**Памятка/шпаргалка по SQL**

[Что такое SQL? 4](#_Toc144743379)

[Почему SQL? 5](#_Toc144743380)

[Процесс SQL 6](#_Toc144743381)

[Команды SQL 7](#_Toc144743382)

[Что такое таблица? 9](#_Toc144743383)

[Что такое поле? 10](#_Toc144743384)

[Что такое запись или строка? 11](#_Toc144743385)

[Что такое колонка? 12](#_Toc144743386)

[Что такое нулевое значение? 13](#_Toc144743387)

[Ограничения 14](#_Toc144743388)

[Целостность данных 15](#_Toc144743389)

[Нормализация БД 16](#_Toc144743390)

[Синтаксис SQL 17](#_Toc144743391)

[Типы данных 20](#_Toc144743392)

[Операторы 25](#_Toc144743393)

[Выражения 29](#_Toc144743394)

[Выражения для работы с датами 34](#_Toc144743395)

[Создание БД 36](#_Toc144743396)

[Удаление БД 38](#_Toc144743397)

[Выбор БД 40](#_Toc144743398)

[Создание таблицы 41](#_Toc144743399)

[Удаление таблицы 43](#_Toc144743400)

[Добавление колонок 44](#_Toc144743401)

[Выборка полей 47](#_Toc144743402)

[Предложение WHERE 49](#_Toc144743403)

[Операторы AND и OR 51](#_Toc144743404)

[Обновление полей 53](#_Toc144743405)

[Удаление записей 54](#_Toc144743406)

[Предложения LIKE и REGEX 55](#_Toc144743407)

[Предложение TOP/LIMIT/ROWNUM 59](#_Toc144743408)

[Предложения ORDER BY и GROUP BY 61](#_Toc144743409)

[Ключевое слово DISTINCT 64](#_Toc144743410)

[Соединения 65](#_Toc144743411)

[Предложение UNION 67](#_Toc144743412)

[Предложение UNION ALL 69](#_Toc144743413)

[Синонимы 70](#_Toc144743414)

[Индексы 73](#_Toc144743415)

[Обновление таблицы 76](#_Toc144743416)

[Очистка таблицы 78](#_Toc144743417)

[Представления 79](#_Toc144743418)

[Создание представления 80](#_Toc144743419)

[Обновление представления 82](#_Toc144743420)

[Удаление представления 84](#_Toc144743421)

[HAVING 85](#_Toc144743422)

[Транзакции 86](#_Toc144743423)

[Временные таблицы 90](#_Toc144743424)

[Клонирование таблицы 91](#_Toc144743425)

[Подзапросы 92](#_Toc144743426)

[Последовательности 95](#_Toc144743427)

# Что такое SQL?

SQL — это язык структурированных запросов (Structured Query Language), позволяющий хранить, манипулировать и извлекать данные из реляционных баз данных (далее — РБД, БД).

# Почему SQL?

SQL позволяет:

* получать доступ к данным в системах управления РБД
* описывать данные (их структуру)
* определять данные в БД и управлять ими
* взаимодействовать с другими языками через модули SQL, библиотеки и предваритальные компиляторы
* создавать и удалять БД и таблицы
* создавать представления, хранимые процедуры (stored procedures) и функции в БД
* устанавливать разрешения на доступ к таблицам, процедурам и представлениям

# Процесс SQL

При выполнении любой SQL-команды в любой RDBMS (Relational Database Management System — система управления РБД, СУБД, например, PostgreSQL, MySQL, MSSQL, SQLite и др.) система определяет наилучший способ выполнения запроса, а движок SQL определяет способ интерпретации задачи.

В данном процессе участвует несколького компонентов:

* диспетчер запросов (Query Dispatcher)
* движок оптимизации (Optimization Engines)
* классический движок запросов (Classic Query Engine)
* движок запросов SQL (SQL Query Engine) и т.д.

Классический движок обрабатывает все не-SQL-запросы, а движок SQL-запросов не обрабатывает логические файлы.

# Команды SQL

Стандартными командами для взаимодействия с РБД являются CREATE, SELECT, INSERT, UPDATE, DELETE и DROP. Эти команды могут быть классифицированы следующим образом:

* DDL — язык определения данных (Data Definition Language)

| **N** | **Команда** | **Описание** |
| --- | --- | --- |
| 1 | CREATE | Создает новую таблицу, представление таблицы или другой объект в БД |
| 2 | ALTER | Модифицирует существующий в БД объект, такой как таблица |
| 3 | DROP | Удаляет существующую таблицу, представление таблицы или другой объект в БД |

* DML — язык изменения данных (Data Manipulation Language)

| **N** | **Команда** | **Описание** |
| --- | --- | --- |
| 1 | SELECT | Извлекает записи из одной или нескольких таблиц |
| 2 | INSERT | Создает записи |
| 3 | UPDATE | Модифицирует записи |
| 4 | DELETE | Удаляет записи |

* DCL — язык управления данными (Data Control Language)

| **N** | **Команда** | **Описание** |
| --- | --- | --- |
| 1 | GRANT | Наделяет пользователя правами |
| 1 | REVOKE | Отменяет права пользователя |

*Обратите внимание*: использование верхнего регистра в названиях команд SQL — это всего лишь соглашение, большинство СУБД нечувствительны к регистру. Тем не менее, форма записи инструкций, когда названия команд пишутся большими буквами, а названия таблиц, колонок и др. — маленькими, позволяет быстро определять назначение производимой с данными операции.

# Что такое таблица?

Данные в СУБД хранятся в объектах БД, называемых таблицами (tables). Таблица, как правило, представляет собой коллекцию связанных между собой данных и состоит из определенного количества колонок и строк.

Таблица — это самая распространенная и простая форма хранения данных в РБД. Вот пример таблицы с пользователями (users):

| **userId** | **userName** | **age** | **city** | **status** |
| --- | --- | --- | --- | --- |
| 1 | Igor | 25 | Moscow | active |
| 2 | Vika | 26 | Ekaterinburg | inactive |
| 3 | Elena | 27 | Ekaterinburg | active |
| 4 | Oleg | 28 | Moscow | inactive |

# Что такое поле?

Каждая таблица состоит из небольших частей — полей (fields). Полями в таблице users являются userId, userName, age, city и status. Поле — это колонка таблицы, предназначенная для хранения определенной информации о каждой записи в таблице.

*Обратите внимание*: вместо userId и userName можно было бы использовать id и name, соответственно. Но при работе с несколькими объектами, содержащими свойство id, бывает сложно понять, какому объекту принадлежит идентификатор, особенно, если вы, как и я, часто прибегаете к деструктуризации. Что касается слова name, то оно часто оказывается зарезервизованным, т.е. уже используется в среде, в которой выполняется код, поэтому я стараюсь его не использовать.

# Что такое запись или строка?

Запись или строка (record/row) — это любое единичное вхождение (entry), существующее в таблице. В таблице users 5 записей. Проще говоря, запись — это горизонтальное вхождение в таблице.

# Что такое колонка?

Колонка (column) — это вертикальное вхождение в таблице, содержащее всю информацию, связанную с определенным полем. В таблице users одной из колонок является city, которая содержит названия городов, в которых проживают пользователи.

# Что такое нулевое значение?

Нулевое значение (NULL) — это значение поля, которое является пустым, т.е. нулевое значение — это значение поля, не имеющего значения. Важно понимать, что нулевое значение отличается от значения 0 и от значения поля, содержащего пробелы (`). Поле с нулевым значением - это такое поля, которое осталось пустым при создании записи. Также, следует учитывать, что в некоторых СУБД пустая строка ('') — этоNULL`, а в некоторых — это разные значения.

# Ограничения

Ограничения (constraints) — это правила, применяемые к данным. Они используются для ограничения данных, которые могут быть записаны в таблицу. Это обеспечивает точность и достоверность данных в БД.

