



# План занятия

1. [Проблемы](#)
2. [SQL](#)
3. [SQLite](#)
4. [SQLite в Android](#)
5. [Итоги](#)



# ПРОБЛЕМЫ

# ПРОБЛЕМА

До этого мы посмотрели, как организовать постоянное хранение данных с помощью файлов (включая SharedPreferences).

Но оба способа не совсем подходят для полноценной работы, поскольку:

- данные из памяти стираются после завершения процесса;
- файлы не предоставляют удобных механизмов поиска, замены и обновления (если вопрос с поиском мы ещё можем эффективно решить с использованием Sequences, то с остальным сложнее)



# ПРОБЛЕМА

Другой вопрос: действительно ли это является проблемой? Если вы делаете приложение, которое получает данные по сети и без сети не работает (совсем), то, конечно же, и проблемы не существует.

Но если вы делаете приложение, которое способно обрабатывать данные и при проблемах с сетью (или вообще не предполагает использования сервера, таких мало, но), то стоит задуматься о хранении данных.



# ХРАНИЛИЩА ДАННЫХ

Мы можем посмотреть в сторону готовых решений, которые бы обеспечивали нужные нам функции.

В качестве первого класса таких систем рассмотрим системы управления базами данных (СУБД), которые часто называют реляционными или SQL\* (мы будем называть SQL).

Примечание\*: современные инструменты часто реализуют такое количество возможностей, что их трудно причислить только к одному классу. Кроме того, сами определения уже давно настолько "размылись" и используются по-разному.

---

# ХРАНИЛИЩА ДАННЫХ

Эти решения обеспечивают нужные нам функции, а также:

- структуру и консистентность информации – защита от того, что приложение внесёт неправильные данные или в неправильном формате;
- запросы – возможность получать данные по различным условиям, обновлять и удалять (т.е. мы можем изменить конкретную запись а не целиком перезаписывать файл)
- транзакционность – выполнение неатомарных операций в виде одного атомарного блока и другие.

Мы переносим часть задач со своей системы на стороннюю, при этом нам придётся научиться ею пользоваться.



## СУБД vs БД

Само хранилище данных называют базой данных (БД), а систему, управляющую им, – СУБД.

Достаточно часто эти термины используют как взаимозаменяемые, поэтому мы не будем заострять внимание на «единственно верных определениях», а просто будем их использовать.



**SQL**



---

# SQL

SQL – общее название языка, используемого в SQL базах данных.

Зачастую он подразделяется на:

- DDL – Data Definition Language
- DML – Data Manipulation Language
- DRL (или DQL) – Data Retrieval (Query) Language

Есть ещё ряд других видов, но нам будет достаточно этих.



# СТАНДАРТЫ и ДИАЛЕКТЫ

На язык SQL разрабатываются стандарты, но ключевое – хоть большинство БД и поддерживает стандарты (либо их часть), почти каждая обладает собственным диалектом.

**Диалект** – это некоторые особенности синтаксиса конкретной БД, возможно, неподдерживаемые в других.



# SQLITE

Android включает в состав платформы [SQLite](#) — библиотеку, которая и предоставляет возможность использовать SQL для доступа к данным из приложения.

# SQL

Вы можете представлять себе SQL базы данных как набор одномерных таблиц, которые друг с другом связаны отношениями.

Давайте начнём с одной:

столбец  
(колонка)



строка



id	author	content	likes	likedByMe
1	Netology	Привет, это ...	100	true
2	Netology	Новые курсы!	999	true

Под термином строка мы будем понимать набор данных (а не строковый тип).

# СТРУКТУРА

число	текст	текст	число	boolean
id	author	content	likes	likedByMe
1	Netology	Привет, это ...	100	true
2	Netology	Новые курсы!	999	true

Структура данных подразумевает, что в эту таблицу можно записать только строки, состоящие из этих 5 полей (как и в объекты Kotlin).

При этом у каждого столбца есть:

1. Тип данных (эти типы отличаются от типов в Kotlin)
2. Название
3. Порядок

# DOMAIN

число	текст	текст	число	boolean
id	author	content	likes	likedByMe
1	Netology	Привет, это ...	100	true
2	Netology	Новые курсы!	999	true

А теперь давайте внимательно посмотрим на таблицу.

*Вопрос: есть ли какие-то ограничения на колонки, кроме непосредственно типа данных?*

---

# DOMAIN

СУБД – достаточно мощный инструмент. Она позволяет установить ограничения на тип данных. Например, мы можем потребовать, чтобы:

- **id** был уникален в рамках всей таблицы и всегда задан;
- **author** всегда был задан и не может быть пустой строкой;
- **likes** был всегда больше или равен 0 (если нельзя уводить посты в минус).

Подобные ограничения (в рамках типа данных) называются доменом.



# PRIMARY KEY



# PRIMARY KEY

id	author	content	likes	likedByMe
1	Netology	Привет, это ...	100	true
2	Netology	Новые курсы!	999	true

**Primary Key** (первичный ключ, далее РК) – столбец или набор столбцов, которые уникальным образом идентифицируют строку в рамках всей таблицы. Обычно делают в виде одного столбца и называют id.

