неограниченно;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

```
SELECT column_name_1 / column_name_2
FROM schema.table_name;
```

Практический пример: из таблицы `employees` необходимо вывести столбцы `first_name` и `salary`. Дополнительным столбцом вывести результат деления столбца `salary` и значения 2, полученному значению присвоить имя `result`.

```
SELECT first_name,
       salary,
       salary / 2 AS result
FROM employees;
```

Выполняем запрос и получаем следующий результат. Обратите внимание на столбцы `salary` и `result`, в них хранятся разные значения.

	ABC first_name	123 salary	123 result
1	Иван	35 000	17 500
2	Петр	45 000	22 500
3	Мария	54 000	27 000
4	Петр	100 000	50 000
5	Екатерина	38 900	19 450
6	Дмитрий	110 000	55 000
7	Анастасия	40 000	20 000
8	Алена	68 000	34 000
9	Артём	120 000	60 000
10	Андрей	42 000	21 000

Остаток от деления (%)

Оператор остаток от деления (%) возвращает в виде результата остаток от деления указанных операндов (значений).

Синтаксис:

- `column_name` – имя столбца или статическое значение, которое может быть задано вручную. Количество значений неограниченно;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

```
SELECT column_name_1 % column_name_2
FROM schema.table_name;
```

Практический пример: из таблицы `employees` необходимо вывести столбцы `first_name` и `salary`. Дополнительным столбцом вывести остаток от деления столбца `salary` и значения 3, полученному значению присвоить имя `result`.

```
SELECT first_name,
       salary,
       salary % 3 AS result
FROM employees;
```

Выполняем запрос и получаем следующий результат. Обратите внимание на столбцы `salary` и `result`, в них хранятся разные значения.

Таблица		ABC first_name ▼	123 salary ▼	123 result ▼
	1	Иван	35 000	2
	2	Петр	45 000	0
	3	Мария	54 000	0
Текст	4	Петр	100 000	1
	5	Екатерина	38 900	2
	6	Дмитрий	110 000	2
	7	Анастасия	40 000	1
Запись	8	Алена	68 000	2
	9	Артём	120 000	0
	10	Андрей	42 000	0

Обновить Save Cancel

Операторы сравнения

Операторы сравнения (Comparison Operators) выполняют сравнение одного определенного значения столбца с другими столбцами таблицы или указанным значением.

Существуют следующие операторы сравнения:

Оператор	Описание
=	Равенство.
<> и !=	Неравенство.
< и <=	Меньше Меньше или равно.
> и >=	Больше Больше или равно.
IN	Находится ли значение в пределах указанного набора значений.
BETWEEN	Находится ли значение в пределах указанного диапазона значений.
IS NULL / IS NOT NULL	Проверка значения на NULL.
LIKE	Поиск значения или его части при помощи метасимволов.

Оператор =

Оператор = выполняет проверку равенства одного столбца таблицы с указанным значением или другим столбцом таблицы (или набором столбцов). Оператор равенства можно применять как в блоке [WHERE](#), так и в других управляющих конструкциях.

Синтаксис:

- `column_list` – список столбцов;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;

- `column_name` – имя столбца, значение которого будет сравниваться со значением `value`;
- `value` – значение, с которым будет сравниваться столбец `column_name`.

```
SELECT column_list
FROM schema.table_name
WHERE column_name = value;
```

Практический пример: из таблицы `employees` необходимо вывести всех сотрудников с именем Петр.

```
SELECT *
FROM employees
WHERE first_name = 'Петр';
```

Выполняем запрос и получаем список сотрудников с именем Петр.

	123 id	abc last_name	abc first_name	abc gender	birthday	abc email	123 id_department	123 id_boss	123 salary
1	2	Петров	Петр	Мужской	1985-12-20	petrov@yandex.ru	2	[NULL]	45 000
2	4	Петров	Петр	Мужской	1987-04-25	[NULL]	4	[NULL]	100 000

Обновить Save Cancel Экспорт данных ... 200 2 2 строк получено - 0ms, 2024-01-18

Оператор `<>` и `!=`

Оператор `<>` выполняет проверку на неравенство один столбец таблицы с указанным значением или другим столбцом таблицы (или набором столбцов). Оператору `<>` эквивалентен оператор `!=`, выполняют они одну и ту же функцию, использовать можно любой вариант.

Указанные операторы можно применять как в блоке [WHERE](#), так и в других управляющих конструкциях.

Синтаксис:

- `column_list` – список столбцов;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `column_name` – имя столбца, значение которого будет сравниваться со значением `value`;
- `value` – значение, с которым будет сравниваться столбец `column_name`.


```
SELECT column_list
FROM schema.table_name
WHERE column_name [<> | !=] value;
```

Практический пример: из таблицы `employees` необходимо вывести всех сотрудников, у которых заработная плата не равна 70 000 рублей.

```
SELECT *
FROM employees
WHERE salary <> 70000;
```

Выполняем запрос и получаем список сотрудников, у которых заработная плата не 70 000 рублей. Обратите внимание, что результат запроса на изображении показан не полностью.

	123 id	ABC last_name	ABC first_name	ABC gender	birthday	ABC email	123 id_department	123 id_boss	123 salary
1	1	Иванов	Иван	Мужской	1990-05-15	ivanov@mail.ru	1	[NULL]	35 000
2	2	Петров	Петр	Мужской	1985-12-20	petrov@yandex.ru	2	[NULL]	45 000
3	3	Сидорова	Мария	Женский	1992-08-10	sidorova@gmail.com	3	[NULL]	54 000
4	4	Петров	Петр	Мужской	1987-04-25	[NULL]	4	[NULL]	100 000
5	5	Васильева	Екатерина	Женский	1995-03-30	vasilieva@yandex.ru	5	[NULL]	38 900
6	6	Попов	Дмитрий	Мужской	1988-11-05	popov@gmail.com	6	[NULL]	110 000
7	7	Соколова	Анастасия	Женский	1991-07-12	sokolova@mail.ru	1	1	40 000
8	8	Лебедева	Алена	Женский	1994-09-18	lebedeva@yandex.ru	2	2	68 000
9	9	Козлов	Артём	Мужской	1986-10-22	kozlov@gmail.com	3	3	120 000
10	10	Новиков	Андрей	Мужской	1989-06-08	novikov@mail.ru	4	4	42 000

Оператор < и <=

Оператор < (меньше) и <= (меньше или равно) осуществляет проверку одного столбца таблицы с указанным значением или другим столбцом таблицы (или набором столбцов).

- оператор < после сравнения выводит все результаты, которые меньше указанного значения или столбца;
- оператор <= после сравнения выводит все результаты, которые меньше либо равны указанному значению или столбцу.

Указанные операторы можно применять как в блоке [WHERE](#), так и в других управляющих конструкциях.

Синтаксис:

- `column_list` – список столбцов;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;

- `column_name` – имя столбца, значение которого будет сравниваться со значением `value`;
- `value` – значение, с которым будет сравниваться столбец `column_name`.

```
SELECT column_list
FROM schema.table_name
WHERE column_name [<| <=] value;
```

Практический пример: из таблицы `employees` необходимо вывести всех сотрудников, у которых заработная плата меньше или равна 85 000 рублей.

```
SELECT *
FROM employees
WHERE salary <= 85000;
```

Выполняем запрос и получаем список сотрудников, у которых заработная плата меньше или равна 85 000 рублей. Обратите внимание, что результат запроса на изображении показан не полностью.

	123 id	ABC last_name	ABC first_name	ABC gender	birthday	ABC email	123 id_department	123 id_boss	123 salary
1	1	Иванов	Иван	Мужской	1990-05-15	ivanov@mail.ru	1	[NULL]	35 000
2	2	Петров	Петр	Мужской	1985-12-20	petrov@yandex.ru	2	[NULL]	45 000
3	3	Сидорова	Мария	Женский	1992-08-10	sidorova@gmail.com	3	[NULL]	54 000
4	5	Васильева	Екатерина	Женский	1995-03-30	vasilieva@yandex.ru	5	[NULL]	38 900
5	7	Соколова	Анастасия	Женский	1991-07-12	sokolova@mail.ru	1	1	40 000
6	8	Лебедева	Алена	Женский	1994-09-18	lebedeva@yandex.ru	2	2	68 000
7	10	Новиков	Андрей	Мужской	1989-06-08	novikov@mail.ru	4	4	42 000
8	11	Морозова	Ольга	Женский	1993-02-14	morozova@yandex.ru	5	5	70 000
9	13	Григорьева	Наталья	Женский	1996-01-25	grigoryeva@mail.ru	1	1	60 000
10	15	Максимова	Вероника	Женский	1990-08-30	maximova@gmail.ru	3	3	72 000

Оператор > и >=

Оператор `>` (больше) и `>=` (больше или равно) осуществляет проверку одного столбца таблицы с указанным значением или другим столбцом таблицы (или набором столбцов).

- оператор `>` после сравнения выводит все результаты, которые больше указанного значения или столбца;
- оператор `>=` после сравнения выводит все результаты, которые больше либо равны указанному значению или столбцу.

Указанные операторы можно применять как в блоке [WHERE](#), так и в других управляющих конструкциях.

Синтаксис:

- `column_list` – список столбцов;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `column_name` – имя столбца, значение которого будет сравниваться со значением `value`;
- `value` – значение, с которым будет сравниваться столбец `column_name`.

```
SELECT column_list
FROM schema.table_name
WHERE column_name [>| >=] value;
```

Практический пример: из таблицы `employees` необходимо вывести всех сотрудников, у которых заработная плата больше или равна 100 000 рублей.

```
SELECT *
FROM employees
WHERE salary >= 100000;
```

Выполняем запрос и получаем список сотрудников, у которых заработная плата больше или равна 100 000 рублей.

	123 id	ABC last_name	ABC first_name	ABC gender	birthday	ABC_email	123 id_department	123 id_boss	123 salary
1	9	Козлов	Артём	Мужской	1986-10-22	kozlov@gmail.com	3	3	120 000
2	12	Павлов	Сергей	Мужской	1987-12-11	pavlov@gmail.com	6	6	130 000
3	21	Григорьев	Михаил	Мужской	1988-10-25	grigoryev@gmail.co	3	9	125 000
4	42	Лебедев	Сергей	Мужской	1996-07-14	lebedev@gmail.com	6	18	120 000

Оператор IN

Оператор `IN` позволяет указать диапазон значений, которые необходимо сравнить со значением в столбце таблицы. Значения, которые указываются в операторе `IN` перечисляются через запятую.



Если в операторе `IN` перечисляются [строковые значения](#) или [дата](#), то их нужно заключать в кавычки.

Преимущества оператора IN:

- когда вы работаете с длинным списком значений, которые нужно сравнить, то использовать оператор `IN` намного удобнее;
- при использовании оператора `IN` вместе с операторами [AND](#), [OR](#) или [NOT](#) становится намного проще управлять результатами выборки;
- списки значений в операторе `IN` быстрее обрабатывается, чем списки оператора `OR` (это заметно только на больших списках);
- в операторе `IN` может содержаться ещё один `SELECT` запрос, так называемый [подзапрос](#). Это позволяет создавать очень гибкие условия для фильтрации результатов.

Синтаксис:

- `column_list` – список столбцов;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `column_name` – имя столбца, значение которого будет сравниваться со значениями `value`;
- `value` – значения, с которыми будет сравниваться столбец `column_name`.

```
-- Оператор IN
SELECT column_list
FROM schema.table_name
WHERE column_name IN (value_1, value_2, value_n);
```

```
-- Оператор IN в комбинации с оператором NOT
SELECT column_list
FROM schema.table_name
WHERE column_name NOT IN (value_1, value_2, value_n);
```

Практический пример: из таблицы `employees` необходимо вывести всех сотрудников с именем Ольга, Игорь и Максим.

```
SELECT *
FROM employees
WHERE first_name IN ('Ольга', 'Игорь', 'Максим');
```

Выполняем запрос и получаем список сотрудников с именем Ольга, Игорь и Максим.

	123 id	ABC last_name	ABC first_name	ABC gender	birthday	ABC email	123 id_department	123 id_boss	123 salary
1	11	Морозова	Ольга	Женский	1993-02-14	morozova@yandex.ru	5	5	70 000
2	14	Борисов	Игорь	Мужской	1985-05-20	borisov@yandex.ru	2	2	95 000
3	30	Прокофьев	Максим	Мужской	1988-04-05	[NULL]	6	12	100 000
4	37	Миронов	Игорь	Мужской	1988-07-15	mirovov@mail.ru	1	13	72 000
5	40	Иванова	Ольга	Женский	1987-10-25	ivanova@mail.ru	4	16	67 000
6	43	Кузнецов	Игорь	Мужской	1985-11-20	[NULL]	1	7	75 000
7	45	Пономарева	Максим	Женский	1984-05-28	ponomareva@gmail.com	3	9	108 000
8	50	Максимов	Максим	Мужской	1988-07-28	maximov@mail.ru	2	14	102 000

Оператор BETWEEN

Оператор `BETWEEN` выполняет сравнение столбца таблицы или указанного значения с заданным диапазоном. Диапазон значений должен задаваться в строгом виде, то есть: должно быть начальное (верхнее) значение диапазона и конечное (нижнее). И эти значения должны быть разделены логическим оператором [AND](#).

При помощи оператора `BETWEEN` можно сравнивать [числа](#), [строки](#) и [даты](#).

Синтаксис:

- `column_list` – список столбцов;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `column_name` – имя столбца, значение которого будет сравниваться со значениями в указанном диапазоне `start_value` и `end_value`;
- `start_value` и `end_value` – начальное и конечное значение диапазона.

```
-- Оператор BETWEEN
SELECT column_list
FROM schema.table_name
WHERE column_name BETWEEN start_value AND end_value;

-- Оператор BETWEEN в комбинации с оператором NOT
SELECT column_list
FROM schema.table_name
WHERE column_name NOT BETWEEN start_value AND end_value;
```

Практический пример: из таблицы `employees` необходимо вывести всех сотрудников, у которых заработная плата входит в диапазон от 85 000 до 95 000 рублей.

```
SELECT *
FROM employees
WHERE salary BETWEEN 85000 AND 95000;
```

Выполняем запрос и получаем список сотрудников, у которых заработная плата входит в диапазон от 85 000 до 95 000 рублей.

	123 id	ABC last_name	ABC first_name	ABC gender	birthday	ABC email	123 id_department	123 id_boss	123 salary
1	14	Борисов	Игорь	Мужской	1985-05-20	borisov@yandex.ru	2	2	95 000
2	33	Фомин	Дмитрий	Мужской	1994-02-14	fomin@gmail.com	3	9	90 000
3	44	Соколов	Михаил	Мужской	1989-08-30	sokolov@yandex.ru	2	14	85 000
4	48	Борисова	Анна	Женский	1990-10-18	borisova@yandex.ru	6	18	95 000

Обновить

Save

Cancel

Экспорт данных ...

200

4

4 строк получено - 1ms, 2024-01-18

Проверка значений NULL

При сравнении значений с NULL нельзя использовать операторы сравнения. Потому что для сравнения значений с NULL существуют два специальных оператора IS NULL и IS NOT NULL.

Оператор IS NULL

Для того, чтобы проверить является ли указанное значение NULL, нужно использовать оператор IS NULL.

Синтаксис:

- `column_list` – список столбцов;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `column_name` – имя столбца, значение которого будет проверяться на значение NULL.

```
SELECT column_list
FROM schema.table_name
WHERE column_name IS NULL;
```

Практический пример: из таблицы `employees` необходимо вывести всех сотрудников, у которых не указан адрес электронной почты.

```
SELECT *
FROM employees
WHERE email IS NULL;
```

Выполняем запрос и получаем список сотрудников, у которых не указан адрес электронной почты.

	123 id	abc last_name	abc first_name	abc gender	birthday	abc_email	123 id_department	123 id_boss	123 salary
1	4	Петров	Петр	Мужской	1987-04-25	[NULL]	4	[NULL]	100 000
2	17	Егоров	Кирилл	Мужской	1986-07-22	[NULL]	5	5	62 000
3	30	Прокофьев	Максим	Мужской	1988-04-05	[NULL]	6	12	100 000
4	43	Кузнецов	Игорь	Мужской	1985-11-20	[NULL]	1	7	75 000

Обновить

Save

Cancel

Экспорт данных ...

200

4

4 строк получено - 0ms, 2024-01-18

Оператор IS NOT NULL

Для того, чтобы проверить не является ли указанное значение NULL, нужно использовать оператор IS NOT NULL.

Синтаксис:

- `column_list` – список столбцов;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `column_name` – имя столбца, значение которого будет проверяться на значение NOT NULL.

```
SELECT column_list
FROM schema.table_name
WHERE column_name IS NOT NULL;
```

Практический пример: из таблицы `employees` необходимо вывести всех сотрудников, у которых указан адрес электронной почты.

```
SELECT *
FROM employees
WHERE email IS NOT NULL;
```

Выполняем запрос и получаем список сотрудников, у которых указан адрес электронной почты.

	123 id	abc last_name	abc first_name	abc gender	birthday	abc email	123 id_department	123 id_boss	123 salary
1	1	Иванов	Иван	Мужской	1990-05-15	ivanov@mail.ru	1	[NULL]	35 000
2	2	Петров	Петр	Мужской	1985-12-20	petrov@yandex.ru	2	[NULL]	45 000
3	3	Сидорова	Мария	Женский	1992-08-10	sidorova@gmail.com	3	[NULL]	54 000
4	5	Васильева	Екатерина	Женский	1995-03-30	vasilieva@yandex.ru	5	[NULL]	38 900
5	6	Попов	Дмитрий	Мужской	1988-11-05	popov@gmail.com	6	[NULL]	110 000
6	7	Соколова	Анастасия	Женский	1991-07-12	sokolova@mail.ru	1	1	40 000
7	8	Лебедева	Алена	Женский	1994-09-18	lebedeva@yandex.ru	2	2	68 000
8	9	Козлов	Артём	Мужской	1986-10-22	kozlov@gmail.com	3	3	120 000
9	10	Новиков	Андрей	Мужской	1989-06-08	novikov@mail.ru	4	4	42 000

Обновить Save Cancel Экспорт данных ... 200 46 ... 46 строк получено - 1ms (1ms получ.)

Оператор LIKE

Поиск данных в таблице в основном сводится к поиску уже известных значений, а как быть, если необходимо найти данные, у которых вы знаете только определенную часть? В этом случае, на помощь приходит оператор `LIKE`.

Оператор `LIKE` позволяет выполнить поиск необходимых значений с использованием метасимволов. При помощи метасимволов можно написать свой шаблон для поиска, который будет выводить необходимые результаты.

Метасимволы - специальные символы, которые применяются для поиска значений.

Шаблон поиска - условие отбора строк, состоящее из метасимволов и любой комбинации текста.

По своей сути метасимволы это специальные знаки, которые особым образом трактуются в блоке [WHERE](#). Чтобы задействовать метасимволы в условиях отбора строк, необходимо написать ключевое слово `LIKE`, это слово информирует СУБД, что следующий шаблон поиска необходимо проанализировать с учётом метасимволов, и не искать точные совпадения.



Осуществлять поиск при помощи метасимволов можно только в текстовых полях.

Метасимвол %

Метасимвол `%` (знак процента) в шаблоне поиска означает, что необходимо найти все совпадения (вхождения) любого символа.



При использовании метасимвола `%`, может показаться, что вы сможете найти все возможные значения, но это не так. Есть одно исключение, и это `NULL`.

Даже если вы воспользуетесь условием для отбора `WHERE column_name LIKE '%'`, вы всё равно не сможете вывести строку, в которой поле содержит `NULL`.

Синтаксис:

- `column_list` – список столбцов;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `column_name` – имя столбца, в котором будет выполнен поиск значений по указанному шаблону.

```
SELECT column_list
FROM schema.table_name
WHERE column_name LIKE '%';
```

Практический пример: из таблицы `employees` необходимо вывести всех сотрудников, у которых электронный адрес находится в домене `@yandex.ru`.

```
SELECT *
FROM employees
WHERE email LIKE '%@yandex.ru';
```

Выполняем запрос и получаем список сотрудников, у которых почта находится в домене `@yandex.ru`. Обратите внимание, что результат запроса на изображении показан не полностью.

	123 id	abc last_name	abc first_name	abc gender	birthday	abc email	123 id_department	123 id_boss	123 salary
1	2	Петров	Петр	Мужской	1985-12-20	petrov@yandex.ru	2	[NULL]	45 000
2	5	Васильева	Екатерина	Женский	1995-03-30	vasilieva@yandex.ru	5	[NULL]	38 900
3	8	Лебедева	Алена	Женский	1994-09-18	lebedeva@yandex.ru	2	2	68 000
4	11	Морозова	Ольга	Женский	1993-02-14	morozova@yandex.ru	5	5	70 000
5	14	Борисов	Игорь	Мужской	1985-05-20	borisov@yandex.ru	2	2	95 000
6	20	Федоров	Егор	Мужской	1993-09-18	f.eg@yandex.ru	2	14	72 000
7	23	Кудряшов	Иван	Мужской	1987-06-20	kudryashov@yandex.ru	5	11	80 000
8	26	Мельникова	Яна	Женский	1993-07-15	melnikova@yandex.ru	2	8	80 000
9	29	Ильина	Алёна	Женский	1995-09-30	ilina@yandex.ru	5	17	75 000
10	32	Карпова	Анна	Женский	1990-06-18	karp@yandex.ru	2	14	75 000

Обновить Save Cancel Экспорт данных ... 200 15 15 строк получено - 3ms, 2024-01-18

Метасимвол

Метасимвол (знак нижнего подчёркивания) в шаблоне поиска означает, что необходимо найти совпадение (вхождение) только одного символа.

Синтаксис:

- `column_list` – список столбцов;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `column_name` – имя столбца, в котором будет выполнен поиск значений по указанному шаблону.

```
SELECT column_list
FROM schema.table_name
WHERE column_name LIKE ' _ ';
```

Практический пример: из таблицы `employees`, необходимо вывести всех сотрудников, у которых имя электронного адреса состоит из 4-х символов.

Примечание к заданию:

- имя электронного адреса (часть адреса, которая находится перед символом @);
- электронный адрес может находиться в любом домене.

```
SELECT *
FROM employees
WHERE email LIKE ' ____@% ';
```

Выполняем запрос и получаем список сотрудников, у которых имя электронного адреса состоит из 4-х символов. Обратите внимание, что результат запроса на изображении показан не полностью.

	123 id	abc last_name	abc first_name	abc gender	birthday	abc email	123 id_department	123 id_boss	123 salary
1	20	Федоров	Егор	Мужской	1993-09-18	f.eg@yandex.ru	2	14	72 000
2	25	Кондратьев	Павел	Мужской	1990-12-14	kpav@mail.ru	1	7	70 000
3	32	Карпова	Анна	Женский	1990-06-18	karp@yandex.ru	2	14	75 000

Текст Обновить Save Cancel Экспорт данных ... 200 3 3 строк получено - 0ms, 2024-01-18

Экранирование символов

Для того, чтобы найти символы подчёркивания (`_`) или процента (`%`), когда используется оператор `LIKE`, необходимо использовать `ESCAPE`-символы. Данные символы используются в шаблоне поиска и указываются непосредственно перед символом подчёркивания или процента, это означает, что следующий за ним символ будет интерпретирован как обычный символ, а не как метасимвол.



В качестве `ESCAPE`-символов, принято использовать знак процента, или любой другой символ.

Синтаксис:

- `column_list` – список столбцов;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `column_name` – имя столбца, в котором будет выполнен поиск значений по указанному шаблону;
- `pattern_search` – шаблон поиска;
- `escape_symbol` – символ экранирования.

```
SELECT column_list
FROM schema.table_name
WHERE column_name LIKE pattern_search ESCAPE 'escape_symbol';
```

Практический пример: из таблицы `employees`, необходимо вывести всех сотрудников, у которых в имени электронного адреса находится символ `%`.

Примечание к заданию:

- имя электронного адреса (часть адреса, которая находится перед символом `@`);
- электронный адрес может находиться в любом домене.

Чтобы запрос вернул нужные данные, необходимо для оператора `LIKE` написать шаблон поиска и выбрать символ для экранирования:

- в качестве символа экранирования выбираем знак восклицания `ESCAPE '!'`;

Теперь рассмотрим шаблон_поиска, он выглядит следующим образом `'%!%%@%'`, рассмотрим его посимвольно:

- `%` – обозначает совпадение (вхождение) всех символов;
- `!%` – так мы экранируем символ `%`, чтобы СУБД не воспринимала его, как метасимвол;

- % – обозначает совпадение (вхождение) всех символов;
- @ – указываем для СУБД, что символ собака должен присутствовать в строке, которая будет сопоставляться с шаблоном_поиска;
- % – обозначает совпадение (вхождение) всех символов.

```
SELECT *
FROM employees
WHERE email LIKE '%!%%@%' ESCAPE '!';
```

Выполняем запрос и получаем список пользователей, у которых в имени электронного адреса содержится символ %.

	123 id	ABC last_name	ABC first_name	ABC gender	birthday	ABC email	123 id_department	123 id_boss	123 salary
1	28	Макарова	Татьяна	Женский	1992-11-18	makaro%va@mail.ru	4	10	65 000

Обновить Save Cancel Экспорт данных ... 200 1 1 строк получено - 0ms, 2024-01-18 в 13

Логические операторы

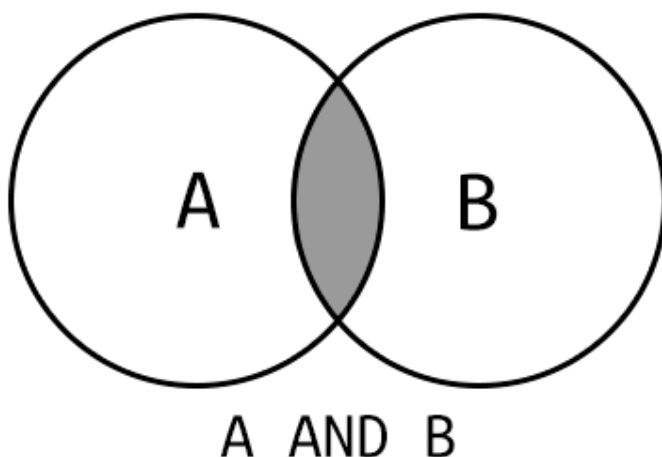
Логические операторы (Logical Operators) позволяют создать сложные и гибкие условия для фильтрации данных в блоке [WHERE](#), за счёт того, что их можно комбинировать между собой.

Существуют следующие логические операторы:

Оператор	Описание
AND	Операция И.
OR	Операция ИЛИ.
NOT	Операция НЕ или ОТРИЦАНИЕ.

Оператор AND

Логический оператор **AND** (И) позволяет осуществить проверку двух и более условий в блоке [WHERE](#). Для наглядности работы оператора **AND** ниже приведено изображение. Заштрихованная область, это данные, которые вернутся если значение **A** и **B** соответствует заданным условиям.



Синтаксис:

- `column_list` – список столбцов;

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `column_name` – имя столбца, значение которого будет сравниваться со значением `value`;
- `value` – значение, с которым будет сравниваться столбец `column_name`.

```
SELECT column_list
FROM schema.table_name
WHERE column_name_1 = value_1
      AND column_name_n = value_n;
```

Практический пример: из таблицы `employees` необходимо вывести сотрудников с именем Ольга и заработной платой больше или равно 70 000 рублей.

```
SELECT *
FROM employees
WHERE first_name = 'Ольга' AND salary >= 70000;
```

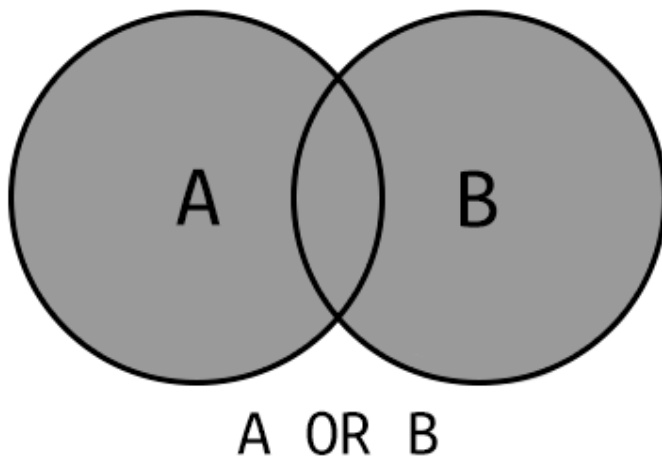
Выполняем запрос и получаем результат, в котором выведен сотрудник с именем Ольга и заработной платой больше или равной 70 000 рублей.

	123 id	ABC last_name	ABC first_name	ABC gender	birthday	ABC email	123 id_department	123 id_boss	123 salary
1	11	Морозова	Ольга	Женский	1993-02-14	morozova@yandex.ru	5	5	70 000

Оператор OR

Логический оператор `OR` (ИЛИ) позволяет осуществить проверку двух и более условий в блоке [WHERE](#). В результате запроса, будут возвращены данные, когда какое-либо из условий возвращает `TRUE`.

Для наглядности работы оператора `OR` ниже приведено изображение. Заштрихованная область, это данные, которые вернутся в том случае, когда значение `A` возвращает `TRUE`, либо значение `B` возвращает `TRUE`.



Синтаксис:

- `column_list` – список столбцов;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `column_name` – имя столбца, значение которого будет сравниваться со значением `value`;
- `value` – значение, с которым будет сравниваться столбец `column_name`.

```
SELECT column_list
FROM schema.table_name
WHERE column_name_1 = value_1
      OR column_name_n = value_n;
```

Практический пример: из таблицы `employees` необходимо вывести сотрудников с заработной платой 70 000 или 80 000 рублей.

```
SELECT *
FROM employees
WHERE salary = 70000 OR salary = 80000;
```

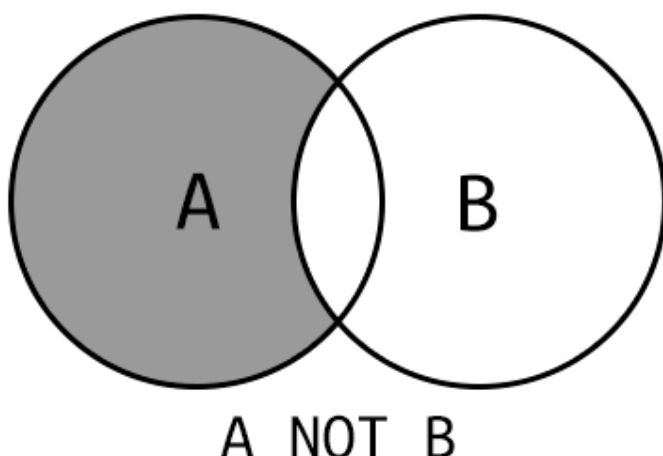
Выполняем запрос и получаем результат, в котором выведены сотрудники с заработной платой 70 000 или 80 000 рублей.

	123 id	ABC last_name	ABC first_name	ABC gender	📅 birthday	ABC email	123 id_department	123 id_boss	123 salary
1	11	Морозова	Ольга	Женский	1993-02-14	morozova@yandex.ru	5	5	70 000
2	23	Кудряшов	Иван	Мужской	1987-06-20	kudryashov@yandex.ru	5	11	80 000
3	25	Кондратьев	Павел	Мужской	1990-12-14	kpav@mail.ru	1	7	70 000
4	26	Мельникова	Яна	Женский	1993-07-15	melnikova@yandex.ru	2	8	80 000
5	31	Семенов	Артём	Мужской	1987-08-12	semenov@mail.ru	1	13	70 000
6	38	Ковалева	Елена	Женский	1995-12-30	kovaleva@yandex.ru	2	8	80 000

Оператор NOT

Оператор NOT (НЕ или ОТРИЦАНИЕ) позволяет осуществить проверку двух и более условий в блоке [WHERE](#) с отрицанием.

Для наглядности работы оператора NOT ниже приведено изображение. Заштрихованная область, это данные, которые вернутся в том случае, когда значение A возвращает TRUE.



Оператор NOT можно комбинировать с такими операторами, как [IS NULL](#), [IN](#), [BETWEEN](#), [EXISTS](#), [LIKE](#) и [ANY \(SOME\) / ALL](#);

Синтаксис:

- `column_list` – список столбцов;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `column_name` – имя столбца, значение которого будет сравниваться со значением `value`;
- `value` – значение, с которым будет сравниваться столбец `column_name`.

```
SELECT column_list
FROM schema.table_name
WHERE NOT column_name = value;
```

Практический пример: из таблицы `employees` необходимо вывести сотрудников мужского пола при помощи оператора NOT.

```
SELECT *
```

```
FROM employees
WHERE NOT gender = 'Женский';
```

Выполняем запрос и получаем результат, в котором выведены все сотрудники мужского пола. Обратите внимание, что результат запроса на изображении показан не полностью.

	123 id	ABC last_name	ABC first_name	ABC gender	birthday	ABC email	123 id_department	123 id_boss	123 salary
1	1	Иванов	Иван	Мужской	1990-05-15	ivanov@mail.ru	1	[NULL]	35 000
2	2	Петров	Петр	Мужской	1985-12-20	petrov@yandex.ru	2	[NULL]	45 000
3	4	Петров	Петр	Мужской	1987-04-25	[NULL]	4	[NULL]	100 000
4	6	Попов	Дмитрий	Мужской	1988-11-05	popov@gmail.com	6	[NULL]	110 000
5	9	Козлов	Артём	Мужской	1986-10-22	kozlov@gmail.com	3	3	120 000
6	10	Новиков	Андрей	Мужской	1989-06-08	novikov@mail.ru	4	4	42 000
7	12	Павлов	Сергей	Мужской	1987-12-11	pavlov@gmail.com	6	6	130 000
8	14	Борисов	Игорь	Мужской	1985-05-20	borisov@yandex.ru	2	2	95 000
9	17	Егоров	Кирилл	Мужской	1986-07-22	[NULL]	5	5	62 000
10	18	Пономарев	Денис	Мужской	1989-04-08	ponomarev@gmail.com	6	6	108 000

Сортировка записей (ORDER BY)

Когда результирующий набор данных получен в неотсортированном виде, то работать с ним достаточно затруднительно. Поэтому данные сначала сортируют при помощи оператора `ORDER BY`, а затем выводят или записывают в таблицу.

Направление сортировки

Существует два направления сортировки:

- по возрастанию, `ASC` (от А до Я)
- по убыванию, `DESC` (от Я до А).

Когда оператор `ORDER BY` используется без указания направления сортировки, то по умолчанию данные сортируются по возрастанию.



При выполнении сортировки нужно обращать внимание на чувствительность символов к регистру. Для уточнения этого вопроса обратитесь к администратору базы данных (DBA).

Сортировка по одному столбцу

Для сортировки результирующего набора по одному столбцу достаточно после оператора `ORDER BY` указать имя нужного столбца и направление сортировки.



Сортировку результатов можно выполнять по столбцам, которые не выводятся на экран. Другими словами, вы можете вывести в блоке [`SELECT`](#) один столбец, а выполнить сортировку по-другому, который вообще не указывается в блоке `SELECT`.

Синтаксис:

- `column_list` – список столбцов;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `condition_filter` – условия для фильтрации;

- `column_name` – имя столбца, по которому будет выполнена сортировка результирующего набора.

```
SELECT column_list
FROM schema.table_name
WHERE condition_filter
ORDER BY column_name [ASC|DESC];
```

Практический пример: из таблицы `employees` необходимо вывести список всех сотрудников, который будет отсортирован по столбцу `first_name` в порядке возрастания.

```
SELECT *
FROM employees
ORDER BY first_name ASC;
```

Выполняем запрос и получаем список сотрудников в отсортированном виде. Обратите внимание, что результат запроса на изображении показан не полностью.

	123 id	ABC last_name	ABC first_name	ABC gender	📅 birthday	ABC email	123 id_department	123 id_boss	123 salary
1	24	Белова	Александра	Женский	1986-12-15	belova@gmail.com	6	12	115 000
2	36	Макаров	Алексей	Мужской	1986-10-08	makarov@gmail.com	6	18	115 000
3	8	Лебедева	Алена	Женский	1994-09-18	lebedeva@yandex.ru	2	2	68 000
4	29	Ильина	Алёна	Женский	1995-09-30	ilina@yandex.ru	5	17	75 000
5	39	Кудряшова	Анастасия	Женский	1990-09-18	kudryashova@gmail.co	3	15	100 000
6	7	Соколова	Анастасия	Женский	1991-07-12	sokolova@mail.ru	1	1	40 000
7	10	Новиков	Андрей	Мужской	1989-06-08	novikov@mail.ru	4	4	42 000
8	48	Борисова	Анна	Женский	1990-10-18	borisova@yandex.ru	6	18	95 000
9	32	Карпова	Анна	Женский	1990-06-18	karp@yandex.ru	2	14	75 000

Сортировка по нескольким столбцам

Не всегда можно добиться желаемого результата, если выполнять сортировку только по одному столбцу. Поэтому оператор `ORDER BY` позволяет осуществлять сортировку по нескольким столбцам одновременно. Для этого необходимо указать имена столбцов через запятую, и для каждого столбца указать порядок сортировки.



Сортировка будет выполнена в том порядке, в каком указаны столбцы в блоке `ORDER BY`.

Синтаксис:

- `column_list` – список столбцов;

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `condition_filter` – условия для фильтрации;
- `column_name` – имя столбца, по которому будет выполнена сортировка результирующего набора.

```
SELECT column_list
FROM schema.table_name
WHERE condition_filter
ORDER BY column_name_1 [ASC|DESC],
        column_name_n [ASC|DESC];
```

Практический пример: из таблицы `employees` необходимо вывести список всех сотрудников, который будет отсортирован сначала по столбцу `id_department` в порядке возрастания, а затем по столбцу `first_name` в порядке убывания.

```
SELECT *
FROM employees
ORDER BY id_department ASC, first_name DESC;
```

Выполняем запрос и получаем список сотрудников в отсортированном виде. Обратите внимание, что результат запроса на изображении показан не полностью.

	123 id	ABC last_name	ABC first_name	ABC gender	birthday	ABC email	123 id_department	123 id_boss	123 salary
1	25	Кондратьев	Павел	Мужской	1990-12-14	kpav@mail.ru	1	7	70 000
2	13	Григорьева	Наталья	Женский	1996-01-25	grigoryeva@mail.ru	1	1	60 000
3	43	Кузнецов	Игорь	Мужской	1985-11-20	[NULL]	1	7	75 000
4	37	Миронов	Игорь	Мужской	1988-07-15	miroнов@mail.ru	1	13	72 000
5	1	Иванов	Иван	Мужской	1990-05-15	ivanov@mail.ru	1	[NULL]	35 000
6	49	Карпов	Денис	Мужской	1995-01-20	karpov@gmail.com	1	13	68 000
7	31	Семенов	Артём	Мужской	1987-08-12	semenov@mail.ru	1	13	70 000
8	19	Семенова	Анна	Женский	1992-02-14	semenova@mail.ru	1	7	66 000
9	7	Соколова	Анастасия	Женский	1991-07-12	sokolova@mail.ru	1	1	40 000

Сортировка по номеру столбца

Порядок сортировки можно задавать не только по именам столбцов, но и по их относительному положению в [SELECT](#) запросе.



Такой способ сортировки лучше не использовать, так как в `SELECT` запросе может измениться порядок столбцов, а порядок столбцов в блоке `ORDER BY` изменен не будет, и сортировка будет работать неправильно.

Синтаксис:

- `column_name` – имя столбца;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `condition_filter` – условия для фильтрации;
- `column_number` – относительный номер столбца в блоке `SELECT`, по которому будет выполнена сортировка результирующего набора.

```
SELECT column_name_1, column_name_2, column_name_3
FROM schema.table_name
WHERE condition_filter
ORDER BY column_number_1 [ASC|DESC],
        column_number_n [ASC|DESC];
```

Практический пример: из таблицы `employees` необходимо вывести следующие столбцы `last_name`, `first_name`, `birthday`, `id_department`. А затем отсортировать полученные данные сначала по номеру столбца `id_department` в порядке убывания, а затем по номеру столбца `birthday` в порядке возрастания.

Сначала реализуем задачу классическим способом с указанием имени столбцов:

```
SELECT last_name,
       first_name,
       birthday,
       id_department
FROM employees
ORDER BY id_department DESC, birthday ASC;
```

Теперь решаем задачу другим методом - сортировка по номеру столбца. В этом случае, имена столбцов не указываются, а указывается их положение в блоке `SELECT` запроса. Давайте разберемся с порядком столбцов:

- столбец `last_name` эквивалентен номеру 1;
- столбец `first_name` эквивалентен номеру 2;
- столбец `birthday` эквивалентен номеру 3;
- столбец `id_department` эквивалентен номеру 4.