Ограничения могут устанавливаться как на уровне колонки, так и на уровне таблицы.

Среди наиболее распространенных ограничений можно назвать следующие:

* NOT NULL — колонка не может иметь нулевое значение
* DEFAULT — значение колонки по умолчанию
* UNIQUE — все значения колонки должны быть уникальными
* PRIMARY KEY — первичный или основной ключ, уникальный идентификатор записи в текущей таблице
* FOREIGN KEY — внешний ключ, уникальный идентификатор записи в другой таблице (таблице, связанной с текущей)
* CHECK — все значения в колонке должны удовлетворять определенному условию
* INDEX — быстрая запись и извлечение данных

Любое ограничение может быть удалено с помощью команды ALTER TABLE и DROP CONSTRAINT + название ограничения. Некоторые реализации предоставляют сокращения для удаления ограничений и возможность отключать ограничения вместо их удаления.

# Целостность данных

В каждой СУБД существуют следующие категории целостности данных:

* целостность объекта (Entity Integrity) — в таблице не должно быть дубликатов (двух и более строк с одинаковыми значениями)
* целостность домена (Domain Integrity) — фильтрация значений по типу, формату или диапазону
* целостность ссылок (Referential integrity) — строки, используемые другими записями (строки, на которые в других записях имеются ссылки), не могут быть удалены
* целостность, определенная пользователем (User-Defined Integrity) — дополнительные правила

# Нормализация БД

Нормализация — это процесс эффективной организации данных в БД. Существует две главных причины, обуславливающих необходимость нормализации:

* предотвращение записи в БД лишних данных, например, хранения одинаковых данных в разных таблицах
* обеспечение "оправданной" связи между данными

Нормализация предполагает соблюдение нескольких форм. Форма — это формат структурирования БД. Существует три главных формы: первая, вторая и, соответственно, третья. Я не буду вдаваться в подробности об этих формах, при желании, вы без труда найдете необходимую информацию.

# Синтаксис SQL

Синтаксис — это уникальный набор правил и рекомендаций. Все инструкции SQL должны начинаться с ключевого слова, такого как SELECT, INSERT, UPDATE, DELETE, ALTER, DROP, CREATE, USE, SHOW и т.п. и заканчиваться точкой с запятой (;) (точка с запятой не входит в синтаксис SQL, но ее наличия, как правило, требуют консольные клиенты СУБД для обозначения окончания ввода команды). SQL не чувствителен к регистру, т.е. SELECT, select и SeLeCt являются идентичными инструкицями. Исключением из этого правила является MySQL, где учитывается регистр в названии таблицы.

**Примеры синтаксиса**

-- выборка

**SELECT** col1, col2, ...colN

**FROM** tableName;

**SELECT** **DISTINCT** col1, col2, ...colN

**FROM** tableName;

**SELECT** col1, col2, ...colN

**FROM** tableName

**WHERE** **condition**;

**SELECT** col1, col2, ...colN

**FROM** tableName

**WHERE** condition1 **AND**|**OR** condition2;

**SELECT** col2, col2, ...colN

**FROM** tableName

**WHERE** colName **IN** (val1, val2, ...valN);

**SELECT** col1, col2, ...colN

**FROM** tableName

**WHERE** colName **BETWEEN** val1 **AND** val2;

**SELECT** col1, col2, ...colN

**FROM** tableName

**WHERE** colName **LIKE** **pattern**;

**SELECT** col1, col2, ...colN

**FROM** tableName

**WHERE** **condition**

**ORDER** **BY** colName [**ASC**|**DESC**];

**SELECT** SUM(colName)

**FROM** tableName

**WHERE** **condition**

**GROUP** **BY** colName;

**SELECT** COUNT(colName)

**FROM** tableName

**WHERE** **condition**;

**SELECT** SUM(colName)

**FROM** tableName

**WHERE** **condition**

**GROUP** **BY** colName

**HAVING** (**function** **condition**);

-- создание таблицы

**CREATE** **TABLE** tableName (

col1 datatype,

col2 datatype,

...

colN datatype,

**PRIMARY** KEY (одна или более колонка)

);

-- удаление таблицы

**DROP** **TABLE** tableName;

-- создание индекса

**CREATE** **UNIQUE** INDEX indexName

**ON** tableName (col1, col2, ...colN);

-- удаление индекса

**ALTER** **TABLE** tableName

**DROP** INDEX indexName;

-- получение описания структуры таблицы

**DESC** tableName;

-- очистка таблицы

**TRUNCATE** **TABLE** tableName;

-- добавление/удаление/модификация колонок

**ALTER** **TABLE** tableName **ADD**|**DROP**|MODIFY colName [datatype];

-- переименование таблицы

**ALTER** **TABLE** tableName RENAME **TO** newTableName;

-- вставка значений

**INSERT** **INTO** tableName (col1, col2, ...colN)

**VALUES** (val1, val2, ...valN)

-- обновление записей

UPDATE tableName

**SET** col1 = val1, col2 = val2, ...colN = valN

[**WHERE** **condition**];

-- удаление записей

**DELETE** **FROM** tableName

**WHERE** **condition**;

-- создание БД

**CREATE** DATABASE [IF **NOT** **EXISTS**] dbName;

-- удаление БД

**DROP** DATABASE [IF **EXISTS**] dbName;

-- выбор БД

USE dbName;

-- завершения транзакции

**COMMIT**;

-- отмена изменений

**ROLLBACK**;

# Типы данных

Каждая колонка, переменная и выражение в SQL имеют определенный тип данных (data type). Основные категории типов данных:

**Точные числовые**

| **Тип данных** | **От** | **До** |
| --- | --- | --- |
| bigint | -9,223,372,036,854,775,808 | 9,223,372,036,854,775,807 |
| int | -2,147,483,648 | 2,147,483,647 |
| smallint | -32,768 | 32,767 |
| tinyint | 0 | 255 |
| bit | 0 | 1 |
| decimal | -10^38 +1 | 10^38 -1 |
| numeric | -10^38 +1 | 10^38 -1 |
| money | -922,337,203,685,477.5808 | +922,337,203,685,477.5807 |
| smallmoney | -214,748.3648 | +214,748.3647 |

**Приблизительные числовые**

| **Тип данных** | **От** | **До** |
| --- | --- | --- |
| float | -1.79E + 308 | 1.79E + 308 |
| real | -3.40E + 38 | 3.40E + 38 |

**Дата и время**

| **Тип данных** | **От** | **До** |
| --- | --- | --- |
| datetime | Jan 1, 1753 | Dec 31, 9999 |
| smalldatetime | Jan 1, 1900 | Jun 6, 2079 |
| date | Дата сохраняется в виде June 30, 1991 |  |
| time | Время сохраняется в виде 12:30 P.M. |  |

**Строковые символьные**

| **N** | **Тип данных** | **Описание** |
| --- | --- | --- |
| 1 | char | Строка длиной до 8,000 символов (не-юникод символы, фиксированной длины) |
| 2 | varchar | Строка длиной до 8,000 символов (не-юникод символы, переменной длины) |
| 3 | text | Не-юникод данные переменной длины, длиной до 2,147,483,647 символов |

**Строковые символьные (юникод)**

| **N** | **Тип данных** | **Описание** |
| --- | --- | --- |
| 1 | nchar | Строка длиной до 4,000 символов (юникод символы, фиксированной длины) |
| 2 | nvarchar | Строка длиной до 4,000 символов (юникод символы, переменной длины) |
| 3 | ntext | Юникод данные переменной длины, длиной до 1,073,741,823 символов |

**Бинарные**

| **N** | **Тип данных** | **Описание** |
| --- | --- | --- |
| 1 | binary | Данные размером до 8,000 байт (фиксированной длины) |
| 2 | varbinary | Данные размером до 8,000 байт (переменной длины) |
| 3 | image | Данные размером до 2,147,483,647 байт (переменной длины) |

**Смешанные**

| **N** | **Тип данных** | **Описание** |
| --- | --- | --- |
| 1 | timestamp | Уникальные числа, обновляющиеся при каждом изменении строки |
| 2 | uniqueidentifier | Глобально-уникальный идентификатор (GUID) |
| 3 | cursor | Объект курсора |
| 4 | table | Промежуточный результат, предназначенный для дальнейшей обработки |

# Операторы

Оператор (operators) — это ключевое слово или символ, которые, в основном, используются в инструкциях WHERE для выполнения каких-либо операций. Они используются как для определения условий, так и для объединения нескольких условий в инструкции.