Ключевое:

- первичный ключ всегда задан
- первичный ключ уникален
- первичный ключ не может быть NULL (см.далее)



# PRIMARY KEY

В большинстве случаев используют суррогатный первичный ключ (т.е. искусственно придуманный).

В простейших случаях это будет целое число, которое сама СУБД будет увеличивать автоматически (а если данные приходят с сервера, то никакого автоувеличения не нужно)

Именно поэтому мы вас с первых лекций приучали использовать `id` в структурах (потому что при хранении в БД он почти всегда будет присутствовать\*).

Примечание\*: наличие первичного ключа — необязательное требование для таблицы, но в большинстве таблиц он будет.



# FOREIGN KEY

## FOREIGN KEY

id	author	content	likes	likedByMe
1	<b>Netology</b>	Привет, это ...	100	true
2	<b>Netology</b>	Новые курсы!	999	true
3	<b>Me</b>	Круто!	0	false

*Вопрос: как вы думаете, записи 1 и 2 принадлежат одному и тому же аккаунту?*

## FOREIGN KEY

id	author	content	likes	likedByMe
1	<b>Netology</b>	Привет, это ...	100	true
2	<b>Netology</b>	Новые курсы!	999	true
3	<b>Me</b>	Круто!	0	false

На самом деле, мы не знаем, поскольку информации недостаточно. В любой соц.сети можно найти людей с одинаковыми именами (или псевдонимами).



## FOREIGN KEY

В БД есть Foreign Key (внешний ключ, далее — FK): мы можем ссылаться из одной таблицы на колонку другой РК (но некоторые СУБД разрешают не только на РК).

# FOREIGN KEY

posts

id	...	author_id
1	...	1
2	...	1
3	...	2

**author\_id** — обычный столбец с обычными данными (числовыми), но с одним **ограничением**: СУБД будет проверять, что записать туда можно **только такое** число, которое **есть в колонке id** таблицы **users**.

users

id	name	...
1	Netology	...
2	Me	...

То же самое с удалением\*: СУБД не даст просто так удалить из **users** запись, если на неё ссылаются из другой таблицы.

Примечание\*: конечно же, это настраивается.

# FOREIGN KEY

posts

id	...	author_id
1	...	1
2	...	1
3	...	2

comments

id	author_id	post_id
1	2	3
2	2	3

Подобными связями можно соединить множество таблиц. Но FK (Foreign Key) всегда может указывать только на одну запись и только из конкретной таблицы.

users

id	name	...
1	Netology	...
2	Me	...





# CONSTRAINTS

# CONSTRAINTS

Constraints – это ограничения, которые мы можем установить на колонку или группу колонок.

Они могут быть следующие:

- **CHECK (<expr>)** – проверка определённого выражения на истину;
- **UNIQUE** – проверка на уникальность значения во всей таблице;
- **NOT NULL** – значение не может быть **NULL**;
- **PRIMARY KEY**;
- **FOREIGN KEY**.

Здесь возникает вопрос: чем **UNIQUE** отличается от **PRIMARY KEY** и что такое **NULL**?



**NULL**

# NULL

В рамках SQL баз данных, **NULL** – это специальный маркер, означающий, что значение в буквальном смысле «не задано» (не путайте с null в Kotlin). В любой столбец (кроме PK) можно записать **NULL**.

**NULL** – **уникальное** значение, оно **не равно ничему, включая самого себя**. Так вот **NOT NULL** говорит, что в столбец нельзя записать **NULL**.

Отличие PK от UNIQUE в том, что в PK нельзя\* записать NULL, а в UNIQUE можно (поскольку NULL уникально и не равно самому себе).

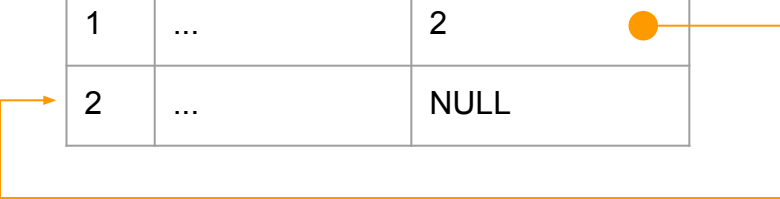
Примечание\*: по стандарту, но есть нестандартные реализации.

# NULL

NULL также можно записать в FK. Это будет означать, что текущей строке нет соответствия в другой таблице (причём другой таблицей может быть та же самая таблица):

posts

id	...	source_id
1	...	2
2	...	NULL



Пример с репостом



# SQLITE



# SQLITEBROWSER

В качестве инструмента, позволяющего потренироваться в использовании SQL и SQLite в частности, мы будем использовать [SQLiteBrowser](#).



# ПРОЦЕСС РАБОТЫ

Весь процесс строится из двух частей:

1. Вы определяете схему базы данных (т.е. структуру информации).
2. Вы осуществляете запросы к базе данных (создание записей, обновление, удаление, выборка)

Всё почти так же, как в Kotlin: мы сначала описываем классы, а потом создаём экземпляры этого класса и работаем с ними.





**DDL**

# CREATE, DROP, ALTER

В рамках этой лекции мы будем работать только с таблицами, поэтому всё будем рассматривать по отношению к ним.