Внесем изменения в запрос и укажем вместо имен столбцов их номера.

```

SELECT last_name,
       first_name,
       birthday,
       id_department
FROM employees
ORDER BY 4 DESC, 3 ASC;

```

Выполняем запрос и получаем список сотрудников в отсортированном виде. Обратите внимание, что результат запроса на изображении показан не полностью.

	ABC last_name	ABC first_name	birthday	123 id_department
1	Макаров	Алексей	1986-10-08	6
2	Белова	Александра	1986-12-15	6
3	Павлов	Сергей	1987-12-11	6
4	Прокофьев	Максим	1988-04-05	6
5	Попов	Дмитрий	1988-11-05	6
6	Пономарев	Денис	1989-04-08	6
7	Борисова	Анна	1990-10-18	6
8	Лебедев	Сергей	1996-07-14	6
9	Егоров	Кирилл	1986-07-22	5
10	Кудряшов	Иван	1987-06-20	5

Подзапросы (SUBQUERIES)

До этого момента мы работали с простыми запросами, а сейчас рассмотрим подзапросы. Подзапросы (SUBQUERIES) — это запросы, которые вложены в другие запросы.

Подзапросы в блоке FROM

Подзапрос можно разместить в блоке [FROM](#) основного запроса, и в дальнейшем использовать данные подзапроса, как источник данных.

Синтаксис:

- `column_list` – список столбцов;
- `subquery` – подзапрос в блоке FROM.

```
SELECT column_list  
FROM (subquery);
```

Практический пример: есть SQL-запрос, который возвращает определенный набор данных, а именно фамилию, имя, пол и дату рождения сотрудника.

```
SELECT last_name, first_name, gender, birthday  
FROM employees  
WHERE id BETWEEN 1 AND 6;
```

Результат выполнения запроса.

	ABC last_name	ABC first_name	ABC gender	🕒 birthday
1	Иванов	Иван	Мужской	1990-05-15
2	Петров	Петр	Мужской	1985-12-20
3	Сидорова	Мария	Женский	1992-08-10
4	Петров	Петр	Мужской	1987-04-25
5	Васильева	Екатерина	Женский	1995-03-30
6	Попов	Дмитрий	Мужской	1988-11-05

Дальше необходимо использовать текущий запрос в роли подзапроса в блоке FROM, и вывести только фамилию и имя сотрудника.

Пояснение к SQL-запросу:

- в блоке `FROM` находится подзапрос, он возвращает фамилию, имя, пол и дату рождения сотрудника, у которых идентификатор находится в диапазоне от 1 до 6. Блоку `FROM` присвоен псевдоним `t`;
- в блоке `SELECT` выводятся столбцы `last_name` и `first_name`, которые были получены из подзапроса в блоке `FROM`, об этом говорит псевдоним `t` перед именем столбца.

```
SELECT t.last_name, t.first_name
FROM (
    SELECT last_name, first_name, gender, birthday
    FROM employees
    WHERE id BETWEEN 1 AND 6
) t;
```

Выполняем запрос и получаем результат, в котором выведены только фамилия и имя сотрудника.

	ABC last_name	ABC first_name
1	Иванов	Иван
2	Петров	Петр
3	Сидорова	Мария
4	Петров	Петр
5	Васильева	Екатерина
6	Попов	Дмитрий

Подзапросы в блоке `WHERE`

Подзапросы в блоке [WHERE](#) можно использовать в тех случаях, когда осуществляется сравнение одного значения с другим, или множеством других значений.

Обычно подзапросы блока `WHERE` используются в связке с такими операторами, как [IN](#), [EXISTS](#) и [предикатами ANY \(SOME\) / ALL](#).

Синтаксис:

- `column_list` – список столбцов;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;

- `column_name` – имя столбца, значение которого будет сравниваться со значениями, которые возвращает подзапрос (`subquery`) в операторе `IN`;
- `subquery` – подзапрос в операторе `IN`.

```
-- Подзапрос с оператором IN
SELECT column_list
FROM schema.table_name
WHERE column_name IN (subquery);
```

Практический пример: при помощи оператора `IN` и подзапроса, необходимо вывести всю информацию о сотрудниках с именами Максим, Елена и Петр из таблицы `employees`.

Пояснение к SQL-запросу:

- в операторе `IN` указываем подзапрос для получения списка с именами сотрудников;

```
SELECT *
FROM employees
WHERE first_name IN (SELECT first_name
                     FROM employees
                     WHERE first_name IN ('Максим', 'Елена', 'Петр'));
```

Выполняем запрос. Получен результирующий набор, в котором присутствует информация о сотрудниках с именами Максим, Елена и Петр. Почему выведены именно эти записи, а не какие-нибудь другие?

Всё очень просто, в подзапросе, который используется в блоке `WHERE` информация отбирается только по сотрудникам, у которых имена Максим, Елена или Петр. И когда начинает выполняться основной запрос, то столбец `first_name` сравнивается уже со значениями, которые находятся в операторе `IN`. Другими словами, подзапрос возвращает в оператор `IN` значения, которые в дальнейшем используются для сравнения со столбцом `first_name`.

	123 id	ABC last_name	ABC first_name	ABC gender	birthday	ABC email	123 id_department	123 id_boss	123 salary
1	2	Петров	Петр	Мужской	1985-12-20	petrov@yandex.ru	2	[NULL]	45 000
2	4	Петров	Петр	Мужской	1987-04-25	[NULL]	4	[NULL]	100 000
3	16	Кузьмина	Елена	Женский	1997-08-15	kuzmina@mail.ru	4	4	105 000
4	30	Прокофьев	Максим	Мужской	1988-04-05	[NULL]	6	12	100 000
5	38	Ковалева	Елена	Женский	1995-12-30	kovaleva@yandex.ru	2	8	80 000
6	45	Пономарева	Максим	Женский	1984-05-28	ponomareva@gmail.com	3	9	108 000
7	50	Максимов	Максим	Мужской	1988-07-28	maximov@mail.ru	2	14	102 000

Подзапросы в блоке SELECT

Ещё один способ использования подзапросов заключается в том, создавать дополнительные столбцы в блоке `SELECT`, в которых будут выводиться дополнительные значения.

Синтаксис:

- `column_name` – имя столбца;
- `subquery` – подзапрос в блоке `SELECT`;
- `alias_column` – псевдоним столбца;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `condition_filter` – условия для фильтрации.

```
SELECT column_name_1,  
       ...  
       column_name_n,  
       (subquery) AS alias_column  
FROM schema.table_name  
WHERE condition_filter;
```

Практический пример: есть SQL-запрос, он возвращает определенный набор данных о сотруднике из таблицы `employees`, а именно фамилию, имя, дату рождения и идентификатор отдела, в котором он трудоустроен.

```
SELECT e.last_name,  
       e.first_name,  
       e.salary,  
       e.id_department  
FROM employees e  
WHERE e.first_name IN ('Максим', 'Елена', 'Петр');
```

Результат выполнения запроса.

	ABC last_name ▼	ABC first_name ▼	123 salary ▼	123 id_department ▼
1	Петров	Петр	45 000	2
2	Петров	Петр	100 000	4
3	Кузьмина	Елена	105 000	4
4	Прокофьев	Максим	100 000	6
5	Ковалева	Елена	80 000	2
6	Пономарева	Максим	108 000	3
7	Максимов	Максим	102 000	2

Дальше необходимо напротив каждого сотрудника вывести название отдела в дополнительном столбце name_department.

Пояснение к SQL-запросу:

- внутри скобок столбца name_department напишем подзапрос, который вернёт название отдела для каждого сотрудника, и в условии WHERE связываем таблицы departments и employees по идентификатору отдела. В этом примере мы зашли немного вперед, и затронули тему [объединения таблиц](#), данную тему мы рассмотрим позже.

```
SELECT e.last_name,
       e.first_name,
       e.salary,
       e.id_department,
       (SELECT name_department
        FROM departments
        WHERE id = e.id_department) AS name_department
FROM employees e
WHERE e.first_name IN ('Максим', 'Елена', 'Петр');
```

Выполняем запрос и получаем результат, в котором напротив каждого сотрудника указано название отдела.

	ABC last_name	ABC first_name	123 salary	123 id_department	ABC name_department
1	Петров	Петр	45 000	2	Отдел финансов
2	Петров	Петр	100 000	4	Отдел кадров
3	Кузьмина	Елена	105 000	4	Отдел кадров
4	Прокофьев	Максим	100 000	6	Отдел качества
5	Ковалева	Елена	80 000	2	Отдел финансов
6	Пономарева	Максим	108 000	3	Отдел разработки
7	Максимов	Максим	102 000	2	Отдел финансов

Оператор EXISTS и подзапросы

Оператор `EXISTS` по своей сути является предикатом, то есть оператор возвращает нам значение либо `TRUE`, либо `FALSE`. И от возвращаемого значения зависит, будет ли выполнено условие `EXISTS`.



Предикат — это логическое выражения, которое может быть, как `TRUE` (правда), `FALSE` (ложь) или `UNKNOWN` (неизвестно).

Также оператор `EXISTS` подразумевает [объединение таблиц](#). Другими словами, у нас происходит объединение внутреннего запроса (подзапроса) с внешним (основным).

Синтаксис:

- `column_name` – имя столбца;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `subquery` – подзапрос в операторе `EXISTS`.

```
SELECT column_name_1, column_name_n
FROM schema.table_name
WHERE EXISTS (subquery);
```

Практический пример: необходимо вывести все идентификаторы и названия отделов из таблицы `departments`, для которых есть соответствие в таблице `employees`. Связь таблицы осуществляется по идентификатору отдела.

Пояснение к SQL-запросу:

- внутри скобок оператора EXISTS напишем запрос, который соединит таблицы departments и employees по идентификаторам отделов;
- в блоке SELECT подзапроса выводим только идентификатор отдела, поскольку нужно проверить наличие идентификатора в таблице employees.

```
SELECT d.*
FROM departments d
WHERE EXISTS (SELECT e.id
              FROM employees e
              WHERE e.id_department = d.id);
```

Выполняем запрос и получаем результат, в котором присутствуют только 6 отделов из 9, которые прошли по условию EXISTS. Если говорить простыми словами, были получены отделы из таблицы departments, которые присутствуют в таблице employees.

	123 id	ABC name_department
1	1	Отдел маркетинга
2	2	Отдел финансов
3	3	Отдел разработки
4	4	Отдел кадров
5	5	Отдел логистики
6	6	Отдел качества

Оператор INSERT

Оператор `INSERT` предназначен для добавления новых записей в таблицу базы данных. Существует несколько вариаций использования оператора `INSERT`, которые будут рассмотрены в этой главе.



По завершению изучения данной главы, восстановите данные в таблицах `employees` и `departments`. Для этого необходимо выполнить весь код из главы – [Тестовые данные для работы](#).

Добавление одной полной строки

Добавление одной полной строки в таблицу реализуется при помощи базового синтаксиса оператора `INSERT`.

Синтаксис:

1 вариант. Сначала указывается ключевое слово `INSERT INTO`, далее указывается `table_name`, в которую нужно добавить данные. В скобках перечисляются имена всех столбцов таблицы, в которые нужно добавить информацию. Затем указывается ключевое слово `VALUES`, и только потом перечисляются значения в скобках, которые будут добавлены в таблицу.

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `column_name` – имя столбца;
- `value` – значение, которое будет вставлено в столбец `column_name`.

```
INSERT INTO schema.table_name (column_name_1, column_name_n)
VALUES (value_1, value_n);
```

2 вариант. В этом варианте не указывается список столбцов таблицы, в которые будут добавляться данные. Этот вариант небезопасен, так как можно ошибиться и добавить данные не в тот столбец. Поэтому лучше использовать первый вариант.

- `schema` – наименование схемы, в которой находится объект;

- `table_name` – имя таблицы;
- `value` – значение, которое будет вставлено в столбец.

-- 2 вариант синтаксиса

```
INSERT INTO schema.table_name
VALUES (value_1, value_n);
```

Практический пример: в таблицу `departments` необходимо добавить две новых записи со следующими значениями:

- `id = 10, name_department = Отдел мониторинга;`
- `id = 11, name_department = Отдел аналитики.`

```
INSERT INTO departments (id, name_department)
VALUES (10, 'Отдел мониторинга');
```

```
INSERT INTO departments (id, name_department)
VALUES (11, 'Отдел аналитики');
```

Выполняем запрос и получаем сообщение, что записи успешно добавлены в таблицу. Далее выводим все данные из таблицы `departments` и видим, что в таблице появилось две новых записи.

Таблица	123 id		ABC name_department	
	1	2	3	4
1	1	2	Отдел маркетинга	
2	2	3	Отдел финансов	
3	3	4	Отдел разработки	
4	4	5	Отдел кадров	
5	5	6	Отдел логистики	
6	6	7	Отдел качества	
7	7	8	Отдел взыскания	
8	8	9	Отдел безопасности	
9	9	10	Отдел поддержки	
10	10	11	Отдел мониторинга	←
11	11		Отдел аналитики	←

Обновить Save Cancel

Добавление части строки

Мы уже знаем, как добавить одну полную строку в таблицу. И основываясь на этих знаниях, мы можем модифицировать метод и добавлять только часть строки. Отличие этого способа от предыдущего лишь в том, что мы можем пропускать определенные столбцы.



Если вы пропускаете столбец, для которого не допускаются значения `NULL` и не заданы значения по умолчанию. СУБД выведет ошибку, и строка не будет добавлена.

Синтаксис:

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `column_name` – имя столбца;
- `value` – значение, которое будет вставлено в столбец `column_name`.

```
INSERT INTO schema.table_name (column_name_1)
VALUES (value_1);
```

Практический пример: предположим такую ситуацию, что в компании создается новый отдел, в который уже трудоустраивают новых сотрудников. Но название отдела пока не утвердили. Поэтому в таблицу `departments` нужно добавить только идентификатор отдела (12), а название не указывать.

Задачу можно решить двумя способами:

- **1 способ:** заполняем данными только столбец `id` и добавляем в него значение 12;
- **2 способ:** заполняем данными оба столбца. В столбец `id` добавляем значение 12, а в столбец `name_department` добавляем значение `NULL`.

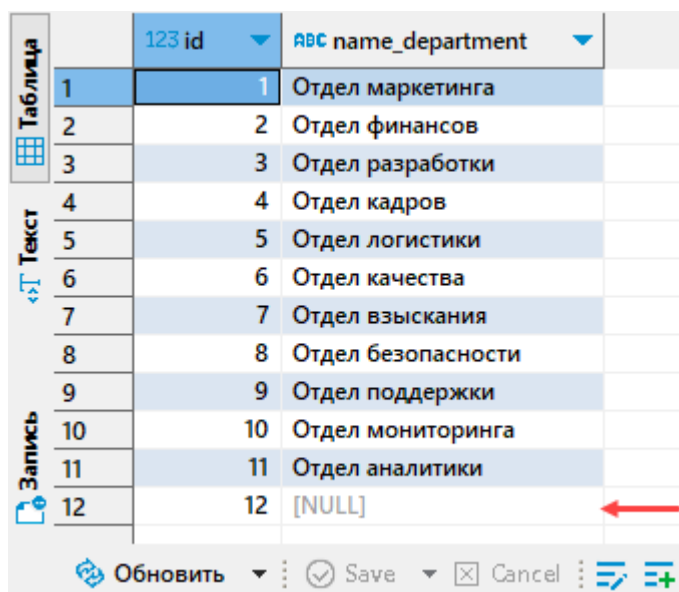
-- 1 способ

```
INSERT INTO departments (id)
VALUES (12);
```

-- 2 способ

```
INSERT INTO departments (id, name_department)
VALUES (12, null);
```

Выполняем запрос и получаем результат, в котором видно, что для отдела с идентификатором 12 не указано название.



	123 id	ABC name_department
1	1	Отдел маркетинга
2	2	Отдел финансов
3	3	Отдел разработки
4	4	Отдел кадров
5	5	Отдел логистики
6	6	Отдел качества
7	7	Отдел взыскания
8	8	Отдел безопасности
9	9	Отдел поддержки
10	10	Отдел мониторинга
11	11	Отдел аналитики
12	12	[NULL]

Добавление нескольких строк

При помощи оператора `INSERT` можно добавлять данные в таблицу не только по одной строке, но и несколько строк одновременно.

Это очень удобно и экономит ресурсы сервера (за счёт того, что оператор `INSERT` будет вызван один раз).

Синтаксис:

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `column_name` – имя столбца;
- `value` – значение, которое будет вставлено в столбец `column_name`.

```
INSERT INTO schema.table_name (column_name_1, ..., column_name_n)
VALUES (value_1, value_2, ..., value_n),
      ...,
      (value_1, value_2, ..., value_n);
```

Практический пример: в таблицу `departments` необходимо добавить две новых записи со следующими значениями:

- `id = 13, name_department = Отдел судопроизводства;`

- `id = 14, name_department = Отдел клининга.`

```
INSERT INTO departments (id, name_department)
VALUES (13, 'Отдел судопроизводства'),
      (14, 'Отдел клининга');
```

Выполняем запрос и получаем сообщение, что записи успешно добавлены в таблицу. Далее выводим все данные из таблицы `departments` и видим, что в таблице появилось две новых записи.

	123 id	ABC name_department
1	1	Отдел маркетинга
2	2	Отдел финансов
3	3	Отдел разработки
4	4	Отдел кадров
5	5	Отдел логистики
6	6	Отдел качества
7	7	Отдел взыскания
8	8	Отдел безопасности
9	9	Отдел поддержки
10	10	Отдел мониторинга
11	11	Отдел аналитики
12	12	[NULL]
13	13	Отдел судопроизводства
14	14	Отдел клининга

Добавление результатов запроса (INSERT SELECT)

Обычно оператор `INSERT` используется для добавления строк, у которых явно заданы значения. Однако, существует ещё одна форма оператора `INSERT`, и она называется `INSERT SELECT`. Эта форма выполняет тоже самое, что оператор [SELECT](#) и [INSERT](#) по отдельности.

Синтаксис:

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `column_name` – имя столбца;
- `table_name_data` – таблица, из которой будут извлечены данные;
- `subquery` – подзапрос.

```

-- 1 вариант
INSERT INTO schema.table_name (column_name_1, column_name_n)
SELECT column_name_1,
       ...,
       column_name_n
FROM schema.table_name_data;

-- 2 вариант
INSERT INTO schema.table_name (column_name_1, column_name_n)
SELECT column_name_1,
       ...,
       column_name_n
FROM (
    subquery
);

```

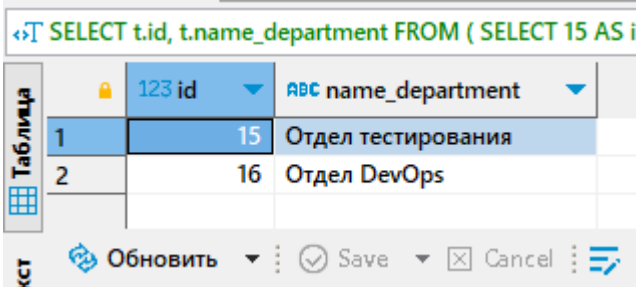
Практический пример: есть SQL-запрос, он возвращает список новых отделов в компании.

```

SELECT t.id, t.name_department
FROM (
    SELECT 15 AS id, 'Отдел тестирования' AS name_department
    UNION ALL
    SELECT 16 AS id, 'Отдел DevOps' AS name_department
) t;

```

Результат, который возвращает запрос.



The screenshot shows a database client interface. At the top, a SQL query is entered: `SELECT t.id, t.name_department FROM (SELECT 15 AS i`. Below the query, a table displays the results. The table has two columns: 'id' and 'name_department'. The first row shows '15' and 'Отдел тестирования'. The second row shows '16' and 'Отдел DevOps'. The table is titled 'Таблица' (Table) on the left. At the bottom, there are buttons for 'Обновить' (Refresh), 'Save', and 'Cancel'.

	123 id	ABC name_department
1	15	Отдел тестирования
2	16	Отдел DevOps

Необходимо добавить полученный список новых отделов в таблицу departments.

Перед запросом, который возвращает список новых отделов добавляем ключевое слово `INSERT INTO`, затем указываем имя таблицы `departments`, а в скобках перечисляем имена столбцов, в которые нужно добавить данные.

```
INSERT INTO departments (id, name_department)
SELECT t.id, t.name_department
FROM (
    SELECT 15 AS id, 'Отдел тестирования' AS name_department
    UNION ALL
    SELECT 16 AS id, 'Отдел DevOps' AS name_department
) t;
```

Выполняем запрос, а затем выводим все данные из таблицы `departments`. В таблице появилось две новых записи.

Таблица	123 id		ABC name_department	
Текст	1	1	Отдел маркетинга	
	2	2	Отдел финансов	
	3	3	Отдел разработки	
	4	4	Отдел кадров	
	5	5	Отдел логистики	
	6	6	Отдел качества	
	7	7	Отдел взыскания	
	8	8	Отдел безопасности	
	9	9	Отдел поддержки	
	10	10	Отдел мониторинга	
	11	11	Отдел аналитики	
	12	12	[NULL]	
Запись	13	13	Отдел судопроизводства	
	14	14	Отдел клининга	
	15	15	Отдел тестирования	←
	16	16	Отдел DevOps	←

Обновить Save Cancel

Оператор UPDATE

Оператор `UPDATE` позволяет обновлять (модифицировать) данные в таблице. Существует два способа обновления данных:

- обновление определенного столбца в таблице;
- обновление нескольких столбцов в таблице.



При использовании оператора `UPDATE`, всегда указывайте предложение [`WHERE`](#). Если не указано предложение `WHERE`, будут обновлены все данные в таблице.

По завершению изучения данной главы, восстановите данные в таблицах `employees` и `departments`. Для этого необходимо выполнить весь код из главы – [Тестовые данные для работы](#).

Обновление определенного столбца в таблице

Чтобы обновить значение определенного столбца в таблице, необходимо указать его имя и новое значение.

Синтаксис:

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `column_name` – имя столбца;
- `value` – новое значение для столбца;
- `condition_filter` – условия для фильтрации.

```
UPDATE schema.table_name  
SET column_name = value  
WHERE condition_filter;
```

Практический пример: в таблице `employees` необходимо обновить информацию о сотруднике с идентификатором 29, поскольку сотрудник сменил фамилию с Ильина на Соболева.

Пояснение к SQL-запросу:

- в блоке `UPDATE` указываем таблицу `employees` для обновления данных;
- в блоке `SET` указываем имя столбца `last_name` и присваиваем новое значение Соболева;
- в блоке `WHERE` указываем условие, которое будет определять, у какого сотрудника нужно обновить информацию.

```
UPDATE employees
SET last_name = 'Соболева'
WHERE id = 29;
```

Выполняем запрос и получаем сообщение, данные в таблице успешно обновлены. Теперь выведем информацию о сотруднике с идентификатором 29.

```
SELECT *
FROM employees
WHERE id = 29;
```

Из полученного результата видно, что у сотрудника было обновлено значение в столбце `last_name`.

	123 id	ABC last_name	ABC first_name	ABC gender	birthday	ABC email	123 id_department	123 id_boss	123 salary
1	29	Соболева	Алёна	Женский	1995-09-30	ilina@yandex.ru	5	17	75 000

Обновление нескольких столбцов в таблице

Чтобы обновить значения нескольких столбцов в таблице, необходимо указать их имена и новые значения через запятую.

Синтаксис:

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `column_name` – имя столбца;
- `value` – новое значение для столбца;
- `condition_filter` – условия для фильтрации.

```
UPDATE schema.table_name
SET column_name_1 = value_1, column_name_n = value_n
WHERE condition_filter;
```


Практический пример: в таблице `employees` необходимо обновить информацию о сотруднике с идентификатором 36, а именно, изменить идентификатор отдела с 6 на 5, и заработную плату с 115 000 на 130 000.

Пояснение к SQL-запросу:

- в блоке `UPDATE` указываем таблицу `employees` для обновления данных;
- в блоке `SET` указываем для столбцов `id_department` и `salary` новые значения;
- в блоке `WHERE` указываем условие, которое будет определять, у какого сотрудника нужно обновить информацию.

```
UPDATE employees
SET id_department = 5, salary = 130000
WHERE id = 36;
```

Выполняем запрос и получаем сообщение, данные в таблице успешно обновлены. Теперь выведем информацию о сотруднике с идентификатором 36.

```
SELECT *
FROM employees
WHERE id = 36;
```

Из полученного результата видно, что у сотрудника были обновлены значения в столбцах `id_department` и `salary`.

	123 id	ABC last_name	ABC first_name	ABC gender	birthday	ABC email	123 id_department	123 id_boss	123 salary
1	36	Макаров	Алексей	Мужской	1986-10-08	makarov@gmail.com	5	18	130 000

Обновить Save Cancel Экспорт данных ... 200 1 1 строк получено - 0ms, 2024-01-23

Оператор DELETE

Оператор `DELETE` предназначен для удаления записей в таблице базы данных.



По завершению изучения данной главы, восстановите данные в таблицах `employees` и `departments`. Для этого необходимо выполнить весь код из главы – [Тестовые данные для работы](#).

Удаление всех строк

Для удаления всех строк в таблице, нужно воспользоваться оператором `DELETE` и опустить блок [WHERE](#).



Если необходимо быстро удалить все строки из таблицы, используйте оператор [TRUNCATE TABLE](#). Принцип работы тот же, но гораздо быстрее, так как действия этого оператора не регистрируются в журнале СУБД.

Синтаксис:

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

```
DELETE FROM schema.table_name;
```

Практический пример: из таблицы `departments` необходимо удалить все строки.

```
DELETE FROM departments;
```

Выполняем запрос, а затем выводим все данные из таблицы `departments`. Данных в таблице нет, так как оператор `DELETE` удалил все записи.

Таблица	123 id	ABC name_department
Текст		

Удаление определенных строк

Для удаления определенных строк в таблице используется оператор `DELETE` и блок [WHERE](#) с условиями фильтрации.

Синтаксис:

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `condition_filter` – условия для фильтрации.

```
DELETE FROM schema.table_name
WHERE condition_filter;
```

Практический пример: из таблицы `employees` необходимо удалить всех сотрудников, у которых идентификатор входит в диапазон от 7 до 50.

```
DELETE FROM employees
WHERE id BETWEEN 7 AND 50;
```

Выполняем запрос, а затем выводим все данные из таблицы `employees`. В таблице остались только сотрудники с идентификаторами от 1 до 6, так как оператор `DELETE` удалил записи о других сотрудниках.

Таблица	123 id	ABC last_name	ABC first_name	ABC gender	birthday	ABC email	123 id_department	123 id_boss	123 salary
Текст	1	Иванов	Иван	Мужской	1990-05-15	ivanov@mail.ru	1	[NULL]	35 000
	2	Петров	Петр	Мужской	1985-12-20	petrov@yandex.ru	2	[NULL]	45 000
	3	Сидорова	Мария	Женский	1992-08-10	sidorova@gmail.com	3	[NULL]	54 000
	4	Петров	Петр	Мужской	1987-04-25	[NULL]	4	[NULL]	100 000
	5	Васильева	Екатерина	Женский	1995-03-30	vasilieva@yandex.ru	5	[NULL]	38 900
	6	Попов	Дмитрий	Мужской	1988-11-05	popov@gmail.com	6	[NULL]	110 000

Оператор TRUNCATE TABLE

Оператор `TRUNCATE TABLE` используется для удаления всех записей из таблицы, без дальнейшей возможности их восстановления. Данный оператор выполняет ту же функцию, как и оператор `DELETE`, но без условия `WHERE`.

Данный оператор используется в тех случаях, когда необходимо быстро и полностью очистить многомиллионную таблицу. Он работает гораздо быстрее, чем оператор [DELETE](#).



Когда происходит удаление данных при помощи оператора `TRUNCATE TABLE`, таблица удаляется и создаётся заново. Все индексы и установленные [права доступа](#) на таблицу пересоздаются, а партии (секции) не пересоздаются.

По завершению изучения данной главы, восстановите данные в таблицах `employees` и `departments`. Для этого необходимо выполнить весь код из главы – [Тестовые данные для работы](#).

Синтаксис:

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

```
TRUNCATE TABLE schema.table_name;
```

Практический пример: необходимо очистить таблицу `employees` при помощи оператора `TRUNCATE TABLE`.

```
TRUNCATE TABLE employees;
```

Выполняем запрос, а затем выводим все данные из таблицы `employees`. Данных в таблице нет, так как оператор `TRUNCATE TABLE` удалил все данные.

	123 id	ABC last_name	ABC first_name	ABC gender	birthday	ABC email	123 id_department	123 id_boss	123 salary
Таблица									
Текст									

Обновить Save Cancel Экспорт данных ... 200 0 Нет данных - 0ms, 2024-01-20

Оператор DROP

Оператор `DROP` используется для удаления [объектов базы данных](#). В качестве объектов базы данных выступают такие сущности, как: таблицы, представления, материализованные представления, триггеры, последовательность, синонимы, процедуры, функции, пакеты и т.д.



По завершению изучения данной главы, восстановите данные в таблицах `employees` и `departments`. Для этого необходимо выполнить весь код из главы – [Тестовые данные для работы](#).

Синтаксис:

- `OBJECT` – тип объекта базы данных, который нужно удалить, например (`table`, `view` и т.д.);
- `IF EXISTS` – если объект существует, он будет удален. Если объект не существует, то данная конструкция позволит проигнорировать ошибку;
- `schema` – наименование схемы, в которой находится объект;
- `name_object` – имя объекта, который требуется удалить;
- `CASCADE` – необязательный параметр. Указывает на каскадное удаление, то есть удаление самого объекта и всех объектов, которые зависят от него;
- `RESTRICT` – необязательный параметр. Указывает на ограничение удаления, то есть удаление самого объекта только в том случае, если на него нет зависимостей.

```
DROP [OBJECT] [IF EXISTS] schema.name_object [CASCADE | RESTRICT];
```

Практический пример: необходимо выполнить удаление таблиц `employees` и `departments`.

```
DROP TABLE IF EXISTS employees;  
DROP TABLE IF EXISTS departments;
```

Если сейчас попробовать вывести данные из таблицы `employees` или `departments`, то будет получена ошибка, что объекта не существует.

Оператор MERGE

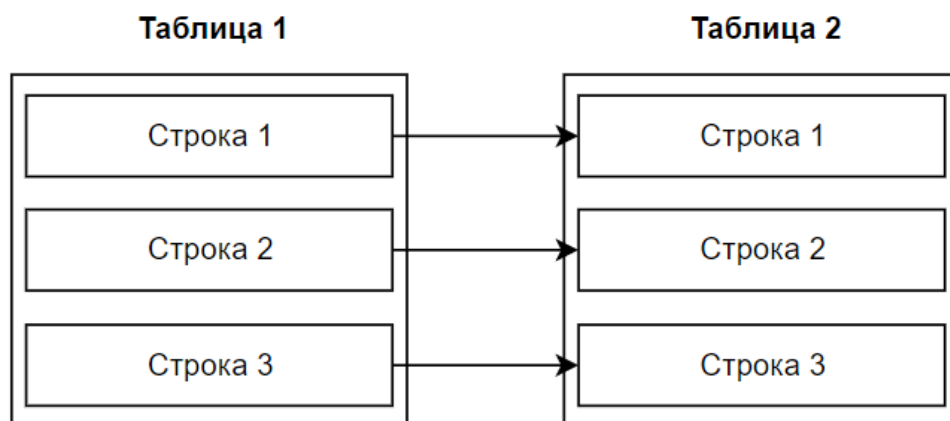
Оператор `MERGE` позволяет выбрать строки из одной таблицы для обновления или вставки в другую таблицу. По своей сути оператор `MERGE` это совмещение двух операторов [INSERT](#) и [UPDATE](#). Использование данного оператора позволяет сократить объём кода.



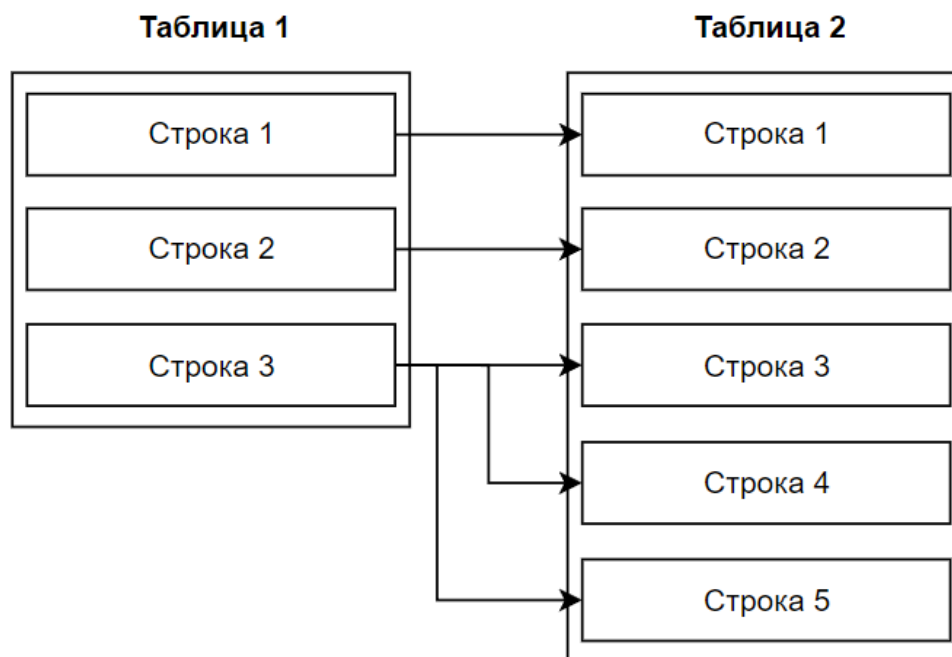
По завершению изучения данной главы, восстановите данные в таблицах `employees` и `departments`. Для этого необходимо выполнить весь код из главы – [Тестовые данные для работы](#).

При работе с данным оператором, нужно быть предельно внимательным и не допускать появления дубликатов строк при [объединении таблиц](#) (данные должны объединяться 1 к 1, а не 1 ко многим). Так как при наличии дубликатов, оператор `MERGE` завершится с ошибкой и не будет выполнен (другими словами, оператор `MERGE` увидит две записи, и не сможет определить с какой строкой ему работать).

На рисунке ниже продемонстрировано объединение 1 к 1. Одна строка из таблицы 1, объединена с одной строкой из таблицы 2.



На рисунке ниже продемонстрировано объединение 1 ко многим. Одна строка из таблицы 1, объединена с несколькими строками из таблицы 2.



Синтаксис:

При использовании данного оператора, не обязательно указывать два блока: `WHEN MATCHED` и `WHEN NOT MATCHED`. Можно указывать один блок, основываясь на вашей задаче.

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `subquery | table_name | view_name` – может быть, как подзапрос, таблица или представление;
- `column_name` – имя столбца;
- `value` – значение для столбца;
- `WHEN MATCHED THEN` – действие, которое будет выполнено при успешном объединении в блоке `ON`;
- `WHEN NOT MATCHED THEN` – действие, которое будет выполнено при неудачном объединении в блоке `ON`;
- `condition_filter` – условия для фильтрации исходной таблицы.

```
MERGE INTO schema.table_name t
USING (subquery | table_name | view_name) c
ON (t.column_name_1 = c.column_name_1)
WHEN MATCHED THEN
    UPDATE SET column_name_1 = c.value_1 ..
```

```
WHEN NOT MATCHED THEN
    INSERT(t.column_name_1) VALUES(c.value_1)
WHERE condition_filter;
```

Чтобы было понятно, как работает данный оператор, разберем его построчно:

1. Указывается имя таблицы, которая будет обновлена.

```
MERGE INTO schema.table_name t
```

2. Указываем источник данных, из которого необходимо обновить или вставить данные в таблицу из пункта 1. В качестве источника может выступать подзапрос, таблица или представление.

```
USING (subquery|table_name|view_name) c
```

3. Указываем условия объединения, при котором оператор `MERGE` либо будет обновлять, либо вставлять данные. Если необходимо указать несколько условий для объединения, то их нужно комбинировать при помощи [логических операторов](#).

```
ON (t.column_name_1 = c.column_name_1)
```

4. В этом блоке указываются действия, которые наступают только при успешном объединении в блоке `ON`. Другими словами, если при объединении найдено совпадение, то будет произведено обновление. Обратите внимание, что псевдоним не указывается для столбца, который нужно обновить.

Если необходимо обновить несколько значений, то их нужно указать через запятую.

```
WHEN MATCHED THEN
    UPDATE SET column_name_1 = c.value_1 ..
```

5. Данный блок выполняется, когда условие `ON` завершилось неудачей и не вернуло совпадений.

```
WHEN NOT MATCHED THEN
    INSERT(t.column_name_1) VALUES(c.value_1);
```

6. В этом блоке указываются дополнительные условия фильтрации для исходной таблицы, которая задаётся в 1 пункте. Это позволяет сократить объём выборки.

```
WHERE condition_filter;
```


Практический пример: необходимо в таблице `employees` выполнить обновление электронного адреса у сотрудников при помощи оператора `MERGE`. Суть обновления заключается в том, чтобы заменить в электронном адресе домен `@yandex.ru` на `@yahoo.com`.

Пояснение к SQL-запросу:

- в блоке `MERGE INTO` указываем имя таблицы, в которой нужно обновить данные;
- в блоке `USING` указываем подзапрос, который выполняет замену домена `@yandex.ru` на `@yahoo.com` в электронном адресе сотрудника, и выводит в запросе идентификатор, имя и обновленный электронный адрес сотрудника. Для замены домена используется функция [REPLACE](#);
- в блоке `ON` указываем условия для объединения таблицы `employees` и подзапроса. В данном случае, объединение будет выполнено по идентификатору и имени сотрудника;
- блок `WHEN MATCHED THEN` отвечает за успешное объединение таблицы `employees` и подзапроса. Поэтому указываем действие, которое будет выполнено при успешном объединении. Действие заключается в том, чтобы обновить значение в столбце `email` таблицы `employees`.

```
MERGE INTO employees e
USING (
    SELECT id,
           first_name,
           REPLACE(email, '@yandex.ru', '@yahoo.com') AS new_email
    FROM employees
    WHERE email LIKE '%yandex.ru'
) c
ON (e.id = c.id AND e.first_name = c.first_name)
WHEN MATCHED THEN
    UPDATE SET email = c.new_email;
```

После выполнения оператора `MERGE`, нужно вывести всех сотрудников с электронной почтой в домене `@yahoo.com`.

```
SELECT *
FROM employees
WHERE email LIKE '@yahoo.com';
```

Выполняем запрос и получаем список сотрудников, у которых почта находится в домене @yahoo.com.

		123 id	ABC last_name	ABC first_name	ABC gender	birthday	ABC email	123 id_department	123 id_boss	123 salary
Таблица	1	2	Петров	Петр	Мужской	1985-12-20	petrov@yahoo.com	2	[NULL]	45 000
	2	5	Васильева	Екатерина	Женский	1995-03-30	vasilieva@yahoo.com	5	[NULL]	38 900
	3	8	Лебедева	Алена	Женский	1994-09-18	lebedeva@yahoo.com	2	2	68 000
	4	11	Морозова	Ольга	Женский	1993-02-14	morozova@yahoo.com	5	5	70 000
	5	14	Борисов	Игорь	Мужской	1985-05-20	borisov@yahoo.com	2	2	95 000
	6	20	Федоров	Егор	Мужской	1993-09-18	f.eg@yahoo.com	2	14	72 000
	7	23	Кудряшов	Иван	Мужской	1987-06-20	kudryashov@yahoo.com	5	11	80 000
	8	26	Мельникова	Яна	Женский	1993-07-15	melnikova@yahoo.com	2	8	80 000
	9	29	Ильина	Алёна	Женский	1995-09-30	ilina@yahoo.com	5	17	75 000
	10	32	Карпова	Анна	Женский	1990-06-18	karp@yahoo.com	2	14	75 000
	11	35	Кузнецова	Екатерина	Женский	1991-07-22	kuznetsova@yahoo.com	5	17	72 000
	12	38	Ковалева	Елена	Женский	1995-12-30	kovaleva@yahoo.com	2	8	80 000
	13	41	Попова	Екатерина	Женский	1993-01-08	popova@yahoo.com	5	11	73 000
	14	44	Соколов	Михаил	Мужской	1989-08-30	sokolov@yahoo.com	2	14	85 000
	15	48	Борисова	Анна	Женский	1990-10-18	borisova@yahoo.com	6	18	95 000

Обновить Save Cancel ... Экспорт данных ... 200 15 ... 15 строк получено - 1ms, 2024-01-23

Агрегатные функции

Агрегатные функции (Aggregate Function) — это функции, которые выполняют арифметические операции над набором данных и возвращают итоговое значение. Существует очень много агрегатных функций, но в этой главе мы рассмотрим только те функции, которые используются чаще всего.