В дальнейших примерах мы будем исходить из предположения, что переменная a имеет значение 10, а b — 20.

**Арифметические**

| **Оператор** | **Описание** | **Пример** |
| --- | --- | --- |
| + (сложение) | Сложение значений | a + b = 30 |
| — (вычитание) | Вычитание правого операнда из левого | b — a = 10 |
| \* (умножение) | Умножение значений | a \* b = 200 |
| / (деление) | Деление левого операнда на правый | b / a = 2 |
| % (деление с остатком/по модулю) | Деление левого операнда на правый с остатком (возвращается остаток) | b % a = 0 |

**Операторы сравнения**

| **Оператор** | **Описание** | **Пример** |
| --- | --- | --- |
| = | Определяет равенство значений | a = b -> false |
| != | Определяет НЕравенство значений | a != b -> true |
| <> | Определяет НЕравенство значений | a <> b -> true |
| > | Значение левого операнда больше значения правого операнда? | a > b -> false |
| < | Значение левого операнда меньше значения правого операнда? | a < b -> true |
| >= | Значение левого операнда больше или равно значению правого операнда? | a >= b -> false |
| <= | Значение левого операнда меньше или равно значению правого операнда? | a <= b -> true |
| !< | Значение левого операнда НЕ меньше значения правого операнда? | a !< b -> false |
| !> | Значение левого операнда НЕ больше значения правого операнда? | a !> b -> true |

**Логические операторы**

| **N** | **Оператор** | **Описание** |
| --- | --- | --- |
| 1 | ALL | Сравнивает все значения |
| 2 | AND | Объединяет условия (все условия должны совпадать) |
| 3 | ANY | Сравнивает одно значение с другим, если последнее совпадает с условием |
| 4 | BETWEEN | Проверяет вхождение значения в диапазон от минимального до максимального |
| 5 | EXISTS | Определяет наличие строки, соответствующей определенному критерию |
| 6 | IN | Выполняет поиск значения в списке значений |
| 7 | LIKE | Сравнивает значение с похожими с помощью операторов подстановки |
| 8 | NOT | Инвертирует (меняет на противоположное) смысл других логических операторов, например, NOT EXISTS, NOT IN и т.д. |
| 9 | OR | Комбинирует условия (одно из условий должно совпадать) |
| 10 | IS NULL | Определяет, является ли значение нулевым |
| 11 | UNIQUE | Определяет уникальность строки |

# Выражения

Выражение (expression) — это комбинация значений, операторов и функций для оценки (вычисления) значения. Выражения похожи на формулы, написанные на языке запросов. Они могут использоваться для извлечения из БД определенного набора данных.

Базовый синтаксис выражения выглядит так:

SELECT col1, col2, ...colN

FROM tableName

WHERE [condition|expression];

Существуют различные типы выражений: логические, числовые и выражения для работы с датами.

**Логические**

Логические выражения извлекают данные на основе совпадения с единичным значением.

SELECT col1, col2, ...colN

FROM tableName

WHERE выражение для поиска совпадения с единичным значением;

Предположим, что в таблице users имеются следующие записи:

| **userId** | **userName** | **age** | **city** | **status** |
| --- | --- | --- | --- | --- |
| 1 | Igor | 25 | Moscow | active |
| 2 | Vika | 26 | Ekaterinburg | inactive |
| 3 | Elena | 27 | Ekaterinburg | active |
| 4 | Oleg | 28 | Moscow | Inactive |

Выполняем поиск активных пользователей:

**SELECT** userId, userName as Name

**FROM** users

**WHERE** status = active;

**SELECT** count(\*) as Count

**FROM** users

**WHERE** status = active;

Результат:

| **userId** | **Name** | **age** | **city** | **status** |
| --- | --- | --- | --- | --- |
| 1 | Igor | 25 | Moscow | active |
| 3 | Elena | 27 | Ekaterinburg | active |

**Числовые**

Используются для выполнения арифметических операций в запросе.

**SELECT** numericalExpression **as** operationName

[**FROM** tableName

**WHERE** **condition**];

Простой пример использования числового выражения:

**SELECT** (10 + 5) **AS** addition;

Результат:

| **addition** |
| --- |
| 15 |

Существует несколько встроенных функций, таких как count(), sum(), avg(), min(), max() и др. для выполнения так называемых агрегирующих вычислений данных таблицы или колонки.

**SELECT** COUNT(\*) **AS** records **FROM** users;

Результат:

| **records** |
| --- |
| 4 |

* AVG — вычисляет среднее значение
* SUM — вычисляет сумму значений
* MIN — вычисляет наименьшее значение
* MAX — вычисляет наибольшее значение
* COUNT — вычисляет количество записей в таблице

Также существует несколько встроенных функций для работы со строками:

* CONCAT — объединение строк
* LENGTH — возвращает количество символов в строке
* TRIM — удаляет пробелы в начале и конце строки
* SUBSTRING — извлекает подстроку из строки
* REPLACE — заменяет подстроку в строке
* LOWER — переводит символы строки в нижний регистр
* UPPER — переводит символы строки в верхний регистр и т.д.

с числами:

* ROUND — округляет число
* TRUNCATE — обрезает дробное число до указанного количества знаков после запятой
* CEILING — возвращает наименьшее целое число, которое больше или равно текущему значению
* FLOOR — возвращает наибольшее целое число, которое меньше или равно текущему значению
* POWER — возводит число в указанную степень
* SQRT — возвращает квадратный корень числа
* RAND — генерирует случайное число с плавающей точкой в диапазоне от 0 до 1

# Выражения для работы с датами

Эти выражения, как правило, возвращают текущую дату и время.

**SELECT** CURRENT\_TIMESTAMP;

Результат:

| **Current\_Timestamp** |
| --- |
| 2021-06-20 12:45:00 |

CURRENT\_TIMESTAMP — это и выражение, и функция (CURRENT\_TIMESTAMP()). Другая функция для получения текущей даты и времени — NOW().

Другие функции для получения текущей даты и времени:

* CURDATE/CURRENT\_DATE — возвращает текущую дату
* CURTIME/CURRENT\_TIME — возвращает текущее время и т.д.

Функции для разбора даты и времени:

* DAYOFMONTH(date) — возвращает день месяца в виде числа
* DAYOFWEEK(date) — возвращает день недели в виде числа
* DAYOFYEAR(date) — возвращает номер дня в году
* MONTH(date) — возвращает месяц
* YEAR(date) — возвращает год
* LAST\_DAY(date) — возвращает последний день месяца в виде даты
* HOUR(time) — возвращает час
* MINUTE(time) — возвращает минуты
* SECOND(time) — возвращает секунды и др.

Функции для манипулирования датами:

* DATE\_ADD(date, interval) — выполняет сложение даты и определенного временного интервала
* DATE\_SUB(date, interval) — выполняет вычитание из даты определенного временного интервала
* DATEDIFF(date1, date2) — возвращает разницу в днях между двумя датами
* TO\_DAYS(date) — возвращает количество дней с 0-го дня года
* TIME\_TO\_SEC(time) — возвращает количество секунд с полуночи и др.