Под «таблицей» мы будем иметь в виду структуру таблицы, а «записи в таблице» – данные в таблице.

Нас интересуют два запроса:

- **CREATE TABLE** – создание таблицы
- **DROP TABLE** – удаление таблицы

# ВОПРОСЫ ИМЕНОВАНИЯ

В отличие от Kotlin, при работе с базами данных существуют разные схемы именования. Например, таблицу с постами можно назвать:

- Post
- Posts (во множественном числе) ← мы будем использовать

Важно: в SQLite названия таблиц регистронезависимы.

А имена, состоящие из нескольких слов, например authorId, так:

- authorId ← мы будем использовать
- author\_id

# CREATE TABLE

```
CREATE TABLE <tbl_name> (  
    <col_name> <type> <col_constraints>,  
    <col_name> <type> <col_constraints> ← запятая не ставится  
); ← ставится точка с запятой
```

**CREATE TABLE** позволяет создать таблицу в БД.

**Важно:** повторный вызов запроса приведёт к ошибке, т.к. таблица уже будет существовать.

---

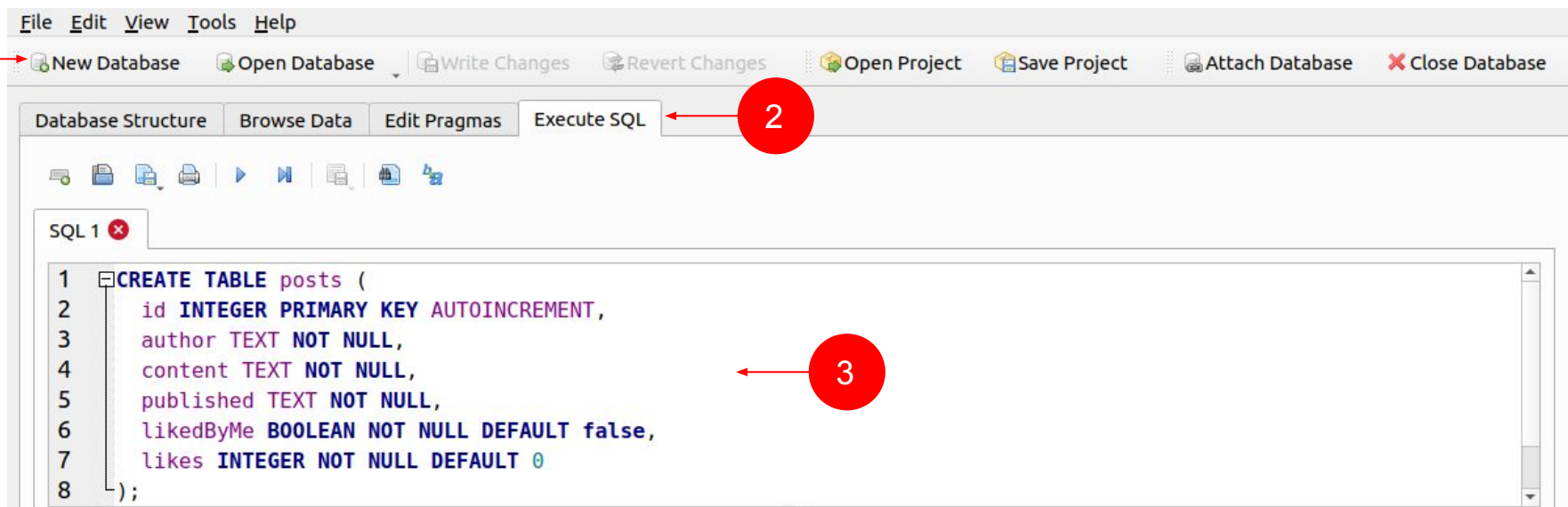
# ТИПЫ ДАННЫХ

В SQLite не так уж много [типов данных](#) (причём часть из них является синонимами других типов). Нас будут интересовать следующие:

- **INTEGER** — целое число
- **TEXT** — текст
- **BOOLEAN** — true/false

# ИСПОЛЬЗОВАНИЕ

В SQLiteBrowser выбираете «создать новую базу данных» (выбираете любой файл, после чего переходите на вкладку Execute SQL):



Чтобы выполнить запрос: **Ctrl + Enter**.

# DATABASE STRUCTURE

После этого на вкладке **Database Structure** можно посмотреть структуру созданной таблицы, а на **Browse Data** сами данные (пока там ничего нет):

Database Structure		
Browse Data   Edit Pragmas   Execute SQL		
Create Table   Create Index   Modify Table   Delete Table   Print		
Name	Type	Schema
▼ Tables (2)		
▼ posts		CREATE TABLE posts ( id INTEGER PRIMARY KEY AUTOINCREMENT, author TEXT NOT NULL, content TEXT NOT NU
id	INTEGER	"id" INTEGER PRIMARY KEY AUTOINCREMENT
author	TEXT	"author" TEXT NOT NULL
content	TEXT	"content" TEXT NOT NULL
published	TEXT	"published" TEXT NOT NULL
likedByMe	BOOLEAN	"likedByMe" BOOLEAN NOT NULL DEFAULT false
likes	INTEGER	"likes" INTEGER NOT NULL DEFAULT 0