Функция	Описание
<code>SUM()</code>	Возвращает сумму значений.
<code>COUNT()</code>	Возвращает количество значений.
<code>AVG()</code>	Возвращает среднее значение.
<code>MAX()</code>	Возвращает максимальное количество.
<code>MIN()</code>	Возвращает минимальное количество.

Важные моменты при работе с агрегатными функциями:

- в большинстве случаев, агрегатные функции применяются с оператором [GROUP BY](#) (об этом поговорим в следующей главе). Если в запросе не используется оператор `GROUP BY`, то это эквивалентно группировке всех строк;
- если в столбце есть значение `NULL`, оно будет проигнорировано;
- можно использовать несколько агрегатных функций в одном запросе.

Синтаксис:

- `aggregate_function` – имя агрегатной функции `SUM()`, `COUNT()`, `AVG()`, `MAX()`, `MIN()` и т.д.;
- `column_name` – имя столбца;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `condition_filter` – условия для фильтрации.

-- Правильный синтаксис

```
SELECT aggregate_function(column_name)
```

```
FROM schema.table_name
WHERE condition_filter;
```

```
-- Неправильный синтаксис
SELECT aggregate_function(column_name),
       column_name_1,
       ...,
       column_name_n
FROM schema.table_name
WHERE condition_filter;
```

Функция SUM()

Функция SUM() возвращает сумму всех значений в указанном столбце.

Синтаксис:

- column_name – имя столбца;
- schema – наименование схемы, в которой находится объект;
- table_name – имя таблицы;
- condition_filter – условия для фильтрации.

```
SELECT SUM(column_name)
FROM schema.table_name
WHERE condition_filter;
```

Практический пример: необходимо определить общую сумму, которую тратит компания на выплату заработной платы своим сотрудникам. Полученному значению присвоить псевдоним sum_salary.

```
SELECT sum(salary) AS sum_salary
FROM employees;
```

Выполняем запрос и получаем результат в виде суммы всех значений столбца salary.

Таблица	123 sum_salary	
1	4 050 900	
Текст		
Обновить Save Cancel		

Функция COUNT()

Функция COUNT() позволяет вычислить количество значений в столбце (значения NULL не учитываются).

Синтаксис:

- column_name – имя столбца;
- schema – наименование схемы, в которой находится объект;
- table_name – имя таблицы;
- condition_filter – условия для фильтрации.

```
SELECT COUNT(column_name)
FROM schema.table_name
WHERE condition_filter;
```

Практический пример: необходимо определить количество строк в столбце email таблицы employees. Полученному значению присвоить псевдоним cnt_email.

```
SELECT count(email) AS cnt_email
FROM employees;
```

Выполняем запрос и получаем результат. Обратите внимание, что получено значение 46, а сотрудников 50. Возникает вопрос, а где еще 4 строки? Это как раз показывает работу агрегатной функции со значением NULL. У некоторых сотрудников значение в столбце email не заполнено, поэтому агрегатная функция проигнорировала эти строки.

Таблица	123 cnt_email	
1	46	
Текст		
Обновить Save Cancel		

Функция AVG()

Функция `AVG()` позволяет рассчитать среднее значение для указанного столбца.

Синтаксис:

- `column_name` – имя столбца;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `condition_filter` – условия для фильтрации.

```
SELECT AVG(column_name)
FROM schema.table_name
WHERE condition_filter;
```

Практический пример: необходимо определить среднее значение в столбце `salary` таблицы `employees`. Полученному значению присвоить псевдоним `avg_salary`.

```
SELECT AVG(salary) AS avg_salary
FROM employees;
```

Выполняем запрос и получаем результат со средним значением столбца `salary`.

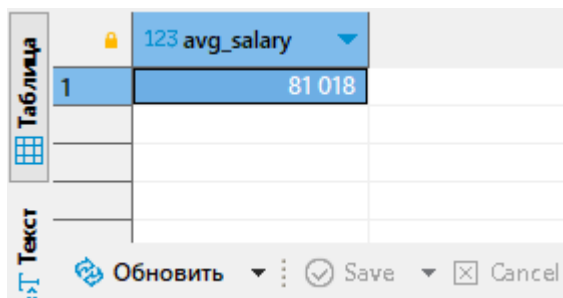


Таблица	123 avg_salary	
1	81 018	

Функция MAX()

Функция `MAX()` позволяет определить максимальное значение в указанном столбце.

Синтаксис:

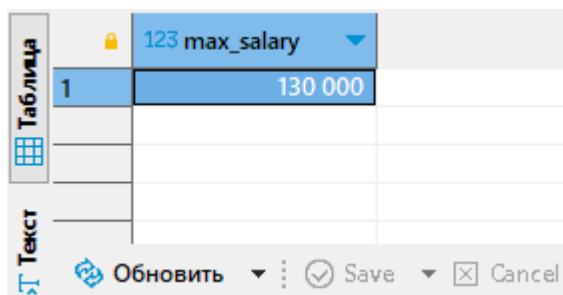
- `column_name` – имя столбца;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `condition_filter` – условия для фильтрации.

```
SELECT MAX(column_name)
FROM schema.table_name
WHERE condition_filter;
```

Практический пример: необходимо определить наибольшее значение в столбце `salary` таблицы `employees`. Полученному значению присвоить псевдоним `max_salary`.

```
SELECT MAX(salary) AS max_salary
FROM employees;
```

Выполняем запрос и получаем результат с наибольшим значением столбца `salary`.



	123 max_salary
1	130 000

Функция MIN()

Функция `MIN()` позволяет определить минимальное значение в указанном столбце.

Синтаксис:

- `column_name` – имя столбца;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `condition_filter` – условия для фильтрации.

```
SELECT MIN(column_name)
FROM schema.table_name
WHERE condition_filter;
```

Практический пример: необходимо определить наименьшее значение в столбце `salary` таблицы `employees`. Полученному значению присвоить псевдоним `min_salary`.

```
SELECT MIN(salary) AS min_salary
FROM employees;
```

Выполняем запрос и получаем результат с наименьшим значением столбца `salary`.

Таблица	123 min_salary	
	1	35 000
Текст		
Обновить Save Cancel		

Группировка данных (GROUP BY)

Оператор `GROUP BY` используется в SQL-запросе тогда, когда необходимо выполнить группировку данных по одному или нескольким столбцам. Группировка данных позволяет выполнить подсчёт строк, общую сумму, вычислять максимальное, минимальное и среднее значение, при этом не извлекая все данные из таблицы.



Когда используется оператор `GROUP BY`, то строки, которые возвращаются в результирующем наборе называются - группами, так как одна группа объединяет несколько строк, которые объединены по каким-то признакам.

Синтаксис:

Список столбцов в блоках `SELECT` и `GROUP BY` должен быть одинаковый, иначе будет получена ошибка. В исключение входят агрегатные функции, их не нужно указывать в блоке `GROUP BY`.

- `column_list` – список столбцов;
- `aggregate_function` – имя агрегатной функции `SUM()`, `COUNT()`, `AVG()`, `MAX()`, `MIN()` и т.д.;
- `column_name` – имя столбца, значения которого передаются в агрегатную функцию;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `condition_filter` – условия для фильтрации.

```
SELECT column_list,  
       aggregate_function(column_name)  
FROM schema.table_name  
WHERE condition_filter  
GROUP BY column_list;
```

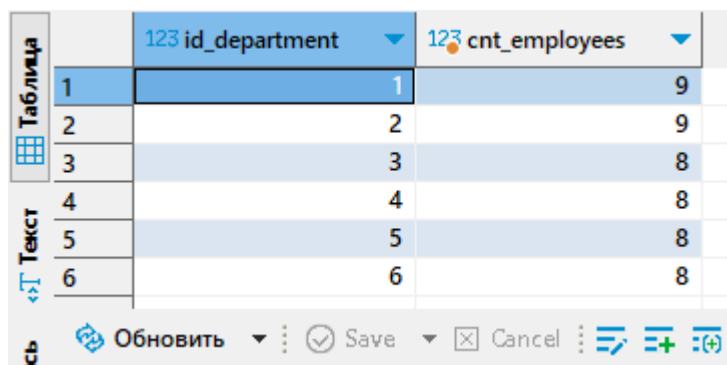
Практический пример 1: необходимо выполнить группировку данных таблицы `employees` по идентификатору отдела и вывести количество сотрудников напротив каждого отдела. Полученному агрегатному выражению присвоить псевдоним `cnt_employees`, а результирующий набор данных отсортировать по столбцу `id_department` в порядке возрастания.

Пояснение к SQL-запросу:

- в блоке `GROUP BY` указываем столбец, по которому будет выполнена группировка;
- в блоке `SELECT` указываем список столбцов, которые будут возвращены. Обратите внимание, что список столбцов в блоке `SELECT` должен быть идентичен списку столбцов, которые указаны в блоке `GROUP BY`;
- дополнительно в блоке `SELECT` указываем агрегатную функцию `COUNT()` для подсчета значений в полученных группах;
- в блоке `ORDER BY` указываем столбец, по которому будет выполнена сортировка результирующего набора.

```
SELECT id_department,  
       COUNT(*) AS cnt_employees  
FROM employees  
GROUP BY id_department  
ORDER BY id_department ASC;
```

Выполняем запрос и получаем результат, в котором выведен список отделов и напротив каждого указано количество сотрудников.



	123 id_department	123 cnt_employees
1	1	9
2	2	9
3	3	8
4	4	8
5	5	8
6	6	8

Практический пример 2: в предыдущем примере была выполнена группировка данных по одному столбцу, теперь необходимо выполнить группировку данных таблицы `employees` по идентификатору отдела и половому признаку, а также вывести напротив каждой полученной группы количество сотрудников. Полученному агрегатному выражению присвоить псевдоним `cnt_employees`, а результирующий набор данных отсортировать по столбцу `id_department` в порядке возрастания.

Принцип работы этого SQL-запроса идентичен предыдущему запросу из примера 1. Поэтому постарайтесь разобраться с этим примером самостоятельно.

```
SELECT id_department,
```

```

        gender,
        COUNT(*) AS cnt_employees
FROM employees
GROUP BY id_department, gender
ORDER BY id_department ASC;

```

Выполняем запрос и получаем результат, в котором выведен список отделов, в котором есть разделение по половому признаку и напротив каждой группы указано количество сотрудников.

	123 id_department	abc gender	123 cnt_employees
1	1	Женский	3
2	1	Мужской	6
3	2	Женский	4
4	2	Мужской	5
5	3	Женский	4
6	3	Мужской	4
7	4	Женский	5
8	4	Мужской	3
9	5	Женский	5
10	5	Мужской	3
11	6	Женский	2
12	6	Мужской	6

Сложные группировки

Ранее был изучен оператор [GROUP BY](#), он позволяет группировать данные по одному или нескольким столбцам, но как бы не был хорош `GROUP BY`, он не умеет выполнять операции агрегации на множественном уровне иерархии. Другими словами, мы можем получить агрегатные значения для полученных групп, но не можем получить общее агрегатное значение для этих групп.

Для решения таких задач используют операторы [ROLLUP](#) и [CUBE](#), они очень похожи, но имеют одно колоссальное различие. Оператор `ROLLUP` генерирует агрегатные значения для выбранных столбцов иерархическим образом, а оператор `CUBE`, наоборот, выполняет генерацию агрегатных значений, которые содержат все возможные комбинации указанных столбцов.

Также существует дополнительный оператор [GROUPING SETS](#). Это расширение оператора `GROUP BY`, которое используется для одновременного создания нескольких наборов группировки данных.

Оператор ROLLUP

Оператор `ROLLUP` позволяет вычислить подитоги и общий итог для множества указанных столбцов. То есть при помощи оператора `ROLLUP` мы можем подсчитать общее агрегатное значение для полученных групп.

На изображении ниже показано, как работает оператор `ROLLUP`.

Группа 1: Значение_1
Группа 1: Значение_2
Агрегатное_значение_группы_1: (Значение_1 + Значение_2)
Группа 2: Значение_1
Группа 2: Значение_2
Агрегатное_значение_группы_2: (Значение_1 + Значение_2)
Общее агрегатное значение групп (Агрегатное_значение_группы_1 + Агрегатное_значение_группы_2)

Синтаксис:

- `column_list` – список столбцов;
- `aggregate_function` – имя агрегатной функции `SUM()`, `COUNT()`, `AVG()`, `MAX()`, `MIN()` и т.д.;
- `column_name` – имя столбца, значения которого передаются в агрегатную функцию;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `condition_filter` – условия для фильтрации;
- `column_group_list` – столбцы для группировки;
- `column_order_list` – столбцы для сортировки.

```
SELECT column_list,  
       aggregate_function(column_name)  
FROM schema.table_name  
WHERE condition_filter  
GROUP BY ROLLUP (column_group_list)  
ORDER BY column_order_list [ASC | DESC];
```

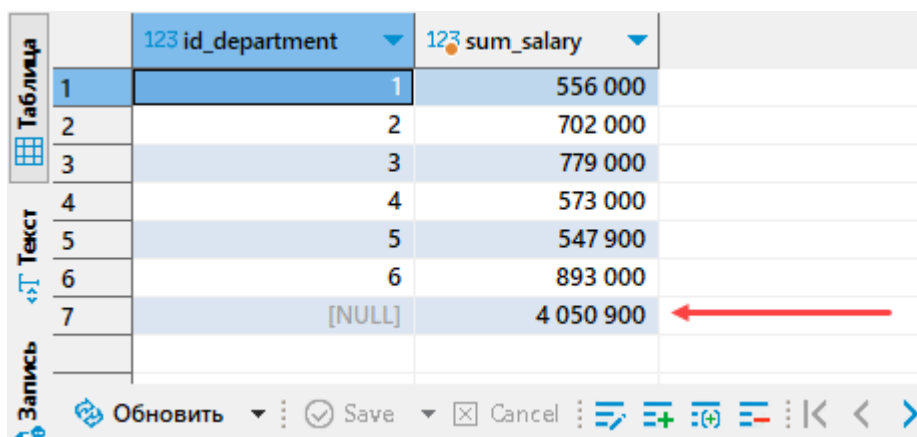
Практический пример: в таблице `employees` хранится информация о сотрудниках компании, необходимо подсчитать суммарную заработную плату сотрудников по каждому отделу, и общую сумму заработных плат всех сотрудников компании.

Пояснение к SQL-запросу:

- используя оператор `GROUP BY ROLLUP` группируем данные таблицы `employees` по столбцу `id_department`;
- в блоке `SELECT` выводим идентификатор отдела и сумму заработной платы при помощи агрегатной функции `SUM()`. Полученному значению присваиваем псевдоним `sum_salary`.

```
SELECT id_department,  
       sum(salary) AS sum_salary  
FROM employees  
GROUP BY ROLLUP(id_department)  
ORDER BY id_department ASC;
```

Выполняем запрос и получаем результат, в котором строка общих итогов `ROLLUP` будет иметь значение `NULL` вместо идентификатора отдела, так как оператор `ROLLUP` не понимает, какое значение нужно подставить.



	123 id_department	123 sum_salary
1	1	556 000
2	2	702 000
3	3	779 000
4	4	573 000
5	5	547 900
6	6	893 000
7	[NULL]	4 050 900

Для того, чтобы в строке с общим итогом не было значения `NULL`, используем функцию [COALESCE](#), она произведет замену значения `NULL` на текст - Все отделы, при этом не будет изменять названия других отделов.

Пояснение к SQL-запросу:

- конструкция `id_department::text` выполняет преобразование данных столбца `id_department` в строку, так как идентификаторы отдела хранятся в числовом виде, а функция `COALESCE` при нахождении значения `NULL` будет возвращать строку. Другими словами, мы устраняем конфликт типов данных.

```
SELECT COALESCE(id_department::text, 'Все отделы') AS id_department,  
       sum(salary) AS sum_salary  
FROM employees  
GROUP BY ROLLUP(id_department)  
ORDER BY id_department ASC;
```

Выполняем запрос и получаем результат, в котором по каждому отделу была подсчитана сумма заработных плат сотрудников, и подсчитана общая сумма заработных плат сотрудников по всей компании.

	ABC id_department	123 sum_salary	
1	1	556 000	
2	2	702 000	
3	3	779 000	
4	4	573 000	
5	5	547 900	
6	6	893 000	
7	Все отделы	4 050 900	←

Практический пример 2: в предыдущем примере был произведён подсчёт заработной платы сотрудников по каждому отделу, и была получена общая сумма заработных плат по всей компании. Теперь усложним задачу и сначала подсчитаем заработную плату сотрудников отдела по половому признаку, а затем получим общую сумму заработных плат сотрудников по всей компании.

Берём предыдущий запрос, и вносим в него небольшие изменения.

Пояснение к SQL-запросу:

- в оператор `GROUP BY ROLLUP` добавляем дополнительный столбец `gender` для группировки;
- в блок `SELECT` добавляем новый столбец `gender`, с использованием функции `COALESCE`.

```
SELECT COALESCE(id_department::text, 'Все отделы') AS id_department,
       COALESCE(gender, 'Оба пола') AS gender,
       sum(salary) AS sum_salary
FROM employees
GROUP BY ROLLUP(id_department, gender)
ORDER BY id_department ASC;
```

Выполняем запрос и получаем результат, в котором можно увидеть, что запрос вернул три результата на каждый отдел:

- заработная плата сотрудников женского пола;
- заработная плата сотрудников мужского пола;
- сумма заработной платы для обоих полов вместе.

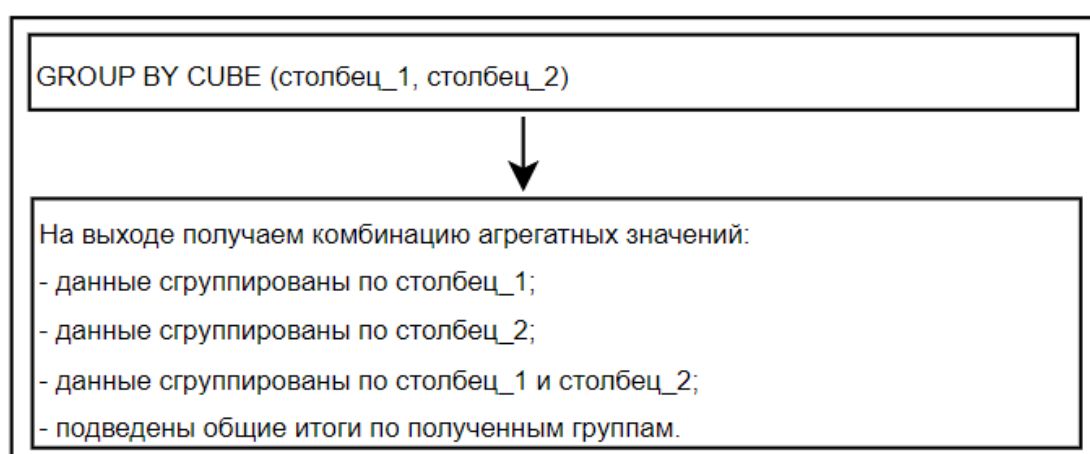
Также была подсчитана общая сумма заработной платы по всем сотрудникам компании.

	ABC id_department	ABC gender	123 sum_salary
1	1	Мужской	390 000
2	1	Женский	166 000
3	1	Оба пола	556 000
4	2	Женский	303 000
5	2	Оба пола	702 000
6	2	Мужской	399 000
7	3	Оба пола	779 000
8	3	Женский	334 000
9	3	Мужской	445 000
10	4	Женский	367 000
11	4	Оба пола	573 000
12	4	Мужской	206 000
13	5	Оба пола	547 900
14	5	Мужской	219 000
15	5	Женский	328 900
16	6	Оба пола	893 000
17	6	Мужской	683 000
18	6	Женский	210 000
19	Все отделы	Оба пола	4 050 900

Оператор CUBE

Оператор `CUBE` позволяет выполнить генерацию агрегатных значений, которые содержат все возможные комбинации указанных столбцов в предложение `GROUP BY CUBE`.

На изображении ниже показано, как работает оператор `CUBE`.



Синтаксис:

- `column_list` – список столбцов;

- `aggregate_function` – имя агрегатной функции `SUM()`, `COUNT()`, `AVG()`, `MAX()`, `MIN()` и т.д.;
- `column_name` – имя столбца, значения которого передаются в агрегатную функцию;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `condition_filter` – условия для фильтрации;
- `column_group_list` – столбцы для группировки;
- `column_order_list` – столбцы для сортировки.

```
SELECT column_list,
       aggregate_function(column_name)
FROM schema.table_name
WHERE condition_filter
GROUP BY CUBE (column_group_list)
ORDER BY column_order_list [ASC | DESC];
```

Практический пример: в таблице `employees` хранится информация о сотрудниках компании, необходимо подсчитать суммарную заработную плату сотрудников, которая сгруппирована по отделу и полу.

Если используется группировка по двум столбцам `id_department` и `gender`, то будет доступно 4 комбинации, по которым можно осуществить группировку заработной платы по отделу и полу. Давайте рассмотрим эти комбинации:

- заработная плата будет сгруппирована по идентификатору отдела (`id_department`) и полу (`gender`);
- заработная плата будет сгруппирована только по полу (`gender`);
- заработная плата будет сгруппирована только по идентификатору отдела (`id_department`);
- будут выведены общие итоги по всем заработным платам сотрудников.

Пояснение к SQL-запросу:

- используя оператор `GROUP BY CUBE`, выполняем группировку данных таблицы `employees` по столбцу `id_department` и `gender`;
- в блоке `SELECT` выводим идентификатор отдела, пол и сумму заработной платы при помощи агрегатной функции `SUM()`. Полученному значению присваиваем псевдоним `sum_salary`;

- столбцы `id_department` и `gender` выводятся при помощи функции [COALESCE](#). Если выводить столбцы без функции `COALESCE`, то в строке общих итогов `CUBE` будет возвращено значение `NULL`, так как оператор не понимает, какое значение ему нужно подставить. Поэтому, чтобы в строке с общим итогом не было значения `NULL`, используется функция `COALESCE`, она произведет замену значения `NULL` на нужный текст, при этом не будет изменять существующие идентификаторы отделов и пол;
- конструкция `id_department::text` выполняет преобразование данных столбца `id_department` в строку, так как идентификаторы отдела хранятся в числовом виде, а функция `COALESCE` при нахождении значения `NULL` будет возвращать строку. Другими словами, мы устраняем конфликт типов данных.

```
SELECT COALESCE(id_department::text, 'Все отделы') AS id_department,
       COALESCE(gender, 'Оба пола') as gender,
       SUM(salary) as sum_salary
FROM employees
GROUP BY CUBE(id_department, gender)
ORDER BY id_department ASC;
```

Выполняем запрос и получаем результат. Давайте найдем все комбинации:

- 1 группа: заработная плата сгруппирована по идентификатору отдела и полу;
- 2 группа: заработная плата сгруппирована по полу;
- 3 группа: общий итог по всем заработным платам всех идентификаторов отделов и полов сотрудников;
- 4 группа: заработная плата сгруппирована по отделу.

	ABC id_department	ABC gender	123 sum_salary	
1	1	Оба пола	556 000	← 4 группа
2	1	Мужской	390 000	← 1 группа
3	1	Женский	166 000	
4	2	Мужской	399 000	← 1 группа
5	2	Женский	303 000	
6	2	Оба пола	702 000	← 4 группа
7	3	Мужской	445 000	← 1 группа
8	3	Женский	334 000	
9	3	Оба пола	779 000	← 4 группа
10	4	Мужской	206 000	← 1 группа
11	4	Оба пола	573 000	← 4 группа
12	4	Женский	367 000	← 1 группа
13	5	Оба пола	547 900	← 4 группа
14	5	Женский	328 900	← 1 группа
15	5	Мужской	219 000	
16	6	Женский	210 000	← 1 группа
17	6	Мужской	683 000	
18	6	Оба пола	893 000	← 4 группа
19	Все отделы	Мужской	2 342 000	← 2 группа
20	Все отделы	Женский	1 708 900	
21	Все отделы	Оба пола	4 050 900	← 3 группа

Оператор GROUPING SETS

Оператор `GROUPING SETS` является расширением функциональных возможностей оператора `GROUP BY`, которое используется для одновременного создания нескольких наборов сгруппированных данных.

Функциональные возможности оператора `GROUPING SETS` эквивалентны оператору `UNION ALL`.

Синтаксис:

- `column_list` – список столбцов;
- `aggregate_function` – имя агрегатной функции `SUM()`, `COUNT()`, `AVG()`, `MAX()`, `MIN()` и т.д.;
- `column_name` – имя столбца, значения которого передаются в агрегатную функцию;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `condition_filter` – условия для фильтрации;

- `column_group_list` – столбцы для группировки;
- `combination_column` – комбинация столбцов для группировки.

```
SELECT column_list,
       aggregate_function(column_name)
FROM schema.table_name
WHERE condition_filter
GROUP BY GROUPING SETS (column_group_list, (combination_column));
```

Практический пример: в таблице `employees` хранится информация о сотрудниках компании, необходимо создать три комбинации для группировки данных, а затем по каждой группе подсчитать сумму заработной платы сотрудников компании. Необходимые комбинации для группировок:

- группировка только по идентификатору отдела (`id_department`);
- группировка только по полу сотрудника (`gender`);
- группировка по идентификатору отдела и полу сотрудника (`id_department` и `gender`).

Пояснение к SQL-запросу:

- используя оператор `GROUP BY GROUPING SETS` выполняем группировку данных таблицы `employees` по группам: 1 группа (`id_department`), 2 группа (`gender`) и 3 группа (`id_department, gender`);
- в блоке `SELECT` выводим идентификатор отдела, пол и сумму заработной платы при помощи агрегатной функции `SUM()`. Полученному значению присваиваем псевдоним `sum_salary`;
- столбцы `id_department` и `gender` выводятся при помощи функции [COALESCE](#). Если выводить столбцы без функции `COALESCE`, то при группировке только по `id_department` или `gender` будет выведено значение `NULL`, так как оператор не понимает, какое значение ему нужно подставить. Поэтому, чтобы при группировке не было значений `NULL`, используем функцию `COALESCE`, она произведет замену значения `NULL` на нужный текст, при этом не будет изменять существующие идентификаторы отделов и пол;
- конструкция `id_department::text` выполняет преобразование данных столбца `id_department` в строку, так как идентификаторы отдела хранятся в числовом

виде, а функция COALESCE при нахождении значения NULL будет возвращать строку. Другими словами, мы устраняем конфликт типов данных.

```
SELECT COALESCE(id_department::text, 'Все отделы') AS id_department,
       COALESCE(gender, 'Оба пола') as gender,
       SUM(salary) as sum_salary
FROM employees
GROUP BY GROUPING SETS(id_department,
                        gender,
                        (id_department, gender));
```

Выполняем запрос и получаем результат. Давайте найдем нужные нам комбинации:

- 1 группа: группировка данных по столбцу id_department;
- 2 группа: группировка данных по столбцу gender;
- 3 группа: группировка данных по столбцу id_department и gender.

	ABC id_department	ABC gender	123 sum_salary	
3	3	Женский	334 000	
4	4	Женский	367 000	
5	1	Женский	166 000	
6	5	Женский	328 900	
7	6	Мужской	683 000	
8	2	Женский	303 000	
9	3	Мужской	445 000	
10	2	Мужской	399 000	
11	1	Мужской	390 000	
12	4	Мужской	206 000	
13	4	Оба пола	573 000	
14	6	Оба пола	893 000	
15	2	Оба пола	702 000	
16	3	Оба пола	779 000	
17	1	Оба пола	556 000	
18	5	Оба пола	547 900	
19	Все отделы	Женский	1 708 900	
20	Все отделы	Мужской	2 342 000	

3 группа: группировка по id_department и gender

1 группа: группировка по id_department

2 группа: группировка по gender

А вот реализация того же самого запроса, но уже при помощи оператора UNION ALL. После выполнения запроса, будут получены точно такие же данные, как в запросе с использованием GROUPING SETS.

```
SELECT id_department::TEXT AS id_department,
       'Оба пола' as gender,
       SUM(salary) as sum_salary
FROM employees
GROUP BY id_department
```

```
UNION ALL
```

```
SELECT 'Все отделы' as id_department,
       gender,
       SUM(salary) as sum_salary
FROM employees
GROUP BY gender
```

```
UNION ALL
```

```
SELECT id_department::TEXT AS id_department,
       gender,
       SUM(salary) as sum_salary
FROM employees
GROUP BY id_department, gender;
```

Функция GROUPING

Функция `GROUPING()` используется для осуществления поиска групп, которые формируют промежуточные итоги в строке, и работает только с оператором [ROLLUP](#) или [CUBE](#).

По своей сути функция `GROUPING()` - это статистическая функция формирующая дополнительный столбец, в котором содержится значение 0 или 1. Значение 1 проставляется в том случае, когда строка добавлена при помощи оператора `CUBE` или `ROLLUP`, в противном случае проставляется значение 0.

Синтаксис:

Функция принимает только один аргумент и это имя столбца, которое используется в предложении `GROUP BY`.

- `column_list` – список столбцов;

- `column_name_group` – столбец для группировки;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `condition_filter` – условия для фильтрации;
- `column_order_list` – столбцы для сортировки.

```
SELECT column_list,
        GROUPING(column_name_group)
FROM schema.table_name
WHERE condition_filter
GROUP BY [CUBE | ROLLUP] (column_name_group)
ORDER BY column_order_list [ASC | DESC];
```

Практический пример: в качестве примера возьмем готовый SQL-запрос, который был написан при рассмотрении оператора `ROLLUP`, и внесем в него небольшие изменения.

Пояснение к SQL-запросу:

- в блок `SELECT` добавим две функции `GROUPING()` с указанием столбцов `id_department` и `gender` по которым осуществляется группировка, а затем присвоим им псевдонимы `group_department` и `group_gender`.

```
SELECT COALESCE(id_department::text, 'Все отделы') AS id_department,
        COALESCE(gender, 'Оба пола') AS gender,
        SUM(salary) AS sum_salary,
        GROUPING(id_department) as group_department,
        GROUPING(gender) as group_gender
FROM employees
GROUP BY ROLLUP(id_department, gender)
ORDER BY id_department ASC;
```

Выполняем запрос и получаем результат, в котором можно увидеть что, функция `GROUPING()` поставила значение 1 для строк, которые были динамически добавлены оператором `ROLLUP`.

		ABC id_department	ABC gender	123 sum_salary	123 group_department	123 group_gender
1	Таблица	1	Мужской	390 000	0	0
2	Текст	1	Женский	166 000	0	0
3		1	Оба пола	556 000	0	1
4	Запись	2	Женский	303 000	0	0
5		2	Оба пола	702 000	0	1
6		2	Мужской	399 000	0	0
7		3	Оба пола	779 000	0	1
8		3	Женский	334 000	0	0
9		3	Мужской	445 000	0	0
10		4	Женский	367 000	0	0
11		4	Оба пола	573 000	0	1
12		4	Мужской	206 000	0	0
13		5	Оба пола	547 900	0	1
14		5	Мужской	219 000	0	0
15		5	Женский	328 900	0	0
16		6	Оба пола	893 000	0	1
17		6	Мужской	683 000	0	0
18		6	Женский	210 000	0	0
19		Все отделы	Оба пола	4 050 900	1	1

Обновить Save Cancel

Экспорт данных ...

200

Фильтрация групп (HAVING)

Можно не только осуществлять группировку результатов запроса при помощи оператора [GROUP BY](#), но и осуществлять фильтрацию полученных групп, то есть, можно указать какие группы должны быть включены в результат запроса, а какие должны быть исключены.

Для осуществления фильтрации групп необходимо написать соответствующие условия, а как мы знаем у нас есть оператор [WHERE](#) при помощи которого можно задавать условия для фильтрации. Но оператор `WHERE` в этом случае нам не подходит, так как он выполняет фильтрацию - строк, а не групп. Для фильтрации групп необходимо воспользоваться оператором `HAVING`, и указывается он после блока `GROUP BY`.



Оператор `HAVING` нельзя использовать без `GROUP BY`.

Синтаксис:

- `column_list` – список столбцов;
- `aggregate_function` – имя агрегатной функции `SUM()`, `COUNT()`, `AVG()`, `MAX()`, `MIN()` и т.д.;
- `column_name` – имя столбца, значения которого передаются в агрегатную функцию;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `condition_filter_string` – условия для фильтрации строк;
- `condition_filter_group` – условия для фильтрации групп.

```
SELECT column_list,  
       aggregate_function(column_name)  
FROM schema.table_name  
WHERE condition_filter  
GROUP BY column_list  
HAVING condition_filter_group;
```

Практический пример: в качестве основы возьмем пример, который ранее был рассмотрен в главе по группировке данных.

Давайте вспомним текст задания: необходимо выполнить группировку данных таблицы `employees` по идентификатору отдела и половому признаку, а также вывести напротив каждой полученной группы количество сотрудников. Полученному агрегатному выражению присвоить псевдоним `cnt_employees`, а результирующий набор данных отсортировать по столбцу `id_department` в порядке возрастания.

```
SELECT id_department,
       gender,
       count(*) AS cnt_employees
FROM employees
GROUP BY id_department, gender
ORDER BY id_department ASC;
```

Выполняем запрос и получаем результат, в котором выведен список, состоящий из 12 групп и напротив каждой группы указано количество строк, которое находится в этой группе.

	123 id_department ▼	ABC gender ▼	123 cnt_employees ▼
Таблица	1	Женский	3
	2	Мужской	6
	3	Женский	4
Текст	4	Мужской	5
	5	Женский	4
	6	Мужской	4
	7	Женский	5
	8	Мужской	3
	9	Женский	5
Запись	10	Мужской	3
	11	Женский	2
	12	Мужской	6

Дальше необходимо осуществить фильтрацию по сгруппированным данным и вывести только те группы, у которых значение в столбце `cnt_employees` больше или равно 4.

Для этого добавляем оператор `HAVING` с условием `count (gender) >= 4`.

```
SELECT id_department,
       gender,
       count(*) AS cnt_employees
FROM employees
GROUP BY id_department, gender
HAVING count (gender) >= 4;
```

```
HAVING count(gender) >= 4
ORDER BY id_department ASC;
```

Выполняем запрос и получаем результат, в котором выведены только те группы, у которых значение в столбце cnt_employees больше или равно 4.

	123 id_department ▼	abc gender ▼	123 cnt_employees ▼
1	1	Мужской	6
2	2	Женский	4
3	2	Мужской	5
4	3	Женский	4
5	3	Мужской	4
6	4	Женский	5
7	5	Женский	5
8	6	Мужской	6

Таблица

Текст

Запись

Обновить

Save

Cancel

Комбинированные запросы

Оператор UNION и UNION ALL

При помощи операторов `UNION` и `UNION ALL` можно объединять несколько [SELECT](#) запросов в одну финальную выборку.



В некоторых СУБД есть ограничения на количество SQL-запросов, которые могут быть объединены при помощи оператора `UNION` и `UNION ALL`. Поэтому лучше обратиться к документации, и убедиться, что СУБД не накладывает никаких ограничений.

Правила при использовании операторов `UNION` и `UNION ALL`:

- оператор `UNION` или `UNION ALL` должен объединять минимум два запроса `SELECT`;
- каждый `SELECT` запрос в операторе `UNION` или `UNION ALL` должен содержать один и тот же набор столбцов, выражений или агрегатных функций;
- типы данных в столбцах не должны отличаться.

Оператор UNION

Использовать оператор `UNION` достаточно просто. Для этого необходимо указать ключевое слово `UNION` между SQL запросами. Применять оператор `UNION` можно многократно.

Оператор `UNION` автоматически удаляет из выборки повторяющиеся строки.

Синтаксис:

- `column_name` – имя столбца;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `condition_filter` – условия для фильтрации.

```
SELECT column_name_1, column_name_n  
FROM schema.table_name  
WHERE condition_filter
```

```
UNION
```

```
SELECT column_name_1, column_name_n
FROM schema.table_name
WHERE condition_filter;
```

Практический пример: есть два SQL-запроса, их необходимо объединить при помощи оператора UNION.

```
-- 1 запрос
SELECT first_name
FROM employees
WHERE first_name = 'Ольга';

-- 2 запрос
SELECT first_name
FROM employees
WHERE first_name IN ('Ольга', 'Кирилл', 'Игорь', 'Петр');
```

Для объединения запросов добавляем оператор UNION между ними.

```
SELECT first_name
FROM employees
WHERE first_name = 'Ольга'

UNION

SELECT first_name
FROM employees
WHERE first_name IN ('Ольга', 'Кирилл', 'Игорь', 'Петр');
```

Выполняем запрос и получаем результат в виде объединения двух запросов без повторяющихся значений. Если вы не разобрались с тем, как работает оператор UNION, то выполните 1 и 2 запрос отдельно друг от друга, и посмотрите сколько результатов вернёт каждый запрос, а потом выполните запрос с оператором UNION и посмотрите сколько он вернет результатов (вернет меньше результатов и без повторяющихся значений).

Таблица		ABC first_name	
	1	Игорь	
	2	Кирилл	
	3	Петр	
	4	Ольга	
Текст			

Обновить Save

Оператор UNION ALL

С работой оператором [UNION](#) мы разобрались, и выяснили, что он позволяет создавать комбинированные запросы и исключает из финальной выборки повторяющиеся значения. А как же быть, если нам всё же нужны повторяющиеся значения?

В этом случае необходимо воспользоваться оператором `UNION ALL`, и повторяющиеся значения будут сохранены.

Синтаксис:

- `column_name` – имя столбца;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `condition_filter` – условия для фильтрации.

```
SELECT column_name_1, column_name_n
FROM schema.table_name
WHERE condition_filter
```

`UNION ALL`

```
SELECT column_name_1, column_name_n
FROM schema.table_name
WHERE condition_filter;
```

Практический пример: есть два SQL-запроса, их необходимо объединить при помощи оператора `UNION ALL`.

```
-- 1 запрос
SELECT first_name
FROM employees
WHERE first_name = 'Ольга';
```

```
-- 2 запрос
SELECT first_name
FROM employees
WHERE first_name IN ('Ольга', 'Кирилл', 'Игорь', 'Петр');
```

Для объединения запросов добавляем оператор `UNION ALL` между ними.

```
SELECT first_name
FROM employees
WHERE first_name = 'Ольга'
```

```
UNION ALL
```

```
SELECT first_name
FROM employees
WHERE first_name IN ('Ольга', 'Кирилл', 'Игорь', 'Петр');
```

Выполняем запрос и получаем результат в виде объединения двух запросов без удаления повторяющихся значений.

Таблица	ABC first_name	
1	Ольга	
2	Ольга	
3	Петр	
4	Петр	
5	Ольга	
6	Игорь	
7	Кирилл	
8	Игорь	
9	Ольга	
10	Игорь	

Обновить Save Cancel

Сортировка результатов выборки

Теперь мы умеем создавать комбинированные запросы при помощи оператора [UNION](#) и [UNION ALL](#). Настало время научиться выполнять сортировку результатов в финальной выборке.

Как мы знаем, результаты [SELECT](#) запроса сортируются при помощи оператора [ORDER BY](#). Когда используются комбинированные запросы, необходимо указывать оператор `ORDER BY` в самом конце, после последнего `SELECT` запроса.



Оператор `ORDER BY` указывается в конце, так как нет смысла сортировать первый запрос одним способом, а второй другим. Поэтому применять несколько операторов `ORDER BY` нельзя.

Синтаксис:

- `column_name` – имя столбца;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `condition_filter` – условия для фильтрации;
- `column_order` – имя столбца для сортировки.

```
SELECT column_name_1, column_name_n
FROM schema.table_name
WHERE condition_filter
```

`UNION ALL`

```
SELECT column_name_1, column_name_n
FROM schema.table_name
WHERE condition_filter
```

```
ORDER BY column_order [ASC|DESC];
```

Практический пример: есть два SQL-запроса, их необходимо объединить при помощи оператора `UNION`. Результирующий набор отсортировать по столбцу `first_name` в порядке возрастания.

```
-- 1 запрос
SELECT first_name, last_name
FROM employees
WHERE first_name = 'Ольга';
```



```
-- 2 запрос
SELECT first_name, last_name
FROM employees
WHERE first_name IN ('Ольга', 'Кирилл', 'Игорь', 'Петр');
```

Для объединения запросов добавляем оператор `UNION` между ними, а также добавляем оператор `ORDER BY` и указываем имя столбца `first_name` с направлением сортировки `ASC`.