Для форматирования даты и времени используются функции DATE\_FORMAT(date, format) и TIME\_FORMAT(date, format), соответственно.

# Создание БД

Для создания БД используется инструкция CREATE DATABASE.

**CREATE** DATABASE dbName;

-- или

**CREATE** DATABASE IF **NOT** **EXISTS** dbName;

Условие IF NOT EXISTS позволяет избежать получения ошибки при попытке создания БД, которая уже существует.

Название БД должно быть уникальным в пределах СУБД.

Создаем БД testDB:

**CREATE** DATABASE testDB;

Получаем список БД:

**SHOW** DATABASES;

Результат:

| **Database** |
| --- |
| information\_schema |
| postgres |
| testDB |

# Удаление БД

Для удаления БД используется инструкция DROP DATABASE.

**DROP** DATABASE dbName;

-- или

**DROP** DATABASE IF **EXISTS** dbName;

Условие IF EXISTS позволяет избежать получения ошибки при попытке удаления несуществующей БД.

Удаляем testDB:

**DROP** DATABASE testDB;

*Обратите внимание*: при удалении БД уничтожаются все данные, которые в ней хранятся, так что будьте предельно внимательны при использовании данной команды.

Проверяем, что БД удалена:

**SHOW** DATABASES;

Для получения списка таблиц используется инструкция SHOW TABLES.

Результат:

| **Database** |
| --- |
| information\_schema |
| postgres |

# Выбор БД

При наличии нескольких БД, перед выполнением каких-либо операций, необходимо выбрать БД. Для этого используется инструкция USE.

USE dbName;

Предположим, что мы не удаляли testDB. Тогда мы можем выбрать ее так:

USE testDB;

# Создание таблицы

Создание таблицы предполагает указание названия таблицы и определение колонок таблицы и их типов данных. Для создания таблицы используется инструкция CREATE TABLE.

**CREATE** **TABLE** tableName (

col1 datatype,

col2 datatype,

...

colN datatype,

**PRIMARY** KEY (хотя бы одна колонка)

);

Для создания таблицы путем копирования другой таблицы используется сочетание CREATE TABLE и SELECT.

Пример создания таблицы users, где первичным ключом являются идентификаторы пользователей, а поля для имени и возраста пользователя не могут быть нулевыми:

**CREATE** **TABLE** users (

userId INT(3),

userName VARCHAR(20) **NOT** **NULL**,

age INT **NOT** **NULL**,

city VARCHAR(20),

status VARCHAR(8),

**PRIMARY** KEY (id)

);

Проверяем, что таблица была создана:

**DESC** users;

Результат:

| **Field** | **Type** | **Null** | **Key** | **Default** | **Extra** |
| --- | --- | --- | --- | --- | --- |
| userId | int(11) | NO | PRI |  |  |
| userName | varchar(20) | NO |  |  |  |
| age | int(11) | NO |  |  |  |
| city | varchar(20) | NO |  |  |  |
| status | varchar(8) | YES |  | NULL |  |

# Удаление таблицы

Для удаления таблицы используется инструкция DROP TABLE.

*Обратите внимание*: при удалении таблицы, навсегда удаляются все хранящиеся в ней данные, индексы, триггеры, ограничения и разрешения, так что будьте предельно внимательны при использовании данной команды.

Удаляем таблицу users:

**DROP** **TABLE** users;

Теперь, если мы попытаемся получить описание users, то получим ошибку:

**DESC** users;

-- ERROR 1146 (42S02): Table 'testDB.users' doesn't exist

# Добавление колонок

Для добавления в таблицу колонок используется инструкция INSERT INTO.

**INSERT** **INTO** tableName (col1, col2, ...colN)

**VALUES** (val1, val2, ...valN);

Названия колонок можно не указывать, однако, в этом случае значения должны перечисляться в правильном порядке.

**INSERT** **INTO** tableName **VALUES** (val1, val2, ...valN);

Во избежание ошибок, рекомендуется всегда перечислять названия колонок.

Предположим, что мы не удаляли таблицу users. Заполним ее пользователями:

**INSERT** **INTO** users (userId, userName, age, city, status)

**VALUES** (1, 'Igor', 25, 'Moscow', 'active');

**INSERT** **INTO** users (userId, userName, age, city, status)

**VALUES** (2, 'Vika', 26, 'Ekaterinburg', 'inactive');

**INSERT** **INTO** users (userId, userName, age, city, status)

**VALUES** (3, 'Elena', 27, 'Ekaterinburg', 'active');

В таблицу можно добавлять несколько строк за один раз.

**INSERT** **INTO** users (userId, userName, age, city, status)

**VALUES**

(1, 'Igor', 25, 'Moscow', 'active'),

(2, 'Vika', 26, 'Ekaterinburg', 'inactive'),

(3, 'Elena', 27, 'Ekaterinburg', 'active');

Также, как было отмечено, при добавлении строки названия полей можно опускать:

**INSERT** **INTO** users

**VALUES** (4, 'Oleg', 28, 'Moscow', 'inactive');

Результат:

| **userId** | **userName** | **age** | **city** | **status** |
| --- | --- | --- | --- | --- |
| 1 | Igor | 25 | Moscow | active |
| 2 | Vika | 26 | Ekaterinburg | inactive |
| 3 | Elena | 27 | Ekaterinburg | active |
| 4 | Oleg | 28 | Moscow | inactive |

**Заполнение таблицы с помощью другой таблицы**

**INSERT** **INTO** tableName [(col1, col2, ...colN)]

**SELECT** col1, col2, ...colN

**FROM** anotherTable

[**WHERE** **condition**];

# Выборка полей

Для выборки полей из таблицы используется инструкция SELECT. Она возвращает данные в виде результирующей таблицы (результирующего набора, result-set).

**SELECT** col1, col2, ...colN

**FROM** tableName;

Для выборки всех полей используется такой синтаксис:

**SELECT** \* **FROM** tableName;

Произведем выборку полей userId, userName и age из таблицы users:

**SELECT** userId, userName, age **FROM** users;

Результат:

| **userId** | **userName** | **age** |
| --- | --- | --- |
| 1 | Igor | 25 |
| 2 | Vika | 26 |
| 3 | Elena | 27 |
| 4 | Oleg | 28 |

# Предложение WHERE

Предложение WHERE используется для фильтрации возвращаемых данных. Оно используется совместно с SELECT, UPDATE, DELETE и другими инструкциями.

**SELECT** col1, col2, ...col2

**FROM** tableName

**WHERE** **condition**;

Условие (condition), которому должны удовлетворять возвращаемые записи, определяется с помощью операторов сравнения или логических операторов типа >, <, =, NOT, LIKE и т.д.

Сделаем выборку полей userId, userName и age активных пользователей:

**SELECT** userId, userName, age

**FROM** users

**WHERE** status = 'active';

Результат:

| **userId** | **userName** | **age** |
| --- | --- | --- |
| 1 | Igor | 25 |
| 3 | Elena | 27 |

Сделаем выборку полей userId, age и city пользователя с именем Vika.

**SELECT** userId, age, city

**FROM** users

**WHERE** userName = 'Vika';

Результат:

| **userId** | **age** | **city** |
| --- | --- | --- |
| 2 | 26 | Ekaterinburg |

*Обратите внимание*: строки в предложении WHERE должны быть обернуты в одинарные кавычки (''), а числа, напротив, указываются как есть.

# Операторы AND и OR

Конъюнктивный оператор AND и дизъюнктивный оператор OR используются для соединения нескольких условий при фильтрации данных.

**AND**

**SELECT** col1, col2, ...colN

**FROM** tableName

**WHERE** condition1 **AND** condition2 ...**AND** conditionN;

Возвращаемые записи должны удовлетворять всем указанным условиям.