# DROP TABLE

**DROP TABLE** <tbl\_name>;

**DROP TABLE** позволяет целиком удалить таблицу (включая записи таблицы).

**Важно:** будьте с этим запросом **очень осторожны!** Данные восстановить уже не получится.





**DML**



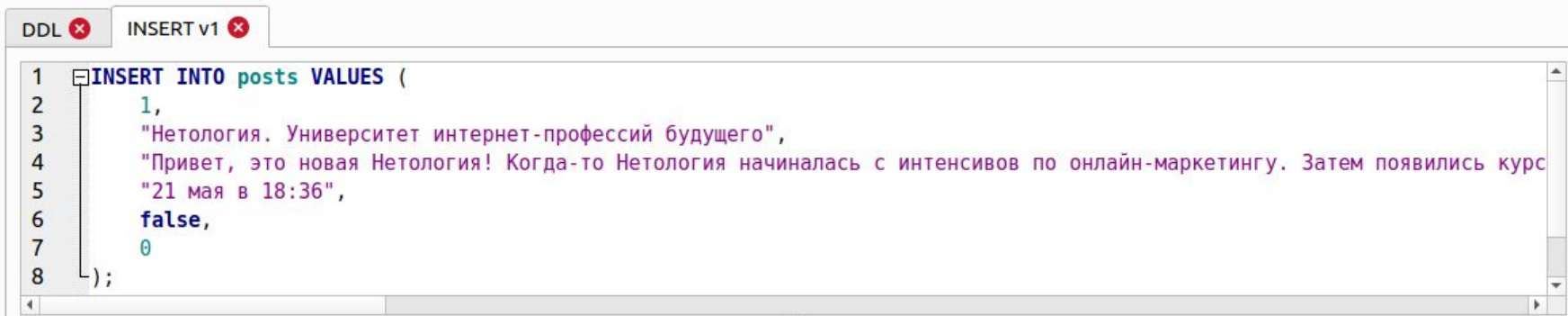
# DML

Мы будем рассматривать три ключевых запроса:

- **INSERT** (вставка данных)
- **UPDATE** (обновление данных)
- **DELETE** (удаление данных)

# INSERT

Первая форма **INSERT**:

A screenshot of a SQL IDE window. The window has two tabs: 'DDL' and 'INSERT v1'. The 'INSERT v1' tab is active. The SQL editor shows the following code:

```
1 INSERT INTO posts VALUES (  
2     1,  
3     "Нетология. Университет интернет-профессий будущего",  
4     "Привет, это новая Нетология! Когда-то Нетология начиналась с интенсивов по онлайн-маркетингу. Затем появились курс  
5     "21 мая в 18:36",  
6     false,  
7     0  
8 );
```

Эта форма используется крайне редко по нескольким причинам:

1. Мы обязаны сами «считать» id (ни о какой автогенерации речи нет)
2. Мы обязаны заполнять все поля

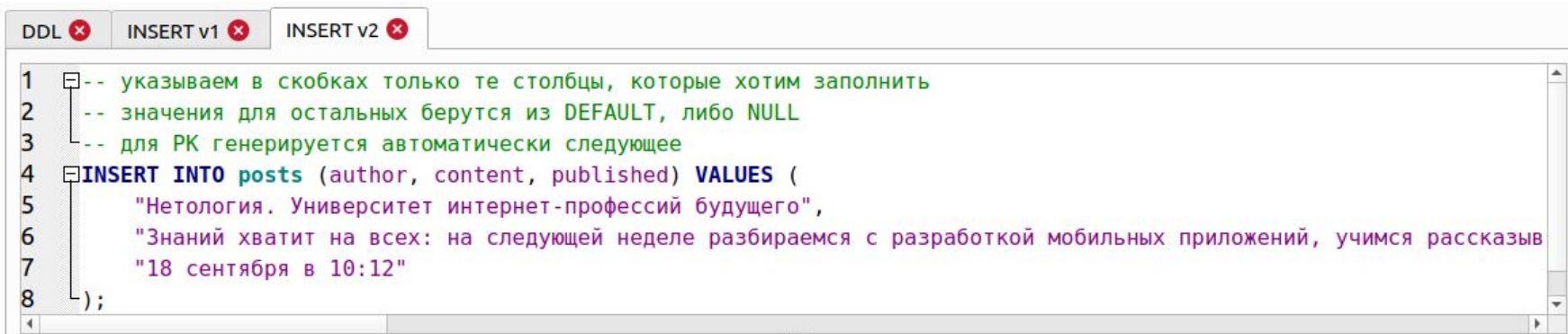
# INSERT

Database Structure   Browse Data   Edit Pragmas   Execute SQL					
Table: posts				New Record	Delete Record
id	author	content	published	likedByMe	likes
Filter	Filter	Filter	Filter	Filter	Filter
1 1	Нетология. Университет интернет-профессий будущего	Привет, это новая Нетология! ...	21 мая в 18:36	0	0

Несмотря на то, что мы указали, что likedByMe — это BOOLEAN, реально это поле будет храниться как INTEGER со значениями 0 и 1.