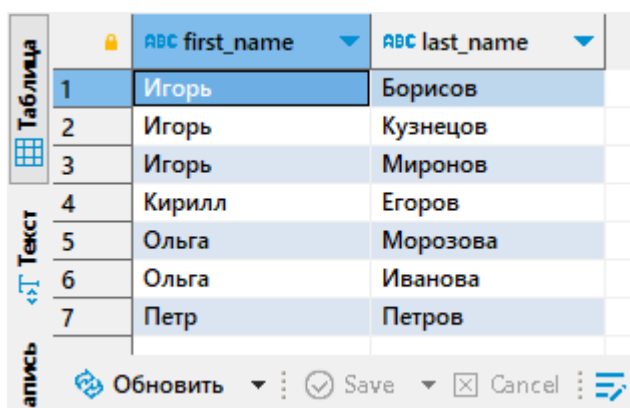
```
SELECT first_name, last_name
FROM employees
WHERE first_name = 'Ольга'

UNION

SELECT first_name, last_name
FROM employees
WHERE first_name IN ('Ольга', 'Кирилл', 'Игорь', 'Петр')

ORDER BY first_name ASC;
```

Выполняем запрос и получаем результат в виде объединения двух запросов с сортировкой по столбцу `first_name`.



	ABC first_name	ABC last_name
1	Игорь	Борисов
2	Игорь	Кузнецов
3	Игорь	Миронов
4	Кирилл	Егоров
5	Ольга	Морозова
6	Ольга	Иванова
7	Петр	Петров

Оператор EXCEPT

Оператор `EXCEPT` выполняет вычитание из первого набора данных второй набор данных, то есть в результате выполнения запроса на экран будут выведены только те строки из первого набора, которые отсутствуют во втором наборе (второй набор — это запрос, который идёт после оператора `EXCEPT`).

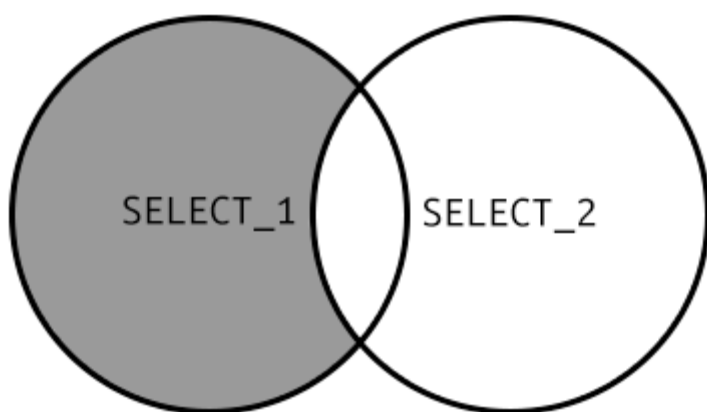


Каждый [SELECT](#) запрос, который используется в запросе с оператором `EXCEPT` должен иметь одинаковое количество столбцов и одинаковый тип данных.

К сожалению, оператор `EXCEPT` в других СУБД носит другое наименование, например:

- оператор `EXCEPT` (используется в PostgreSQL, MySQL, Microsoft SQL Server, SQLite и т.д.);
- оператор `MINUS` (используется в Oracle);

Давайте отобразим работу оператора `EXCEPT` при помощи изображения. Оператор `EXCEPT` вернет данные из заштрихованной области. Эти данные существуют только в запросе `SELECT_1`, а не в `SELECT_2`.



Синтаксис:

- `column_name` – имя столбца;
- `schema` – наименование схемы, в которой находится объект;
- `table_name_1` и `table_name_2` – имя первой и второй таблицы;
- `condition_filter` – условия для фильтрации;
- `column_order` – имя столбца для сортировки.

```
SELECT column_name_1, column_name_n  
FROM schema.table_name_1  
WHERE condition_filter
```

```
EXCEPT
```

```
SELECT column_name_1, column_name_n
FROM schema.table_name_2
WHERE condition_filter
```

```
ORDER BY column_order [ASC|DESC];
```

Практический пример: есть два SQL-запроса, необходимо вычесть из первого набора данных второй набор, при помощи оператора EXCEPT.

-- 1 запрос

```
SELECT first_name
FROM employees
WHERE first_name IN ('Ольга', 'Игорь', 'Петр');
```

-- 2 запрос

```
SELECT first_name
FROM employees
WHERE first_name IN ('Максим', 'Иван', 'Ольга', 'Анастасия');
```

Добавляем оператор EXCEPT для выполнения вычитания.

```
SELECT first_name
FROM employees
WHERE first_name IN ('Ольга', 'Игорь', 'Петр')
```

```
EXCEPT
```

```
SELECT first_name
FROM employees
WHERE first_name IN ('Максим', 'Иван', 'Ольга', 'Анастасия');
```

Выполняем запрос и получаем результат, в котором присутствуют имена Игорь и Петр. Другими словами, мы получили имена, которые отсутствуют во втором запросе.

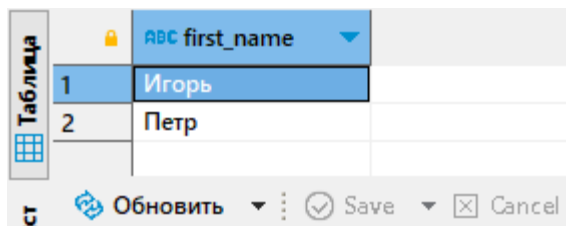


Таблица	ABC first_name
1	Игорь
2	Петр

Стр. Обновить Save Cancel

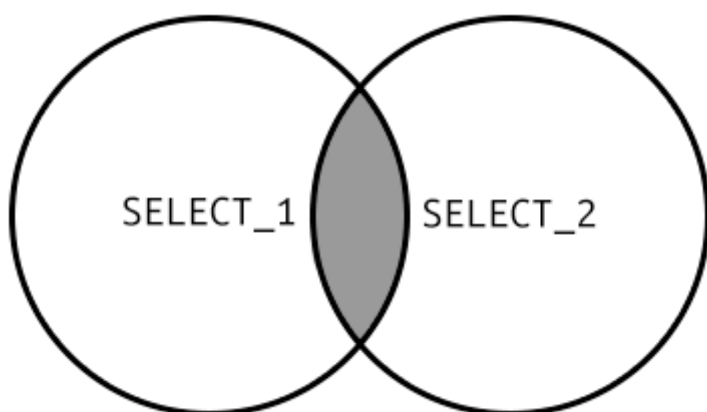
Оператор INTERSECT

Оператор `INTERSECT` чем-то похож на ранее изученный оператор [EXCEPT](#), но суть работы оператора `INTERSECT` заключается в том, чтобы выводить только те данные, которые есть и в первом, и во втором наборе (запросе). Другими словами, оператор `INTERSECT` возвращает пересечение множеств.



Каждый [SELECT](#) запрос, который используется в запросе с оператором `INTERSECT` должен иметь одинаковое количество полей и одинаковый тип данных.

Отобразим работу оператора `INTERSECT` при помощи изображения. Оператор `INTERSECT` вернет данные из заштрихованной области. Эти данные существуют и в запросе `SELECT_1`, и в `SELECT_2`.



Синтаксис:

- `column_name` – имя столбца;
- `schema` – наименование схемы, в которой находится объект;
- `table_name_1` и `table_name_2` – имя первой и второй таблицы;
- `condition_filter` – условия для фильтрации;
- `column_order` – имя столбца для сортировки.

```
SELECT column_name_1, column_name_n
FROM schema.table_name_1
WHERE condition_filter
```

INTERSECT

```
SELECT column_name_1, column_name_n  
FROM schema.table_name_2  
WHERE condition_filter
```

```
ORDER BY column_order [ASC|DESC];
```

Практический пример: есть два SQL-запроса, необходимо вывести на экран данные, которые существуют и в первом, и во втором наборе данных при помощи оператора INTERSECT.

-- 1 запрос

```
SELECT first_name  
FROM employees  
WHERE first_name IN ('Ольга', 'Игорь', 'Петр', 'Наталья', 'Елена');
```

-- 2 запрос

```
SELECT first_name  
FROM employees  
WHERE first_name IN ('Максим', 'Иван', 'Ольга', 'Анастасия', 'Елена');
```

Добавляем оператор INTERSECT для нахождения пересечения множеств.

```
SELECT first_name  
FROM employees  
WHERE first_name IN ('Ольга', 'Игорь', 'Петр', 'Наталья', 'Елена')
```

INTERSECT

```
SELECT first_name  
FROM employees  
WHERE first_name IN ('Максим', 'Иван', 'Ольга', 'Анастасия', 'Елена');
```

Выполняем запрос и получаем результат, в котором присутствуют имена Ольга и Елена. Другими словами, мы получили имена, которые присутствуют и в первом, и во втором запросе.

Таблица		ABC first_name	
	1	Ольга	
	2	Елена	
Текст			
Обновить			Save Cancel

Предикаты ANY (SOME) / ALL

Предикаты `ANY` (`SOME`) и `ALL` позволяют сравнить исходные значения с каждым значением [подзапроса](#). Данные предикаты используются с предложением [WHERE](#) или [HAVING](#).

Предикат — это логическое выражение, которое может быть, как `TRUE` (правда), `FALSE` (ложь) или `UNKNOWN` (неизвестно).



Предикат `ANY` является синонимом предиката `SOME`, другими словами, `ANY` и `SOME` выполняют одну и ту же функцию.

Предикаты `ANY` (`SOME`) и `ALL` используются в блоке `WHERE`, как результат сравнения значения поля в указанном столбце со списком значений. То есть сначала название поля, а затем уже подзапрос.

Предикат ANY (SOME)

Предикат `ANY` (`SOME`) возвращает значение `TRUE`, если какое-либо из значений подзапроса соответствует условию.

Синтаксис:

- `column_list` – список столбцов;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `column_name` – имя столбца;
- `comparison_operator` – оператор сравнения;
- `subquery` – подзапрос.

```
SELECT column_list
FROM schema.table_name
WHERE column_name [comparison_operator] ANY (subquery);
```

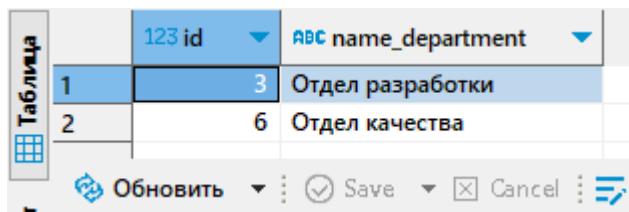
Практический пример: из таблицы `departments` необходимо вывести отделы, в которых сотрудники получают заработную плату больше или равную 110 000 рублей.

Пояснение к SQL-запросу:

- в блоке `FROM` указываем имя таблицы `departments`, из нее будут получены основные данные;
- в блоке `WHERE` задаём условие, в котором будем сравнивать столбец `id` из таблицы `departments` с предикатом `ANY`, а внутри предиката напишем подзапрос, который вернет идентификаторы отдела из таблицы `employees`, в которых сотрудники получают заработную плату больше или равную 110 000 рублей.

```
SELECT d.id,
       d.name_department
FROM departments d
WHERE d.id >= ANY (SELECT e.id_department
                  FROM employees e
                  WHERE e.salary >= 110000);
```

Выполняем запрос и получаем список отделов, в которых заработная плата у сотрудников больше или равна 110 000 рублей.



	123 id	ABC name_department
1	3	Отдел разработки
2	6	Отдел качества

Предикат ALL

Предикат `ALL` возвращает значение `TRUE`, если все значения возвращаемого подзапроса соответствуют условию.

Синтаксис:

- `column_list` – список столбцов;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `column_name` – имя столбца;
- `comparison_operator` – оператор сравнения;
- `subquery` – подзапрос.

```
SELECT column_list
FROM schema.table_name
WHERE column_name [comparison_operator] ALL (subquery);
```


Практический пример: из таблицы `departments` необходимо вывести идентификаторы и названия отделов, у которых идентификатор, больше идентификаторов отделов, которые есть в таблице `employees`.

Пояснение к SQL-запросу:

- в блоке `FROM` указываем имя таблицы `departments`, из нее будут получены основные данные;
- в блоке `WHERE` задаём условие, в котором будем сравнивать столбец `id` из таблицы `departments` с предикатом `ALL`, а внутри предиката напишем подзапрос, который вернет уникальные идентификаторы отделов из таблицы `employees`.

```
SELECT d.id,  
       d.name_department  
FROM departments d  
WHERE d.id > ALL (SELECT DISTINCT e.id_department  
                  FROM employees e);
```

Выполняем запрос и получаем список отделов из таблицы `departments`, у которых идентификатор, больше идентификаторов отделов, которые есть в таблице `employees`.

Таблица		123 id	ABC name_department
	1	7	Отдел взыскания
	2	8	Отдел безопасности
	3	9	Отдел поддержки
Текст	Обновить Save Cancel		

Объединение таблиц (JOIN)

Это один из основных механизмов, который позволяет выполнять объединение таблиц на лету и получать необходимые данные. **Объединение таблиц** — это самые мощные операции, которые можно выполнить при помощи оператора `SELECT`, поэтому каждый разработчик, аналитик или администратор базы данных должен понимать, как работает этот механизм.

Объединение таблиц нужно в том случае, когда данные хранятся в разных таблицах, а вам необходимо связать эти данные между собой и вывести в одном запросе.

Существует несколько видов объединений таблиц:

- внутреннее объединение (`INNER JOIN`);
- левое внешнее объединение (`LEFT OUTER JOIN`);
- правое внешнее объединение (`RIGHT OUTER JOIN`);
- полное внешнее объединение (`FULL OUTER JOIN`);
- перекрёстное объединение (`CROSS JOIN`);
- само объединение (`SELF JOIN`);
- естественное объединение (`NATURAL JOIN`).

Классический способ объединения

Объединение таблиц происходит при помощи оператора `JOIN`, но можно использовать и классический вариант объединения в блоке [FROM](#), который тоже работает и выполняет свои функции.



Данный способ объединения таблиц лучше не использовать, так как снижается читабельность кода и увеличивается размер запроса при объединении больших таблиц.

Синтаксис:

- `column_list` — список столбцов;
- `schema` — наименование схемы, в которой находится объект;
- `table_name_1` и `table_name_2` — имена таблиц;

- `alias_table_1` и `alias_table_2` – псевдонимы таблиц;
- `column_name` – имя столбца.

В блоке `WHERE` задаётся условие для объединения таблиц, и условия для фильтрации. Если вы не укажете условие для объединения таблиц, то вы получите [перекрёстное соединение](#) (Декартово произведение).

```
SELECT column_list
FROM schema.table_name_1 AS alias_table_1,
     schema.table_name_2 AS alias_table_2
WHERE alias_table_1.column_name = alias_table_2.column_name;
```

Практический пример: необходимо объединить таблицы `employees` и `departments` по идентификатору отдела и вывести информацию о всех сотрудниках, а также наименование отдела, в котором трудоустроен сотрудник.

Пояснение к SQL-запросу:

- в блоке `FROM` указываем список таблиц для объединения и присваиваем им псевдонимы;
- в блоке `WHERE` указываем условие для объединения таблиц по идентификатору отдела и дополнительно указываем, что нужны сотрудники, которые работают в отделе и с идентификатором 5.

```
SELECT e.id,
       e.first_name,
       e.last_name,
       e.gender,
       e.birthday,
       e.email,
       d.name_department,
       e.salary
FROM employees e, departments d
WHERE e.id_department = d.id
      AND e.id_department = 5;
```

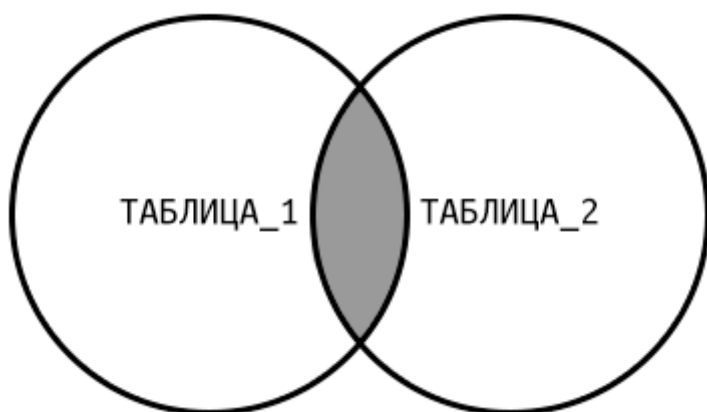
Выполняем запрос и получаем список сотрудников, которые работают в отделе с идентификатором 5.

	123 id	abc first_name	abc last_name	abc gender	birthday	abc email	abc name_department	123 salary
1	5	Екатерина	Васильева	Женский	1995-03-30	vasilieva@yandex.ru	Отдел логистики	38 900
2	11	Ольга	Морозова	Женский	1993-02-14	morozova@yandex.r	Отдел логистики	70 000
3	17	Кирилл	Егоров	Мужской	1986-07-22	[NULL]	Отдел логистики	62 000
4	23	Иван	Кудряшов	Мужской	1987-06-20	kudryashov@yandex	Отдел логистики	80 000
5	29	Алёна	Ильина	Женский	1995-09-30	ilina@yandex.ru	Отдел логистики	75 000
6	35	Екатерина	Кузнецова	Женский	1991-07-22	kuznetsova@yandex	Отдел логистики	72 000
7	41	Екатерина	Попова	Женский	1993-01-08	popova@yandex.ru	Отдел логистики	73 000
8	47	Павел	Григорьев	Мужской	1987-12-04	grigoryev@mail.ru	Отдел логистики	77 000

Внутреннее объединение (JOIN/INNER JOIN)

Внутреннее объединение таблиц осуществляется при помощи оператора `INNER JOIN` или просто `JOIN`. Когда происходит объединение таблиц методом внутреннего соединения, то результатом запроса будут данные, которые полностью удовлетворяют условию `ON`.

Для наглядности, работа оператора `JOIN` представлена ниже на изображение. Заштрихованная область, это данные, которые соответствуют условию `ON`.



Синтаксис:

- `column_list` – список столбцов;
- `schema` – наименование схемы, в которой находится объект;
- `table_name_1` и `table_name_2` – имена таблиц;
- `alias_table_1` и `alias_table_2` – псевдонимы таблиц;
- `column_name` – имя столбца;
- `condition_filter` – условия для фильтрации.

После блока [FROM](#) указывается сначала ключевое слово `INNER JOIN` или `JOIN`, а затем имя таблицы, с которой необходимо выполнить объединение. Далее идёт ключевое слово `ON`, и только потом необходимые условия для объединения.

```

SELECT column_list
FROM schema.table_name_1 AS alias_table_1
[INNER JOIN|JOIN] schema.table_name_2 AS alias_table_2
ON alias_table_2.column_name = alias_table_1.column_name
WHERE condition_filter;

```

Практический пример: необходимо объединить таблицы `employees` и `departments` по идентификатору отдела и вывести информацию о всех сотрудниках, а также наименование отдела, в котором трудоустроен сотрудник.

Пояснение к SQL-запросу:

- в блоке `FROM` указываем таблицу `employees`, из нее будут получены основные данные о сотрудниках;
- в операторе `JOIN` указываем таблицу `departments`, с которой будет происходить объединение. После ключевого слова `ON` указываем условия объединения;
- в блоке `WHERE` указываем дополнительное условие, что нужны сотрудники, которые работают в отделе с идентификатором 5.

```

SELECT e.id,
       e.first_name,
       e.last_name,
       e.gender,
       e.birthday,
       e.email,
       d.name_department,
       e.salary
FROM employees e
JOIN departments d
ON d.id = e.id_department
WHERE e.id_department = 5;

```

Выполняем запрос и получаем список сотрудников, которые работают в отделе с идентификатором 5.

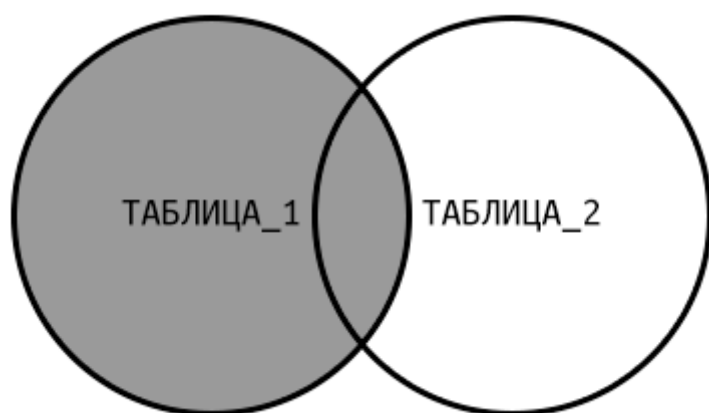
	123 id	abc first_name	abc last_name	abc gender	birthday	abc email	abc name_department	123 salary
1	5	Екатерина	Васильева	Женский	1995-03-30	vasilieva@yandex.ru	Отдел логистики	38 900
2	11	Ольга	Морозова	Женский	1993-02-14	morozova@yandex.ru	Отдел логистики	70 000
3	17	Кирилл	Егоров	Мужской	1986-07-22	[NULL]	Отдел логистики	62 000
4	23	Иван	Кудряшов	Мужской	1987-06-20	kudryashov@yandex.ru	Отдел логистики	80 000
5	29	Алёна	Ильина	Женский	1995-09-30	ilina@yandex.ru	Отдел логистики	75 000
6	35	Екатерина	Кузнецова	Женский	1991-07-22	kuznetsova@yandex.ru	Отдел логистики	72 000
7	41	Екатерина	Попова	Женский	1993-01-08	popova@yandex.ru	Отдел логистики	73 000
8	47	Павел	Григорьев	Мужской	1987-12-04	grigoryev@mail.ru	Отдел логистики	77 000

Левое внешнее соединение (LEFT OUTER JOIN)

Оператор `LEFT OUTER JOIN` или его еще называют `LEFT JOIN` предназначен для соединения таблиц и вывода результатов, в которых данные полностью удовлетворяют условию, указанному после ключевого слова `ON`, и дополняются записями из левой таблицы (первой по порядку), даже если они не соответствуют условию объединения.

У записей левой таблицы (первой), которые не соответствуют условию, значения столбца из правой таблицы (второй) будут равняться `NULL` (неопределёнными).

Наглядная работа оператора `LEFT JOIN` представлена на изображении ниже. Заштрихованная область, это данные, которые соответствуют условию `ON`.



Синтаксис:

При объединении таблиц можно указывать, как `LEFT OUTER JOIN`, так и `LEFT JOIN`, разницы между ними нет, поскольку выполняют одну и ту же функцию.

- `column_list` – список столбцов;
- `schema` – наименование схемы, в которой находится объект;
- `table_name_1` и `table_name_2` – имена таблиц;
- `alias_table_1` и `alias_table_2` – псевдонимы таблиц;

- `column_name` – имя столбца;
- `condition_filter` – условия для фильтрации.

```
SELECT column_list
FROM schema.table_name_1 AS alias_table_1
[LEFT OUTER JOIN | LEFT JOIN] schema.table_name_2 AS alias_table_2
ON alias_table_2.column_name = alias_table_1.column_name
WHERE condition_filter;
```

Практический пример: необходимо объединить таблицы `departments` и `employees` по идентификатору отдела и вывести информацию о всех сотрудниках с именем Екатерина, которые работают в отделе с идентификатором 5.

Пояснение к SQL-запросу:

- в блоке `FROM` указываем таблицу `departments`, из нее будут получены основные данные о сотрудниках;
- в операторе `LEFT JOIN` указываем таблицу `employees`, с которой будет происходить объединение. После ключевого слова `ON` указываем условия объединения по идентификатору отдела, а также задаем ограничения на выборку данных из таблицы `employees`. То есть, так мы указываем, что нужны сотрудники с именем Екатерина, которые работают в отделе с идентификатором 5.

```
SELECT e.id,
       e.first_name,
       e.last_name,
       e.gender,
       e.birthday,
       e.email,
       d.id AS id_department,
       d.name_department,
       e.salary
FROM departments d
LEFT JOIN employees e
ON d.id = e.id_department
   AND e.id_department = 5
   AND e.first_name = 'Екатерина';
```

Выполняем запрос и получаем результат, в котором присутствуют не только сотрудники с именем Екатерина работающие в отделе с идентификатором 5, а также дополнительные строки. Чтобы с этим разобраться, прочитайте внимательно пояснение к полученному результату.

Пояснение к полученному результату:

- строки с 5-7 содержат информацию о сотруднике и отделе, в котором он работает, так как они прошли по условию объединения в блоке ON;
- строки под номерами 1-4 и 8-11 были добавлены оператором LEFT JOIN. Другими словами, в этих строках указана информация об отделах из таблицы departments, которым не нашлось совпадения в таблице employees, поэтому в этих строках нет информации о сотрудниках.

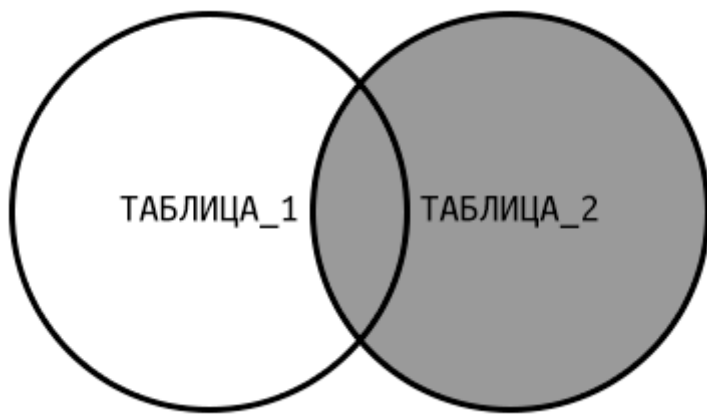
	123 id	abc first_name	abc last_name	abc gender	birthday	abc email	123 id_department	abc name_department	123 salary
1	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]	1	Отдел маркетинга	[NULL]
2	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]	2	Отдел финансов	[NULL]
3	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]	3	Отдел разработки	[NULL]
4	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]	4	Отдел кадров	[NULL]
5	41	Екатерина	Попова	Женский	1993-01-08	popova@yandex.ru	5	Отдел логистики	73 000
6	35	Екатерина	Кузнецова	Женский	1991-07-22	kuznetsova@yandex.ru	5	Отдел логистики	72 000
7	5	Екатерина	Васильева	Женский	1995-03-30	vasileva@yandex.ru	5	Отдел логистики	38 900
8	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]	6	Отдел качества	[NULL]
9	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]	7	Отдел взыскания	[NULL]
10	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]	8	Отдел безопасности	[NULL]
11	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]	9	Отдел поддержки	[NULL]

Правое внешнее соединение (RIGHT OUTER JOIN)

Оператор RIGHT OUTER JOIN или как его еще называют RIGHT JOIN предназначен для соединения таблиц и вывода результатов, в которых данные полностью удовлетворяют условию, указанному после ключевого слова ON, и дополняются записями из правой таблицы (второй по порядку), даже если они не соответствуют условию объединения.

У записей правой таблицы (второй), которые не соответствуют условию, значения столбца из левой таблицы (первой) будут равняться NULL (неопределёнными).

Наглядная работа оператора RIGHT JOIN представлена на изображении ниже. Заштрихованная область, это данные, которые соответствуют условию ON.



Синтаксис:

При объединении таблиц можно указывать, как `RIGHT OUTER JOIN`, так и `RIGHT JOIN`, разницы между ними нет, поскольку выполняют одну и ту же функцию.

- `column_list` – список столбцов;
- `schema` – наименование схемы, в которой находится объект;
- `table_name_1` и `table_name_2` – имена таблиц;
- `alias_table_1` и `alias_table_2` – псевдонимы таблиц;
- `column_name` – имя столбца;
- `condition_filter` – условия для фильтрации.

```
SELECT column_list
FROM schema.table_name_1 AS alias_table_1
[RIGHT OUTER JOIN | RIGHT JOIN] schema.table_name_2 AS alias_table_2
ON alias_table_2.column_name = alias_table_1.column_name
WHERE condition_filter;
```

Практический пример: необходимо объединить таблицы `employees` и `departments` по идентификатору отдела, а затем вывести информацию о всех сотрудниках и отделах, в которых они трудоустроены.

Пояснение к SQL-запросу:

- в блоке `FROM` указываем таблицу `employees`, из нее будут получены основные данные о сотрудниках;
- в операторе `RIGHT JOIN` указываем таблицу `departments`, с которой будет происходить объединение. После ключевого слова `ON` указываем условия объединения по идентификатору отдела.

```

SELECT e.id,
       e.first_name,
       e.last_name,
       e.gender,
       e.birthday,
       e.email,
       d.id AS id_department,
       d.name_department,
       e.salary
FROM employees e
RIGHT JOIN departments d
ON d.id = e.id_department;

```

Выполняем запрос и получаем результат, в котором присутствует информация о сотрудниках и отделах, но также присутствуют дополнительные строки. Чтобы с этим разобраться, прочитайте внимательно пояснение к полученному результату.

Пояснение к полученному результату:

- строки с 1-50 содержат информацию о сотруднике и отделе, в котором он работает, так как они прошли по условию объединения в блоке ON;
- строки 51-53 были добавлены оператором `RIGHT JOIN`. Другими словами, в этих строках указана информация об отделах из таблицы `departments`, которым не нашлось совпадения в таблице `employees`, поэтому в этих строках нет информации о сотрудниках.

	123 id	ABC first_name	ABC last_name	ABC gender	ABC birthday	ABC email	123 id_department	ABC name_department	123 salary
42	5	Екатерина	Васильева	Женский	1995-03-30	vasilieva@yandex.ru	5	Отдел логистики	38 900
43	48	Анна	Борисова	Женский	1990-10-18	borisova@yandex.ru	6	Отдел качества	95 000
44	42	Сергей	Лебедев	Мужской	1996-07-14	lebedev@gmail.com	6	Отдел качества	120 000
45	36	Алексей	Макаров	Мужской	1986-10-08	makarov@gmail.com	6	Отдел качества	115 000
46	30	Максим	Прокофьев	Мужской	1988-04-05	[NULL]	6	Отдел качества	100 000
47	24	Александра	Белова	Женский	1986-12-15	belova@gmail.com	6	Отдел качества	115 000
48	18	Денис	Пономарев	Мужской	1989-04-08	ponomarev@gmail.com	6	Отдел качества	108 000
49	12	Сергей	Павлов	Мужской	1987-12-11	pavlov@gmail.com	6	Отдел качества	130 000
50	6	Дмитрий	Попов	Мужской	1988-11-05	popov@gmail.com	6	Отдел качества	110 000
51	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]	7	Отдел взыскания	[NULL]
52	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]	8	Отдел безопасности	[NULL]
53	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]	9	Отдел поддержки	[NULL]

Полное внешнее соединение (FULL OUTER JOIN)

Оператор `RIGHT OUTER JOIN` или как его еще называют `FULL JOIN` предназначен для соединения таблиц и вывода результатов, в которых данные полностью удовлетворяют условию, указанному после ключевого слова `ON`, и дополняются записями из левой

таблицы (первой по порядку) и правой таблицы (второй по порядку), даже если они не соответствуют условию объединения.



По завершению изучения оператора `FULL JOIN`, восстановите данные в таблицах `employees` и `departments`. Для этого необходимо выполнить весь код из главы – [Тестовые данные для работы](#).

У записей, которые не соответствуют условию, значение столбцов из другой таблицы будет равно `NULL` (неопределённым).

Наглядная работа оператора `FULL JOIN` представлена на изображении ниже.



Синтаксис:

При объединении таблиц можно указывать, как `FULL OUTER JOIN`, так и `FULL JOIN`, разницы между ними нет, поскольку выполняют одну и ту же функцию.

- `column_list` – список столбцов;
- `schema` – наименование схемы, в которой находится объект;
- `table_name_1` и `table_name_2` – имена таблиц;
- `alias_table_1` и `alias_table_2` – псевдонимы таблиц;
- `column_name` – имя столбца;
- `condition_filter` – условия для фильтрации.

```
SELECT column_list
FROM schema.table_name_1 AS alias_table_1
[FULL OUTER JOIN | FULL JOIN] schema.table_name_2 AS alias_table_2
```

```
ON alias_table_2.column_name = alias_table_1.column_name
WHERE condition_filter;
```

Практический пример: необходимо объединить таблицы `employees` и `departments` по идентификатору отдела, а затем вывести информацию о всех сотрудниках и отделах, в которых они трудоустроены.

Для демонстрации работы оператора `FULL JOIN` необходимо добавить в таблицу `employees` новую запись о сотруднике, у которого будет заполнен только идентификатор, фамилия, имя, пол, дата рождения и электронный адрес:

```
INSERT INTO employees
(id, last_name, first_name, gender, birthday, email, id_department,
id_boss, salary)
VALUES
(51, 'Петренко', 'Василий', 'Мужской', '1991-11-25',
'pet.vasya@google.com', NULL, NULL, NULL);
```

Дальше необходимо объединить таблицы `employees` и `departments` при помощи оператора `FULL JOIN`.

Пояснение к SQL-запросу:

- в блоке `FROM` указываем таблицу `employees`, из нее будут получены основные данные о сотрудниках;
- в операторе `FULL JOIN` указываем таблицу `departments`, с которой будет происходить объединение. После ключевого слова `ON` указываем условия объединения по идентификатору отдела.

```
SELECT e.id,
       e.first_name,
       e.last_name,
       e.gender,
       e.birthday,
       e.email,
       d.id AS id_department,
       d.name_department,
       e.salary
FROM employees e
```

```
FULL JOIN departments d
ON d.id = e.id_department;
```

Выполняем запрос и получаем результат, в котором присутствует информация о сотрудниках и отделах, но также присутствуют дополнительные строки. Чтобы с этим разобраться, прочитайте внимательно пояснение к полученному результату.

Пояснение к полученному результату:

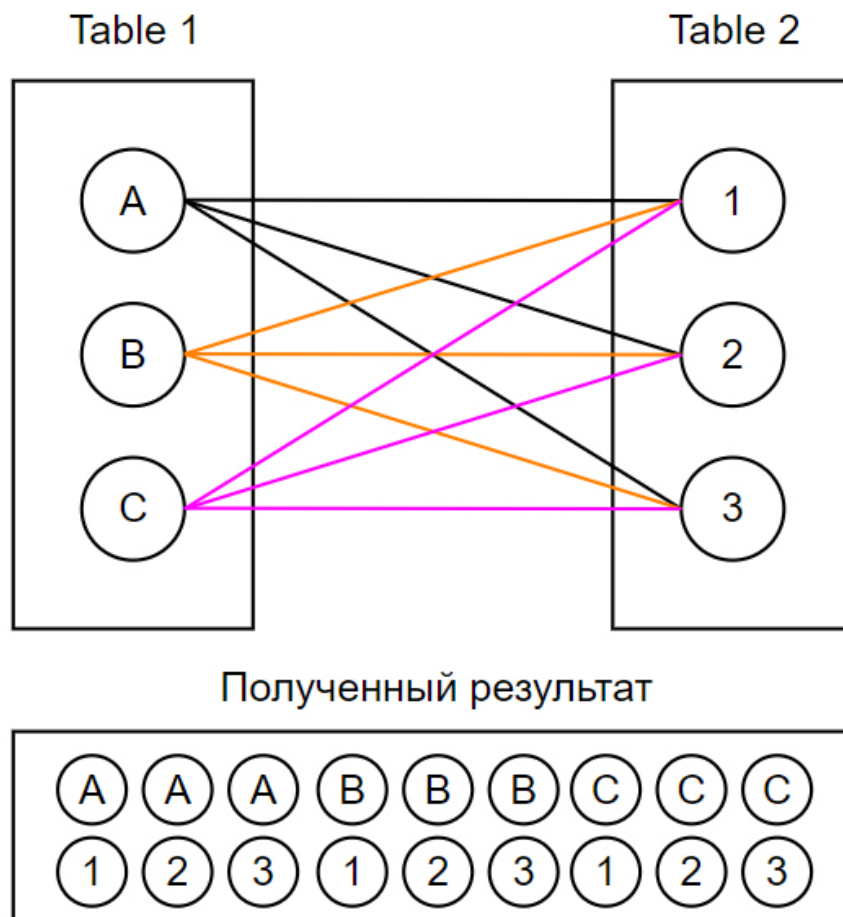
- строки с 1-50 содержат информацию о сотруднике и отделе, в котором он работает, так как они прошли по условию объединения в блоке ON;
- строки 51-53 были добавлены оператором FULL JOIN, так им не нашлось совпадений в таблице employees;
- строка 54 также была добавлена оператором FULL JOIN, так как ей не нашлось совпадения в таблице departments;

	123 id	abc first_name	abc last_name	abc gender	birthday	abc email	123 id_department	abc name_department	123 salary
43	48	Анна	Борисова	Женский	1990-10-18	borisova@yandex.ru	6	Отдел качества	95 000
44	42	Сергей	Лебедев	Мужской	1996-07-14	lebedev@gmail.com	6	Отдел качества	120 000
45	36	Алексей	Макаров	Мужской	1986-10-08	makarov@gmail.com	6	Отдел качества	115 000
46	30	Максим	Прокофьев	Мужской	1988-04-05	[NULL]	6	Отдел качества	100 000
47	24	Александра	Белова	Женский	1986-12-15	belova@gmail.com	6	Отдел качества	115 000
48	18	Денис	Пономарев	Мужской	1989-04-08	ponomarev@gmail.com	6	Отдел качества	108 000
49	12	Сергей	Павлов	Мужской	1987-12-11	pavlov@gmail.com	6	Отдел качества	130 000
50	6	Дмитрий	Попов	Мужской	1988-11-05	popov@gmail.com	6	Отдел качества	110 000
51	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]	7	Отдел взыскания	[NULL]
52	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]	8	Отдел безопасности	[NULL]
53	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]	9	Отдел поддержки	[NULL]
54	51	Василий	Петренко	Мужской	1991-11-25	pet.vasya@google.com	[NULL]	[NULL]	[NULL]

Перекрёстное соединение (CROSS JOIN)

Оператор CROSS JOIN позволяет выполнить перекрёстное соединение (Декартово произведение) двух таблиц. Результатом работы оператора CROSS JOIN будет объединение первой строки первой таблицы, с каждой строкой второй таблицы.

Для наглядности работы оператора CROSS JOIN, ниже находится изображение, на котором показано, как происходит перекрёстное соединение таблиц. Для удобства отображения результатов, они представлены в колоночном виде, а не в построчном.



Синтаксис:

Оператор `CROSS JOIN` в отличие от других способов соединения таблиц не требует никаких условий для объединения таблиц (`ON`). Существует несколько способов создать перекрёстное соединение (Декартово произведение).

- `column_list` – список столбцов;
- `schema` – наименование схемы, в которой находится объект;
- `table_name_1` и `table_name_2` – имена таблиц;
- `alias_table_1` и `alias_table_2` – псевдонимы таблиц;
- `condition_filter` – условия для фильтрации.

```
-- 1 вариант: при помощи оператора CROSS JOIN
SELECT column_list
FROM schema.table_name_1 AS alias_table_1
CROSS JOIN schema.table_name_2 AS alias_table_2
WHERE condition_filter;
```

```
-- 2 вариант: список таблиц в блоке FROM
SELECT column_list
FROM schema.table_name_1 AS alias_table_1,
     schema.table_name_2 AS alias_table_2
WHERE condition_filter;

-- 3 вариант: при помощи оператора INNER JOIN
SELECT column_list
FROM schema.table_name_1 AS alias_table_1
INNER JOIN schema.table_name_2 AS alias_table_2
ON TRUE
WHERE condition_filter;
```

Практический пример: необходимо объединить таблицы `employees` и `departments` при помощи оператора `CROSS JOIN`, а затем вывести информацию о сотруднике и отделе.

Пояснение к SQL-запросу:

- в блоке `FROM` указываем таблицу `employees`, из нее будут получены основные данные о сотрудниках;
- в операторе `CROSS JOIN` указываем таблицу `departments`, с которой будет выполнено объединение;
- в блоке `WHERE` указываем условие, что нужны сотрудники с идентификаторами 49 и 50 из таблицы `employees`.

```
SELECT e.id,
       e.last_name,
       e.first_name,
       e.gender,
       e.birthday,
       e.email,
       d.id AS id_department,
       d.name_department,
       e.salary
FROM employees e
CROSS JOIN departments d
WHERE e.id IN (49, 50)
ORDER BY e.id ASC, d.id ASC;
```

Выполняем запрос и получаем результат, в котором можно увидеть, что каждый сотрудник из таблицы `employees` был соединен со всеми отделами таблицы `departments`.