Сделаем выборку полей userId, userName и age активных пользователей старше 26 лет:

**SELECT** userId, userName, age

**FROM** users

**WHERE** status = active **AND** age > 26;

Результат:

| **userId** | **userName** | **AGE** |
| --- | --- | --- |
| 3 | Elena | 27 |

**OR**

**SELECT** col1, col2, ...colN

**FROM** tableName

**WHERE** condition1 **OR** condition2 ...**OR** conditionN;

Возвращаемые записи должны удовлетворять хотя бы одному условию.

Сделаем выборку тех же полей неактивных пользователей или пользователей, младше 27 лет:

**SELECT** userId, userName, age

**FROM** users

**WHERE** status = inactive **OR** age < 27;

Результат:

| **userId** | **userName** | **age** |
| --- | --- | --- |
| 1 | Igor | 25 |
| 2 | Vika | 26 |

# Обновление полей

Для обновления полей используется инструкция UPDATE ... SET. Эта инструкция, обычно, используется в сочетании с предложением WHERE.

UPDATE tableName

**SET** col1 = val1, col2 = val2, ...colN = valN

[**WHERE** **condition**];

Обновим возраст пользователя с именем Igor:

UPDATE users

**SET** age = 30

**WHERE** username = 'Igor';

Если в данном случае опустить WHERE, то будет обновлен возраст всех пользователей.

# Удаление записей

Для удаления записей используется инструкция DELETE. Эта инструкция также, как правило, используется в сочетании с предложением WHERE.

**DELETE** **FROM** tableName

[**WHERE** **condition**];

Удалим неактивных пользователей:

**DELETE** **FROM** users

**WHERE** status = 'inactive';

Если в данном случае опустить WHERE, то из таблицы users будут удалены все записи.

# Предложения LIKE и REGEX

**LIKE**

Предложение LIKE используется для сравнения значений с помощью операторов с подстановочными знаками. Существует два вида таких операторов:

* проценты (%)
* нижнее подчеркивание (\_)

% означает 0, 1 или более символов. \_ означает точно 1 символ.

**SELECT** col1, col2, ...colN **FROM** tableName

**WHERE** col **LIKE** 'xxx%'

-- или

**WHERE** col **LIKE** '%xxx%'

-- или

**WHERE** col **LIKE** '%xxx'

-- или

**WHERE** col **LIKE** 'xxx\_'

-- и т.д.

Примеры:

| **N** | **Инструкция** | **Результат** |
| --- | --- | --- |
| 1 | WHERE col LIKE 'foo%' | Любые значения, начинающиеся с foo |
| 2 | WHERE col LIKE '%foo%' | Любые значения, содержащие foo |
| 3 | WHERE col LIKE '\_oo%' | Любые значения, содержащие oo на второй и третьей позициях |
| 4 | WHERE col LIKE 'f*%*%' | Любые значения, начинающиеся с f и состоящие как минимум из 1 символа |
| 5 | WHERE col LIKE '%oo' | Любые значения, оканчивающиеся на oo |
| 6 | WHERE col LIKE '\_o%o' | Любые значения, содержащие o на второй позиции и оканчивающиеся на o |
| 7 | WHERE col LIKE 'f\_o' | Любые значения, содержащие f и o на первой и третьей позициях, соответственно, и состоящие из трех символов |

Сделаем выборку неактивных пользователей:

**SELECT** \* **FROM** users

**WHERE** status **LIKE** 'in%';

Результат:

| **userId** | **userName** | **age** | **city** | **status** |
| --- | --- | --- | --- | --- |
| 2 | Vika | 26 | Ekaterinburg | inactive |
| 4 | Oleg | 28 | Moscow | inactive |

Сделаем выборку пользователей 30 лет и старше:

**SELECT** \* **FROM** users

**WHERE** age **LIKE** '3\_';

Результат:

| **userId** | **userName** | **age** | **city** | **status** |
| --- | --- | --- | --- | --- |
| 1 | Igor | 30 | Moscow | active |

**REGEX**

Предложение REGEX позволяет определять регулярное выражение, которому должна соответствовать запись.

**SELECT** col1, col2, ...colN **FROM** tableName

**WHERE** colName REGEXP регулярное выражение;

В регулярное выражении могут использоваться следующие специальные символы:

* ^ — начало строки
* $ — конец строки
* . — любой символ
* [символы] — любой из указанных в скобках символов
* [начало-конец] — любой символ из диапазона
* | — разделяет шаблоны

Сделаем выборку пользователей с именами Igor и Vika:

**SELECT** \* **FROM** users

**WHERE** userName REGEXP 'Igor|Vika';

Результат:

| **userId** | **userName** | **age** | **city** | **status** |
| --- | --- | --- | --- | --- |
| 1 | Igor | 30 | Moscow | active |
| 2 | Vika | 26 | Ekaterinburg | inactive |

# Предложение TOP/LIMIT/ROWNUM

Данные предложения позволяют извлекать указанное количество или процент записей с начала таблицы. Разные СУБД поддерживают разные предложения.

**SELECT** TOP number|**percent** col1, col2, ...colN

**FROM** tableName

[**WHERE** **condition**];

Сделаем выборку первых трех пользователей:

**SELECT** TOP 3 \* **FROM** users;

Результат:

| **userId** | **userName** | **age** | **city** | **status** |
| --- | --- | --- | --- | --- |
| 1 | Igor | 30 | Moscow | active |
| 2 | Vika | 26 | Ekaterinburg | inactive |
| 3 | Elena | 27 | Ekaterinburg | active |

В mysql:

**SELECT** \* **FROM** users

LIMIT 3, [**offset**];

Параметр offset (смещение) определяет количество пропускаемых записей. Например, так можно извлечь первых двух пользователей, начиная с третьего:

**SELECT** \* **FROM** users

LIMIT 2, 2;

В oracle:

**SELECT** \* **FROM** users

**WHERE** ROWNUM <= 3;

# Предложения ORDER BY и GROUP BY

**ORDER BY**

Предложение ORDER BY используется для сортировки данных по возрастанию (ASC) или убыванию (DESC). Многие СУБД по умолчанию выполняют сортировку по возрастанию.

**SELECT** col1, col2, ...colN

**FROM** tableName

[**WHERE** **condition**]

[**ORDER** **BY** col1, col2, ...colN] [**ASC** | **DESC**];

*Обратите внимание*: колонки для сортировки должны быть указаны в списке колонок для выборки.

Сделаем выборку пользователей, отсортировав их по городу и возрасту:

**SELECT** \* **FROM** users

**ORDER** **BY** city, age;

Результат:

| **userId** | **userName** | **age** | **city** | **status** |
| --- | --- | --- | --- | --- |
| 2 | Vika | 26 | Ekaterinburg | inactive |
| 3 | Elena | 27 | Ekaterinburg | active |
| 1 | Igor | 25 | Moscow | active |
| 4 | Oleg | 28 | Moscow | inactive |

Теперь выполним сортировку по убыванию:

**SELECT** \* **FROM** users

**ORDER** **BY** city, age **DESC**;

Определим собственный порядок сортировки по убыванию:

**SELECT** \* **FROM** users

**ORDER** **BY** (**CASE** city

**WHEN** 'Ekaterinburg' **THEN** 1

**WHEN** 'Moscow' **THEN** 2

**ELSE** 100 **END**) **ASC**, city **DESC**;

**GROUP BY**

Предложение GROUP BY используется совместно с инструкцией SELECT для группировки записей. Оно указывается после WHERE и перед ORDER BY.