# INSERT

Вторая форма **INSERT** (наиболее частая):

A screenshot of a SQL IDE window with three tabs: 'DDL', 'INSERT v1', and 'INSERT v2'. The 'INSERT v2' tab is active, displaying a SQL statement. The statement is an INSERT INTO query for a table named 'posts'. It specifies columns 'author', 'content', and 'published'. The values are: 'Нетология. Университет интернет-профессий будущего', 'Знаний хватит на всех: на следующей неделе разбираемся с разработкой мобильных приложений, учимся рассказыв', and '18 сентября в 10:12'. There are also three green comments at the top of the statement explaining the column and value handling.

```
1 -- указываем в скобках только те столбцы, которые хотим заполнить
2 -- значения для остальных берутся из DEFAULT, либо NULL
3 -- для PK генерируется автоматически следующее
4 INSERT INTO posts (author, content, published) VALUES (
5     "Нетология. Университет интернет-профессий будущего",
6     "Знаний хватит на всех: на следующей неделе разбираемся с разработкой мобильных приложений, учимся рассказыв
7     "18 сентября в 10:12"
8 );
```

При этом id будет теперь генерироваться автоматически.

Существует ещё третья форма, но поскольку в коде она используется редко, мы рассматривать её не будем.

# DELETE

Для удаления данных предназначен запрос **DELETE** (важно: **DROP** удаляет таблицу вместе с записями, **DELETE** удаляет только записи из таблицы).

Общий формат вот такой:

**DELETE FROM <tbl\_name> WHERE <expr>;**



необязательно, но лучше привыкнуть всегда писать,  
иначе аккуратно «почистите» всю таблицу

# WHERE

в таком виде будете использовать чаще всего

```
DELETE FROM posts WHERE id = 1;
```

Условие **WHERE** может содержать проверки:

1. На равенство **id = 1** (обратите внимание **одно равно**)
2. На неравенство **id != 1** или **id <> 1**
3. Сравнения: **id < 1, id > 1, id >= 1, id <= 1**
4. На входжение в список: **id IN (1, 10, 100)**
5. На нахождение между границами (обе включены): **id BETWEEN 1 AND 10**

Кроме того, можно использовать функции и комбинировать выражения с помощью **AND, OR, NOT**.

# NULL

С NULL логические операторы работают по-особому:

<i>a</i>	<i>b</i>	<i>a AND b</i>	<i>a OR b</i>
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

<i>a</i>	NOT <i>a</i>
TRUE	FALSE
FALSE	TRUE
NULL	NULL



# UPDATE

Запрос **UPDATE** позволяет обновлять нужные поля строк, соответствующих условию:

Общий формат вот такой:

**UPDATE <tbl\_name> SET <col1> = <val1>, <col2> = <val2> WHERE <expr>;**



не обязательно, но лучше привыкнуть всегда писать, иначе аккуратно обновите всю таблицу

При этом можно ссылаться на значение, которое было до этого:

**UPDATE posts SET likes = likes + 1 WHERE id = 1;**



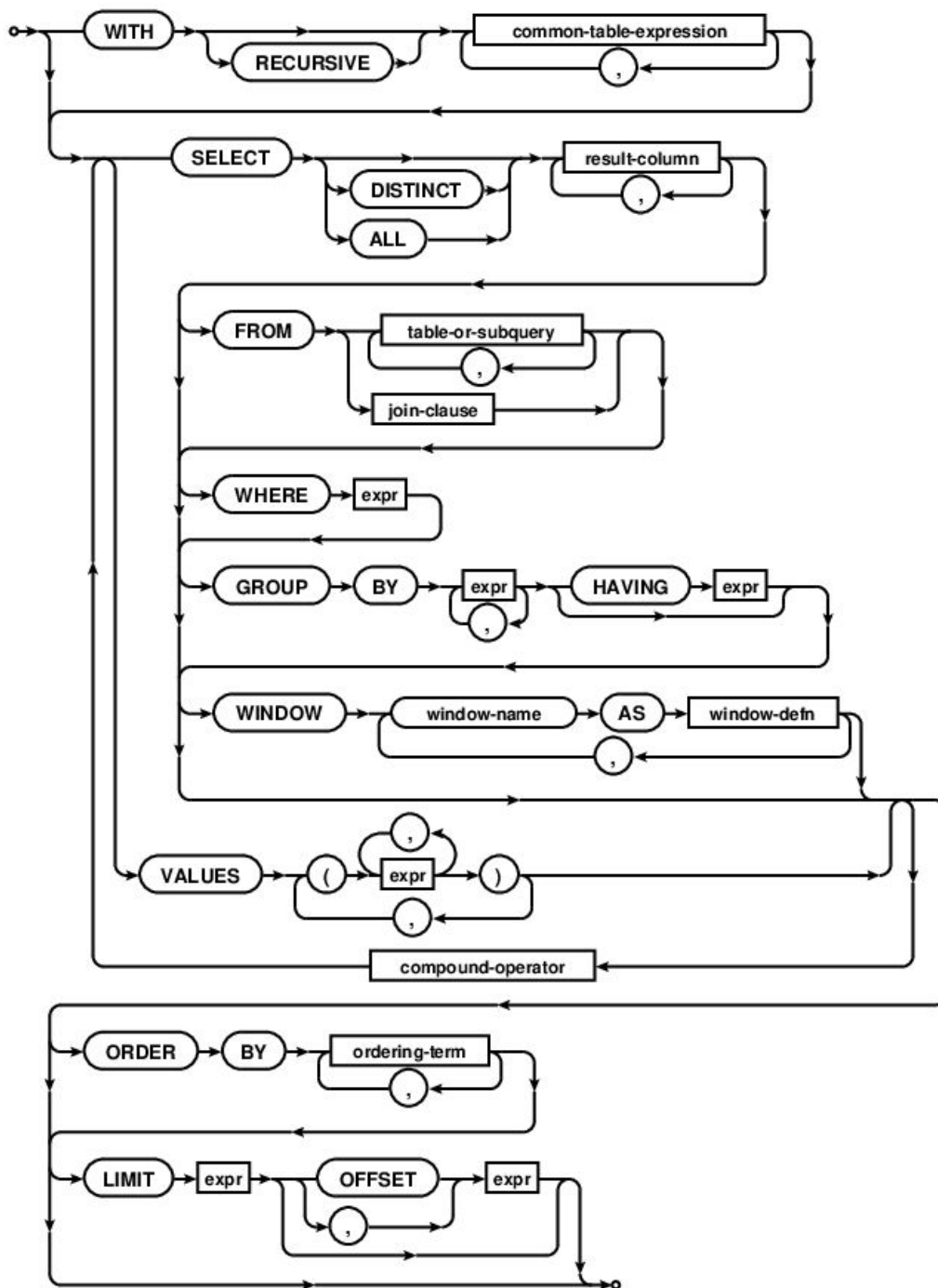
но **likes += 1** или **likes++** нельзя  
(как и подчёркивания в числах)