Таблица

Текст

Запись

	123 id	ABC last_name	ABC first_name	ABC gender	birthday	ABC email	123 id_department	ABC name_department	123 salary
1	49	Карпов	Денис	Мужской	1995-01-20	karpov@gmail.com	1	Отдел маркетинга	68 000
2	49	Карпов	Денис	Мужской	1995-01-20	karpov@gmail.com	2	Отдел финансов	68 000
3	49	Карпов	Денис	Мужской	1995-01-20	karpov@gmail.com	3	Отдел разработки	68 000
4	49	Карпов	Денис	Мужской	1995-01-20	karpov@gmail.com	4	Отдел кадров	68 000
5	49	Карпов	Денис	Мужской	1995-01-20	karpov@gmail.com	5	Отдел логистики	68 000
6	49	Карпов	Денис	Мужской	1995-01-20	karpov@gmail.com	6	Отдел качества	68 000
7	49	Карпов	Денис	Мужской	1995-01-20	karpov@gmail.com	7	Отдел взыскания	68 000
8	49	Карпов	Денис	Мужской	1995-01-20	karpov@gmail.com	8	Отдел безопасности	68 000
9	49	Карпов	Денис	Мужской	1995-01-20	karpov@gmail.com	9	Отдел поддержки	68 000
10	50	Максимов	Максим	Мужской	1988-07-28	maximov@mail.ru	1	Отдел маркетинга	102 000
11	50	Максимов	Максим	Мужской	1988-07-28	maximov@mail.ru	2	Отдел финансов	102 000
12	50	Максимов	Максим	Мужской	1988-07-28	maximov@mail.ru	3	Отдел разработки	102 000
13	50	Максимов	Максим	Мужской	1988-07-28	maximov@mail.ru	4	Отдел кадров	102 000
14	50	Максимов	Максим	Мужской	1988-07-28	maximov@mail.ru	5	Отдел логистики	102 000
15	50	Максимов	Максим	Мужской	1988-07-28	maximov@mail.ru	6	Отдел качества	102 000
16	50	Максимов	Максим	Мужской	1988-07-28	maximov@mail.ru	7	Отдел взыскания	102 000
17	50	Максимов	Максим	Мужской	1988-07-28	maximov@mail.ru	8	Отдел безопасности	102 000
18	50	Максимов	Максим	Мужской	1988-07-28	maximov@mail.ru	9	Отдел поддержки	102 000

Обновить

Save

Cancel

Самообъединение (SELF JOIN)

`SELF JOIN` — это запрос, который предназначен для самообъединения таблицы, то есть таблица соединяется сама с собой.



Оператора `SELF JOIN` не существует, самообъединение таблицы достигается за счёт использования таких операторов, как [INNER JOIN](#), [LEFT JOIN](#) или [RIGHT JOIN](#).

Синтаксис:

Для формирования запроса `SELF JOIN` нужно указать одну и ту же таблицу дважды, но присвоить им разные [псевдонимы](#) (алиасы).

- `column_list` – список столбцов;
- `schema` – наименование схемы, в которой находится объект;
- `table_name_1` – имя таблицы;
- `alias_table_1` и `alias_table_2` – псевдонимы таблиц;
- `column_name` – имя столбца;
- `condition_filter` – условия для фильтрации.

```
SELECT list_column
FROM schema.table_name_1 AS alias_table_1
[INNER JOIN|LEFT JOIN|RIGHT JOIN] schema.table_name_1 AS alias_table_2
```



```
ON alias_table_2.column_name = alias_table_1.column_name
WHERE condition_filter;
```

Практический пример: в таблице `employees` хранится информация о сотрудниках компании, необходимо для каждого сотрудника найти руководителя и вывести его фамилию и имя.

Пояснение к SQL-запросу:

- запрос сформирован при помощи оператора `LEFT JOIN` и дважды ссылается на таблицу `employees`, сначала для поиска сотрудника, а затем для поиска его руководителя. Псевдоним `e1` используется для сотрудника, а `e2` для руководителя;
- в блоке `ON` указывается условие для самообъединения таблицы `employees` и означает, что будут выведены только те записи, которые соответствуют условию `id_руководителя = id_руководителя_сотрудника`.

```
SELECT e1.id,
       e1.last_name,
       e1.first_name,
       e1.gender,
       e1.id_boss,
       e2.last_name||' '||e2.first_name AS name_boss
FROM employees e1
LEFT JOIN employees e2
ON e2.id = e1.id_boss
WHERE e1.id_department = 4
ORDER BY e1.id ASC;
```

Выполняем запрос и получаем результат со списком сотрудников, которые работают в отделе с идентификатором 4 и напротив каждого сотрудника в столбце `name_boss` указана фамилия и имя его руководителя.

Пояснение к полученному результату:

- обратите внимание на строку 1, в ней не заполнены столбцы `id_boss` и `name_boss`. Это связано с тем, что при выполнении самообъединения таблицы `employees` сотруднику не нашлось совпадения по условию объединения. Другими словами, сотрудник в строке 1 является руководителем отдела и у него за заполнен столбец `id_boss`, поэтому значение в столбце `name_boss` отсутствует.

	123 id	ABC last_name	ABC first_name	ABC gender	123 id_boss	ABC name_boss
1	4	Петров	Петр	Мужской	[NULL]	[NULL]
2	10	Новиков	Андрей	Мужской	4	Петров Петр
3	16	Кузьмина	Елена	Женский	4	Петров Петр
4	22	Миронова	Марина	Женский	10	Новиков Андрей
5	28	Макарова	Татьяна	Женский	10	Новиков Андрей
6	34	Денисова	Вероника	Женский	16	Кузьмина Елена
7	40	Иванова	Ольга	Женский	16	Кузьмина Елена
8	46	Семенов	Иван	Мужской	10	Новиков Андрей

Естественное объединение (NATURAL JOIN)

Оператор `NATURAL JOIN` позволяет объединить таблицы при помощи неявного объединения, то есть не нужно указывать условие для объединения (`ON`) и объединение таблиц будет выполнено на основании общего столбца или столбцов этих таблиц. Другими словами, таблицы будут объединены по столбу или столбцам, которые есть и в той, и в другой таблице.



Используйте оператор `NATURAL JOIN` только тогда, когда это возможно, потому что это может привести к неожиданному результату.

По завершению изучения оператора `NATURAL JOIN`, восстановите данные в таблицах `employees` и `departments`. Для этого необходимо выполнить весь код из главы – [Тестовые данные для работы](#).

Синтаксис:

- `column_list` – список столбцов;
- `schema` – наименование схемы, в которой находится объект;
- `table_name_1` и `table_name_2` – имена таблиц;
- `alias_table_1` и `alias_table_2` – псевдонимы таблиц;
- `condition_filter` – условия для фильтрации.

```
SELECT column_list
FROM schema.table_name_1 AS alias_table_1
NATURAL [INNER, LEFT, RIGHT] JOIN schema.table_name_2 AS alias_table_2
WHERE condition_filter;
```

Практический пример: необходимо объединить две таблицы `employees` и `departments` при помощи оператора `NATURAL JOIN`, а затем вывести информацию о сотрудниках, которые работают в отделе идентификатором 3.

Для демонстрации работы оператора `NATURAL JOIN`, необходимо внести изменения в таблицу `departments`, а именно изменить имя столбца `id` на `id_department`:

```
ALTER TABLE departments RENAME COLUMN id TO id_department;
```

Дальше нужно написать запрос для объединения таблиц при помощи `NATURAL JOIN`.

Пояснение к SQL-запросу:

- таблицы `employees` и `departments` будут неявно объединены, то есть объединение будет выполнено по столбцу `id_department`, так как он есть и в той, и в другой таблице;
- в блоке `WHERE` указываем условие, что нужны сотрудники, которые работают в отделе с идентификатором 3.

```
SELECT e.id,  
       e.last_name,  
       e.first_name,  
       e.gender,  
       e.birthday,  
       e.email,  
       e.id_boss,  
       d.id_department,  
       d.name_department,  
       e.salary  
FROM employees e  
NATURAL JOIN departments d  
WHERE e.id_department = 3  
ORDER BY e.id ASC;
```

А вот тот же самый запрос, но он написан при помощи оператора `INNER JOIN`. Эти запросы вернут идентичные результаты, так как эквивалентны запросу выше.

```
-- 1 запрос  
SELECT e.id,  
       e.last_name,
```

```

        e.first_name,
        e.gender,
        e.birthday,
        e.email,
        e.id_boss,
        d.id_department,
        d.name_department,
        e.salary
FROM employees e
INNER JOIN departments d USING (id_department)
WHERE e.id_department = 3
ORDER BY e.id ASC;

```

-- 2 запрос

```

SELECT e.id,
       e.last_name,
       e.first_name,
       e.gender,
       e.birthday,
       e.email,
       e.id_boss,
       d.id_department,
       d.name_department,
       e.salary
FROM employees e
INNER JOIN departments d
ON d.id_department = e.id_department
WHERE e.id_department = 3
ORDER BY e.id ASC;

```

Выполняем запрос, который написан при помощи оператора `NATURAL JOIN` и получаем список сотрудников, которые работают в отделе с идентификатором 3.

	123 id	last_name	first_name	gender	birthday	email	id_boss	id_department	name_department	salary
1	8	Сидорова	Мария	Женский	1992-08-10	sidorova@gmail.com	[NULL]	3	Отдел разработки	54 000
2	9	Козлов	Артём	Мужской	1986-10-22	kozlov@gmail.com	3	3	Отдел разработки	120 000
3	15	Максимова	Вероника	Женский	1990-08-30	maximova@gmail.com	3	3	Отдел разработки	72 000
4	21	Григорьев	Михаил	Мужской	1988-10-25	grigoryev@gmail.com	9	3	Отдел разработки	125 000
5	27	Афанасьев	Григорий	Мужской	1995-01-08	afanasyev@gmail.com	15	3	Отдел разработки	110 000
6	33	Фомин	Дмитрий	Мужской	1994-02-14	fomin@gmail.com	9	3	Отдел разработки	90 000
7	39	Кудряшова	Анастасия	Женский	1990-09-18	kudryashova@gmail.com	15	3	Отдел разработки	100 000
8	45	Пономарева	Максим	Женский	1984-05-28	ponomareva@gmail.com	9	3	Отдел разработки	108 000

Обновить

Save

Cancel

Экспорт данных ...

200

8

8 строк получено - 1ms, 2024-01-22 в 16:29:54

Обобщённое табличное выражение (WITH)

Обобщённое табличное выражение или **СТЕ** (Common Table Expressions) — это временный результирующий набор данных, который хранится в оперативной памяти сервера (RAM) и доступен только в пределах текущего SQL-запроса. После выполнения SQL-запроса данные из оперативной памяти удаляются и повторно обратиться к результирующему набору данных нельзя.

Для создания обобщённого табличного выражения используется оператор `WITH`. При помощи данного оператора можно улучшить наглядное представление SQL кода, и расширить функциональность вашего запроса.



Можно использовать несколько операторов `WITH` одновременно, а также комбинировать их с процедурами, функциями или операторами для объединения таблиц.

Не стоит усердствовать с использованием оператора `WITH`, так как временные наборы данных заполняют оперативную память сервера.

Синтаксис для одного оператора `WITH`:

- `name_cte` — имя СТЕ;
- `subquery` — подзапрос;
- `list_column` — список столбцов;
- `alias_cte` — псевдоним для СТЕ;
- `schema` — наименование схемы, в которой находится объект;
- `table_name` — имя таблицы;
- `alias_table` — псевдоним для таблицы;
- `column_name` — имя столбца;
- `condition_filter` — условия для фильтрации.

```
-- 1 вариант
WITH name_cte AS (
    subquery
)
SELECT list_column
FROM name_cte AS alias_cte
WHERE condition_filter;
```

```
-- 2 вариант
WITH name_cte AS (
    subquery
)
SELECT column_list
FROM schema.table_name AS alias_table
[LEFT|FULL|RIGHT] JOIN name_cte AS alias_cte
ON alias_cte.column_name = alias_table.column_name
WHERE condition_filter;
```

Синтаксис для нескольких операторов WITH:

- name_cte_1 и name_cte_2 – имя CTE;
- subquery – подзапрос;
- list_column – список столбцов;
- alias_cte_1 и alias_cte_2 – псевдонимы для CTE;
- schema – наименование схемы, в которой находится объект;
- table_name – имя таблицы;
- alias_table – псевдоним для таблицы;
- column_name – имя столбца
- condition_filter – условия для фильтрации.

```
-- 1 вариант
WITH name_cte_1 AS (
    subquery
),
name_cte_2 AS (
    subquery
)
SELECT column_list
FROM name_cte_1 AS alias_cte_1
[LEFT|FULL|RIGHT] JOIN name_cte_2 AS alias_cte_2
ON alias_cte_1.column_name = alias_cte_2.column_name
WHERE condition_filter;
```

```
-- 2 вариант
```

```

WITH name_cte_1 AS (
    subquery
),
name_cte_2 AS (
    subquery
)
SELECT column_list
FROM schema.table_name AS alias_table
[LEFT | FULL | RIGHT] JOIN name_cte_1 AS alias_cte_1
ON alias_cte_1.column_name = alias_table.column_name
[LEFT|FULL|RIGHT] JOIN name_cte_2 AS alias_cte_2
ON alias_cte_2.column_name = alias_cte_1.column_name
WHERE condition_filter;

```

Практический пример: есть SQL-запрос, он возвращает информацию о сотрудниках компании, а также название отдела, в котором трудоустроен сотрудник.

```

SELECT e.id,
       e.first_name,
       e.last_name,
       e.gender,
       e.birthday,
       e.email,
       d.name_department,
       e.salary
FROM employees e
JOIN departments d
ON d.id = e.id_department;

```

Результат выполнения запроса.

	123 id	ABC first_name	ABC last_name	ABC gender	ABC birthday	ABC email	ABC name_department	123 salary
1	49	Денис	Карпов	Мужской	1995-01-20	karpov@gmail.com	Отдел маркетинга	68 000
2	43	Игорь	Кузнецов	Мужской	1985-11-20	[NULL]	Отдел маркетинга	75 000
3	37	Игорь	Миронов	Мужской	1988-07-15	mironov@mail.ru	Отдел маркетинга	72 000
4	31	Артём	Семенов	Мужской	1987-08-12	semenov@mail.ru	Отдел маркетинга	70 000
5	25	Павел	Кондратьев	Мужской	1990-12-14	kpav@mail.ru	Отдел маркетинга	70 000
6	19	Анна	Семенова	Женский	1992-02-14	semenova@mail.ru	Отдел маркетинга	66 000
7	13	Наталья	Григорьева	Женский	1996-01-25	grigoryeva@mail.ru	Отдел маркетинга	60 000
8	7	Анастасия	Соколова	Женский	1991-07-12	sokolova@mail.ru	Отдел маркетинга	40 000
9	1	Иван	Иванов	Мужской	1990-05-15	ivanov@mail.ru	Отдел маркетинга	35 000
10	50	Максим	Максимов	Мужской	1988-07-28	maximov@mail.ru	Отдел финансов	102 000

Дальше необходимо создать обобщённое табличное выражение `get_employees`, в котором нужно разместить текущий SQL-запрос, а затем обратиться к созданному табличному выражению `get_employees` и вывести из него информацию о всех сотрудниках, которые работают в отделе маркетинга.

Пояснение к SQL-запросу:

- создаем табличное выражение `get_employees` и размещаем в нем запрос, который возвращает информацию о сотрудниках;
- обращаемся к табличному выражению `get_employees`, как к таблице и выводим сотрудников, которые работают в отделе маркетинга.

```
WITH get_employees AS (  
    SELECT e.id,  
           e.first_name,  
           e.last_name,  
           e.gender,  
           e.birthday,  
           e.email,  
           d.name_department,  
           e.salary  
  
    FROM employees e  
    JOIN departments d  
    ON d.id = e.id_department  
)  
SELECT *  
FROM get_employees  
WHERE name_department = 'Отдел маркетинга';
```

Выполняем запрос и получаем список с сотрудниками, которые работают в отделе маркетинга.

	123 id	ABC first_name	ABC last_name	ABC gender	birthday	ABC email	ABC name_department	123 salary
1	1	Иван	Иванов	Мужской	1990-05-15	ivanov@mail.ru	Отдел маркетинга	35 000
2	7	Анастасия	Соколова	Женский	1991-07-12	sokolova@mail.ru	Отдел маркетинга	40 000
3	13	Наталья	Григорьева	Женский	1996-01-25	grigoryeva@mail.ru	Отдел маркетинга	60 000
4	19	Анна	Семенова	Женский	1992-02-14	semenova@mail.ru	Отдел маркетинга	66 000
5	25	Павел	Кондратьев	Мужской	1990-12-14	kpav@mail.ru	Отдел маркетинга	70 000
6	31	Артём	Семенов	Мужской	1987-08-12	semenov@mail.ru	Отдел маркетинга	70 000
7	37	Игорь	Миронов	Мужской	1988-07-15	mironov@mail.ru	Отдел маркетинга	72 000
8	43	Игорь	Кузнецов	Мужской	1985-11-20	[NULL]	Отдел маркетинга	75 000
9	49	Денис	Карпов	Мужской	1995-01-20	karpov@gmail.com	Отдел маркетинга	68 000

Обновить
Save
Cancel
Экспорт данных ...
200
9
9 строк получено

Функции преобразования

Получить дополнительную информацию по функциям преобразования можно на страницах [официального руководства](#) PostgreSQL, так как в этой главе будут рассмотрены только некоторые функции.

Функция TO_NUMBER

Функции TO_NUMBER выполняет преобразование строки в число.

Синтаксис:

- `value` – статическое значение или имя столбца;
- `format` – необязательный параметр, который отвечает за формат числа, соответствующий входной строке. Пример шаблонов: 9 - позиция цифры, S - знак числа (зависит от локальных настроек), D - десятичная точка (зависит от локальных настроек), . - период и т.д.;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

```
SELECT TO_NUMBER(value, format)
FROM schema.table_name;
```

Также можно преобразовать строку в число с плавающей точкой при помощи конструкции `::numeric` или `::float`, а для целочисленных значений можно использовать `::integer`. После `::` можно указать нужный [числовой тип данных](#).

```
SELECT value::numeric
FROM schema.table_name;
```

Практический пример: необходимо преобразовать строку '-154.31' в число при помощи функции TO_NUMBER.

```
-- 1 вариант
SELECT TO_NUMBER('-154.31', 'S999.99');
```

```
-- 2 вариант
SELECT '-154.31'::numeric;
```

Выполняем запрос и получаем результат работы функции TO_NUMBER.

Таблица	123 to_number	
	1	-154,31
ст	Обновить Save	

Функция TO_CHAR

Функции TO_CHAR выполняет преобразование числа или даты в строку.

Синтаксис:

- value – статическое значение или имя столбца;
- format – формат, который будет использоваться для преобразования значения в строку. Формат задаётся при помощи цифр 9, например: для числа 451.14 будет формат 999.99, а для числа 17.2 будет 99.9;
- schema – наименование схемы, в которой находится объект;
- table_name – имя таблицы.

```
SELECT TO_CHAR(value, format)
FROM schema.table_name;
```

Также можно преобразовать число или дату в строку при помощи конструкции ::text. После :: можно указать нужный [символьный тип данных](#).

```
SELECT value::text
FROM schema.table_name;
```

Практический пример: есть SQL-запрос, он возвращает столбец numbers, в котором хранится числовой набор данных. Необходимо преобразовать столбец numbers тремя способами при помощи функции TO_CHAR с использованием форматов 999.999, 999.99 и 999.9.

```
SELECT t.numbers,
       to_char(t.numbers, '999.999') AS convert_1,
       to_char(t.numbers, '999.99') AS convert_2,
       to_char(t.numbers, '999.9') AS convert_3
FROM (
  SELECT 623.452 AS numbers
  UNION ALL
```

```

SELECT 603.316 AS numbers
UNION ALL
SELECT 593.214 AS numbers
) t;

```

Выполняем запрос и получаем результат работы функции TO_CHAR.

	123 numbers	ABC convert_1	ABC convert_2	ABC convert_3
1	623,452	623.452	623.45	623.5
2	603,316	603.316	603.32	603.3
3	593,214	593.214	593.21	593.2

Функция TO_DATE

Функции TO_DATE выполняет преобразование строки в дату.

Синтаксис:

- value – статическое значение или имя столбца;
- format – формат, который будет использоваться для преобразования строки в дату. Форматов существует очень много, поэтому рассмотрим самые основные:
 YYYY - год;
 MM - месяц;
 DD - день;
 Q - квартал;
 DAY - название дня;
 HH24 - час (0-24 ч);
 HH/HH12 - час дня (1-12);
 MI - минута (0-59);
 SS - секунда (0-59).
- schema – наименование схемы, в которой находится объект;
- table_name – имя таблицы.

```

SELECT TO_DATE(value, format)
FROM schema.table_name;

```

Также можно преобразовать строку в дату при помощи конструкции `::date`. После `::` можно указать нужный [временной тип данных](#).

```
SELECT value::date
FROM schema.table_name;
```

Практический пример: есть SQL-запрос, он возвращает столбец `dates`, в котором хранится текстовый набор данных. Необходимо преобразовать столбец `dates` при помощи функции `TO_DATE` с использованием формата `DD.MM.YYYY`.

```
SELECT t.dates,
       to_date(t.dates, 'DD.MM.YYYY') AS convert_date
FROM (
    SELECT '30.01.2024' AS dates
    UNION ALL
    SELECT '12.01.2024' AS dates
    UNION ALL
    SELECT '02.01.2024' AS dates
) t;
```

Выполняем запрос и получаем результат работы функции `TO_DATE`. Обратите внимание, отображение результатов в столбце `convert_date` отличается от установленного формата в функции `TO_DATE`, это из-за настроек клиента.

Таблица		ABC dates	convert_date
	1	30.01.2024	2024-01-30
	2	12.01.2024	2024-01-12
	3	02.01.2024	2024-01-02
Текст			
Обновить Save Cancel			

Математические функции

Получить дополнительную информацию по математическим функциям можно на страницах [официального руководства](#) PostgreSQL, так как в этой главе будут рассмотрены только некоторые функции.

Функция ABS

Функция `ABS` возвращает абсолютное значение числа, то есть модуль числа.

Синтаксис:

- `value` – статическое значение или имя столбца;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

```
SELECT ABS(value)
FROM schema.table_name;
```

Практический пример: есть SQL-запрос, он возвращает столбец `number`, в котором хранится список чисел. Необходимо для каждого числа определить модуль при помощи функции `ABS`.

```
SELECT t.numbers,
       ABS(t.numbers) AS abs_result
FROM (
    SELECT -2 AS numbers
    UNION ALL
    SELECT 4 AS numbers
    UNION ALL
    SELECT -8 AS numbers
) t;
```

Выполняем запрос и получаем результат работы функции `ABS`.

Таблица	123 numbers		123 abs_result	
	1	-2	2	
	2	4	4	
	3	-8	8	
Текст	Обновить		Save	Cancel

Функция CEIL

Функция `CEIL` возвращает наименьшее целое число, которое больше или равно указанному числу. Другими словами, функция округляет значение вверх до целого числа.

Синтаксис:

- `value` – статическое значение или имя столбца;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

```
SELECT CEIL(value)
FROM schema.table_name;
```

Практический пример: есть SQL-запрос, он возвращает столбец `number`, в котором хранится список чисел. Необходимо каждое значение округлить до целого числа вверх при помощи функции `CEIL`.

```
SELECT t.numbers,
       CEIL(t.numbers) AS ceil_result
FROM (
    SELECT 8200.43 AS numbers
    UNION ALL
    SELECT 14293.29 AS numbers
    UNION ALL
    SELECT 0 AS numbers
    UNION ALL
    SELECT 4383.39 AS numbers
    UNION ALL
    SELECT 9005.76 AS numbers
    UNION ALL
    SELECT 1731.12 AS numbers
    UNION ALL
```



```
SELECT -345.9 AS numbers
) t;
```

Выполняем запрос и получаем результат работы функции `CEIL`.

	123 numbers	123 ceil_result
1	8 200,43	8 201
2	14 293,29	14 294
3	0	0
4	4 383,39	4 384
5	9 005,76	9 006
6	1 731,12	1 732
7	-345,9	-345

Функция `FLOOR`

Функция `FLOOR` возвращает наибольшее целое значение, равное или меньше, чем число. Другими словами, функция округляет значение до ближайшего меньшего числа.

Синтаксис:

- `value` – статическое значение или имя столбца;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

```
SELECT FLOOR(value)
FROM schema.table_name;
```

Практический пример: есть SQL-запрос, он возвращает столбец `number`, в котором хранится список чисел. Необходимо каждое значение округлить до ближайшего меньшего числа при помощи функции `FLOOR`.

```
SELECT t.numbers,
       FLOOR(t.numbers) AS floor_result
FROM (
  SELECT 8200.43 AS numbers
  UNION ALL
  SELECT 14293.29 AS numbers
  UNION ALL
  SELECT 0 AS numbers
```

```

UNION ALL
SELECT 4383.39 AS numbers
UNION ALL
SELECT 9005.76 AS numbers
UNION ALL
SELECT 1731.12 AS numbers
UNION ALL
SELECT -345.9 AS numbers
) t;

```

Выполняем запрос и получаем результат работы функции FLOOR.

	123 numbers	123 floor_result
1	8 200,43	8 200
2	14 293,29	14 293
3	0	0
4	4 383,39	4 383
5	9 005,76	9 005
6	1 731,12	1 731
7	-345,9	-346

Функция MOD

Функция MOD возвращает остаток от деления.

Синтаксис:

- `numerator` – числитель, статическое значение или имя столбца;
- `denominator` – знаменатель, статическое значение или имя столбца;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

```

SELECT MOD(numerator, denominator)
FROM schema.table_name;

```

Практический пример: есть SQL-запрос, он возвращает столбец `number`, в котором хранится список чисел. Необходимо каждое число разделить на 3 и вернуть остаток от деления при помощи функции MOD.

```

SELECT t.numbers,
       MOD(t.numbers, 3) AS mod_result
FROM (
  SELECT 15 AS numbers
  UNION ALL
  SELECT 19 AS numbers
  UNION ALL
  SELECT 23 AS numbers
  UNION ALL
  SELECT 20 AS numbers
) t;

```

Выполняем запрос и получаем результат работы функции MOD.

Таблица	123 numbers	123 mod_result
1	15	0
2	19	1
3	23	2
4	20	2

Функция POWER

Функция POWER выполняет возведение числа в указанную степень.

Синтаксис:

- value – статическое значение или имя столбца;
- degree_numeric – степень, в которую нужно возвести указанное значение;
- schema – наименование схемы, в которой находится объект;
- table_name – имя таблицы.

```

SELECT POWER(value, degree_numeric)
FROM schema.table_name;

```

Практический пример: есть SQL-запрос, он возвращает столбец number, в котором хранится список чисел. Необходимо каждое число возвести в степень 2 при помощи функции POWER.

```

SELECT t.numbers,

```

```

        POWER(t.numbers, 2) AS power_result
FROM (
    SELECT 2 AS numbers
    UNION ALL
    SELECT 4 AS numbers
    UNION ALL
    SELECT 8 AS numbers
) t;

```

Выполняем запрос и получаем результат работы функции `POWER`.

Таблица	123 numbers	123 power_result
1	2	4
2	4	16
3	8	64

Текст Обновить Save Cancel

Функция `ROUND`

Функция `ROUND` выполняет округление числа.

Синтаксис:

- `value` – статическое значение или имя столбца;
- `number_characters` – количество знаков после запятой;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

```

SELECT ROUND(value, number_characters)
FROM schema.table_name;

```

Практический пример: есть SQL-запрос, он возвращает столбец `number`, в котором хранится список чисел. Необходимо каждое число округлить до 2 знака после запятой при помощи функции `ROUND`.

```

SELECT t.numbers,
       ROUND(t.numbers, 3) AS round_result
FROM (
    SELECT 15.36578 AS numbers
    UNION ALL

```

```

SELECT 19.6587 AS numbers
UNION ALL
SELECT 23.11258 AS numbers
UNION ALL
SELECT 20.17785 AS numbers
) t;

```

Выполняем запрос и получаем результат работы функции `ROUND`.

	123 numbers	123 round_result
1	15,36578	15,366
2	19,6587	19,659
3	23,11258	23,113
4	20,17785	20,178

Функция SIGN

Функция `SIGN` определяет знака числа, и возвращает значение 0, -1, 1 или `NULL` после проверки.

Пояснение к работе функции:

- если число < 0 , то возвращается -1;
- если число $= 0$, то возвращается 0;
- если число > 0 , то возвращается 1;
- если число $= \text{NULL}$, то возвращается `NULL`.

Синтаксис:

- `value` – статическое значение или имя столбца;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

```

SELECT SIGN(value)
FROM schema.table_name;

```

Практический пример: есть SQL-запрос, он возвращает столбец `number`, в котором хранится список чисел. Необходимо для каждого числа определить знак при помощи функции `SIGN`.

```

SELECT t.numbers,
       SIGN(t.numbers) AS sign_result
FROM (
    SELECT -15 AS numbers
    UNION ALL
    SELECT 0 AS numbers
    UNION ALL
    SELECT 18 AS numbers
    UNION ALL
    SELECT null AS numbers
) t;

```

Выполняем запрос и получаем результат работы функции `SIGN`.

Таблица	123 numbers	123 round_result
1	-15	-1
2	0	0
3	18	1
4	[NULL]	[NULL]

Функция SQRT

Функция `SQRT` возвращает квадратный корень числа.

Синтаксис:

- `value` – статическое значение или имя столбца;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

```

SELECT SQRT(value)
FROM schema.table_name;

```

Практический пример: есть SQL-запрос, он возвращает столбец `number`, в котором хранится список чисел. Необходимо для каждого числа найти квадратный корень при помощи функции `SQRT`.

```

SELECT t.numbers,
       SQRT(t.numbers) AS sqrt_result

```

```
FROM (
    SELECT 64 AS numbers
    UNION ALL
    SELECT 121 AS numbers
    UNION ALL
    SELECT 169 AS numbers
) t;
```

Выполняем запрос и получаем результат работы функции SQRT.

Таблица	123 numbers	123 sqrt_result
1	64	8
2	121	11
3	169	13

Функция TRUNC

Функция TRUNC выполняет усечение числа до определённого количества знаков после запятой.

Синтаксис:

- `value` – статическое значение или имя столбца;
- `number_characters` – количество знаков после запятой. Если не указано количество знаков после запятой, то значение будет усечено до целого числа;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

```
SELECT TRUNC(value, number_characters)
FROM schema.table_name;
```

Практический пример: есть SQL-запрос, он возвращает столбец `number`, в котором хранится список чисел. Необходимо каждое число усечь до 2 знака после запятой при помощи функции TRUNC.

```
SELECT t.numbers,
       TRUNC(t.numbers, 2) AS trunc_result
FROM (
    SELECT 15.36578 AS numbers
```

```

UNION ALL
SELECT 19.6587 AS numbers
UNION ALL
SELECT 23.11258 AS numbers
UNION ALL
SELECT 20.17785 AS numbers
) t;

```

Выполняем запрос и получаем результат работы функции TRUNC.

	123 numbers	123 trunc_result
1	15,36578	15,36
2	19,6587	19,65
3	23,11258	23,11
4	20,17785	20,17

Обновить Save Cancel

Работа с датой и временем

Получить дополнительную информацию по функциям для работы с датой и временем можно на страницах [официального руководства](#) PostgreSQL, так как в этой главе будут рассмотрены только некоторые функции.

Функция NOW

Функция `NOW` возвращает текущую дату и время в формате `timestamp`.

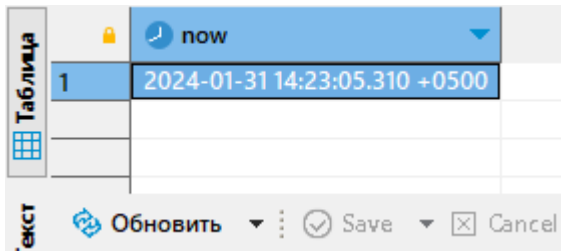
Синтаксис:

```
SELECT NOW();
```

Практический пример: необходимо вывести текущую дату и время при помощи функции `NOW()`.

```
SELECT NOW();
```

Выполняем запрос и получаем результат работы функции `NOW`.



The screenshot shows a table with one row containing the timestamp '2024-01-31 14:23:05.310 +0500'. The table has a column header 'now' and a row number '1'.

	now
1	2024-01-31 14:23:05.310 +0500

Функция CURRENT_DATE

Функция `CURRENT_DATE` возвращает текущую дату, без времени и временной зоны.

Синтаксис:

```
SELECT CURRENT_DATE;
```

Практический пример: необходимо вывести текущую дату при помощи функции `CURRENT_DATE`.

```
SELECT CURRENT_DATE;
```

Выполняем запрос и получаем результат работы функции `CURRENT_DATE`.

Таблица		current_date
	1	2024-01-31

екст Обновить Save

Функция CURRENT_TIME

Функция `CURRENT_TIME` возвращает текущее время без даты.

Синтаксис:

```
SELECT CURRENT_TIME;
```

Практический пример: необходимо вывести текущее время при помощи функции `CURRENT_TIME`.

```
SELECT CURRENT_TIME;
```

Выполняем запрос и получаем результат работы функции `CURRENT_TIME`.

Таблица		current_time
	1	14:34:07 +0500

екст Обновить Save

Функция EXTRACT

Функция `EXTRACT` позволяет извлечь конкретные значения (год, месяц, день, часы, минуты и т.д.) из даты или временного интервала.

Синтаксис:

- `column_name` – имя столбца из которого будут извлечены нужные значения;
- `extract_value` – значение, которое нужно извлечь. Например: `YEAR`, `MONTH`, `DAY`, `HOURL`, `MINUTE`, `SECOND` и т.д.;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

```
SELECT EXTRACT(extract_value FROM column_name)
```

```
FROM shema.table_name;
```

Практический пример: есть SQL-запрос, он возвращает столбец `dates`, в котором хранится список с датами. Необходимо извлечь из каждой строки столбца `dates` значения `YEAR`, `MONTH`, `DAY`, `HOURL`, `MINUTE` и `SECOND` при помощи функции `EXTRACT`.

```
SELECT t.dates,  
       extract(YEAR FROM t.dates) AS ext_year,  
       extract(MONTH FROM t.dates) AS ext_month,  
       extract(DAY FROM t.dates) AS ext_day,  
       extract(HOUR FROM t.dates) AS ext_hour,  
       extract(MINUTE FROM t.dates) AS ext_minute,  
       extract(SECOND FROM t.dates) AS ext_second  
FROM (  
    SELECT '2024.01.15 14:53:36'::timestamp AS dates  
    UNION ALL  
    SELECT '2024.01.30 18:31:05'::timestamp AS dates  
) t;
```

Выполняем запрос и получаем результат работы функции `EXTRACT`.

	dates	123 ext_year	123 ext_month	123 ext_day	123 ext_hour	123 ext_minute	123 ext_second
1	2024-01-15 14:53:36.000	2024	1	15	14	53	36
2	2024-01-30 18:31:05.000	2024	1	30	18	31	5

Функция AGE

Функция `AGE` используется для вычисления разницы между двумя датами и временем. Она возвращает результат в виде интервала.

Синтаксис:

- `column_name_1` и `column_name_2` – имена столбцов, в которых хранятся значения дат, между которыми нужно вычислить разницу;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

```
SELECT (column_name_1, column_name_2)  
FROM shema.table_name;
```

Практический пример: необходимо вычислить разницу между датами 18.01.2024 и 10.01.2024 при помощи функции AGE.

```
SELECT AGE('18.01.2024'::date, '10.01.2024'::date);
```

Выполняем запрос и получаем результат работы функции AGE.

Таблица		age	
	1	8 days	
экст	Обновить Save		

Символьные/строчные функции

Получить дополнительную информацию по символьным/строчным функциям можно на страницах [официального руководства](#) PostgreSQL, так как в этой главе будут рассмотрены только некоторые функции.

Функция ASCII

Функция `ASCII` возвращает числовое представление символа.

Синтаксис:


- `value` – символ, для которого будет выведен числовой код;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.








```
SELECT ASCII(value)
FROM schema.table_name;
```

Практический пример: необходимо вывести числовой код символов А, В, С, D, Е и F при помощи функции `ASCII`.

```
SELECT ASCII('A') AS A,
       ASCII('B') AS B,
       ASCII('C') AS C,
       ASCII('D') AS D,
       ASCII('E') AS E,
       ASCII('F') AS F;
```

Выполняем запрос и получаем результат работы функции `ASCII`.

Таблица	 123 a	123 b	123 c	123 d	123 e	123 f	
	1	65	66	67	68	69	70

кст  Обновить Save Cancel      

Функция CHR

Функция `CHR` возвращает символ на основании числового кода. Данная функция противоположна функции [ASCII](#).

Синтаксис:


- `value` – числовой код символа;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.












```
SELECT CHR(value)
FROM schema.table_name;
```

Практический пример: необходимо преобразовать числовой набор 65, 66, 67, 68, 69 и 70 в символы при помощи функции `CHR`.

```
SELECT CHR(65) AS sym_65,
       CHR(66) AS sym_66,
       CHR(67) AS sym_67,
       CHR(68) AS sym_68,
       CHR(69) AS sym_69,
       CHR(70) AS sym_70;
```

Выполняем запрос и получаем результат работы функции `CHR`.

Таблица	 ABC sym_65	ABC sym_66	ABC sym_67	ABC sym_68	ABC sym_69	ABC sym_70	
	1	A	B	C	D	E	F

кст  Обновить  Save  Cancel        Экспорт данных ... 

Функция LENGTH

Функция `LENGTH` возвращает длину строки.

Синтаксис:

- `value` – статическое значение или имя столбца, для которого нужно вернуть длину строки;
- `schema` – наименование схемы, в которой находится объект;

- `table_name` – имя таблицы.

```
SELECT LENGTH(value)
FROM schema.table_name;
```

Практический пример: есть SQL-запрос, он возвращает список имен сотрудников в столбце `full_name`. Необходимо вычислить длину каждой строки в столбце `full_name` при помощи функции `LENGTH`.

```
SELECT t.full_name,
       length(t.full_name) AS result_length
FROM (
  SELECT 'Александра Зайцева' AS full_name
  UNION ALL
  SELECT 'Анна Волкова' AS full_name
  UNION ALL
  SELECT 'Фёдор Ким' AS full_name
) t;
```

Выполняем запрос и получаем результат работы функции `LENGTH`.

Таблица	ABC full_name		123 result_length	
	1	Александра Зайцева		18
	2	Анна Волкова		12
	3	Фёдор Ким		9

Текст Обновить Save Cancel

Функции LOWER / UPPER / INITCAP

Функции `LOWER` / `UPPER` / `INITCAP` позволяют выполнять манипуляции с регистром строк:

- `INITCAP` (возводит первый символ каждого слова в верхний регистр);
- `LOWER` (приводит все символы в нижний регистр);
- `UPPER` (возводит все символы в верхний регистр).

Синтаксис:

- `value` – статическое значение или имя столбца, для которого нужно изменить регистр;
- `schema` – наименование схемы, в которой находится объект;

- `table_name` – имя таблицы.

```
-- Функция INITCAP
SELECT INITCAP(value)
FROM schema.table_name;
```

```
-- Функция LOWER
SELECT LOWER(value)
FROM schema.table_name;
```

```
-- Функция UPPER
SELECT UPPER(value)
FROM schema.table_name;
```

Практический пример: есть SQL-запрос, он возвращает список имен сотрудников в столбце `full_name`. Необходимо выполнить преобразование значений столбца `full_name` тремя разными способами при помощи функций `INITCAP`, `LOWER` и `UPPER`.

```
SELECT t.full_name,
       INITCAP(t.full_name) AS result_initcap,
       LOWER(t.full_name) AS result_lower,
       UPPER(t.full_name) AS result_upper
FROM (
  SELECT 'КоВАлев И. А.' AS full_name
  UNION ALL
  SELECT 'ПоНОмареВ Б. г.' AS full_name
  UNION ALL
  SELECT 'ЕГоров д. А.' AS full_name
) t;
```

Выполняем запрос и получаем результат работы функций `LOWER`, `UPPER` и `INITCAP`.

	ABC full_name	ABC result_initcap	ABC result_lower	ABC result_upper
1	КоВАлев И. А.	Ковалев И. А.	ковалев и. а.	КОВАЛЕВ И. А.
2	ПоНОмареВ Б. г.	Пономарев Б. Г.	пономарев б. г.	ПОНОМАРЕВ Б. Г.
3	ЕГоров д. А.	Егоров Д. А.	егоров д. а.	ЕГОРОВ Д. А.

Функции TRIM / RTRIM / LTRIM

Выполнить удаление лишних символов можно при помощи следующих функции:

- функция `TRIM` (удаление указанных символов в двух сторон строки);
- функция `RTRIM` (удаление символов с правого края);
- Функция `LTRIM` (удаление символов с левого края).