**SELECT** col1, col2, ...colN

**FROM** tableName

**WHERE** **condition**

**GROUP** **BY** col1, col2, ...colN

**ORDER** **BY** col1, col2, ...colN;

Сгруппируем активных пользователей по городам:

**SELECT** city, COUNT(city) **AS** amount **FROM** users

**WHERE** status = active

**GROUP** **BY** city

**ORDER** **BY** city;

Результат:

| **city** | **amount** |
| --- | --- |
| Ekaterinburg | 2 |
| Moscow | 2 |

# Ключевое слово DISTINCT

Ключевое слово DISTINCT используется совместно с инструкцией SELECT для возврата только уникальных записей (без дубликатов).

**SELECT** **DISTINCT** col1, col2, ...colN

**FROM** tableName

[**WHERE** **condition**];

Сделаем выборку городов проживания пользователей:

**SELECT** **DISTINCT** city

**FROM** users;

Результат:

| **city** |
| --- |
| Ekaterinburg |
| Moscow |

# Соединения

Соединения (joins) используются для комбинации записей двух и более таблиц.

Предположим, что кроме users, у нас имеется таблица orders с заказами пользователей следующего содержания:

| **orderId** | **date** | **userId** | **amount** |
| --- | --- | --- | --- |
| 101 | 2021-06-21 00:00:00 | 2 | 3000 |
| 102 | 2021-06-20 00:00:00 | 2 | 1500 |
| 103 | 2021-06-19 00:00:00 | 3 | 2000 |
| 104 | 2021-06-18 00:00:00 | 3 | 1000 |

Сделаем выборку полей userId, userName, age и amount из наших таблиц посредством их соединения:

**SELECT** userId, userName, age, amount

**FROM** users, orders

**WHERE** users.userId = orders.userId;

Результат:

| **userId** | **userName** | **age** | **amount** |
| --- | --- | --- | --- |
| 2 | Vika | 26 | 3000 |
| 2 | Vika | 26 | 1500 |
| 3 | Elena | 27 | 2000 |
| 3 | Elena | 27 | 1000 |

При соединении таблиц могут использоваться такие операторы, как =, <, >, <>, <=, >=, !=, BETWEEN, LIKE и NOT, однако наиболее распространенным является =.

Существуют разные типы объединений:

* INNER JOIN — возвращает записи, имеющиеся в обеих таблицах
* LEFT JOIN — возвращает записи из левой таблицы, даже если такие записи отсутствуют в правой таблице
* RIGHT JOIN — возвращает записи из правой таблицы, даже если такие записи отсутствуют в левой таблице
* FULL JOIN — возвращает все записи объединяемых таблиц
* CROSS JOIN — возвращает все возможные комбинации строк обеих таблиц
* SELF JOIN — используется для объединения таблицы с самой собой

# Предложение UNION

Предложение/оператор UNION используется для комбинации результатов двух и более инструкций SELECT. При этом, возвращаются только уникальные записи.

В случае с UNION, каждая инструкция SELECT должна иметь:

* одинаковый набор колонок для выборки
* одинаковое количество выражений
* одинаковые типы данных колонок и
* одинаковый порядок колонок

Однако, они могут быть разной длины.

**SELECT** col1, col2, ...colN

**FROM** table1

[**WHERE** **condition**]

**UNION**

**SELECT** col1, col2, ...colN

**FROM** table2

[**WHERE** **condition**];

Объединим наши таблицы users и orders:

**SELECT** userId, userName, amount, date

**FROM** users

**LEFT** **JOIN** orders

**ON** users.useId = orders.userId

**UNION**

**SELECT** userId, userName, amount, date

**FROM** users

**RIGHT** **JOIN** orders

**ON** users.userId = orders.userId;

Результат:

| **userId** | **userName** | **amount** | **date** |
| --- | --- | --- | --- |
| 1 | Igor | NULL | NULL |
| 2 | Vika | 3000 | 2021-06-21 00:00:00 |
| 2 | Vika | 1500 | 2021-06-20 00:00:00 |
| 3 | Elena | 2000 | 2021-06-19 00:00:00 |
| 3 | Elena | 1000 | 2021-06-18 00:00:00 |
| 4 | Alex | NULL | NULL |

# Предложение UNION ALL

Предложение UNION ALL также используется для объединения результатов двух и более инструкций SELECT. При этом, возвращаются все записи, включая дубликаты.

**SELECT** col1, col2, ...colN

**FROM** table1

[**WHERE** **condition**]

**UNION** **ALL**

**SELECT** col1, col2, ...colN

**FROM** table2

[**WHERE** **condition**];

Существует еще два предложения, похожих на UNION:

* INTERSECT — используется для комбинации результатов двух и более SELECT, но возвращаются только строки из первого SELECT, совпадающие со строками из второго SELECT
* EXCEPT|MINUS — возвращаются только строки из первого SELECT, отсутствующие во втором SELECT

# Синонимы

Синонимы (aliases) позволяют временно изменять названия таблиц и колонок. "Временно" означает, что новое название используется только в текущем запросе, в БД название остается прежним.

Синтаксис синонима таблицы:

**SELECT** col1, col2, ...colN

**FROM** tableName **AS** aliasName

[**WHERE** **condition**];

Синтаксис синонима колонки:

**SELECT** colName **AS** aliasName

**FROM** tableName

[**WHERE** **condition**];

Пример использования синонимов таблиц:

**SELECT** U.userId, U.userName, U.age, O.amount

**FROM** users **AS** U, orders **AS** O

**WHERE** U.userId = O.userId;

Результат:

| **userId** | **userName** | **age** | **amount** |
| --- | --- | --- | --- |
| 2 | Vika | 26 | 3000 |
| 2 | Vika | 26 | 1500 |
| 3 | Elena | 27 | 2000 |
| 3 | Elena | 27 | 1000 |

Пример использования синонимов колонок:

**SELECT** userId **AS** user\_id, userName **AS** user\_name, age **AS** user\_age

**FROM** users

**WHERE** status = active;

Результат:

| **user\_id** | **user\_name** | **user\_age** |
| --- | --- | --- |
| 1 | Igor | 30 |
| 3 | Elena | 27 |

# Индексы

**Создание индексов**

Индексы — это специальные поисковые таблицы (lookup tables), которые используются движком БД в целях более быстрого извлечения данных. Проще говоря, индекс — это указатель или ссылка на данные в таблице.

Индексы ускоряют работу инструкции SELECT и предложения WHERE, но замедляют работу инструкций UPDATE и INSERT. Индексы могут создаваться и удаляться, не оказывая никакого влияния на данные.

Для создания индекса используется инструкция CREATE INDEX, позволяющая определять название индекса, индексируемые колонки и порядок индексации (по возрастанию или по убыванию).

К индексам можно применять ограничение UNIQUE для того, чтобы обеспечить их уникальность.

Синтаксис создания индекса:

**CREATE** INDEX indexName **ON** tableName;

Синтаксис создания индекса для одной колонки:

**CREATE** INDEX indexName

**ON** tableName (colName);

Синтакис создания уникальных индексов (такие индексы используются не только для повышения производительности, но и для обеспечения согласованности данных):

**CREATE** **UNIQUE** INDEX indexName

**ON** tableName (colName);

Синтаксис создания индексов для нескольких колонок (композиционный индекс):

**CREATE** INDEX indexName

**ON** tableName (col1, col2, ...colN);

Решение о создании индексов для одной или нескольких колонок следует принимать на основе того, какие колонки будут часто использоваться в запросе WHERE в качестве условия для сортировки строк.

Для ограничений PRIMARY KEY и UNIQUE автоматически создаются неявные индексы.

**Удаление индексов**

Для удаления индексов используется инструкция DROP INDEX:

**DROP** INDEX indexName;

Несмотря на то, что индексы предназначены для повышения производительности БД, существуют ситуации, в которых их использования лучше избегать.

К таким ситуациям относится следующее:

* индексы не должны использоваться в маленьких таблицах
* в таблицах, которые часто и в большом объеме обновляются или перезаписываются
* в колонках, которые содержат большое количество нулевых значений
* в колонках, над которыми часто выполняются операции

# Обновление таблицы

Команда ALTER TABLE используется для добавления, удаления и модификации колонок существующей таблицы. Также эта команда используется для добавления и удаления ограничений.