# SELECT

SELECT – это запрос, позволяющий осуществлять выборку данных из таблиц (одной или сразу нескольких).

[Общая схема запроса SELECT  
\(кликабельная версия на сайте\)](#)






# SELECT

Ключевое, что нужно запомнить: **SELECT** (выполненный без ошибок) **всегда возвращает таблицу (набор строк и столбцов).**

Даже если в наборе 1 столбец и 0 строк, это всё равно таблица.

# SELECT

Начнём рассмотрение с простых форм:

1. `SELECT * FROM <tbl_name>` — антипаттерн 
2. `SELECT <col1>, <col2> FROM <tbl_name> WHERE <expr>`
3. `2 + ORDER BY <col>` — с сортировкой по колонке
4. `2 или 3 + LIMIT <count>` — лимит выборки
5. `2 или 3 или 4 + OFFSET <count>` — отступ

не обязательно,  
будут выбраны все строки

`ORDER BY <col>` по умолчанию сортирует по возрастанию (полная запись `ORDER BY <col> ASC`), может и по убыванию (но тогда нужно писать направление): `ORDER BY <col> DESC`.

# SELECT

DDL	INSERT v1	INSERT v2	SELECT
1	SELECT id, author, published FROM posts;		
	id	author	published
1	1	Нетология. Университет интернет-профессий будущего	21 мая в 18:36
2	2	Нетология. Университет интернет-профессий будущего	18 сентября в 10:12



# SQLITE B ANDROID



## SQLITE В ANDROID

Теперь, когда мы познакомились с базовым синтаксисом SQLite, давайте попробуем сразу использовать его в Android.

Важно: несмотря на то, что мы говорили, что SQL облегчит нам жизнь, судя по коду, этого не скажешь.



# SQLOPENHELPER

Специальный вспомогательный класс, позволяющий создавать базу при первом запуске и обновлять при изменении версии (когда пользователь обновляет приложение, мы можем обновить версию):

```
class DbHelper(context: Context, dbVersion: Int, dbName: String, private val DDLs: Array<String>) :  
    SQLiteOpenHelper(context, dbName, factory: null, dbVersion) {  
    override fun onCreate(db: SQLiteDatabase) {  
        DDLs.forEach { it: String  
            db.execSQL(it)  
        }  
    }  
  
    override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {  
        TODO(reason: "Not implemented")  
    }  
  
    override fun onDowngrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {  
        TODO(reason: "Not implemented")  
    }  
}
```

---

# DAO

Всю непосредственную работу по запросам мы вынесем в DAO (Data Access Object). Мы так могли делать и раньше, но наш код был не особо большой, поэтому мы уместяли всё в репозитории:

```
1 ↓ interface PostDao {  
1 ↓     fun getAll(): List<Post>  
1 ↓     fun save(post: Post): Post  
1 ↓     fun likeById(id: Long)  
1 ↓     fun removeById(id: Long)  
    }
```

---

# DAO IMPLEMENTATION

```
class PostDaoImpl(private val db: SQLiteDatabase) : PostDao {  
    companion object {...}  
  
    object PostColumns {...}  
  
    override fun getAll(): List<Post> {...}  
  
    override fun save(post: Post): Post {...}  
  
    override fun likeById(id: Long) {...}  
  
    override fun removeById(id: Long) {...}  
  
    private fun map(cursor: Cursor): Post {...}  
}
```