Синтаксис функции LTRIM:

- `string_1` – строка, которая усекается с левой стороны;
- `trim_string` – строка, которая будет удалена с левой стороны;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

```
SELECT LTRIM(string_1, trim_string)
FROM schema.table_name;
```

Синтаксис функции RTRIM:

- `string_1` – строка, которая усекается с правой стороны;
- `trim_string` – строка, которая будет удалена с правой стороны;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

```
SELECT RTRIM(string_1, trim_string)
FROM schema.table_name;
```

Синтаксис функции TRIM:

- `EADING` – удалит `trim_character` с начала `string_1`;
- `TRAILING` – удалит `trim_character` с конца `string_1`;
- `BOTH` – удалит `trim_character` с начала и с конца `string_1`;
- `trim_string` – строка, которая будет удалена из `string_1`;
- `string_1` – строка для удаления;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

```
SELECT TRIM([EADING|TRAILING|BOTH] trim_string FROM string_1)
```

```
FROM schema.table_name;
```

Практический пример: есть SQL-запрос, он возвращает столбец `string_1`, в котором хранятся текстовые значения. Необходимо обработать столбец `string_1` тремя разными способами, то есть удалить символ (-) при помощи функций `LTRIM`, `RTRIM` и `TRIM`.

```
SELECT t.string_1,  
       LTRIM(t.string_1, '-') AS result_ltrim,  
       RTRIM(t.string_1, '-') AS result_rtrim,  
       TRIM(BOTH '-' FROM t.string_1) AS result_trim  
FROM (  
    SELECT 'Это тестовая строка-' AS string_1  
    UNION ALL  
    SELECT '-Это тестовая строка' AS string_1  
    UNION ALL  
    SELECT '-Это тестовая строка-' AS string_1  
) t;
```

Выполняем запрос и получаем результат работы функций `LTRIM`, `RTRIM` и `TRIM`.

Таблица	ABC string_1	ABC result_ltrim	ABC result_rtrim	ABC result_trim
	1	Это тестовая строка-	Это тестовая строка	Это тестовая строка
	2	-Это тестовая строка	Это тестовая строка	Это тестовая строка
	3	-Это тестовая строка-	Это тестовая строка	Это тестовая строка
Текст	Обновить Save Cancel Экспо			

Функция SUBSTR

Функция `SUBSTR` извлекает подстроку из указанной строки

Синтаксис:

- `string` – строка, в которой осуществляется поиск;
- `start_position` – позиция для начала извлечения подстроки;
- `length` – количество символов для извлечения;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

```
SELECT SUBSTR(string, start_position, length)
```

```
FROM schema.table_name;
```

Практический пример: есть SQL-запрос, он возвращает столбец `string_1`, в котором хранятся текстовые значения. Необходимо выбрать первые 5 символов из каждой строки столбца `string_1` при помощи функции `SUBSTR`.

```
SELECT t.string_1,  
       SUBSTR(t.string_1, 1, 5) AS result_substr  
FROM (  
    SELECT '6861234' AS string_1  
    UNION ALL  
    SELECT '87213453231' AS string_1  
    UNION ALL  
    SELECT '132345854' AS string_1  
) t;
```

Выполняем запрос и получаем результат работы функции `SUBSTR`.

Таблица	ABC string_1		ABC result_substr	
	1	6861234	68612	
	2	87213453231	87213	
	3	132345854	13234	
Текст	Обновить Save Cancel			

Функция REPLACE

Функция `REPLACE` выполняет замену символов в строке.

Синтаксис:

- `string` – строка, в которой будет выполнена замена;
- `string_search` – значение, которое необходимо заменить в строке `string`;
- `string_replace` – значение, которое будет вместо `string_search`;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

```
SELECT REPLACE(string, string_search, string_replace)  
FROM schema.table_name;
```

Практический пример: есть SQL-запрос, он возвращает список имен сотрудников в столбце `full_name`. Необходимо заменить символ пробела на символ (-) в каждой строке столбца `full_name` при помощи функции `REPLACE`.

```
SELECT t.full_name,  
       REPLACE(t.full_name, ' ', '-') AS result_replace  
FROM (  
    SELECT 'Фадеева П. М.' AS full_name  
    UNION ALL  
    SELECT 'Фомин М. М.' AS full_name  
    UNION ALL  
    SELECT 'Абрамов И. Т.' AS full_name  
) t;
```

Выполняем запрос и получаем результат работы функции `REPLACE`.

Таблица	ABC full_name		ABC result_replace	
	1	Фадеева П. М.	Фадеева-П.-М.	
	2	Фомин М. М.	Фомин-М.-М.	
	3	Абрамов И. Т.	Абрамов-И.-Т.	
Текст	Обновить Save Cancel			

Условные выражения

Получить дополнительную информацию по условным выражениям можно на страницах [официального руководства PostgreSQL](#).

Оператор CASE

Оператор `CASE` позволяет выбрать для выполнения одну из нескольких последовательностей команд в зависимости от условия или значения. Существует несколько разновидностей команды `CASE`:

- простая команда `CASE`;
- поисковая команда `CASE`.

Простая команда CASE

Простая команда `CASE` связывает одну или несколько последовательностей команд с соответствующими значениями.

Синтаксис:

- `column_list` – список столбцов;
- `search_value` – имя столбца или статическое значение указанное вручную;
- `compare_value` – значения, которые будут сравниваться с указанным значением или столбцом;
- `result` – значения, которые будут проставлены в столбец при совпадении;
- `else_value` – значение, которое будет проставлено в столбец, в том случае, когда не удалось сравнить указанное значение или столбец со значениями в блоке `WHEN`;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `condition_filter` – условия для фильтрации.

```
SELECT column_list,  
       CASE search_value  
         WHEN compare_value_1 THEN result_1  
         WHEN compare_value_2 THEN result_2  
         ...  
         ELSE else_value  
       END;
```

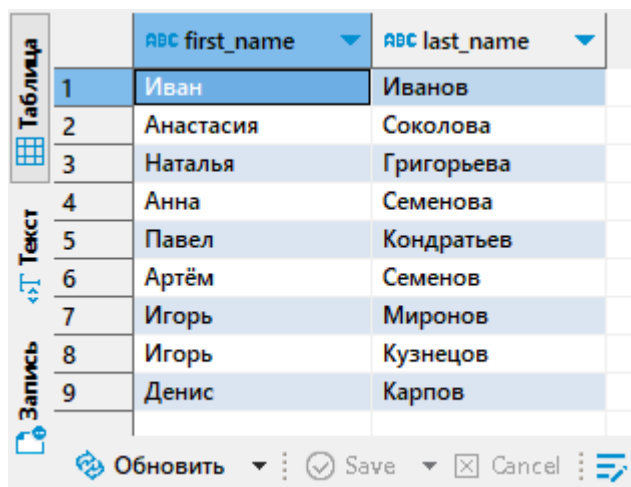
```
FROM schema.table_name  
WHERE condition_filter;
```

Практический пример: из таблицы `employees` необходимо вывести фамилию и имя сотрудников, которые работают в отделе с идентификатором 1. Затем при помощи оператора `CASE` необходимо сравнить имена всех сотрудников отдела с именами Иван и Игорь, если имя совпадает, то в столбец добавить значение 1, если совпадений не найдено то значение 0. Полученному столбцу присвоить псевдоним `emp_chk`.

Сначала напишем запрос, который выведет имя и фамилию всех сотрудников, которые работают в отделе с идентификатором 1.

```
SELECT first_name,  
       last_name  
FROM employees  
WHERE id_department = 1;
```

Результат выполнения запроса.



	ABC first_name	ABC last_name
1	Иван	Иванов
2	Анастасия	Соколова
3	Наталья	Григорьева
4	Анна	Семенова
5	Павел	Кондратьев
6	Артём	Семенов
7	Игорь	Миронов
8	Игорь	Кузнецов
9	Денис	Карпов

Дальше необходимо сравнить полученные имена сотрудников с именами Иван и Игорь при помощи оператора `CASE`.

Пояснение к SQL-запросу:

- в операторе `CASE` указываем имя столбца, его значения будут использоваться для сравнения;

- конструкция `WHEN 'Иван' THEN 1` означает, что если найдено совпадение значения столбца `first_name` и значения в блоке `WHEN`, то в столбец будет проставлено значение 1;
- конструкция `ELSE 0` отвечает за то, что если не будет найдено ни одного совпадения, то в столбец будет проставлено значение 0;
- закрываем оператор `CASE` при помощи ключевого слова `END`, и присваиваем псевдоним `emp_chk` полученной конструкции.

```
SELECT first_name,
       last_name,
       CASE first_name
         WHEN 'Иван' THEN 1
         WHEN 'Игорь' THEN 1
         ELSE 0
       END AS emp_chk
FROM employees
WHERE id_department = 1;
```

Выполняем запрос и получаем результат, в котором есть дополнительный столбец `emp_chk`, и в нем проставлены значения 1 для имён Иван и Игорь, а для других имен значение 0.

	ABC first_name ▼	ABC last_name ▼	123 emp_chk ▼
1	Иван	Иванов	1
2	Анастасия	Соколова	0
3	Наталья	Григорьева	0
4	Анна	Семенова	0
5	Павел	Кондратьев	0
6	Артём	Семенов	0
7	Игорь	Миронов	1
8	Игорь	Кузнецов	1
9	Денис	Карпов	0

Поисковая команда CASE

Поисковая команда `CASE` выбирает для выполнения одну или несколько последовательностей команд в зависимости от результатов проверки списка логических значений.

Синтаксис:

- `column_list` – список столбцов;
- `condition_1` и `condition_2` – это выражение, которое будет проверяться на истинность;
- `result_1` и `result_2` – значение, которое будет проставлено в столбец при истинности указанного выражения;
- `else_value` – значение, которое будет проставлено в столбец, в том случае, когда все указанные выражения в блоках `WHEN` оказались ложными;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `condition_filter` – условия для фильтрации.

```
SELECT column_list,  
       CASE  
         WHEN condition_1 THEN result_1  
         WHEN condition_2 THEN result_2  
         ...  
         ELSE else_value  
       END;  
FROM schema.table_name  
WHERE condition_filter;
```

Практический пример: из таблицы `employees` необходимо вывести имя, фамилию и заработную плату всех сотрудников, которые работают в отделе с идентификатором 1. Затем при помощи оператора `CASE` проверить заработную плату каждого сотрудника и вывести значение в новом столбце `up_salary` в зависимости от условия:

- если заработная плата меньше или равна 50 000 рублей, то вывести текст 'Увеличить ЗП на 15%';
- если заработная плата меньше или равна 70 000 рублей, то вывести текст 'Увеличить ЗП на 5%';
- для остальных случаев, вывести текст 'Не увеличивать ЗП'.

Сначала напишем запрос, который выведет имя, фамилию и заработную плату всех сотрудников, которые работают в отделе с идентификатором 1.

```
SELECT first_name,
```



```

        last_name,
        salary
FROM employees
WHERE id_department = 1
ORDER BY salary ASC;

```

Результат выполнения запроса.

	ABC first_name ▼	ABC last_name ▼	123 salary ▼
1	Иван	Иванов	35 000
2	Анастасия	Соколова	40 000
3	Наталья	Григорьева	60 000
4	Анна	Семенова	66 000
5	Денис	Карпов	68 000
6	Артём	Семенов	70 000
7	Павел	Кондратьев	70 000
8	Игорь	Миронов	72 000
9	Игорь	Кузнецов	75 000

Дальше проверяем заработную плату каждого сотрудника при помощи оператора CASE и выводим значение в дополнительном столбце up_salary в зависимости от условия.

Пояснение к SQL-запросу:

- открываем оператор CASE;
- конструкция WHEN salary <= 50000 THEN 'Увеличить ЗП на 15%' означает, что если заработная плата сотрудника меньше или равна 50 000 рублей, то в столбец up_salary будет проставлено значение 'Увеличить ЗП на 15%'. Аналогичным образом работает следующее условие в блоке WHEN для проверки заработной платы сотрудника;
- конструкция ELSE 'Не увеличивать ЗП' отвечает за то, что если не будет выполнено ни одно условие из блока WHEN, то в столбец up_salary будет проставлено значение 'Не увеличивать ЗП';
- закрываем оператор CASE при помощи ключевого слова END, и присваиваем псевдоним up_salary полученной конструкции.

```

SELECT first_name,
       last_name,
       salary,

```

```

CASE
    WHEN salary <= 50000 THEN 'Увеличить ЗП на 15%'
    WHEN salary <= 70000 THEN 'Увеличить ЗП на 5%'
    ELSE 'Не увеличивать ЗП'
END AS up_salary
FROM employees
WHERE id_department = 1
ORDER BY salary ASC;

```

Выполняем запрос и получаем результат, в котором напротив каждого сотрудника в столбце up_salary проставлено значение в зависимости от условия в операторе CASE.

	ABC first_name	ABC last_name	123 salary	ABC up_salary
1	Иван	Иванов	35 000	Увеличить ЗП на 15%
2	Анастасия	Соколова	40 000	Увеличить ЗП на 15%
3	Наталья	Григорьева	60 000	Увеличить ЗП на 5%
4	Анна	Семенова	66 000	Увеличить ЗП на 5%
5	Денис	Карпов	68 000	Увеличить ЗП на 5%
6	Артём	Семенов	70 000	Увеличить ЗП на 5%
7	Павел	Кондратьев	70 000	Увеличить ЗП на 5%
8	Игорь	Миронов	72 000	Не увеличивать ЗП
9	Игорь	Кузнецов	75 000	Не увеличивать ЗП

Функция COALESCE

Функция COALESCE возвращает первое ненулевое выражение из списка. Если все выражения определены как NULL, то функция COALESCE вернет NULL.

Синтаксис:

- value – статическое значение или имя столбца;
- schema – наименование схемы, в которой находится объект;
- table_name – имя таблицы.

```

SELECT COALESCE(value_1, value_2, ..., value_n)
FROM schema.table_name;

```

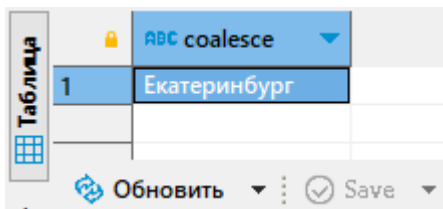
Практический пример: есть 4 значения NULL, NULL, Екатеринбург и Москва, их необходимо проверить функцией COALESCE и вернуть первое не нулевое значение.

```

SELECT COALESCE(NULL, NULL, 'Екатеринбург', 'Москва');

```

Выполняем запрос и получаем результат, в котором получено первое не нулевое значение. Другими словами, функция `COALESCE` проверила 1 значение оно равно `NULL`, функция пошла проверять второе значение, оно тоже равно `NULL`, функция пошла проверять 3 значение, и оно уже не `NULL`. Поэтому функция сразу вернула 3 значение.



Функция NULLIF

Функция `NULLIF` сравнивает значение_1 и значение_2. Если значение_1 и значение_2 равны, то функция возвращает `NULL`. В противном случае, она возвращает значение_1.

Примечание к работе функции:

- функция возвращает `NULL`, если значение_1 и значение_2 равны;
- функция возвращает значение_1, если значение_1 и значение_2 не равны.

Синтаксис:

- `value_1` и `value_2` – статическое значение или имя столбца. Значения должны быть одного типа данных;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

```
SELECT NULLIF(value_1, value_2)
FROM schema.table_name;
```

Практический пример: есть SQL-запрос, он возвращает два столбца `str_1` и `str_2`. Необходимо сравнить строки столбцов `str_1` и `str_2` при помощи функции `NULLIF`.

```
SELECT t.str_1,
       t.str_2,
       NULLIF(t.str_1, t.str_2) AS result_ifnull
FROM (
  SELECT '15' AS str_1, '15' AS str_2
  UNION ALL
  SELECT NULL AS str_1, NULL AS str_2
```

```

UNION ALL
SELECT '5' AS str_1, NULL AS str_2
) t;

```

Выполняем запрос и получаем результат работы функции `NULLIF`.

Таблица	ABC str_1	ABC str_2	ABC result_ifnull
1	15	15	[NULL]
2	[NULL]	[NULL]	[NULL]
3	5	[NULL]	5

Текст Обновить Save Cancel

Функция `GREATEST`

Функция `GREATEST` возвращает наибольшее значение в списке выражений.

Синтаксис:

- `value` – статическое значение или имя столбца;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

```

SELECT GREATEST(value_1, value_2, ..., value_n)
FROM schema.table_name;

```

Практический пример: есть числовая последовательность 1, 50, 10, 47, 90 и 17, при помощи функции `GREATEST` нужно найти наибольшее значение в этой последовательности.

```

SELECT GREATEST(1, 50, 10, 47, 90, 17);

```

Выполняем запрос и получаем результат работы функции `GREATEST`.

Таблица	123 greatest
1	90

Текст Обновить Save

Функция LEAST

Функция `LEAST` возвращает наименьшее значение в списке выражений.

Синтаксис:

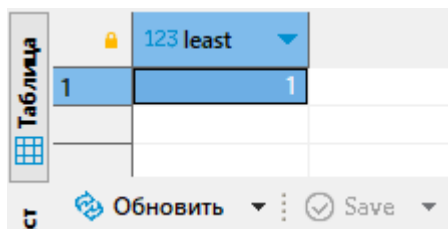
- `value` – статическое значение или имя столбца;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы.

```
SELECT LEAST(value_1, value_2, ..., value_n)
FROM schema.table_name;
```

Практический пример: есть числовая последовательность 1, 50, 10, 47, 90 и 17, при помощи функции `LEAST` нужно найти наименьшее значение в этой последовательности.

```
SELECT LEAST(1, 50, 10, 47, 90, 17);
```

Выполняем запрос и получаем результат работы функции `LEAST`.



least
1

Оконные функции

Что такое оконные функции?

Оконные функции — это очень мощный инструмент, с помощью которого можно решать большое количество задач, благодаря тому что их можно применять в таких блоках, как: [SELECT](#), [WHERE](#), [GROUP BY](#), [HAVING](#) или [ORDER BY](#)

Оконные функции позволяют выполнить ряд вычислений над определенным набором строк, которые объединены по какому-то конкретному признаку или признакам, как например: идентификатор клиента. Сразу можно предположить, что оконные функции выполняют такие же функции, как и `GROUP BY`, но нет, это не так.

Оконные функции значительно отличаются от `GROUP BY` и выполняют совершенно другие функции. Чтобы понять, чем же всё-таки отличаются оконные функции от `GROUP BY` давайте посмотрим несколько отличий:

- оконные функции не уменьшают количество строк в финальном результате, и возвращают столько же значений, сколько получили на входе;
- в отличие от `GROUP BY`, оконные функции могут обращаться к другим строкам;
- оконные функции могут посчитать кумулятивные суммы и скользящие средние значение.

После того, как мы немного разобрались с отличием оконных функций от `GROUP BY`, у вас может возникнуть закономерный вопрос «Что значит оконные?».

При обычном `SELECT` запросе, все множество строк обрабатывается одним цельным фрагментом, для которого в дальнейшем рассчитываются агрегатные значения. А когда используются оконные функции, то `SELECT` запрос разделяется на части (окна) и уже для каждой из отдельных частей считаются свои агрегатные значения.

На рисунке ниже продемонстрировано отличие обычного `SELECT` запроса от запроса, который использует оконные функции.

Обычный запрос	Запрос с оконной функцией

Синтаксис оконных функций

При использовании оконных функций данные запроса разбираются на определенные окна (части), в которых вычисляются агрегатные значения.

Определение оконной функции начинается с указания обязательной конструкции `OVER`. Сначала разберёмся с тем, как выглядит общий синтаксис, а затем подробно разберём каждую часть команды.

Общий синтаксис:

- `name_function` – имя оконной функции, которая будет применяться к окну;
- `column_name` – имя столбца, значение которого будет передано в `name_function`;
- `column_group_list` – список столбцов для группировки окна;
- `column_order_list` – список столбцов для сортировки в окне;
- `condition_filter_group` – выражение для ограничения строк в пределах окна;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `condition_filter` – условия для фильтрации.

```
SELECT name_function(column_name)
      OVER (PARTITION BY column_group_list
            ORDER BY column_order_list
            ROWS или RANGE condition_filter_group)
```

```
FROM schema.table_name
WHERE condition_filter;
```

OVER()

При помощи конструкции `OVER()` определяется оконная функция. Внутри конструкции `OVER()` задаются параметры для группировки и сортировки данных, другими словами, мы определяем правила создания окон (частей).



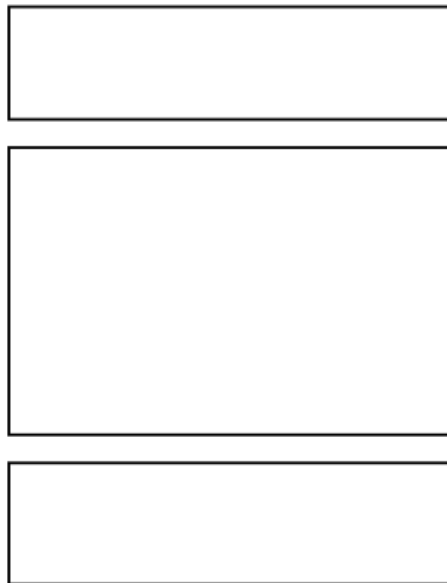
Когда используется конструкция `OVER()` без дополнительных параметров, система автоматически создаёт одно окно для всего набора данных, и к этому окну не применяется сортировка.

На рисунке ниже показаны отличия конструкции `OVER()` без параметров от конструкции `OVER()` с параметрами.

OVER() без параметров



OVER() с параметрами



Практический пример: для закрепления знаний напомним простой запрос, который будет возвращать сквозную сумму столбца `salary` из таблицы `employees`.

Для начала выведем нужные столбцы из таблицы `employees`:

```
SELECT e.id,
       e.last_name,
```



```
        e.first_name,  
        e.gender,  
        e.birthday,  
        e.email,  
        e.id_department,  
        e.salary  
FROM employees e;
```

Сейчас необходимо подсчитать сквозную сумму для столбца `salary`, то есть общую сумму всех значений столбца `salary`.

Пояснение к SQL-запросу:

- добавляем функцию `SUM()` в запрос, внутри неё указываем имя столбца `salary`;
- далее идёт ключевое слово `OVER()` без параметров;
- присваиваем полученной конструкции псевдоним `total_salary`.

```
SELECT e.id,  
       e.last_name,  
       e.first_name,  
       e.gender,  
       e.birthday,  
       e.email,  
       e.id_department,  
       e.salary,  
       sum(salary) OVER() AS total_salary  
FROM employees e;
```

Выполняем запрос и получаем дополнительный столбец `total_salary`, в котором отображается одно значение для всех записей. Это и есть сквозная сумма столбца `salary`.

	123 id	ABC last_name	ABC first_name	ABC gender	birthday	ABC email	123 id_department	123 salary	123 total_salary
1	1	Иванов	Иван	Мужской	1990-05-15	ivanov@mail.ru	1	35 000	4 050 900
2	2	Петров	Петр	Мужской	1985-12-20	petrov@yandex.ru	2	45 000	4 050 900
3	3	Сидорова	Мария	Женский	1992-08-10	sidorova@gmail.com	3	54 000	4 050 900
4	4	Петров	Петр	Мужской	1987-04-25	[NULL]	4	100 000	4 050 900
5	5	Васильева	Екатерина	Женский	1995-03-30	vasilieva@yandex.ru	5	38 900	4 050 900
6	6	Попов	Дмитрий	Мужской	1988-11-05	popov@gmail.com	6	110 000	4 050 900
7	7	Соколова	Анастасия	Женский	1991-07-12	sokolova@mail.ru	1	40 000	4 050 900
8	8	Лебедева	Алена	Женский	1994-09-18	lebedeva@yandex.ru	2	68 000	4 050 900
9	9	Козлов	Артём	Мужской	1986-10-22	kozlov@gmail.com	3	120 000	4 050 900
10	10	Новиков	Андрей	Мужской	1989-06-08	novikov@mail.ru	4	42 000	4 050 900
11	11	Морозова	Ольга	Женский	1993-02-14	morozova@yandex.ru	5	70 000	4 050 900
12	12	Павлов	Сергей	Мужской	1987-12-11	pavlov@gmail.com	6	130 000	4 050 900
13	13	Григорьева	Наталья	Женский	1996-01-25	grigoryeva@mail.ru	1	60 000	4 050 900
14	14	Борисов	Игорь	Мужской	1985-05-20	borisov@yandex.ru	2	95 000	4 050 900
15	15	Максимова	Вероника	Женский	1990-08-30	maximova@gmail.com	3	72 000	4 050 900
16	16	Кучмина	Елена	Женский	1997-08-15	kuzmina@mail.ru	4	105 000	4 050 900

PARTITION BY

`PARTITION BY` это основной параметр, который определяет, как будет происходить разделение данных на окна. Другими словами, параметр `PARTITION BY` позволяет задать столбец или список столбцов, по которым будет выполняться группировка данных.

Практический пример: чтобы понять, как работает параметр `PARTITION BY` возьмем запрос, который был написан при рассмотрении конструкции [OVER\(\)](#), и немного его модифицируем.

Пояснение к SQL-запросу:

- в блоке `WHERE` добавляем условие, что нужны отделы с идентификаторами 1, 2 и 3. Это нужно для того, чтобы сократить выборку из таблицы `employees`;
- в конструкцию `OVER()` добавляем параметр `PARTITION BY`, и указываем имя столбца `id_department`, по нему будет выполнена группировка данных (разбиение на окна).

```
SELECT e.id,
       e.last_name,
       e.first_name,
       e.gender,
       e.birthday,
       e.email,
       e.id_department,
       e.salary,
       sum(salary) OVER(PARTITION BY e.id_department) AS total_salary
FROM employees e
WHERE e.id_department IN (1, 2, 3);
```

Выполняем запрос и получаем результат, в котором данные таблицы `employees` были сгруппированы по столбцу `id_department`, и для каждой группы была рассчитана сквозная сумма столбца `salary`.

Список полученных групп:

- красным цветом выделена группа, которая относится к отделу с идентификатором 1;
- оранжевым цветом выделена группа, которая относится к отделу с идентификатором 2;
- зелёным цветом выделена группа, которая относится к отделу с идентификатором 3.

	123 id	abc last_name	abc first_name	abc gender	birthday	abc email	123 id_department	123 salary	123 total_salary
1	19	Семенова	Анна	Женский	1992-02-14	semenova@mail.ru	1	66 000	556 000
2	1	Иванов	Иван	Мужской	1990-05-15	ivanov@mail.ru	1	35 000	556 000
3	37	Миронов	Игорь	Мужской	1988-07-15	mironov@mail.ru	1	72 000	556 000
4	25	Кондратьев	Павел	Мужской	1990-12-14	kpav@mail.ru	1	70 000	556 000
5	49	Карпов	Денис	Мужской	1995-01-20	karpov@gmail.com	1	68 000	556 000
6	31	Семенов	Артём	Мужской	1987-08-12	semenov@mail.ru	1	70 000	556 000
7	13	Григорьева	Наталья	Женский	1996-01-25	grigoryeva@mail.ru	1	60 000	556 000
8	7	Соколова	Анастасия	Женский	1991-07-12	sokolova@mail.ru	1	40 000	556 000
9	43	Кузнецов	Игорь	Мужской	1985-11-20	[NULL]	1	75 000	556 000
10	50	Максимов	Максим	Мужской	1988-07-28	maximov@mail.ru	2	102 000	702 000
11	2	Петров	Петр	Мужской	1985-12-20	petrov@yandex.ru	2	45 000	702 000
12	8	Лебедева	Алена	Женский	1994-09-18	lebedeva@yandex.ru	2	68 000	702 000
13	14	Борисов	Игорь	Мужской	1985-05-20	borisov@yandex.ru	2	95 000	702 000
14	20	Федоров	Егор	Мужской	1993-09-18	f.eg@yandex.ru	2	72 000	702 000
15	26	Мельникова	Яна	Женский	1993-07-15	melnikova@yandex.ru	2	80 000	702 000
16	32	Карпова	Анна	Женский	1990-06-18	karp@yandex.ru	2	75 000	702 000
17	38	Ковалева	Елена	Женский	1995-12-30	kovaleva@yandex.ru	2	80 000	702 000
18	44	Соколов	Михаил	Мужской	1989-08-30	sokolov@yandex.ru	2	85 000	702 000
19	21	Григорьев	Михаил	Мужской	1988-10-25	grigoryev@gmail.com	3	125 000	779 000
20	9	Козлов	Артём	Мужской	1986-10-22	kozlov@gmail.com	3	120 000	779 000
21	39	Кудряшова	Анастасия	Женский	1990-09-18	kudryashova@gmail.com	3	100 000	779 000
22	15	Максимова	Вероника	Женский	1990-08-30	maximova@gmail.com	3	72 000	779 000
23	3	Сидорова	Мария	Женский	1992-08-10	sidорова@gmail.com	3	54 000	779 000
24	27	Афанасьев	Григорий	Мужской	1995-01-08	afanasyev@gmail.com	3	110 000	779 000
25	45	Пономарева	Максим	Женский	1984-05-28	ponomareva@gmail.com	3	108 000	779 000
26	33	Фомин	Дмитрий	Мужской	1994-02-14	fomin@gmail.com	3	90 000	779 000

ORDER BY

Параметр `ORDER BY` позволяет осуществить сортировку значений внутри окна. По своей сути параметр `ORDER BY` очень простой, но в оконных функциях у него есть одна особенность, и заключается она в том, что, когда используется параметр `ORDER BY` с разными функциями, на выходе будут получены совершенно разные результаты.



При использовании параметра `ORDER BY` с функцией `SUM()` в результате будет получен нарастающий итог (сумма значений каждого поля с предыдущим значением).

При использовании параметра `ORDER BY` с функцией `ROW_NUMBER()` в результате будет получен отсортированный список по нужному полю.

Практический пример: для рассмотрения работы параметра `ORDER BY` возьмем запрос, который был написан при рассмотрении параметра [`PARTITION BY`](#), и внесем в него изменения.

Пояснение к SQL-запросу:

- внутри конструкции `OVER()`, после основного параметра `PARTITION BY` добавим ключевое слово `ORDER BY`, а затем указываем столбец `salary` и направление сортировки `ASC`. Если говорить другими словами, то внутри каждого окна будет выполнена сортировка по столбцу `salary` в порядке возрастания.

```
SELECT e.id,  
       e.last_name,  
       e.first_name,  
       e.gender,  
       e.birthday,  
       e.email,  
       e.id_department,  
       e.salary,  
       sum(salary) OVER(PARTITION BY e.id_department ORDER BY salary  
ASC) AS total_salary  
FROM employees e  
WHERE e.id_department IN (1, 2, 3);
```

Выполняем запрос и получаем результат, в котором видно, что в каждой группе была выполнена сортировка по столбцу `salary` в порядке возрастания, также обратите внимание на столбец `total_salary` в нем был получен нарастающий итог.

	123 id	abc last_name	abc first_name	abc gender	birthday	abc email	123 id_department	123 salary	123 total_salary
1	1	Иванов	Иван	Мужской	1990-05-15	ivanov@mail.ru	1	35 000	35 000
2	7	Соколова	Анастасия	Женский	1991-07-12	sokolova@mail.ru	1	40 000	75 000
3	13	Григорьева	Наталья	Женский	1996-01-25	grigoryeva@mail.ru	1	60 000	135 000
4	19	Семенова	Анна	Женский	1992-02-14	semenova@mail.ru	1	66 000	201 000
5	49	Карпов	Денис	Мужской	1995-01-20	karpov@gmail.com	1	68 000	269 000
6	25	Кондратьев	Павел	Мужской	1990-12-14	kpav@mail.ru	1	70 000	409 000
7	31	Семенов	Артём	Мужской	1987-08-12	semenov@mail.ru	1	70 000	409 000
8	37	Миронов	Игорь	Мужской	1988-07-15	mironov@mail.ru	1	72 000	481 000
9	43	Кузнецов	Игорь	Мужской	1985-11-20	[NULL]	1	75 000	556 000
10	2	Петров	Петр	Мужской	1985-12-20	petrov@yandex.ru	2	45 000	45 000
11	8	Лебедева	Алена	Женский	1994-09-18	lebedeva@yandex.ru	2	68 000	113 000
12	20	Федоров	Егор	Мужской	1993-09-18	f.eg@yandex.ru	2	72 000	185 000
13	32	Карпова	Анна	Женский	1990-06-18	karp@yandex.ru	2	75 000	260 000
14	26	Мельникова	Яна	Женский	1993-07-15	melnikova@yandex.ru	2	80 000	420 000
15	38	Ковалева	Елена	Женский	1995-12-30	kovaleva@yandex.ru	2	80 000	420 000
16	44	Соколов	Михаил	Мужской	1989-08-30	sokolov@yandex.ru	2	85 000	505 000
17	14	Борисов	Игорь	Мужской	1985-05-20	borisov@yandex.ru	2	95 000	600 000
18	50	Максимов	Максим	Мужской	1988-07-28	maximov@mail.ru	2	102 000	702 000
19	3	Сидорова	Мария	Женский	1992-08-10	sidorova@gmail.com	3	54 000	54 000
20	15	Максимова	Вероника	Женский	1990-08-30	maximova@gmail.com	3	72 000	126 000
21	33	Фомин	Дмитрий	Мужской	1994-02-14	fomin@gmail.com	3	90 000	216 000
22	39	Кудряшова	Анастасия	Женский	1990-09-18	kudryashova@gmail.com	3	100 000	316 000
23	45	Пономарева	Максим	Женский	1984-05-28	ponomareva@gmail.com	3	108 000	424 000
24	27	Афанасьев	Григорий	Мужской	1995-01-08	afanasyev@gmail.com	3	110 000	534 000
25	9	Козлов	Артём	Мужской	1986-10-22	kozlov@gmail.com	3	120 000	654 000
26	21	Григорьев	Михаил	Мужской	1988-10-25	grigoryev@gmail.com	3	125 000	779 000

ROWS и RANGE

Параметр `ROWS` позволяет осуществить ограничение строк в окне, указывая фиксированное количество строк, предшествующих или следующих за текущей.

Параметр `RANGE`, в отличие от параметра `ROWS`, работает не со строками, а с диапазоном строк в параметре `ORDER BY`. То есть под понятием одной строки для параметра `RANGE` может пониматься несколько физических строк, которые одинаковы по рангу.

Параметры `ROWS` и `RANGE` всегда используются совместно с параметром `ORDER BY`.

Для ограничения строк в параметрах `ROWS` и `RANGE` можно указывать дополнительные ключевые слова:

- `UNBOUNDED PRECEDING` – указывает, что окно начинается с первой строки группы;
- `UNBOUNDED FOLLOWING` – указывает, что окно заканчивается на последней строке группы;
- `CURRENT ROW` – указывает, что окно начинается или заканчивается на текущей строке;
- `BETWEEN "граница_окна" AND "граница_окна"` – задаётся нижняя и верхняя граница окна;
- `"значение" PRECEDING` – позволяет установить количество строк перед текущей строкой (не допускается использование в параметре `RANGE`);
- `"значение" FOLLOWING` – позволяет установить количество строк после текущей строки (не допускается использование в параметре `RANGE`).

Виды оконных функций

Существует большое количество оконных функций, и рассматривать мы будем только самые основные. Для начала разобьём оконные функции на следующие группы:

- агрегатные функции;
- ранжирующие функции;
- функции смещения;
- аналитические функции.

Агрегатные функции

Агрегатные функции — это функции, которые выполняют арифметические операции над набором данных и возвращают итоговое значение.

Функция SUM()

Функция `SUM()` возвращает сумму всех значений в указанном столбце.



При использовании параметра `ORDER BY` с функцией `SUM()` в результате будет получен нарастающий итог (сумма значений каждого поля с предыдущим значением).

Синтаксис:

- `column_name` — имя столбца, значение которого будет передано в функцию;
- `column_group_list` — список столбцов для группировки окна;
- `column_order_list` — список столбцов для сортировки в окне;
- `schema` — наименование схемы, в которой находится объект;
- `table_name` — имя таблицы;
- `condition_filter` — условия для фильтрации.

```
SELECT SUM(column_name)
       OVER (PARTITION BY column_group_list
             ORDER BY column_order_list)
FROM schema.table_name
WHERE condition_filter;
```

Практический пример: при помощи оконной функции необходимо выполнить группировку данных таблицы `employees` по идентификатору отдела, а затем подсчитать сумму значений столбца `salary` в каждой полученной группе при помощи функции `SUM()`.

Пояснение к SQL-запросу:

- в блоке `SELECT` выводим все необходимые столбцы таблицы `employees`;
- дополнительным столбцом указываем агрегатную функцию `SUM()` и в качестве аргумента передаем ей столбец `salary`;
- при помощи конструкции `OVER()` определяем оконную функцию, внутри неё задаём правило `PARTITION BY` для формирования окон. В данном случае окна будут формироваться по идентификатору отдела;
- полученной конструкции присваиваем псевдоним `sum_salary`;
- в блоке `WHERE` добавляем условие, что нужны отделы с идентификаторами 1 и 2. Это нужно для того, чтобы сократить выборку из таблицы `employees`.

```
SELECT e.id,  
       e.last_name,  
       e.first_name,  
       e.gender,  
       e.birthday,  
       e.email,  
       e.id_department,  
       e.salary,  
       sum(e.salary) OVER(PARTITION BY e.id_department) AS sum_salary  
FROM employees e  
WHERE e.id_department IN (1, 2);
```

Выполняем запрос и получаем результат, в котором данные таблицы `employees` были сгруппированы по столбцу `id_department`, и для каждой группы была рассчитана сквозная сумма столбца `salary`.

	123 id	abc last_name	abc first_name	abc gender	birthday	abc email	123 id_department	123 salary	123 sum_salary
1	7	Соколова	Анастасия	Женский	1991-07-12	sokolova@mail.ru	1	40 000	556 000
2	31	Семенов	Артём	Мужской	1987-08-12	semenov@mail.ru	1	70 000	556 000
3	19	Семенова	Анна	Женский	1992-02-14	semenova@mail.ru	1	66 000	556 000
4	37	Миронов	Игорь	Мужской	1988-07-15	mironov@mail.ru	1	72 000	556 000
5	13	Григорьева	Наталья	Женский	1996-01-25	grigoryeva@mail.ru	1	60 000	556 000
6	43	Кузнецов	Игорь	Мужской	1985-11-20	[NULL]	1	75 000	556 000
7	25	Кондратьев	Павел	Мужской	1990-12-14	kpav@mail.ru	1	70 000	556 000
8	49	Карпов	Денис	Мужской	1995-01-20	karpov@gmail.com	1	68 000	556 000
9	1	Иванов	Иван	Мужской	1990-05-15	ivanov@mail.ru	1	35 000	556 000
10	50	Максимов	Максим	Мужской	1988-07-28	maximov@mail.ru	2	102 000	702 000
11	2	Петров	Петр	Мужской	1985-12-20	petrov@yandex.ru	2	45 000	702 000
12	8	Лебедева	Алена	Женский	1994-09-18	lebedeva@yandex.ru	2	68 000	702 000
13	14	Борисов	Игорь	Мужской	1985-05-20	borisov@yandex.ru	2	95 000	702 000
14	20	Федоров	Егор	Мужской	1993-09-18	f.eg@yandex.ru	2	72 000	702 000
15	26	Мельникова	Яна	Женский	1993-07-15	melnikova@yandex.ru	2	80 000	702 000
16	32	Карпова	Анна	Женский	1990-06-18	karp@yandex.ru	2	75 000	702 000
17	38	Ковалева	Елена	Женский	1995-12-30	kovaleva@yandex.ru	2	80 000	702 000
18	44	Соколов	Михаил	Мужской	1989-08-30	sokolov@yandex.ru	2	85 000	702 000

Функция COUNT()

Функция COUNT () позволяет вычислить количество значений в столбце (значения NULL не учитываются).

Синтаксис:

- column_name – имя столбца, значение которого будет передано в функцию;
- column_group_list – список столбцов для группировки окна;
- column_order_list – список столбцов для сортировки в окне;
- schema – наименование схемы, в которой находится объект;
- table_name – имя таблицы;
- condition_filter – условия для фильтрации.

```
SELECT COUNT(column_name)
      OVER (PARTITION BY column_group_list
            ORDER BY column_order_list)
FROM schema.table_name
WHERE condition_filter;
```

Практический пример: при помощи оконной функции необходимо выполнить группировку данных таблицы employees по идентификатору отдела, а затем определить количество строк в столбце salary в каждой полученной группе при помощи функции COUNT () .