Синтаксис:

-- добавление новой колонки

**ALTER** **TABLE** tableName **ADD** colName datatype;

-- удаление колонки

**ALTER** **TABLE** tableName **DROP** **COLUMN** colName;

-- изменение типа данных колонки

**ALTER** **TABLE** tableName MODIFY **COLUMN** colName newDatatype;

-- добавление ограничения `NOT NULL`

**ALTER** **TABLE** tableName MODIFY colName datatype **NOT** **NULL**;

-- добавление ограничения `UNIQUE`

**ALTER** **TABLE** tableName

**ADD** **CONSTRAINT** myUniqueConstraint **UNIQUE** (col1, col2, ...colN);

-- добавление ограничения `CHECK`

**ALTER** **TABLE** tableName

**ADD** **CONSTRAINT** myUniqueConstraint **CHECK** (**condition**);

-- добавление первичного ключа

**ALTER** **TABLE** tableName

**ADD** **CONSTRAINT** myPrimaryKey **PRIMARY** KEY (col1, col2, ...colN);

-- удаление ограничения

**ALTER** **TABLE** tableName

**DROP** **CONSTRAINT** myUniqueContsraint;

-- mysql

**ALTER** **TABLE** tableName

**DROP** INDEX myUniqueContsraint;

-- удаление первичного ключа

**ALTER** **TABLE** tableName

**DROP** **CONSTRAINT** myPrimaryKey;

-- mysql

**ALTER** **TABLE** tableName

**DROP** **PRIMARY** KEY;

Добавляем в таблицу users новую колонку — пол пользователя:

**ALTER** **TABLE** users **ADD** sex char(1);

Удаляем эту колонку:

**ALTER** **TABLE** users **DROP** sex;

# Очистка таблицы

Команда TRUNCATE TABLE используется для очистки таблицы. Ее отличие от DROP TABLE состоит в том, что сохраняется структура таблицы (DROP TABLE полностью удаляет таблицу и все ее данные).

**TRUNCATE** **TABLE** tableName;

Очищаем таблицу users:

**TRUNCATE** **TABLE** users;

Проверяем, что users пустая:

**SELECT** \* **FROM** users;

-- Empty set (0.00 sec)

# Представления

Представление (view) — это не что иное, как инструкция, записанная в БД под определенным названием. Другими словами, представление — это композиция таблицы в форме предварительно определенного запроса.

Представления могут содержать все или только некоторые строки таблицы. Представление может быть создано на основе одной или нескольких таблиц (это зависит от запроса для создания представления).

Представления — это виртутальные таблицы, позволяющие делать следующее:

* структурировать данные способом, который пользователи находят наиболее естественным или интуитивно понятным
* ограничивать доступ к данным таким образом, что пользователь может просматривать и (иногда) модифицировать только то, что ему нужно и ничего более
* объединять данные из нескольких таблиц для формирования отчетов

# Создание представления

Для создания представления используется инструкция CREATE VIEW. Как было отмечено, представления могут создаваться на основе одной или нескольких таблиц, и даже на основе другого представления.

**CREATE** **VIEW** viewName **AS**

**SELECT** col1, col2, ...colN

**FROM** tableName

[**WHERE** **condition**];

Создаем представление для имен и возраста пользователей:

**CREATE** **VIEW** usersView **AS**

**SELECT** userName, age

**FROM** users;

Получаем данные с помощью представления:

**SELECT** \* **FROM** usersView;

Результат:

| **userName** | **age** |
| --- | --- |
| Igor | 30 |
| Vika | 26 |
| Elena | 27 |
| Oleg | 28 |

**WITH CHECK OPTION**

WITH CHECK OPTION — это настройка инструкции CREATE VIEW. Она позволяет обеспечить соответствие всех UPDATE и INSERT условию, определенном в представлении.

Если условие не удовлетворяется, выбрасывается исключение.

**CREATE** **VIEW** usersView **AS**

**SELECT** userName, age

**FROM** users

**WHERE** age **IS** **NOT** **NULL**

**WITH** **CHECK** OPTION;

# Обновление представления

Представление может быть обновлено при соблюдении следующих условий:

* SELECT не содержит ключевого слова DISTINCT
* SELECT не содержит агрегирующих функций
* SELECT не содержит функций установки значений
* SELECT не содержит операций установки значений
* SELECT не содержит предложения ORDER BY
* FROM не содержит больше одной таблицы
* WHERE не содержит подзапросы
* запрос не содержит GROUP BY или HAVING
* вычисляемые колонки не обновляются
* все ненулевые колонки из базовой таблицы включены в представление в том же порядке, в каком они указаны в запросе INSERT

Пример обновления возраста пользователя с именем Igor в представлении:

UPDATE usersView

**SET** age = 31

**WHERE** userName = 'Igor';

*Обратите внимание*: обновление строки в представлении приводит к ее обновлению в базовой таблице.

В представление могут добавляться новые строки с помощью команды INSERT. При выполнении этой команды должны соблюдаться те же правила, что и при выполнении команды UPDATE.

С помощью команды DELETE можно удалять строки из представления.

Удаляем из представления пользователя, возраст которого составляет 26 лет:

**DELETE** **FROM** usersView

**WHERE** age = 26;

*Обратите внимание*: удаление строки в представлении приводит к ее удалению в базовой таблице.

# Удаление представления

Для удаления представления используется инструкция DROP VIEW:

**DROP** **VIEW** viewName;

Удаляем представление usersView:

**DROP** **VIEW** usersView;

HAVING

Предложение HAVING используется для фильтрации результатов группировки. WHERE используется для применения условий к колонкам, а HAVING — к группам, созданным с помощью GROUP BY.

HAVING должно указываться после GROUP BY, но перед ORDER BY (при наличии).

**SELECT** col1, col2, ...colN

**FROM** table1, table2, ...tableN

[**WHERE** **condition**]

**GROUP** **BY** col1, col2, ...colN

**HAVING** **condition**

**ORDER** **BY** col1, col2, ...colN;

# Транзакции

Транзакция — это единица работы или операции, выполняемой над БД. Это последовательность операций, выполняемых в логическом порядке. Эти операции могут запускаться как пользователем, так и какой-либо программой, функционирующей в БД.

Транзакция — это применение одного или более изменения к БД. Например, при создании/обновлении/удалении записи мы выполняем транзакцию. Важно контролировать выполнение таких операций в целях обеспечения согласованности данных и обработки возможных ошибок.

На практике, запросы, как правило, не отправляются в БД по одному, они группируются и выполняются как часть транзакции.

**Свойства транзакции**

Транзакции имеют 4 стандартных свойства (ACID):

* атомарность (atomicity) — все операции транзакции должны быть успешно завершены. В противном случае, транзакция прерывается, а все изменения отменяются (происходит откат к предыдущему состоянию)
* согласованность (consistency) — состояние должно изменться в полном соответствии с операциями транзакции
* изоляция или автономность (isolation) — транзакции не зависят друг от друга и не оказывают друг на друга никакого влияния
* долговечность (durability) — результат звершенной транзакции должен сохраняться при поломке системы

**Управление транзакцией**

Для управления транзакцией используются следующие команды:

* BEGIN|START TRANSACTION — запуск транзакции
* COMMIT — сохранение изменений
* ROLLBACK — отмена изменений
* SAVEPOINT — контрольная точка для отмены изменений
* SET TRANSACTION — установка характеристик текущей транзакции

Команды для управления транзакцией могут использоваться только совместно с такими запросами как INSERT, UPDATE и DELETE. Они не могут использоваться во время создания и удаления таблиц, поскольку эти операции автоматически отправляются в БД.