# POSTCOLUMNS

Для удобства все названия таблиц, колонок мы храним в отдельном объекте:

```
object PostColumns {  
    const val TABLE = "posts"  
    const val COLUMN_ID = "id"  
    const val COLUMN_AUTHOR = "author"  
    const val COLUMN_CONTENT = "content"  
    const val COLUMN_PUBLISHED = "published"  
    const val COLUMN_LIKED_BY_ME = "likedByMe"  
    const val COLUMN_LIKES = "likes"  
    val ALL_COLUMNS = arrayOf(  
        COLUMN_ID,  
        COLUMN_AUTHOR,  
        COLUMN_CONTENT,  
        COLUMN_PUBLISHED,  
        COLUMN_LIKED_BY_ME,  
        COLUMN_LIKES  
    )  
}
```

# GETALL



```
override fun getAll(): List<Post> {  
    val posts = mutableListOf<Post>()  
    db.query(  
        PostColumns.TABLE,  
        PostColumns.ALL_COLUMNS,  
        selection: null,  
        selectionArgs: null,  
        groupBy: null,  
        having: null,  
        orderBy: "${PostColumns.COLUMN_ID} DESC"  
    ).use { it: Cursor!  
        while (it.moveToNext()) {  
            posts.add(map(it))  
        }  
    }  
    return posts  
}
```

# CURSOR

`Cursor` — это специальный объект (реализующий интерфейс `Closeable`), который позволяет перемещаться по выбранным строкам.

По умолчанию он указывает на позицию **до первой строки данных**. Вызов `moveToNext()` перемещает курсор на следующую позицию и возвращает `true`, если там есть данные или `false`, если данных нет.

# MAPPING

Мы написали специальную функцию `map`, которая умеет отображать строку в объект `Post`:

```
private fun map(cursor: Cursor): Post {  
    with(cursor) { this: Cursor  
        return Post(  
            id = getLong(getColumnIndexOrThrow(PostColumns.COLUMN_ID)),  
            author = getString(getColumnIndexOrThrow(PostColumns.COLUMN_AUTHOR)),  
            content = getString(getColumnIndexOrThrow(PostColumns.COLUMN_CONTENT)),  
            published = getString(getColumnIndexOrThrow(PostColumns.COLUMN_PUBLISHED)),  
            likedByMe = getInt(getColumnIndexOrThrow(PostColumns.COLUMN_LIKED_BY_ME)) != 0,  
            likes = getInt(getColumnIndexOrThrow(PostColumns.COLUMN_LIKES)),  
        )  
    }  
}
```

Курсор не предоставляет функций для извлечения `Boolean` (вы можете написать extension), поэтому пришлось `!= 0`.

# MAPPING

Курсор не предоставляет функций для получения значения по имени колонки (т.е. если вы их поменяете местами, то будет либо Exception, либо данные попадут не в те поля), поэтому приходится по имени искать индекс (но можно написать extension-функции):

```
private fun map(cursor: Cursor): Post {  
    with(cursor) { this: Cursor  
        return Post(  
            id = getLong(getColumnIndexOrThrow(PostColumns.COLUMN_ID)),  
            author = getString(getColumnIndexOrThrow(PostColumns.COLUMN_AUTHOR)),  
            content = getString(getColumnIndexOrThrow(PostColumns.COLUMN_CONTENT)),  
            published = getString(getColumnIndexOrThrow(PostColumns.COLUMN_PUBLISHED)),  
            likedByMe = getInt(getColumnIndexOrThrow(PostColumns.COLUMN_LIKED_BY_ME)) != 0,  
            likes = getInt(getColumnIndexOrThrow(PostColumns.COLUMN_LIKES)),  
        )  
    }  
}
```



# CURSOR

Всего два варианта исполнения запросов:

1. Специализированные методы с большим количеством параметров вроде: `query`, `insert`, `update`, `replace`, `delete`
2. «Raw» — позволяющие писать SQL-запросы в виде строки (вроде `execSQL` и `rawQuery`)

В специализированные передаются массивы и части SQL без ключевых слов:

```
PostColumns.TABLE,  
PostColumns.ALL_COLUMNS,  
selection: null,  
selectionArgs: null,  
groupBy: null,  
having: null,  
→ orderBy: "${PostColumns.COLUMN_ID} DESC"
```

# SAVE

```
override fun save(post: Post): Post {
    val values = ContentValues().apply { this: ContentValues
        if (post.id != 0L) {
            put(PostColumns.COLUMN_ID, post.id)
        }
        // TODO: remove hardcoded values
        put(PostColumns.COLUMN_AUTHOR, "Me")
        put(PostColumns.COLUMN_CONTENT, post.content)
        put(PostColumns.COLUMN_PUBLISHED, "now")
    }
    val id = db.replace(PostColumns.TABLE, nullColumnHack: null, values)
    db.query(
        PostColumns.TABLE,
        PostColumns.ALL_COLUMNS,
        selection: "${PostColumns.COLUMN_ID} = ?",
        arrayOf(id.toString()),
        groupBy: null,
        having: null,
        orderBy: null,
    ).use { it: Cursor!
        it.moveToNext()
        return map(it)
    }
}
```



# SAVE

**ContentValues** — специальный тип, позволяющий задавать значения, которые потом будут использоваться в insert/update.

**?** — placeholder, на место которого будет подставляться конкретное значение (используется при любых запросах, чтобы не получить SQL Injection).