Пояснение к SQL-запросу:

- в блоке SELECT выводим все необходимые столбцы таблицы employees;
- дополнительным столбцом указываем агрегатную функцию COUNT () и в качестве аргумента передаем ей столбец salary;

- при помощи конструкции `OVER()` определяем оконную функцию, внутри неё задаём правило `PARTITION BY` для формирования окон. В данном случае окна будут формироваться по идентификатору отдела;
- полученной конструкции присваиваем псевдоним `cnt_salary`;
- в блоке `WHERE` добавляем условие, что нужны отделы с идентификаторами 1 и 2. Это нужно для того, чтобы сократить выборку из таблицы `employees`.

```
SELECT e.id,
       e.last_name,
       e.first_name,
       e.gender,
       e.birthday,
       e.email,
       e.id_department,
       e.salary,
       count(e.salary) OVER(PARTITION BY e.id_department) AS
cnt_salary
FROM employees e
WHERE e.id_department IN (1, 2);
```

Выполняем запрос и получаем результат, в котором данные таблицы `employees` были сгруппированы по столбцу `id_department`, и для каждой группы было посчитано количество строк в столбце `salary`.

	123 id	ABC last_name	ABC first_name	ABC gender	birthday	ABC email	123 id_department	123 salary	123 cnt_salary
1	7	Соколова	Анастасия	Женский	1991-07-12	sokolova@mail.ru	1	40 000	9
2	31	Семенов	Артём	Мужской	1987-08-12	semenov@mail.ru	1	70 000	9
3	19	Семенова	Анна	Женский	1992-02-14	semenova@mail.ru	1	66 000	9
4	37	Миронов	Игорь	Мужской	1988-07-15	miroнов@mail.ru	1	72 000	9
5	13	Григорьева	Наталья	Женский	1996-01-25	grigoryeva@mail.ru	1	60 000	9
6	43	Кузнецов	Игорь	Мужской	1985-11-20	[NULL]	1	75 000	9
7	25	Кондратьев	Павел	Мужской	1990-12-14	kpav@mail.ru	1	70 000	9
8	49	Карпов	Денис	Мужской	1995-01-20	karpov@gmail.com	1	68 000	9
9	1	Иванов	Иван	Мужской	1990-05-15	ivanov@mail.ru	1	35 000	9
10	50	Максимов	Максим	Мужской	1988-07-28	maximov@mail.ru	2	102 000	9
11	2	Петров	Петр	Мужской	1985-12-20	petrov@yandex.ru	2	45 000	9
12	8	Лебедева	Алена	Женский	1994-09-18	lebedeva@yandex.ru	2	68 000	9
13	14	Борисов	Игорь	Мужской	1985-05-20	borisov@yandex.ru	2	95 000	9
14	20	Федоров	Егор	Мужской	1993-09-18	f.eg@yandex.ru	2	72 000	9
15	26	Мельникова	Яна	Женский	1993-07-15	melnikova@yandex.ru	2	80 000	9
16	32	Карпова	Анна	Женский	1990-06-18	karp@yandex.ru	2	75 000	9
17	38	Ковалева	Елена	Женский	1995-12-30	kovaleva@yandex.ru	2	80 000	9
18	44	Соколов	Михаил	Мужской	1989-08-30	sokolov@yandex.ru	2	85 000	9

Функция AVG()

Функция `AVG()` позволяет рассчитать среднее значение для указанного столбца.

Синтаксис:

- `column_name` – имя столбца, значение которого будет передано в функцию;
- `column_group_list` – список столбцов для группировки окна;
- `column_order_list` – список столбцов для сортировки в окне;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `condition_filter` – условия для фильтрации.

```
SELECT AVG(column_name)
      OVER (PARTITION BY column_group_list
            ORDER BY column_order_list)
FROM schema.table_name
WHERE condition_filter;
```

Практический пример: при помощи оконной функции необходимо выполнить группировку данных таблицы `employees` по идентификатору отдела, а затем определить среднее значение столбца `salary` в каждой полученной группе при помощи функции `AVG()`.

Пояснение к SQL-запросу:

- в блоке `SELECT` выводим все необходимые столбцы таблицы `employees`;
- дополнительным столбцом указываем агрегатную функцию `AVG()` и в качестве аргумента передаем ей столбец `salary`;
- при помощи конструкции `OVER()` определяем оконную функцию, внутри неё задаём правило `PARTITION BY` для формирования окон. В данном случае окна будут формироваться по идентификатору отдела;
- полученной конструкции присваиваем псевдоним `avg_salary`;
- в блоке `WHERE` добавляем условие, что нужны отделы с идентификаторами 1 и 2. Это нужно для того, чтобы сократить выборку из таблицы `employees`.

```
SELECT e.id,
       e.last_name,
       e.first_name,
       e.gender,
       e.birthday,
       e.email,
       e.id_department,
       e.salary,
       AVG(e.salary) OVER(PARTITION BY e.id_department) AS avg_salary
```

```
FROM employees e
WHERE e.id_department IN (1, 2);
```

Выполняем запрос и получаем результат, в котором данные таблицы `employees` были сгруппированы по столбцу `id_department`, и для каждой группы было посчитано среднее значение столбца `salary`.

	123 id	ABC last_name	ABC first_name	ABC gender	birthday	ABC email	123 id_department	123 salary	123 avg_salary
1	7	Соколова	Анастасия	Женский	1991-07-12	sokolova@mail.ru	1	40 000	61 777,7777777778
2	31	Семенов	Артём	Мужской	1987-08-12	semenov@mail.ru	1	70 000	61 777,7777777778
3	19	Семенова	Анна	Женский	1992-02-14	semenova@mail.ru	1	66 000	61 777,7777777778
4	37	Миронов	Игорь	Мужской	1988-07-15	mironov@mail.ru	1	72 000	61 777,7777777778
5	13	Григорьева	Наталья	Женский	1996-01-25	grigoryeva@mail.ru	1	60 000	61 777,7777777778
6	43	Кузнецов	Игорь	Мужской	1985-11-20	[NULL]	1	75 000	61 777,7777777778
7	25	Кондратьев	Павел	Мужской	1990-12-14	kpav@mail.ru	1	70 000	61 777,7777777778
8	49	Карпов	Денис	Мужской	1995-01-20	karpov@gmail.com	1	68 000	61 777,7777777778
9	1	Иванов	Иван	Мужской	1990-05-15	ivanov@mail.ru	1	35 000	61 777,7777777778
10	50	Максимов	Максим	Мужской	1988-07-28	maximov@mail.ru	2	102 000	78 000
11	2	Петров	Петр	Мужской	1985-12-20	petrov@yandex.ru	2	45 000	78 000
12	8	Лебедева	Алена	Женский	1994-09-18	lebedeva@yandex.ru	2	68 000	78 000
13	14	Борисов	Игорь	Мужской	1985-05-20	borisov@yandex.ru	2	95 000	78 000
14	20	Федоров	Егор	Мужской	1993-09-18	f.eg@yandex.ru	2	72 000	78 000
15	26	Мельникова	Яна	Женский	1993-07-15	melnikova@yandex.ru	2	80 000	78 000
16	32	Карпова	Анна	Женский	1990-06-18	karpov@yandex.ru	2	75 000	78 000
17	38	Ковалева	Елена	Женский	1995-12-30	kovaleva@yandex.ru	2	80 000	78 000
18	44	Соколов	Михаил	Мужской	1989-08-30	sokolov@yandex.ru	2	85 000	78 000

Функция MAX()

Функция `MAX()` позволяет определить максимальное значение в указанном столбце.

Синтаксис:

- `column_name` – имя столбца, значение которого будет передано в функцию;
- `column_group_list` – список столбцов для группировки окна;
- `column_order_list` – список столбцов для сортировки в окне;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `condition_filter` – условия для фильтрации.

```
SELECT MAX(column_name)
      OVER (PARTITION BY column_group_list
            ORDER BY column_order_list)
FROM schema.table_name
WHERE condition_filter;
```

Практический пример: при помощи оконной функции необходимо выполнить группировку данных таблицы `employees` по идентификатору отдела, а затем определить

максимальное значение столбца `salary` в каждой полученной группе при помощи функции `MAX()`.

Пояснение к SQL-запросу:

- в блоке `SELECT` выводим все необходимые столбцы таблицы `employees`;
- дополнительным столбцом указываем агрегатную функцию `MAX()` и в качестве аргумента передаем ей столбец `salary`;
- при помощи конструкции `OVER()` определяем оконную функцию, внутри неё задаём правило `PARTITION BY` для формирования окон. В данном случае окна будут формироваться по идентификатору отдела;
- полученной конструкции присваиваем псевдоним `max_salary`;
- в блоке `WHERE` добавляем условие, что нужны отделы с идентификаторами 1 и 2. Это нужно для того, чтобы сократить выборку из таблицы `employees`.

```
SELECT e.id,  
       e.last_name,  
       e.first_name,  
       e.gender,  
       e.birthday,  
       e.email,  
       e.id_department,  
       e.salary,  
       MAX(e.salary) OVER(PARTITION BY e.id_department) AS max_salary  
FROM employees e  
WHERE e.id_department IN (1, 2);
```

Выполняем запрос и получаем результат, в котором данные таблицы `employees` были сгруппированы по столбцу `id_department`, и для каждой группы было определено максимальное значение в столбце `salary`.

	123 id	ABC last_name	ABC first_name	ABC gender	birthday	ABC email	123 id_department	123 salary	123 max_salary
1	7	Соколова	Анастасия	Женский	1991-07-12	sokolova@mail.ru	1	40 000	75 000
2	31	Семенов	Артём	Мужской	1987-08-12	semenov@mail.ru	1	70 000	75 000
3	19	Семенова	Анна	Женский	1992-02-14	semenova@mail.ru	1	66 000	75 000
4	37	Миронов	Игорь	Мужской	1988-07-15	mironov@mail.ru	1	72 000	75 000
5	13	Григорьева	Наталья	Женский	1996-01-25	grigoryeva@mail.ru	1	60 000	75 000
6	43	Кузнецов	Игорь	Мужской	1985-11-20	[NULL]	1	75 000	75 000
7	25	Кондратьев	Павел	Мужской	1990-12-14	kpav@mail.ru	1	70 000	75 000
8	49	Карпов	Денис	Мужской	1995-01-20	karpov@gmail.com	1	68 000	75 000
9	1	Иванов	Иван	Мужской	1990-05-15	ivanov@mail.ru	1	35 000	75 000
10	50	Максимов	Максим	Мужской	1988-07-28	maximov@mail.ru	2	102 000	102 000
11	2	Петров	Петр	Мужской	1985-12-20	petrov@yandex.ru	2	45 000	102 000
12	8	Лебедева	Алена	Женский	1994-09-18	lebedeva@yandex.ru	2	68 000	102 000
13	14	Борисов	Игорь	Мужской	1985-05-20	borisov@yandex.ru	2	95 000	102 000
14	20	Федоров	Егор	Мужской	1993-09-18	f.eg@yandex.ru	2	72 000	102 000
15	26	Мельникова	Яна	Женский	1993-07-15	melnikova@yandex.ru	2	80 000	102 000
16	32	Карпова	Анна	Женский	1990-06-18	karp@yandex.ru	2	75 000	102 000
17	38	Ковалева	Елена	Женский	1995-12-30	kovaleva@yandex.ru	2	80 000	102 000
18	44	Сokolov	Михаил	Мужской	1989-08-30	sokolov@yandex.ru	2	85 000	102 000

Функция MIN()

Функция MIN () позволяет определить минимальное значение в указанном столбце.

Синтаксис:

- column_name – имя столбца, значение которого будет передано в функцию;
- column_group_list – список столбцов для группировки окна;
- column_order_list – список столбцов для сортировки в окне;
- schema – наименование схемы, в которой находится объект;
- table_name – имя таблицы;
- condition_filter – условия для фильтрации.

```
SELECT MIN(column_name)
      OVER (PARTITION BY column_group_list
            ORDER BY column_order_list)
FROM schema.table_name
WHERE condition_filter;
```

Практический пример: при помощи оконной функции необходимо выполнить группировку данных таблицы employees по идентификатору отдела, а затем определить минимальное значение столбца salary в каждой полученной группе при помощи функции MIN () .

Пояснение к SQL-запросу:

- в блоке SELECT выводим все необходимые столбцы таблицы employees;
- дополнительным столбцом указываем агрегатную функцию MIN () и в качестве аргумента передаем ей столбец salary;

- при помощи конструкции `OVER()` определяем оконную функцию, внутри неё задаём правило `PARTITION BY` для формирования окон. В данном случае окна будут формироваться по идентификатору отдела;
- полученной конструкции присваиваем псевдоним `min_salary`;
- в блоке `WHERE` добавляем условие, что нужны отделы с идентификаторами 1 и 2. Это нужно для того, чтобы сократить выборку из таблицы `employees`.

```
SELECT e.id,
       e.last_name,
       e.first_name,
       e.gender,
       e.birthday,
       e.email,
       e.id_department,
       e.salary,
       MIN(e.salary) OVER(PARTITION BY e.id_department) AS min_salary
FROM employees e
WHERE e.id_department IN (1, 2);
```

Выполняем запрос и получаем результат, в котором данные таблицы `employees` были сгруппированы по столбцу `id_department`, и для каждой группы было определено минимальное значение в столбце `salary`.

	123 id	ABC last_name	ABC first_name	ABC gender	📅 birthday	ABC email	123 id_department	123 salary	123 min_salary
1	7	Соколова	Анастасия	Женский	1991-07-12	sokolova@mail.ru	1	40 000	35 000
2	31	Семенов	Артём	Мужской	1987-08-12	semenov@mail.ru	1	70 000	35 000
3	19	Семенова	Анна	Женский	1992-02-14	semenova@mail.ru	1	66 000	35 000
4	37	Миронов	Игорь	Мужской	1988-07-15	miroнов@mail.ru	1	72 000	35 000
5	13	Григорьева	Наталья	Женский	1996-01-25	grigoryeva@mail.ru	1	60 000	35 000
6	43	Кузнецов	Игорь	Мужской	1985-11-20	[NULL]	1	75 000	35 000
7	25	Кондратьев	Павел	Мужской	1990-12-14	kpav@mail.ru	1	70 000	35 000
8	49	Карпов	Денис	Мужской	1995-01-20	karpov@gmail.com	1	68 000	35 000
9	1	Иванов	Иван	Мужской	1990-05-15	ivanov@mail.ru	1	35 000	35 000
10	50	Максимов	Максим	Мужской	1988-07-28	maximov@mail.ru	2	102 000	45 000
11	2	Петров	Петр	Мужской	1985-12-20	petrov@yandex.ru	2	45 000	45 000
12	8	Лебедева	Алена	Женский	1994-09-18	lebedeva@yandex.ru	2	68 000	45 000
13	14	Борисов	Игорь	Мужской	1985-05-20	borisov@yandex.ru	2	95 000	45 000
14	20	Федоров	Егор	Мужской	1993-09-18	f.eg@yandex.ru	2	72 000	45 000
15	26	Мельникова	Яна	Женский	1993-07-15	melnikova@yandex.ru	2	80 000	45 000
16	32	Карпова	Анна	Женский	1990-06-18	karp@yandex.ru	2	75 000	45 000
17	38	Ковалева	Елена	Женский	1995-12-30	kovaleva@yandex.ru	2	80 000	45 000
18	44	Соколов	Михаил	Мужской	1989-08-30	sokolov@yandex.ru	2	85 000	45 000

Ранжирующие функции

Ранжирующие функции — это функции, которые осуществляют ранжирование (сортировку) значения для каждой строки внутри окна

Функция ROW_NUMBER()

Функция ROW_NUMBER() осуществляет нумерацию строки внутри окна.

Синтаксис:

- column_group_list – список столбцов для группировки окна;
- column_order_list – список столбцов для сортировки в окне;
- schema – наименование схемы, в которой находится объект;
- table_name – имя таблицы;
- condition_filter – условия для фильтрации.

```
SELECT ROW_NUMBER()  
       OVER (PARTITION BY column_group_list  
            ORDER BY column_order_list)  
FROM schema.table_name  
WHERE condition_filter;
```

Практический пример: при помощи оконной функции необходимо выполнить группировку данных таблицы `employees` по идентификатору отдела, а затем пронумеровать строки в каждой полученной группе при помощи функции ROW_NUMBER(). Сортировка внутри окна должна выполняться по столбцу `birthday` в порядке возрастания.

Пояснение к SQL-запросу:

- в блоке SELECT выводим все необходимые столбцы таблицы `employees`;
- дополнительным столбцом указываем ранжирующую функцию ROW_NUMBER() без аргумента;
- при помощи конструкции OVER() определяем оконную функцию, внутри неё задаём правило PARTITION BY для формирования окон. В данном случае окна будут формироваться по идентификатору отдела, а сортировка окна должна выполняться по столбцу `birthday` в порядке возрастания;
- полученной конструкции присваиваем псевдоним `rn`;
- в блоке WHERE добавляем условие, что нужны отделы с идентификаторами 1 и 2. Это нужно для того, чтобы сократить выборку из таблицы `employees`.

```
SELECT e.id,  
       e.last_name,  
       e.first_name,
```

```

        e.gender,
        e.birthday,
        e.email,
        e.id_department,
        e.salary,

        ROW_NUMBER() OVER(PARTITION BY e.id_department ORDER BY
e.birthday ASC) AS rn
FROM employees e
WHERE e.id_department IN (1, 2);

```

Выполняем запрос и получаем результат, в котором данные таблицы `employees` были сгруппированы по столбцу `id_department` и отсортированы по столбцу `birthday` в порядке возрастания. Номера строк каждой группы пронумерованы и отображаются в столбце `rn`.

	123 id	abc last_name	abc first_name	abc gender	birthday	abc email	123 id_department	123 salary	123 rn
1	43	Кузнецов	Игорь	Мужской	1985-11-20	[NULL]	1	75 000	1
2	31	Семенов	Артём	Мужской	1987-08-12	semenov@mail.ru	1	70 000	2
3	37	Мионов	Игорь	Мужской	1988-07-15	mirovov@mail.ru	1	72 000	3
4	1	Иванов	Иван	Мужской	1990-05-15	ivanov@mail.ru	1	35 000	4
5	25	Кондратьев	Павел	Мужской	1990-12-14	kpav@mail.ru	1	70 000	5
6	7	Соколова	Анастасия	Женский	1991-07-12	sokolova@mail.ru	1	40 000	6
7	19	Семенова	Анна	Женский	1992-02-14	semenova@mail.ru	1	66 000	7
8	49	Карпов	Денис	Мужской	1995-01-20	karpov@gmail.com	1	68 000	8
9	13	Григорьева	Наталья	Женский	1996-01-25	grigoryeva@mail.ru	1	60 000	9
10	14	Борисов	Игорь	Мужской	1985-05-20	borisov@yandex.ru	2	95 000	1
11	2	Петров	Петр	Мужской	1985-12-20	petrov@yandex.ru	2	45 000	2
12	50	Максимов	Максим	Мужской	1988-07-28	maximov@mail.ru	2	102 000	3
13	44	Соколов	Михаил	Мужской	1989-08-30	sokolov@yandex.ru	2	85 000	4
14	32	Карпова	Анна	Женский	1990-06-18	karp@yandex.ru	2	75 000	5
15	26	Мельникова	Яна	Женский	1993-07-15	melnikova@yandex.ru	2	80 000	6
16	20	Федоров	Егор	Мужской	1993-09-18	f.eg@yandex.ru	2	72 000	7
17	8	Лебедева	Алена	Женский	1994-09-18	lebedeva@yandex.ru	2	68 000	8
18	38	Ковалева	Елена	Женский	1995-12-30	kovaleva@yandex.ru	2	80 000	9

Функция RANK()

Функция `RANK()` возвращает номер ранга для каждой строки. Если при анализе данных обнаружены одинаковые значения, то для них проставляется одинаковый ранг, и пропускается следующее значение ранга.

Синтаксис:

- `column_group_list` – список столбцов для группировки окна;
- `column_order_list` – список столбцов для сортировки в окне;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `condition_filter` – условия для фильтрации.


```

SELECT RANK()
       OVER (PARTITION BY column_group_list
             ORDER BY column_order_list)
FROM schema.table_name
WHERE condition_filter;

```

Практический пример: при помощи оконной функции необходимо выполнить группировку данных таблицы `employees` по идентификатору отдела, а затем проставить ранг для каждой строки в каждой полученной группе при помощи функции `RANK()`. Сортировка внутри окна должна выполняться по столбцу `salary` в порядке возрастания.

Пояснение к SQL-запросу:

- в блоке `SELECT` выводим все необходимые столбцы таблицы `employees`;
- дополнительным столбцом указываем ранжирующую функцию `RANK()` без аргумента;
- при помощи конструкции `OVER()` определяем оконную функцию, внутри неё задаём правило `PARTITION BY` для формирования окон. В данном случае окна будут формироваться по идентификатору отдела, а сортировка окна должна выполняться по столбцу `salary` в порядке возрастания;
- полученной конструкции присваиваем псевдоним `rn_rank`;
- в блоке `WHERE` добавляем условие, что нужны отделы с идентификаторами 1 и 2. Это нужно для того, чтобы сократить выборку из таблицы `employees`.

```

SELECT e.id,
       e.last_name,
       e.first_name,
       e.gender,
       e.birthday,
       e.email,
       e.id_department,
       e.salary,
       RANK() OVER(PARTITION BY e.id_department ORDER BY e.salary ASC)
AS rn_rank
FROM employees e
WHERE e.id_department IN (1, 2);

```

Выполняем запрос и получаем результат, в котором данные таблицы `employees` были сгруппированы по столбцу `id_department` и отсортированы по столбцу `salary` в порядке возрастания. Также проставлен ранг для каждой строки в каждой группе, он отображается в столбце `rn_rank`.

Сейчас нужно разобраться, как проставляется ранг для каждой строки в группе, для этого нужно внимательно прочитать пояснение ниже.

Пояснение к полученному результату:

- рассмотрим группу, которая была сформирована по идентификатору отдела с номером 1;
- чтобы проставить ранги для строк, система сравнивает значения в столбце `salary`, если они одинаковые, то для них устанавливается одинаковый ранг и следующее значение ранга пропускается. В данном случае обратим внимание на строки 6 и 7, у них проставлен одинаковый ранг, так как значение заработной платы одинаковое. А следующая строка под номером 8 имеет уже ранг 8, так как ранг под номером 7 был пропущен.

	123 id	ABC last_name	ABC first_name	ABC gender	birthday	ABC email	123 id_department	123 salary	123 rn_rank
1	1	Иванов	Иван	Мужской	1990-05-15	ivanov@mail.ru	1	35 000	1
2	7	Соколова	Анастасия	Женский	1991-07-12	sokolova@mail.ru	1	40 000	2
3	13	Григорьева	Наталья	Женский	1996-01-25	grigoryeva@mail.ru	1	60 000	3
4	19	Семенова	Анна	Женский	1992-02-14	semenova@mail.ru	1	66 000	4
5	49	Карпов	Денис	Мужской	1995-01-20	karpov@gmail.com	1	68 000	5
6	31	Семенов	Артём	Мужской	1987-08-12	semenov@mail.ru	1	70 000	6
7	25	Кондратьев	Павел	Мужской	1990-12-14	kpav@mail.ru	1	70 000	6
8	37	Миронов	Игорь	Мужской	1988-07-15	mironov@mail.ru	1	72 000	8
9	43	Кузнецов	Игорь	Мужской	1985-11-20	[NULL]	1	75 000	9
10	2	Петров	Петр	Мужской	1985-12-20	petrov@yandex.ru	2	45 000	1
11	8	Лебедева	Алена	Женский	1994-09-18	lebedeva@yandex.ru	2	68 000	2
12	20	Федоров	Егор	Мужской	1993-09-18	f.eg@yandex.ru	2	72 000	3
13	32	Карпова	Анна	Женский	1990-06-18	karp@yandex.ru	2	75 000	4
14	26	Мельникова	Яна	Женский	1993-07-15	melnikova@yandex.ru	2	80 000	5
15	38	Ковалева	Елена	Женский	1995-12-30	kovaleva@yandex.ru	2	80 000	5
16	44	Соколов	Михаил	Мужской	1989-08-30	sokolov@yandex.ru	2	85 000	7
17	14	Борисов	Игорь	Мужской	1985-05-20	borisov@yandex.ru	2	95 000	8
18	50	Максимов	Максим	Мужской	1988-07-28	maximov@mail.ru	2	102 000	9

Функция DENSE_RANK()

Функция `DENSE_RANK()` возвращает номер ранга для каждой строки. Если при анализе данных обнаружены одинаковые значения, то для них проставляется одинаковый ранг, и НЕ пропускается следующее значение ранга.

Синтаксис:

- `column_group_list` – список столбцов для группировки окна;
- `column_order_list` – список столбцов для сортировки в окне;

- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `condition_filter` – условия для фильтрации.

```
SELECT DENSE_RANK()
       OVER (PARTITION BY column_group_list
             ORDER BY column_order_list)
FROM schema.table_name
WHERE condition_filter;
```

Практический пример: при помощи оконной функции необходимо выполнить группировку данных таблицы `employees` по идентификатору отдела, а затем проставить ранг для каждой строки в каждой полученной группе при помощи функции `DENSE_RANK()`. Сортировка внутри окна должна выполняться по столбцу `salary` в порядке возрастания.

Пояснение к SQL-запросу:

- в блоке `SELECT` выводим все необходимые столбцы таблицы `employees`;
- дополнительным столбцом указываем ранжирующую функцию `DENSE_RANK()` без аргумента;
- при помощи конструкции `OVER()` определяем оконную функцию, внутри неё задаём правило `PARTITION BY` для формирования окон. В данном случае окна будут формироваться по идентификатору отдела, а сортировка окна должна выполняться по столбцу `salary` в порядке возрастания;
- полученной конструкции присваиваем псевдоним `rn_dense_rank`;
- в блоке `WHERE` добавляем условие, что нужны отделы с идентификаторами 1 и 2. Это нужно для того, чтобы сократить выборку из таблицы `employees`.

```
SELECT e.id,
       e.last_name,
       e.first_name,
       e.gender,
       e.birthday,
       e.email,
       e.id_department,
       e.salary,
       DENSE_RANK() OVER(PARTITION BY e.id_department ORDER BY
e.salary ASC) AS rn_dense_rank
```

```
FROM employees e
WHERE e.id_department IN (1, 2);
```

Выполняем запрос и получаем результат, в котором данные таблицы `employees` были сгруппированы по столбцу `id_department` и отсортированы по столбцу `salary` в порядке возрастания. Также проставлен ранг для каждой строки в каждой группе, он отображается в столбце `rn_dense_rank`.

Сейчас нужно разобраться, как проставляется ранг для каждой строки в группе, для этого нужно внимательно прочитать пояснение ниже.

Пояснение к полученному результату:

- рассмотрим группу, которая была сформирована по идентификатору отдела с номером 1;
- чтобы проставить ранги для строк, система сравнивает значения в столбце `salary`, если они одинаковые, то для них устанавливается одинаковый ранг и следующее значение ранга НЕ пропускается. В данном случае обратим внимание на строки 6 и 7, у них проставлен одинаковый ранг, так как значение заработной платы одинаковое. А следующая строка под номером 8 имеет уже ранг 7, так как ранг под номером 7 НЕ был пропущен.

	123 id	ABC last_name	ABC first_name	ABC gender	birthday	ABC email	123 id_department	123 salary	123 rn_dense_rank
1	1	Иванов	Иван	Мужской	1990-05-15	ivanov@mail.ru	1	35 000	1
2	7	Соколова	Анастасия	Женский	1991-07-12	sokolova@mail.ru	1	40 000	2
3	13	Григорьева	Наталья	Женский	1996-01-25	grigoryeva@mail.ru	1	60 000	3
4	19	Семенова	Анна	Женский	1992-02-14	semenova@mail.ru	1	66 000	4
5	49	Карпов	Денис	Мужской	1995-01-20	karpov@gmail.com	1	68 000	5
6	31	Семенов	Артём	Мужской	1987-08-12	semenov@mail.ru	1	70 000	6
7	25	Кондратьев	Павел	Мужской	1990-12-14	kpav@mail.ru	1	70 000	6
8	37	Миронов	Игорь	Мужской	1988-07-15	mironov@mail.ru	1	72 000	7
9	43	Кузнецов	Игорь	Мужской	1985-11-20	[NULL]	1	75 000	8
10	2	Петров	Петр	Мужской	1985-12-20	petrov@yandex.ru	2	45 000	1
11	8	Лебедева	Алена	Женский	1994-09-18	lebedeva@yandex.ru	2	68 000	2
12	20	Федоров	Егор	Мужской	1993-09-18	f.eg@yandex.ru	2	72 000	3
13	32	Карпова	Анна	Женский	1990-06-18	karp@yandex.ru	2	75 000	4
14	26	Мельникова	Яна	Женский	1993-07-15	melnikova@yandex.ru	2	80 000	5
15	38	Ковалева	Елена	Женский	1995-12-30	kovaleva@yandex.ru	2	80 000	5
16	44	Соколов	Михаил	Мужской	1989-08-30	sokolov@yandex.ru	2	85 000	6
17	14	Борисов	Игорь	Мужской	1985-05-20	borisov@yandex.ru	2	95 000	7
18	50	Максимов	Максим	Мужской	1988-07-28	maximov@mail.ru	2	102 000	8

Функция NTILE()

Функция `NTILE()` позволяет разделить значения текущего окна (группы) на заданное количество групп. Количество групп указывается в скобках.

Синтаксис:

- `count_group` – количество групп;
- `column_group_list` – список столбцов для группировки окна;
- `column_order_list` – список столбцов для сортировки в окне;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `condition_filter` – условия для фильтрации.

```
SELECT NTILE(count_group)
      OVER (PARTITION BY column_group_list
            ORDER BY column_order_list)
FROM schema.table_name
WHERE condition_filter;
```

Практический пример: при помощи оконной функции необходимо выполнить группировку данных таблицы `employees` по идентификатору отдела, а затем разделить каждое окно на 3 части при помощи функции `NTILE()`. Сортировка внутри окна должна выполняться по столбцу `salary` в порядке возрастания.

Пояснение к SQL-запросу:

- в блоке `SELECT` выводим все необходимые столбцы таблицы `employees`;
- дополнительным столбцом указываем ранжирующую функцию `NTILE()` и в качестве аргумента передаем ей значение 3, оно укажет функции на сколько групп нужно разделить текущее окно;
- при помощи конструкции `OVER()` определяем оконную функцию, внутри неё задаём правило `PARTITION BY` для формирования окон. В данном случае окна будут формироваться по идентификатору отдела, а сортировка окна должна выполняться по столбцу `salary` в порядке возрастания;
- полученной конструкции присваиваем псевдоним `grp_ntile`;
- в блоке `WHERE` добавляем условие, что нужны отделы с идентификаторами 1 и 2. Это нужно для того, чтобы сократить выборку из таблицы `employees`.

```
SELECT e.id,
       e.last_name,
       e.first_name,
       e.gender,
       e.birthday,
       e.email,
```

```

        e.id_department,
        e.salary,
        NTILE(3) OVER(PARTITION BY e.id_department ORDER BY e.salary
ASC) AS grp_ntile
FROM employees e
WHERE e.id_department IN (1, 2);

```

Выполняем запрос и получаем результат, в котором данные таблицы `employees` были сгруппированы по столбцу `id_department`, и каждое окно было разделено на 3 группы. Номер группы, в которую входит строка отображается в столбце `grp_ntile`.

Сейчас нужно разобраться, как проставляются номера групп для строк в окне, для этого нужно внимательно прочитать пояснение ниже.

Пояснение к полученному результату:

- рассмотрим группу, которая была сформирована по идентификатору отдела с номером 1;
- функция `NTILE()` анализирует все значения в столбце `salary` и пытается их разделить на равные части. В данном случае функция присваивает строкам 1-3 группу 1, строкам 4-6 группу 2, а строкам 7-9 группу 3.

	123 id	ABC last_name	ABC first_name	ABC gender	🕒 birthday	ABC email	123 id_department	123 salary	123 grp_ntile
1	1	Иванов	Иван	Мужской	1990-05-15	ivanov@mail.ru	1	35 000	1
2	7	Соколова	Анастасия	Женский	1991-07-12	sokolova@mail.ru	1	40 000	1
3	13	Григорьева	Наталья	Женский	1996-01-25	grigoryeva@mail.ru	1	60 000	1
4	19	Семенова	Анна	Женский	1992-02-14	semenova@mail.ru	1	66 000	2
5	49	Карпов	Денис	Мужской	1995-01-20	karpov@gmail.com	1	68 000	2
6	31	Семенов	Артём	Мужской	1987-08-12	semenov@mail.ru	1	70 000	2
7	25	Кондратьев	Павел	Мужской	1990-12-14	kpav@mail.ru	1	70 000	3
8	37	Миронов	Игорь	Мужской	1988-07-15	mironov@mail.ru	1	72 000	3
9	43	Кузнецов	Игорь	Мужской	1985-11-20	[NULL]	1	75 000	3
10	2	Петров	Петр	Мужской	1985-12-20	petrov@yandex.ru	2	45 000	1
11	8	Лебедева	Алена	Женский	1994-09-18	lebedeva@yandex.ru	2	68 000	1
12	20	Федоров	Егор	Мужской	1993-09-18	f.eg@yandex.ru	2	72 000	1
13	32	Карпова	Анна	Женский	1990-06-18	karp@yandex.ru	2	75 000	2
14	26	Мельникова	Яна	Женский	1993-07-15	melnikova@yandex.ru	2	80 000	2
15	38	Ковалева	Елена	Женский	1995-12-30	kovaleva@yandex.ru	2	80 000	2
16	44	Соколов	Михаил	Мужской	1989-08-30	sokolov@yandex.ru	2	85 000	3
17	14	Борисов	Игорь	Мужской	1985-05-20	borisov@yandex.ru	2	95 000	3
18	50	Максимов	Максим	Мужской	1988-07-28	maximov@mail.ru	2	102 000	3

Функции смещения

Функции смещения — это функции, которые предназначены для перемещения и обращения к разным строкам в окне, относительно текущей строки, а также обращаться к значениям, которые находятся в начале или в конце окна.

Функция LAG()

Функция `LAG()` позволяет обратиться к предыдущей строке в текущем окне (группе). Если предыдущее значение отсутствует, то функция вернёт значение `NULL`.

Синтаксис:

- `column_name` – имя столбца, значение которого будет передано в функцию;
- `offset` – сдвиг;
- `default_value` – значение по умолчанию;
- `column_group_list` – список столбцов для группировки окна;
- `column_order_list` – список столбцов для сортировки в окне;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `condition_filter` – условия для фильтрации.

```
SELECT LAG(column_name, offset, default_value)
       OVER (PARTITION BY column_group_list
             ORDER BY column_order_list)
FROM schema.table_name
WHERE condition_filter;
```

Практический пример: при помощи оконной функции необходимо выполнить группировку данных таблицы `employees` по идентификатору отдела, а затем для каждой строки окна вывести предыдущее значение столбца `salary` при помощи функции `LAG()`. Сортировка внутри окна должна выполняться по столбцу `salary` в порядке возрастания.

Пояснение к SQL-запросу:

- в блоке `SELECT` выводим все необходимые столбцы таблицы `employees`;
- дополнительным столбцом указываем функцию смещения `LAG()` и в качестве аргумента передаем ей столбец `salary`;
- при помощи конструкции `OVER()` определяем оконную функцию, внутри неё задаём правило `PARTITION BY` для формирования окон. В данном случае окна будут формироваться по идентификатору отдела, а сортировка окна должна выполняться по столбцу `salary` в порядке возрастания;
- полученной конструкции присваиваем псевдоним `lag_salary`;

- в блоке `WHERE` добавляем условие, что нужны отделы с идентификаторами 1 и 2. Это нужно для того, чтобы сократить выборку из таблицы `employees`.

```
SELECT e.id,  
       e.last_name,  
       e.first_name,  
       e.gender,  
       e.birthday,  
       e.email,  
       e.id_department,  
       e.salary,  
       LAG(e.salary) OVER(PARTITION BY e.id_department ORDER BY  
e.salary ASC) AS lag_salary  
FROM employees e  
WHERE e.id_department IN (1, 2);
```

Выполняем запрос и получаем результат, в котором данные таблицы `employees` были сгруппированы по столбцу `id_department` и отсортированы по столбцу `salary` в порядке возрастания. Также для каждой строки окна было проставлено предыдущее значение столбца `salary`, оно отображается в столбце `lag_salary`.

Сейчас нужно разобраться, как проставляется предыдущее значения для каждой строки в группе, для этого нужно внимательно прочитать пояснение ниже.

Пояснение к полученному результату:

- рассмотрим группу, которая была сформирована по идентификатору отдела с номером 1;
- первая строка в группе - это сотрудник Иванов Иван, поскольку это первая строка в группе, то к предыдущей строке невозможно обратиться, так как её не существует, поэтому функция `LAG()` вернёт значение `NULL`;
- вторая строка в группе – это сотрудник Соколова Анастасия, и для неё уже возвращается предыдущее значение столбца `salary`, то есть в столбце `lag_salary` отображается заработная плата предыдущего сотрудника.

	123 id	abc last_name	abc first_name	abc gender	birthday	abc email	123 id_department	123 salary	123 lag_salary
1	1	Иванов	Иван	Мужской	1990-05-15	ivanov@mail.ru	1	35 000	[NULL]
2	7	Соколова	Анастасия	Женский	1991-07-12	sokolova@mail.ru	1	40 000	35 000
3	13	Григорьева	Наталья	Женский	1996-01-25	grigoryeva@mail.ru	1	60 000	40 000
4	19	Семенова	Анна	Женский	1992-02-14	semenova@mail.ru	1	66 000	60 000
5	49	Карпов	Денис	Мужской	1995-01-20	karpov@gmail.com	1	68 000	66 000
6	31	Семенов	Артём	Мужской	1987-08-12	semenov@mail.ru	1	70 000	68 000
7	25	Кондратьев	Павел	Мужской	1990-12-14	kpav@mail.ru	1	70 000	70 000
8	37	Миронов	Игорь	Мужской	1988-07-15	mironov@mail.ru	1	72 000	70 000
9	43	Кузнецов	Игорь	Мужской	1985-11-20	[NULL]	1	75 000	72 000
10	2	Петров	Петр	Мужской	1985-12-20	petrov@yandex.ru	2	45 000	[NULL]
11	8	Лебедева	Алена	Женский	1994-09-18	lebedeva@yandex.ru	2	68 000	45 000
12	20	Федоров	Егор	Мужской	1993-09-18	f.eg@yandex.ru	2	72 000	68 000
13	32	Карпова	Анна	Женский	1990-06-18	karp@yandex.ru	2	75 000	72 000
14	26	Мельникова	Яна	Женский	1993-07-15	melnikova@yandex.ru	2	80 000	75 000
15	38	Ковалева	Елена	Женский	1995-12-30	kovaleva@yandex.ru	2	80 000	80 000
16	44	Соколов	Михаил	Мужской	1989-08-30	sokolov@yandex.ru	2	85 000	80 000
17	14	Борисов	Игорь	Мужской	1985-05-20	borisov@yandex.ru	2	95 000	85 000
18	50	Максимов	Максим	Мужской	1988-07-28	maximov@mail.ru	2	102 000	95 000

Функция LEAD()

Функция `LEAD()` позволяет обратиться к следующей строке в текущем окне (группе). Если следующее значение отсутствует, то функция вернёт значение `NULL`.

Синтаксис:

- `column_name` – имя столбца, значение которого будет передано в функцию;
- `offset` – сдвиг;
- `default_value` – значение по умолчанию;
- `column_group_list` – список столбцов для группировки окна;
- `column_order_list` – список столбцов для сортировки в окне;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `condition_filter` – условия для фильтрации.

```
SELECT LEAD(column_name, offset, default_value)
      OVER (PARTITION BY column_group_list
            ORDER BY column_order_list)
FROM schema.table_name
WHERE condition_filter;
```

Практический пример: при помощи оконной функции необходимо выполнить группировку данных таблицы `employees` по идентификатору отдела, а затем для каждой строки окна вывести следующее значение столбца `salary` при помощи функции `LEAD()`. Сортировка внутри окна должна выполняться по столбцу `salary` в порядке возрастания.

Пояснение к SQL-запросу:

- в блоке `SELECT` выводим все необходимые столбцы таблицы `employees`;
- дополнительным столбцом указываем функцию смещения `LEAD()` и в качестве аргумента передаем ей столбец `salary`;
- при помощи конструкции `OVER()` определяем оконную функцию, внутри неё задаём правило `PARTITION BY` для формирования окон. В данном случае окна будут формироваться по идентификатору отдела, а сортировка окна должна выполняться по столбцу `salary` в порядке возрастания;
- полученной конструкции присваиваем псевдоним `lead_salary`;
- в блоке `WHERE` добавляем условие, что нужны отделы с идентификаторами 1 и 2. Это нужно для того, чтобы сократить выборку из таблицы `employees`.

```
SELECT e.id,
       e.last_name,
       e.first_name,
       e.gender,
       e.birthday,
       e.email,
       e.id_department,
       e.salary,
       LEAD(e.salary) OVER(PARTITION BY e.id_department ORDER BY
e.salary ASC) AS lead_salary
FROM employees e
WHERE e.id_department IN (1, 2);
```

Выполняем запрос и получаем результат, в котором данные таблицы `employees` были сгруппированы по столбцу `id_department` и отсортированы по столбцу `salary` в порядке возрастания. Также для каждой строки окна было проставлено следующее значение столбца `salary`, оно отображается в столбце `lead_salary`.