Удаляем пользователя, возраст которого составляет 26 лет, и отправляем изменения в БД:

**BEGIN** TRANSACTION

**DELETE** **FROM** users

**WHERE** age = 26;

**COMMIT**;

Удаляем пользователя с именем Oleg и отменяем эту операцию:

**BEGIN**

**DELETE** **FROM** users

**WHERE** username = 'Oleg';

**ROLLBACK**;

Контрольные точки создаются с помощью такого синтаксиса:

**SAVEPOINT** savepointName;

Возврат к контрольной точке выполняется так:

**ROLLBACK** **TO** savepointName;

Выполняем три запроса на удаление данных из users, создавая контрольные точки перед каждый удалением:

**START** TRANSACTION

**SAVEPOINT** sp1;

**DELETE** **FROM** users

**WHERE** age = 26;

**SAVEPOINT** sp2;

**DELETE** **FROM** users

**WHERE** userName = 'Oleg';

**SAVEPOINT** sp3;

**DELETE** **FROM** users

**WHERE** status = 'inactive';

Отменяем два последних удаления, возвращаясь к контрльной точке sp2, созданной после первого удаления:

**ROLLBACK** **TO** sp2;

Делаем выборку пользователей:

**SELECT** \* **FROM** users;

Результат:

| **userId** | **userName** | **age** | **city** | **status** |
| --- | --- | --- | --- | --- |
| 1 | Igor | 31 | Moscow | active |
| 3 | Elena | 27 | Ekaterinburg | active |
| 4 | Oleg | 28 | Moscow | inactive |

Как видим, из таблицы был удален только пользователь с возрастом 26 лет.

Для удаление контрольной точки используется команда RELEASE SAVEPOINT. Естественно, после удаления контрольной точки, к ней нельзя будет вернуться с помощью ROLLBACK TO.

Команда SET TRANSACTION используется для инициализации транзакции, т.е. начала ее выполнения. При этом, можно определять некоторые характеристики транзакции. Например, так можно определить уровень доступа транзакции (доступна только для чтения или для записи тоже):

**SET** TRANSACTION [READ WRITE | READ **ONLY**];

# Временные таблицы

Некоторые СУБД поддерживают так называемые временные таблицы (temporary tables). Такие таблицы позволяют хранить и обрабатывать промежуточные результаты с помощью таких же запросов, как и при работе с обычными таблицами.

Временные таблицы могут быть очень полезными при необходимости хранения временных данных. Одной из главных особенностей таких таблиц является то, что они удаляются по завершении текущей сессии. При запуске скрипта временная таблица удаляется после завершения выполнения этого скрипта. При доступе к БД с помощью клиентской программы, такая таблица будет удалена после закрытия этой программы.

Временная таблица создается с помощью инструкции CREATE TEMPORARY TABLE, в остальном синтаксис создания таких таблиц идентичен синтаксису создания обычных таблиц.

Временная таблица удаляется точно также, как и обычная таблица, с помощью инструкции DROP TABLE.

# Клонирование таблицы

Может возникнуть ситуация, когда потребуется получить точную копию существующей таблицы, а CREATE TABLE или SELECT окажется недостаточно в силу того, что мы хотим получить не только идентичную структуру, но также индексы, значения по умолчанию и т.д. копируемой таблицы.

В mysql, например, это можно сделать так:

* вызываем команду SHOW CREATE TABLE для получения инструкции, выполненной при создании таблицы, включая индексы и прочее
* меняем название таблицы и выполняем запрос. Получаем точную копию таблицы
* опционально: если требуется содержимое копируемой таблицы, можно также использовать инструкции INSERT INTO или SELECT

# Подзапросы

Подзапрос — это внутренний (вложенный) запрос другого запроса, встроенный (вставленный) с помощью WHERE или других инструкций.

Подзапрос используется для получения данных, которые будут использованы основным запросом в качестве условия для фильтрации возвращаемых записей.

Подзапросы могут использоваться в инструкциях SELECT, INSERT, UPDATE и DELETE, а также с операторами =, <, >, >=, <=, IN, BETWEEN и т.д.

Правила использования подзапросов:

* они должны быть обернуты в круглые скобки
* подзапрос должен содержать только одну колонку для выборки, если основной запрос не содержит несколько таких колонок, которые сравниваются в подзапросе
* в подзапросе нельзя использовать команду ORDER BY, это можно сделать в основном запросе. В подзапросе для замены ORDER BY можно использовать GROUP BY
* подзапросы, возвращающие несколько значений, могут использоваться только с операторами, которые работают с наборами значений, такими как IN
* список SELECT не может содержать ссылки на значения, которые оцениваются (вычисляются) как BLOB, ARRAY, CLOB или NCLOB
* подзапрос не может быть сразу передан в функцию для установки значений
* команду BETWEEN нельзя использовать совместно с подзапросом. Тем не менее, в самомподзапросе указанную команду использовать можно

Подзапросы, обычно, используются в инструкции SELECT.

**SELECT** col1, col2, ...colN

**FROM** table1, table2, ...tableN

**WHERE** colName operator

(**SELECT** col1, col2, ...colN

**FROM** table1, table2, tableN

[**WHERE** **condition**]);

Пример:

**SELECT** \* **FROM** users

**WHERE** userId **IN** (

**SELECT** userId **FROM** users

**WHERE** status = 'active'

);

Результат:

| **userId** | **userName** | **age** | **city** | **status** |
| --- | --- | --- | --- | --- |
| 1 | Igor | 30 | Moscow | active |
| 3 | Elena | 27 | Ekaterinburg | active |

Подзапросы могут использоваться в инструкции INSERT. Эта инструкция добавляет в таблицу данные, возвращаемые подзапросом. При этом, данные, возвращаемые подзапросом, могут быть модифицированы любыми способами.

**INSERT** **INTO** tableName col1, col2, ...colN

**SELECT** col1, col2, ...colN

**FROM** table1, table2, ...tableN

[**WHERE** operator [**value**]];

Подзапросы могут использоваться в инструкции UPDATE. При этом, данные из подзапроса могут использоваться для обновления любого количества колонок.

UPDATE tableName

**SET** col = newVal

[**WHERE** operator [**value**]

(

**SELECT** colName

**FROM** tableName

[**WHERE** **condition**]

)

];

Данные, возвращаемые подзапросом, могут использоваться и для удаления записей.

**DELETE** **FROM** tableName

[**WHERE** operator [**value**]

(

**SELECT** colName

**FROM** tableName

[**WHERE** **condition**]

)

];

# Последовательности

Последовательность — это набор целых чисел (1, 2, 3 и т.д.), генерируемых автоматически. Последовательности часто используются в БД, поскольку многие приложения нуждаются в уникальных значениях, используемых для идентификации строк.

Приведенные ниже примеры рассчитаны на mysql.

Простейшим способом определения последовательности является использование AUTO\_INCREMENT при создании таблицы:

**CREATE** **TABLE** tableName (

id INT UNSIGNED **NOT** **NULL** AUTO\_INCREMENT,

**PRIMARY** KEY (id),

-- другие строки

);

Для того, чтобы заново пронумеровать строки с помощью автоматически генерируемых значений (например, при удалении большого количества строк), можно удалить колонку, содержащую такие значения и создать ее заново. *Обратите внимание*: такая таблица не должна быть частью объединения.

**ALTER** **TABLE** tableName **DROP** id;

**ALTER** **TABLE** tableName

**ADD** id INT UNSIGNED **NOT** **NULL** AUTO\_INCREMENT **FIRST**,

**ADD** **PRIMARY** KEY (id);

По умолчанию значения, генерируемые с помощью AUTO\_INCREMENT, начинаются с 1. Для того, чтобы установить другое начальное значение достаточно указать, например, AUTO\_INCREMENT = 100 — в этом случае нумерация строк начнется со 100.