**replace** — делает insert, если возникает конфликт (в данном случае по id), делает update.

# LIKEBYID

Пример «сложного» запроса с чистым SQL:

```
override fun likeById(id: Long) {  
    db.execSQL(  
        """  
        UPDATE posts SET  
            likes = likes + CASE WHEN likedByMe THEN -1 ELSE 1 END,  
            likedByMe = CASE WHEN likedByMe THEN 0 ELSE 1 END  
        WHERE id = ?;  
        """.trimIndent(), arrayOf(id)  
    )  
}
```

# REMOVEBYID



```
override fun removeById(id: Long) {  
    db.delete(  
        PostColumns.TABLE,  
        whereClause: "${PostColumns.COLUMN_ID} = ?",  
        arrayOf(id.toString())  
    )  
}
```

# REPOSITORY

В нашем случае  
репозиторий "кэширует" в  
памяти данные для  
ускорения доступа.

Поскольку в текущей  
реализации с базой  
работает только он (через  
DAO), то это вполне  
допустимо.

```
class PostRepositorySQLiteImpl(
    private val dao: PostDao
) : PostRepository {
    private var posts = emptyList<Post>()
    private val data = MutableLiveData(posts)

    init {
        posts = dao.getAll()
        data.value = posts
    }

    override fun getAll(): LiveData<List<Post>> = data

    override fun save(post: Post) {
        val id = post.id
        val saved = dao.save(post)
        posts = if (id == 0L) {
            listOf(saved) + posts
        } else {
            posts.map { it: Post
                if (it.id != id) it else saved
            }
        }
        data.value = posts
    }
}
```

# APPDB

```
class AppDb private constructor(db: SQLiteDatabase) {
    val postDao: PostDao = PostDaoImpl(db)

    companion object {
        @Volatile
        private var instance: AppDb? = null

        fun getInstance(context: Context): AppDb {
            return instance ?: synchronized( lock: this) {
                instance ?: AppDb(
                    buildDatabase(context, arrayOf(PostDaoImpl.DDL))
                ).also { instance = it }
            }
        }

        private fun buildDatabase(context: Context, DDLs: Array<String>) = DbHelper(
            context, dbVersion: 1, dbName: "app.db", DDLs,
        ).writableDatabase
    }
}
```



## SYNCHRONIZED & VOLATILE

Ключевая идея: мы хотим получить синглтон для доступа к БД. Пока мы не говорили о многопоточности и стоит воспринимать это как «шаблонную версию» создания синглтона (о многопоточности мы будем говорить с вами на следующем курсе).



# СУБД

Сама локальная СУБД хранится в каталоге databases приложения:

▼ ru.netology.nmedia

- > cache
- > code\_cache
- ▼ databases
  - app.db
  - app.db-journal
- > files
- > shared\_prefs

# VIEWMODEL

```
class PostViewModel(application: Application) : AndroidViewModel(application) {  
    // упрощённый вариант  
    private val repository: PostRepository = PostRepositorySQLiteImpl(  
        AppDb.getInstance(application).postDao  
    )  
    val data = repository.getAll()  
    val edited = MutableLiveData(empty)  
  
    fun save() {...}  
  
    fun edit(post: Post) {...}  
  
    fun changeContent(content: String) {...}  
  
    fun likeById(id: Long) = repository.likeById(id)  
    fun removeById(id: Long) = repository.removeById(id)  
}
```

# DATABASE INSPECTOR

View -> Tool Windows -> Database Inspector:

The screenshot shows the Database Inspector tool window. At the top, it says 'Database Inspector' with a settings icon. Below that, the connection path is 'Pixel 2 API 30 > ru.netology.nmedia'. The 'Databases' section on the left shows a tree view with 'app.db' expanded, and 'posts' selected. To the right of the tree, the 'posts' table is selected, and its schema is listed: 'id : INTEGER', 'author : TEXT, NOT NULL', 'content : TEXT, NOT NULL', 'published : TEXT, NOT NULL', 'likedByMe : NUMERIC, NOT NULL', and 'likes : INTEGER, NOT NULL'. The main area displays a table with one row of data. The table has columns: 'id', 'author', 'content', 'published', 'likedByMe', and 'likes'. The row contains the values: '1', 'Me', 'Hello world', 'now', '0', and '0'. At the bottom, it says 'Results are read-only' and has a pagination control showing '50'.

id	author	content	published	likedByMe	likes
1	Me	Hello world	now	0	0



# ИТОГИ

# ИТОГИ

Если попробовать представить, что будет, когда у нас появится несколько сущностей для хранения в БД, то мы увидим типовые операции:

1. Маппинг таблиц БД на объекты и обратно.
2. Запросы вроде «выбрать всё», «выбрать по id», «удалить по id» (просто написать сам запрос — выборка полей типовая).
3. и т.д.

Хотелось бы более высокоуровневого инструмента, который сможет это делать за нас. На следующей лекции мы как раз и приступим к его изучению (библиотека Room).