Сейчас нужно разобраться, как проставляется следующее значение для каждой строки в группе, для этого нужно внимательно прочитать пояснение ниже.

Пояснение к полученному результату:

- рассмотрим группу, которая была сформирована по идентификатору отдела с номером 1;

- первая строка в группе – это сотрудник Иванов Иван, и для неё было возвращено следующее значение столбца salary, то есть в столбце lead_salary отображается заработная плата следующего сотрудника;
- обратите внимание на последнюю строку в группе - это сотрудник Кузнецов Игорь, поскольку это последняя строка в группе, то к следующей строке невозможно обратиться, так как её не существует, поэтому функция LEAD() вернёт значение NULL;

	123 id	ABC last_name	ABC first_name	ABC gender	birthday	ABC email	123 id_department	123 salary	123 lead_salary
1	1	Иванов	Иван	Мужской	1990-05-15	ivanov@mail.ru	1	35 000	40 000
2	7	Соколова	Анастасия	Женский	1991-07-12	sokolova@mail.ru	1	40 000	60 000
3	13	Григорьева	Наталья	Женский	1996-01-25	grigoryeva@mail.ru	1	60 000	66 000
4	19	Семенова	Анна	Женский	1992-02-14	semenova@mail.ru	1	66 000	68 000
5	49	Карпов	Денис	Мужской	1995-01-20	karpov@gmail.com	1	68 000	70 000
6	31	Семенов	Артём	Мужской	1987-08-12	semenov@mail.ru	1	70 000	70 000
7	25	Кондратьев	Павел	Мужской	1990-12-14	kpav@mail.ru	1	70 000	72 000
8	37	Миронов	Игорь	Мужской	1988-07-15	mironov@mail.ru	1	72 000	75 000
9	43	Кузнецов	Игорь	Мужской	1985-11-20	[NULL]	1	75 000	[NULL]
10	2	Петров	Петр	Мужской	1985-12-20	petrov@yandex.ru	2	45 000	68 000
11	8	Лебедева	Алена	Женский	1994-09-18	lebedeva@yandex.ru	2	68 000	72 000
12	20	Федоров	Егор	Мужской	1993-09-18	f.eg@yandex.ru	2	72 000	75 000
13	32	Карпова	Анна	Женский	1990-06-18	karp@yandex.ru	2	75 000	80 000
14	26	Мельникова	Яна	Женский	1993-07-15	melnikova@yandex.ru	2	80 000	80 000
15	38	Ковалева	Елена	Женский	1995-12-30	kovaleva@yandex.ru	2	80 000	85 000
16	44	Соколов	Михаил	Мужской	1989-08-30	sokolov@yandex.ru	2	85 000	95 000
17	14	Борисов	Игорь	Мужской	1985-05-20	borisov@yandex.ru	2	95 000	102 000
18	50	Максимов	Максим	Мужской	1988-07-28	maximov@mail.ru	2	102 000	[NULL]

Функция FIRST_VALUE()

Функция FIRST_VALUE() позволяет получить первое значение в группе (окне).

Синтаксис:

- column_name – имя столбца, значение которого будет передано в функцию;
- column_group_list – список столбцов для группировки окна;
- column_order_list – список столбцов для сортировки в окне;
- schema – наименование схемы, в которой находится объект;
- table_name – имя таблицы;
- condition_filter – условия для фильтрации.

```
SELECT FIRST_VALUE(column_name)
      OVER (PARTITION BY column_group_list
            ORDER BY column_order_list)
FROM schema.table_name
WHERE condition_filter;
```

Практический пример: при помощи оконной функции необходимо выполнить группировку данных таблицы `employees` по идентификатору отдела, а затем определить первое значение столбца `salary` в каждом окне при помощи функции `FIRST_VALUE()`.

Пояснение к SQL-запросу:

- в блоке `SELECT` выводим все необходимые столбцы таблицы `employees`;
- дополнительным столбцом указываем функцию смещения `FIRST_NAME()` и в качестве аргумента передаем ей столбец `salary`;
- при помощи конструкции `OVER()` определяем оконную функцию, внутри неё задаём правило `PARTITION BY` для формирования окон. В данном случае окна будут формироваться по идентификатору отдела;
- полученной конструкции присваиваем псевдоним `first_salary`;
- в блоке `WHERE` добавляем условие, что нужны отделы с идентификаторами 1 и 2. Это нужно для того, чтобы сократить выборку из таблицы `employees`.

```
SELECT e.id,
       e.last_name,
       e.first_name,
       e.gender,
       e.birthday,
       e.email,
       e.id_department,
       e.salary,
       FIRST_VALUE(e.salary) OVER(PARTITION BY e.id_department) AS
first_salary
FROM employees e
WHERE e.id_department IN (1, 2);
```

Выполняем запрос и получаем результат, в котором данные таблицы `employees` были сгруппированы по столбцу `id_department` и для каждого окна было получено первое значение столбца `salary`, оно отображается в столбце `first_salary`.

	123 id	ABC last_name	ABC first_name	ABC gender	birthday	ABC email	123 id_department	123 salary	123 first_salary
1	7	Соколова	Анастасия	Женский	1991-07-12	sokolova@mail.ru	1	40 000	40 000
2	31	Семенов	Артём	Мужской	1987-08-12	semenov@mail.ru	1	70 000	40 000
3	19	Семенова	Анна	Женский	1992-02-14	semenova@mail.ru	1	66 000	40 000
4	37	Миронов	Игорь	Мужской	1988-07-15	miroнов@mail.ru	1	72 000	40 000
5	13	Григорьева	Наталья	Женский	1996-01-25	grigoryeva@mail.ru	1	60 000	40 000
6	43	Кузнецов	Игорь	Мужской	1985-11-20	[NULL]	1	75 000	40 000
7	25	Кондратьев	Павел	Мужской	1990-12-14	kpav@mail.ru	1	70 000	40 000
8	49	Карпов	Денис	Мужской	1995-01-20	karpov@gmail.com	1	68 000	40 000
9	1	Иванов	Иван	Мужской	1990-05-15	ivanov@mail.ru	1	35 000	40 000
10	50	Максимов	Максим	Мужской	1988-07-28	maximov@mail.ru	2	102 000	102 000
11	2	Петров	Петр	Мужской	1985-12-20	petrov@yandex.ru	2	45 000	102 000
12	8	Лебедева	Алена	Женский	1994-09-18	lebedeva@yandex.ru	2	68 000	102 000
13	14	Борисов	Игорь	Мужской	1985-05-20	borisov@yandex.ru	2	95 000	102 000
14	20	Федоров	Егор	Мужской	1993-09-18	feg@yandex.ru	2	72 000	102 000
15	26	Мельникова	Яна	Женский	1993-07-15	melnikova@yandex.ru	2	80 000	102 000
16	32	Карпова	Анна	Женский	1990-06-18	karp@yandex.ru	2	75 000	102 000
17	38	Ковалева	Елена	Женский	1995-12-30	kovaleva@yandex.ru	2	80 000	102 000
18	44	Соколов	Михаил	Мужской	1989-08-30	sokolov@yandex.ru	2	85 000	102 000

Функция LAST_VALUE()

Функция LAST_VALUE() позволяет получить последнее значение в группе (окне).

Синтаксис:

- column_name – имя столбца, значение которого будет передано в функцию;
- column_group_list – список столбцов для группировки окна;
- column_order_list – список столбцов для сортировки в окне;
- schema – наименование схемы, в которой находится объект;
- table_name – имя таблицы;
- condition_filter – условия для фильтрации.

```
SELECT LAST_VALUE(column_name)
      OVER (PARTITION BY column_group_list
            ORDER BY column_order_list)
FROM schema.table_name
WHERE condition_filter;
```

Практический пример: при помощи оконной функции необходимо выполнить группировку данных таблицы employees по идентификатору отдела, а затем определить последнее значение столбца salary в каждом окне при помощи функции LAST_VALUE().

Пояснение к SQL-запросу:

- в блоке SELECT выводим все необходимые столбцы таблицы employees;
- дополнительным столбцом указываем функцию смещения LAST_VALUE() и в качестве аргумента передаем ей столбец salary;

- при помощи конструкции `OVER()` определяем оконную функцию, внутри неё задаём правило `PARTITION BY` для формирования окон. В данном случае окна будут формироваться по идентификатору отдела;
- полученной конструкции присваиваем псевдоним `last_salary`;
- в блоке `WHERE` добавляем условие, что нужны отделы с идентификаторами 1 и 2. Это нужно для того, чтобы сократить выборку из таблицы `employees`.

```
SELECT e.id,
       e.last_name,
       e.first_name,
       e.gender,
       e.birthday,
       e.email,
       e.id_department,
       e.salary,
       LAST_VALUE(e.salary) OVER(PARTITION BY e.id_department) AS
last_salary
FROM employees e
WHERE e.id_department IN (1, 2);
```

Выполняем запрос и получаем результат, в котором данные таблицы `employees` были сгруппированы по столбцу `id_department` и для каждого окна было получено последнее значение столбца `salary`, оно отображается в столбце `last_salary`.

	123 id	ABC last_name	ABC first_name	ABC gender	birthday	ABC email	123 id_department	123 salary	123 last_salary
1	7	Соколова	Анастасия	Женский	1991-07-12	sokolova@mail.ru	1	40 000	35 000
2	31	Семенов	Артём	Мужской	1987-08-12	semenov@mail.ru	1	70 000	35 000
3	19	Семенова	Анна	Женский	1992-02-14	semenova@mail.ru	1	66 000	35 000
4	37	Миронов	Игорь	Мужской	1988-07-15	mironov@mail.ru	1	72 000	35 000
5	13	Григорьева	Наталья	Женский	1996-01-25	grigoryeva@mail.ru	1	60 000	35 000
6	43	Кузнецов	Игорь	Мужской	1985-11-20	[NULL]	1	75 000	35 000
7	25	Кондратьев	Павел	Мужской	1990-12-14	kpav@mail.ru	1	70 000	35 000
8	49	Карпов	Денис	Мужской	1995-01-20	karpov@gmail.com	1	68 000	35 000
9	1	Иванов	Иван	Мужской	1990-05-15	ivanov@mail.ru	1	35 000	35 000
10	50	Максимов	Максим	Мужской	1988-07-28	maximov@mail.ru	2	102 000	85 000
11	2	Петров	Петр	Мужской	1985-12-20	petrov@yandex.ru	2	45 000	85 000
12	8	Лебедева	Алена	Женский	1994-09-18	lebedeva@yandex.ru	2	68 000	85 000
13	14	Борисов	Игорь	Мужской	1985-05-20	borisov@yandex.ru	2	95 000	85 000
14	20	Федоров	Егор	Мужской	1993-09-18	f.eg@yandex.ru	2	72 000	85 000
15	26	Мельникова	Яна	Женский	1993-07-15	melnikova@yandex.ru	2	80 000	85 000
16	32	Карпова	Анна	Женский	1990-06-18	karp@yandex.ru	2	75 000	85 000
17	38	Ковалева	Елена	Женский	1995-12-30	kovaleva@yandex.ru	2	80 000	85 000
18	44	Соколов	Михаил	Мужской	1989-08-30	sokolov@yandex.ru	2	85 000	85 000

Аналитические функции

Аналитические функции — это функции, которые позволяют вернуть информацию о распределении данных в окне. В основном аналитические функции используются для выполнения статического анализа.

Функция CUME_DIST()

Функция `CUME_DIST()` позволяет вычислить относительное положение (интегральное распределение) строк в окне (группе).

Данная функция используется для вычисления кумулятивного распределения (*cumulative distribution*) для каждой строки в окне. Кумулятивное распределение — это функция, которая показывает, какая доля данных в наборе значений имеет значение меньше или равно данного значения.

Функция `CUME_DIST()` возвращает значение от 0 до 1, где 0 означает, что значение является наименьшим в окне, а 1 - что значение является наибольшим в окне. Значение, меньшее или равное 0, обозначает, что окно пустое.

Синтаксис:

- `column_group_list` – список столбцов для группировки окна;
- `column_order_list` – список столбцов для сортировки в окне;
- `schema` – наименование схемы, в которой находится объект;
- `table_name` – имя таблицы;
- `condition_filter` – условия для фильтрации.

```
SELECT CUME_DIST()  
      OVER (PARTITION BY column_group_list  
            ORDER BY column_order_list)  
FROM schema.table_name  
WHERE condition_filter;
```

Практический пример: при помощи оконной функции необходимо выполнить группировку данных таблицы `employees` по идентификатору отдела, а затем определить относительное положение для каждой строки окна при помощи функции `CUME_DIST()`. Сортировка внутри окна должна выполняться по столбцу `salary` в порядке возрастания.

Пояснение к SQL-запросу:

- в блоке `SELECT` выводим все необходимые столбцы таблицы `employees`;
- дополнительным столбцом указываем аналитическую функцию `CUME_DIST()` без аргумента;
- при помощи конструкции `OVER()` определяем оконную функцию, внутри неё задаём правило `PARTITION BY` для формирования окон. В данном случае окна будут формироваться по идентификатору отдела, а сортировка окна должна выполняться по столбцу `salary` в порядке возрастания;
- полученной конструкции присваиваем псевдоним `cdist_rank`;
- в блоке `WHERE` добавляем условие, что нужны отделы с идентификаторами 1 и 2. Это нужно для того, чтобы сократить выборку из таблицы `employees`.

```
SELECT e.id,
       e.last_name,
       e.first_name,
       e.gender,
       e.birthday,
       e.email,
       e.id_department,
       e.salary,
       CUME_DIST() OVER(PARTITION BY e.id_department ORDER BY e.salary
ASC) AS cdist_rank
FROM employees e
WHERE e.id_department IN (1, 2);
```

Выполняем запрос и получаем результат, в котором данные таблицы `employees` были сгруппированы по столбцу `id_department` и отсортированы по столбцу `salary` в порядке возрастания. Также для каждой строки окна было определено относительное положение, оно отображается в столбце `cdist_rank`.

	123 id	abc last_name	abc first_name	abc gender	birthday	abc email	123 id_department	123 salary	123 cdist_rank
1	1	Иванов	Иван	Мужской	1990-05-15	ivanov@mail.ru	1	35 000	0,1111111111
2	7	Соколова	Анастасия	Женский	1991-07-12	sokolova@mail.ru	1	40 000	0,2222222222
3	13	Григорьева	Наталья	Женский	1996-01-25	grigoryeva@mail.ru	1	60 000	0,3333333333
4	19	Семенова	Анна	Женский	1992-02-14	semenova@mail.ru	1	66 000	0,4444444444
5	49	Карпов	Денис	Мужской	1995-01-20	karpov@gmail.com	1	68 000	0,5555555556
6	31	Семенов	Артём	Мужской	1987-08-12	semenov@mail.ru	1	70 000	0,7777777778
7	25	Кондратьев	Павел	Мужской	1990-12-14	kpav@mail.ru	1	70 000	0,7777777778
8	37	Миронов	Игорь	Мужской	1988-07-15	miroнов@mail.ru	1	72 000	0,8888888889
9	43	Кузнецов	Игорь	Мужской	1985-11-20	[NULL]	1	75 000	1
10	2	Петров	Петр	Мужской	1985-12-20	petrov@yandex.ru	2	45 000	0,1111111111
11	8	Лебедева	Алена	Женский	1994-09-18	lebedeva@yandex.ru	2	68 000	0,2222222222
12	20	Федоров	Егор	Мужской	1993-09-18	f.eg@yandex.ru	2	72 000	0,3333333333
13	32	Карпова	Анна	Женский	1990-06-18	karp@yandex.ru	2	75 000	0,4444444444
14	26	Мельникова	Яна	Женский	1993-07-15	melnikova@yandex.ru	2	80 000	0,6666666667
15	38	Ковалева	Елена	Женский	1995-12-30	kovaleva@yandex.ru	2	80 000	0,6666666667
16	44	Соколов	Михаил	Мужской	1989-08-30	sokolov@yandex.ru	2	85 000	0,7777777778
17	14	Борисов	Игорь	Мужской	1985-05-20	borisov@yandex.ru	2	95 000	0,8888888889
18	50	Максимов	Максим	Мужской	1988-07-28	maximov@mail.ru	2	102 000	1

Функция PERCENT_RANK()

Функция PERCENT_RANK () вычисляет относительный ранг строки * в текущем окне.

* Относительный ранг строки в текущем окне в функции PERCENT_RANK () — это значение, которое показывает, какая часть результирующего набора данных находится до текущей строки в отсортированном порядке. Другими словами, это процентное значение, указывающее, насколько текущая строка выше или ниже остальных строк в отсортированном наборе данных.

Значение относительного ранга вычисляется при помощи следующей формулы:

$$PERCENT_RANK = \frac{RANK-1}{TOTAL\ ROWS-1},$$

где RANK - ранг текущей строки, а TOTAL ROWS - общее количество строк в отсортированном наборе данных.

Итоговое значение PERCENT_RANK будет находиться в диапазоне от 0 до 1. Значение 0 означает, что текущая строка находится в самом начале отсортированного набора данных, а значение 1, что в самом конце.

Синтаксис:

- column_group_list – список столбцов для группировки окна;
- column_order_list – список столбцов для сортировки в окне;
- schema – наименование схемы, в которой находится объект;
- table_name – имя таблицы;
- condition_filter – условия для фильтрации.

```
SELECT PERCENT_RANK()
```

```

        OVER (PARTITION BY column_group_list
              ORDER BY column_order_list)
FROM schema.table_name
WHERE condition_filter;

```

Практический пример: при помощи оконной функции необходимо выполнить группировку данных таблицы `employees` по идентификатору отдела, а затем для каждой строки окна определить относительный ранг при помощи функции `PERCENT_RANK()`. Сортировка внутри окна должна выполняться по столбцу `salary` в порядке возрастания.

Пояснение к SQL-запросу:

- в блоке `SELECT` выводим все необходимые столбцы таблицы `employees`;
- дополнительным столбцом указываем аналитическую функцию `PERCENT_RANK()` без аргумента;
- при помощи конструкции `OVER()` определяем оконную функцию, внутри неё задаём правило `PARTITION BY` для формирования окон. В данном случае окна будут формироваться по идентификатору отдела, а сортировка окна должна выполняться по столбцу `salary` в порядке возрастания;
- полученной конструкции присваиваем псевдоним `prct_rank`;
- в блоке `WHERE` добавляем условие, что нужны отделы с идентификаторами 1 и 2. Это нужно для того, чтобы сократить выборку из таблицы `employees`.

```

SELECT e.id,
       e.last_name,
       e.first_name,
       e.gender,
       e.birthday,
       e.email,
       e.id_department,
       e.salary,
       PERCENT_RANK() OVER(PARTITION BY e.id_department ORDER BY
e.salary ASC) AS prct_rank
FROM employees e
WHERE e.id_department IN (1, 2);

```

Выполняем запрос и получаем результат, в котором данные таблицы `employees` были сгруппированы по столбцу `id_department` и отсортированы по столбцу `salary` в порядке

возрастания. Также для каждой строки окна был определен относительный ранг, он отображается в столбце `prct_rank`.

	123 id	ABC last_name	ABC first_name	ABC gender	birthday	ABC email	123 id_department	123 salary	123 prct_rank
Таблица	1	Иванов	Иван	Мужской	1990-05-15	ivanov@mail.ru	1	35 000	0
	2	Соколова	Анастасия	Женский	1991-07-12	sokolova@mail.ru	1	40 000	0,125
	3	Григорьева	Наталья	Женский	1996-01-25	grigoryeva@mail.ru	1	60 000	0,25
Текст	4	Семенова	Анна	Женский	1992-02-14	semenova@mail.ru	1	66 000	0,375
	5	Карпов	Денис	Мужской	1995-01-20	karpov@gmail.com	1	68 000	0,5
	6	Семенов	Артём	Мужской	1987-08-12	semenov@mail.ru	1	70 000	0,625
	7	Кондратьев	Павел	Мужской	1990-12-14	kpav@mail.ru	1	70 000	0,625
	8	Миронов	Игорь	Мужской	1988-07-15	mironov@mail.ru	1	72 000	0,875
	9	Кузнецов	Игорь	Мужской	1985-11-20	[NULL]	1	75 000	1
Запись	10	Петров	Петр	Мужской	1985-12-20	petrov@yandex.ru	2	45 000	0
	11	Лебедева	Алена	Женский	1994-09-18	lebedeva@yandex.ru	2	68 000	0,125
	12	Федоров	Егор	Мужской	1993-09-18	f.eg@yandex.ru	2	72 000	0,25
	13	Карпова	Анна	Женский	1990-06-18	karp@yandex.ru	2	75 000	0,375
	14	Мельникова	Яна	Женский	1993-07-15	melnikova@yandex.ru	2	80 000	0,5
	15	Ковалева	Елена	Женский	1995-12-30	kovaleva@yandex.ru	2	80 000	0,5
	16	Соколов	Михаил	Мужской	1989-08-30	sokolov@yandex.ru	2	85 000	0,75
	17	Борисов	Игорь	Мужской	1985-05-20	borisov@yandex.ru	2	95 000	0,875
	18	Максимов	Максим	Мужской	1988-07-28	maximov@mail.ru	2	102 000	1

Представления (VIEW)

Что такое представления?

Представления (VIEW) — это виртуальные таблицы, которые чаще всего называют вьюшками. В отличие от таблиц, которые хранят данные, представления хранят только SQL-запросы для динамического извлечения данных по мере надобности. Другими словами, когда происходит обращение к представлению, то сначала выполняется код, который хранится в этом представлении, а затем выводится результирующий набор данных.

Чтобы грамотно использовать представления, вы должны понимать и знать структуру таблиц, из которых происходит выборка данных.

Основные причины для применения представлений (VIEW):

- повторное использование операторов SQL;
- доступ к определенным частям вместо всей таблицы;
- не нужно писать повторно SQL запрос на получение данных, достаточно обратиться к созданному представлению;
- благодаря представлениям вы можете обеспечить безопасность ваших данных, то есть пользователи будут видеть только часть данных, которые им необходимы, остальные данные они видеть не будут.

Правила и ограничения для представлений:

- представления должны иметь уникальные имена;
- на количество создаваемых представлений не накладывается никаких ограничений;
- чтобы создавать представления нужен доступ от отдела безопасности;
- представления могут быть вложенными, другими словами, это значит, что одно представление может быть построено с использованием запроса, извлекающего данные из другого представления;
- оператор ORDER BY допускается использовать в представлении, но его действие может быть отменено, если другое предложение `ORDER BY` используется в операторе SELECT для извлечения данных из представления;
- представления нельзя индексировать и применять в них триггеры.

Создание представления

Представления создаются при помощи оператора `CREATE VIEW`.



Оператор `CREATE VIEW` не выводит никаких результатов. Но если выполнить код в программном обеспечении DBeaver, то на экране появится информационное сообщение, что запрос выполнен.

Синтаксис:

- `schema` – наименование схемы, в которой находится объект;
- `view_name` – имя представления;
- `sql_query` – SQL-запрос.

```
CREATE VIEW schema.view_name AS  
sql_query;
```

Практический пример: есть SQL-запрос, он возвращает информацию о сотрудниках компании. Необходимо при помощи данного запроса создать представление с именем `v_employees`.

Пояснение к SQL-запросу:

- левое соединение с таблицей `employees` необходимо для того, чтобы вывести фамилию и имя руководителя сотрудника;
- внутреннее объединение с таблицей `departments` необходимо для того, чтобы вывести наименование подразделения, в котором трудоустроен сотрудник;
- функция `COALESCE` заменяет значения `NULL` на значение 'Начальник отдела' в столбце `name_boss`, так как у руководителя это значение не проставляется.

```
SELECT e1.id,  
       e1.first_name,  
       e1.last_name,  
       e1.gender,  
       e1.birthday,  
       e1.email,  
       e1.id_department,  
       d.name_department,
```

```

        COALESCE(e2.first_name||' '||e2.last_name, 'Начальник отдела')
AS name_boss,
        e1.salary
FROM employees e1
LEFT JOIN employees e2
ON e2.id = e1.id_boss
JOIN departments d
ON d.id = e1.id_department
ORDER BY e1.id ASC;

```

Результат выполнения запроса.

	123 id	abc first_name	abc last_name	abc gender	birthday	abc email	123 id_department	abc name_department	abc name_boss	123 salary
1	1	Иван	Иванов	Мужской	1990-05-15	ivanov@mail.ru	1	Отдел маркетинга	Начальник отдела	35 000
2	2	Петр	Петров	Мужской	1985-12-20	petrov@yandex.ru	2	Отдел финансов	Начальник отдела	45 000
3	3	Мария	Сидорова	Женский	1992-08-10	sidorova@gmail.com	3	Отдел разработки	Начальник отдела	54 000
4	4	Петр	Петров	Мужской	1987-04-25	[NULL]	4	Отдел кадров	Начальник отдела	100 000
5	5	Екатерина	Васильева	Женский	1995-03-30	vasileva@yandex.ru	5	Отдел логистики	Начальник отдела	38 900
6	6	Дмитрий	Попов	Мужской	1988-11-05	popov@gmail.com	6	Отдел качества	Начальник отдела	110 000
7	7	Анастасия	Соколова	Женский	1991-07-12	sokolova@mail.ru	1	Отдел маркетинга	Иван Иванов	40 000
8	8	Алена	Лебедева	Женский	1994-09-18	lebedeva@yandex.ru	2	Отдел финансов	Петр Петров	68 000
9	9	Артём	Козлов	Мужской	1986-10-22	kozlov@gmail.com	3	Отдел разработки	Мария Сидорова	120 000
10	10	Андрей	Новиков	Мужской	1989-06-08	novikov@mail.ru	4	Отдел кадров	Петр Петров	42 000
11	11	Ольга	Морозова	Женский	1993-02-14	morozova@yandex.ru	5	Отдел логистики	Екатерина Васильева	70 000
12	12	Сергей	Павлов	Мужской	1987-12-11	pavlov@gmail.com	6	Отдел качества	Дмитрий Попов	130 000

Дальше необходимо из текущего SQL-запроса сделать представление с именем v_employees, для этого используем оператор CREATE VIEW.

```

CREATE VIEW v_employees AS
SELECT e1.id,
        e1.first_name,
        e1.last_name,
        e1.gender,
        e1.birthday,
        e1.email,
        e1.id_department,
        d.name_department,
        COALESCE(e2.first_name||' '||e2.last_name, 'Начальник отдела')
AS name_boss,
        e1.salary
FROM employees e1
LEFT JOIN employees e2
ON e2.id = e1.id_boss
JOIN departments d

```

```
ON d.id = e1.id_department
ORDER BY e1.id ASC;
```

Представление создано и сейчас нужно обратиться к нему, как к обычной таблице и вывести всю информацию.

```
SELECT *
FROM v_employees;
```

Результат выполнения запроса. Сначала был выполнен SQL-запрос, который хранится в представлении, а затем был выведен результирующий набор данных.

	123 id	first_name	last_name	gender	birthday	email	123 id_department	name_department	name_boss	salary
1	1	Иван	Иванов	Мужской	1990-05-15	ivanov@mail.ru	1	Отдел маркетинга	Начальник отдела	35 000
2	2	Петр	Петров	Мужской	1985-12-20	petrov@yandex.ru	2	Отдел финансов	Начальник отдела	45 000
3	3	Мария	Сидорова	Женский	1992-08-10	sidorova@gmail.com	3	Отдел разработки	Начальник отдела	54 000
4	4	Петр	Петров	Мужской	1987-04-25	[NULL]	4	Отдел кадров	Начальник отдела	100 000
5	5	Екатерина	Васильева	Женский	1995-03-30	vasilieva@yandex.ru	5	Отдел логистики	Начальник отдела	38 900
6	6	Дмитрий	Попов	Мужской	1988-11-05	popov@gmail.com	6	Отдел качества	Начальник отдела	110 000
7	7	Анастасия	Соколова	Женский	1991-07-12	sokolova@mail.ru	1	Отдел маркетинга	Иван Иванов	40 000
8	8	Алена	Лебедева	Женский	1994-09-18	lebedeva@yandex.ru	2	Отдел финансов	Петр Петров	68 000
9	9	Артём	Козлов	Мужской	1986-10-22	kozlov@gmail.com	3	Отдел разработки	Мария Сидорова	120 000
10	10	Андрей	Новиков	Мужской	1989-06-08	novikov@mail.ru	4	Отдел кадров	Петр Петров	42 000
11	11	Ольга	Морозова	Женский	1993-02-14	morozova@yandex.ru	5	Отдел логистики	Екатерина Васильева	70 000
12	12	Сергей	Павлов	Мужской	1987-12-11	pavlov@gmail.com	6	Отдел качества	Дмитрий Попов	130 000

Сейчас мы убедимся, что представления бывают очень полезны, так как представление `v_employees` возвращает информацию о всех сотрудниках компании, то мы без проблем можем получить информацию о сотрудниках другого отдела и нам не придется снова писать SQL-запрос.

В качестве примера выведем информацию о сотрудниках отдела с идентификатором 5. Для этого добавляем в блоке `WHERE` нужное условие фильтрации.

```
SELECT *
FROM v_employees
WHERE id_department = 5;
```

Выполняем запрос и получаем результат со списком сотрудников, которые трудоустроены в отделе с идентификатором 5.

	123 id	first_name	last_name	gender	birthday	email	123 id_department	name_department	name_boss	salary
1	5	Екатерина	Васильева	Женский	1995-03-30	vasilieva@yandex.ru	5	Отдел логистики	Начальник отдела	38 900
2	11	Ольга	Морозова	Женский	1993-02-14	morozova@yandex.ru	5	Отдел логистики	Екатерина Васильева	70 000
3	17	Кирилл	Егоров	Мужской	1986-07-22	[NULL]	5	Отдел логистики	Екатерина Васильева	62 000
4	23	Иван	Кудряшов	Мужской	1987-06-20	kudryashov@yandex.ru	5	Отдел логистики	Ольга Морозова	80 000
5	29	Алёна	Ильина	Женский	1995-09-30	ilina@yandex.ru	5	Отдел логистики	Кирилл Егоров	75 000
6	35	Екатерина	Кузнецова	Женский	1991-07-22	kuznetsova@yandex.ru	5	Отдел логистики	Кирилл Егоров	72 000
7	41	Екатерина	Попова	Женский	1993-01-08	popova@yandex.ru	5	Отдел логистики	Ольга Морозова	73 000
8	47	Павел	Григорьев	Мужской	1987-12-04	grigoryev@mail.ru	5	Отдел логистики	Ольга Морозова	77 000

Обновление представления

Обновить ранее созданное представление можно двумя способами:

- **1 способ:** можно удалить представление при помощи оператора [DROP VIEW](#), а затем создать его заново;
- **2 способ:** можно воспользоваться составным оператором `CREATE OR REPLACE VIEW`, он создаёт представление, если оно не существует, или заменяет его, если оно существует.

Синтаксис:

- `schema` – наименование схемы, в которой находится объект;
- `view_name` – имя представления;
- `sql_query` – SQL-запрос.

```
CREATE OR REPLACE VIEW schema.view_name AS  
sql_query;
```

Удаление представления

Чтобы удалить представление, необходимо воспользоваться оператором `DROP VIEW`.

Синтаксис:

- `schema` – наименование схемы, в которой находится объект;
- `view_name` – имя представления.

```
DROP VIEW schema.view_name;
```


Управление доступом к данным

Каждый пользователь в базе данных имеет привилегии, благодаря которым он может выполнять определенные действия. Исходя из этого можно сказать, что привилегии (права доступа) — это действия, которые может выполнить пользователь в базе данных.

Для чего нужны привилегии?

При помощи привилегий можно контролировать доступ пользователей к объектам базы данных.

Кто выдаёт права доступа к объектам?

Для каждого пользователя привилегии назначаются администратором базы данных или владельцем объекта в индивидуальном порядке. Все назначенные привилегии записываются в системные таблицы, и в дальнейшем можно вносить изменения, то есть удалять неактуальные привилегии и добавлять новые.

Владелец объекта — это пользователь, который создал объект и имеет на него все права.

Виды привилегий

Существует два вида привилегий: объектные и системные.

Объектные привилегии

Объектные привилегии (`object privilege`) позволяют выполнять определенные действия над конкретным объектом базы данных, такие как таблицы, представления, процедуры и т.д. Объектные привилегии влияют на возможность выполнения определенных действий с объектом.

Существуют следующие объектные привилегии:

- `SELECT` (позволяет читать данные из таблицы или представления);
- `INSERT` (позволяет вставлять новые строки в таблицу);
- `UPDATE` (позволяет обновлять существующие строки в таблице);
- `DELETE` (позволяет удалять строки из таблицы);
- `TRUNCATE` (позволяет удалить все строки из таблицы, при этом схема таблицы остается неизменной);

- REFERENCES (позволяет создавать внешние ключи, которые ссылаются на данную таблицу);
- TRIGGER (позволяет создавать и удалять триггеры на таблицу);
- USAGE (позволяет использовать объект базы данных, например, скалярную функцию или процедуру);
- EXECUTE (позволяет выполнять объект базы данных, такие как хранимые процедуры или пользовательские функции);
- CREATE (позволяет создавать новые объекты базы данных, такие как таблицы, представления или индексы);
- ALTER (позволяет изменять существующие объекты базы данных, такие как таблицы или представления);
- DROP (позволяет удалять объекты базы данных);
- ALL PRIVILEGES (предоставляет все перечисленные выше привилегии для данного объекта).

Системные привилегии

В PostgreSQL есть несколько уровней системных привилегий (`system privilege`), которые позволяют управлять доступом и выполнением различных операций. Вот некоторые из основных системных привилегий:

- SUPERUSER (привилегия дает пользователю полный контроль над всей системой. Обладатели этой привилегии могут создавать базы данных, управлять пользователями, изменять конфигурацию сервера и выполнять другие административные задачи);
- CREATEDB (пользователь имеющий эту привилегию, может создавать новые базы данных);
- CREATEROLE (привилегия позволяет пользователю создавать новых пользователей и роли в системе);
- LOGIN (позволяет пользователю входить в систему. Без нее пользователь не сможет авторизоваться и выполнять запросы);
- REPLICATION (пользователь имеющий эту привилегию, может создавать потоковые репликации, что позволяет ему создавать резервные копии данных и масштабировать систему);
- и т.д.

Информация о пользователях, ролях и привилегиях?

Для того, чтобы найти информацию о пользователях и привилегиях, достаточно написать SQL-запрос к нужным таблицам в информационной схеме.

```
-- Вывод информации о пользователях
SELECT *
FROM pg_user;

-- Вывод информации о ролях
SELECT *
FROM pg_roles;

-- Вывод информации о привилегиях
SELECT *
FROM information_schema.table_privileges;

-- Вывод информации о пользователях, ролях и привилегиях
SELECT u.username, r.rolname, p.privilege_type
FROM pg_user u
LEFT JOIN pg_auth_members m
ON (u.usesysid = m.member)
LEFT JOIN pg_roles r
ON (m.roleid = r.oid)
LEFT JOIN information_schema.table_privileges p
ON (r.rolname = p.grantee);
```

Оператор GRANT

Для предоставления доступа пользователям на объекты базы данных необходимо воспользоваться оператором `GRANT`. Права доступа на объекты в базе данных выдаются на основании типов объектов. Ранее мы рассмотрели, какие бывают - виды и списки привилегий.

Синтаксис:

- `privileges` – привилегии на объект базы данных, которые необходимо предоставить пользователю (можно перечислять через запятую);
- `schema` – наименование схемы, в которой находится объект;

- `object_name` – имя объекта, к которому необходимо предоставить доступ;
- `[role | user]` – имя пользователя или роли, кому будут выданы права доступа.

```
GRANT privileges ON schema.object_name TO [role | user];
```

Аргументы ALL и PUBLIC

У оператора `GRANT` есть дополнительные аргументы, которые имеют специальное назначение: `ALL PRIVILEGES` (или просто `ALL`) и `PUBLIC`.

`ALL` используется вместо перечисления всех привилегий в операторе `GRANT` для передачи всех привилегий для объекта базы данных.

```
GRANT ALL PRIVILEGES ON schema.object_name TO [role | user];
```

`PUBLIC` - по большей части относится к пользователям, а не к привилегиям. Когда выдаются права доступа с аргументом `PUBLIC`, все пользователи получают их автоматически. Очень часто аргумент `PUBLIC` используют в том случае, когда необходимо массово предоставить привилегии на чтение определенной таблицы.

```
GRANT SELECT ON schema.object_name TO PUBLIC;
```

Когда вы используете аргумент `PUBLIC` при выдаче прав доступа, доступ получают не только существующие пользователи базы данных, но и новый пользователь, который подключается к базе данных.

Передача привилегий с использованием GRANT OPTION

Иногда у владельца объектов возникает необходимость в том, чтобы пользователи базы данных могли самостоятельно передавать привилегии на эти объекты другим пользователям. Это можно осуществить, используя дополнительное предложение `WITH GRANT OPTION` в команде `GRANT`.

Синтаксис:

- `privileges` – привилегии на объект базы данных, которые необходимо предоставить пользователю (можно перечислять через запятую);
- `schema` – наименование схемы, в которой находится объект;
- `object_name` – имя объекта, к которому необходимо предоставить доступ;
- `user` – имя пользователя, кому будут выданы права доступа.

```
GRANT privileges ON schema.object_name TO user  
WITH GRANT OPTION;
```

Оператор REVOKE

Любой владелец объекта или администратор базы данных может лишить пользователя или список пользователей определенных привилегий при помощи оператора REVOKE.



Когда вы лишаете привилегий пользователя, у которого есть возможность передачи привилегий другим пользователям, то сначала лишается привилегий основной пользователь, а затем лишаются привилегий пользователи, которым он успел передать привилегии. Другими словами, происходит каскадное лишение привилегий.

Синтаксис:

Синтаксис команды REVOKE аналогичен синтаксису команды GRANT, но имеет противоположное значение.

- `privileges` – привилегии на объект базы данных, которых необходимо лишить пользователя (можно перечислять через запятую);
- `schema` – наименование схемы, в которой находится объект;
- `object_name` – имя объекта, к которому необходимо ограничить доступ;
- `user` – имя пользователя, кому будут выданы права доступа.

```
REVOKE privileges ON schema.object_name FROM user;
```

Оператор DENY

Оператор DENY позволяет создать запрет на выполнение определенных действий, даже в том случае, когда пользователь состоит в группе или роли, которая имеет полномочия на выполнение этих действий.



Оператор DENY имеет приоритет над разрешениями (оператор GRANT).

Синтаксис:

- `privileges` – привилегии на объект базы данных, на которые нужно отказать в доступе (можно перечислять через запятую);
- `schema` – наименование схемы, в которой находится объект;
- `object_name` – имя объекта, к которому необходимо ограничить доступ;
- `user` – имя пользователя, кому будет отказано в доступе.

```
DENY privileges ON schema.object_name TO user;
```

Заключение

Дойдя до этой главы, вы проделали сложный путь и проявили упорство. Сейчас вы обладаете основными знаниями языка SQL и можете написать, как простой, так и сложный SQL-запрос, чтобы получить необходимую информацию из таблиц базы данных.

Теперь, вы готовы к самостоятельному изучению таких важных и сложных тем, как: оптимизация SQL-запросов, проектирование, нормализация и администрирование баз данных.

Если вам понравилась книга и вы нашли её полезной и познавательной, хочу попросить вас оставить свой отзыв на странице книги. Для меня это очень важно, и заранее я хочу сказать вам спасибо.



Страница книги

URL-адрес: <https://artemsannikov.ru/books/sql-without-tears>



Страница ВКонтакте

URL-адрес: https://vk.com/topic-38309255_49561618

Полезные ресурсы

Список дополнительных материалов для дальнейшего изучения:

1. официальное руководство PostgreSQL, ссылка: <https://www.postgresql.org/docs/>
2. официальная справочная система WIKI по СУБД PostgreSQL, ссылка: <https://wiki.postgresql.org/>
3. русскоязычная документация от компании СУБД Postgres Pro, ссылка: <https://postgrespro.ru/docs/>
4. пошаговые обучающие уроки по информационным технологиям, ссылка: <https://artemsannikov.ru/>
5. практическое руководство по PostgreSQL, предназначенное для администраторов баз данных и разработчиков приложений, ссылка: <https://www.postgresqltutorial.com/>