

## Основы реляционных баз данных

Предположим, что мы делаем игру, например, «Сапер» и в ней предполагается сохранять результаты игр и профили игроков. Обычно, для этих целей создают базу данных (БД) с набором таблиц. Пусть это будут таблицы «Игроки» (users) и «Игры» (games).

### Раздаточный материал № 1



Каждая таблица имеет заданную структуру (набор определенных столбцов) и определенное количество записей – строк. Например, так:

### Раздаточный материал № 2

users				
id	sex	name	level	score
1	2	Сергей	2	26116
2	2	Михаил	2	13832
3	1	Мария	1	0

games			
id	user_id	score	time
15	1	1514	1559470775
16	1	2310	1559471049
17	1	1706	1559471132
18	1	11	1559568764
19	1	1260	1559568871
20	1	20	1559569080
21	1	1165	1559627469
22	1	1259	1559627528

Структура первой таблицы users определяется набором полей (столбцов):

id (тип integer) – уникальный идентификатор записи (строки);  
sex (integer) – пол игрока (1 – женский; 2 – мужской);  
name (text) – имя игрока;  
level (integer) – уровень игрока;  
score (integer) – максимальное число очков, заработанных игроком.

Структура второй таблицы games, следующая:

id (тип integer) – уникальный идентификатор записи (строки);  
user\_id (integer) – внешний ключ для связи с таблицей users;  
score (integer) – число очков, набранных в игре;  
time (integer) – время начала игры.

Структура каждой таблицы определяется разработчиком и может быть самой разной.

В приведенном примере, таблица games имеет внешний ключ user\_id, содержащий id игрока, участвующего в соответствующих играх. Затем, по этому ключу можно будет выполнять связывание таблиц и получать сводные данных по играм и данным игрока. Такая связь по-английски звучит как relation, откуда и пошло название реляционные базы данных, то есть, базы, содержащие таблицы с возможностью связываться между собой.

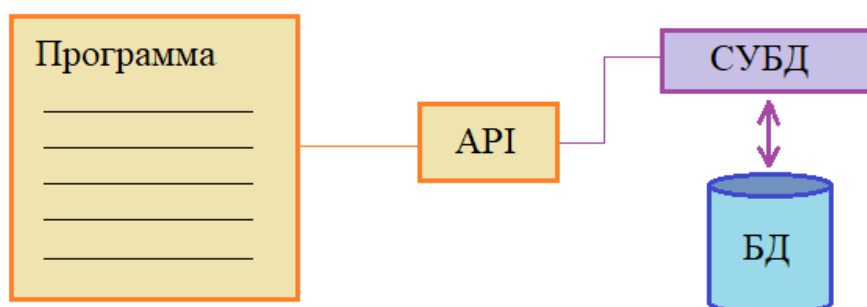
### Раздаточный материал № 3

games			
id	user_id	score	time
15	1	1514	1559470775
16	1	2310	1559471049
17	1	1706	1559471132

users				
id	sex	name	level	score
1	2	Сергей	2	26116
2	2	Михаил	2	13832
3	1	Мария	1	0

Программисты пользуются различными СУБД для организации хранения различных данных. После выбора определенной СУБД программист получает доступ к ее API (Application Programming Interface) – программному интерфейсу для взаимодействия с СУБД. Фактически, к набору функций, через которые производится работа с базами данных.

### Раздаточный материал № 4



При программировании на Python популярными являются следующие: PyMySQL, Python SQLite и Python PostgreSQL

SQLite используется для создания легковесной дисковой БД, то есть, эта СУБД не поддерживает сетевое взаимодействие (разве что удаленный доступ к файлу БД, но не более того). Также она имеет ограничение при многопользовательском доступе: запись данных может осуществлять только один поток или процесс в один момент времени. А вот на чтение таких ограничений нет. SQLite работает по принципу «один пишет – многие читают» и старается развиваться и функционировать по правилу «минимальный, но полный набор». Она идеально подходит для хранения данных различных приложений.

Все взаимодействие с СУБД происходит с помощью нескольких методов, главным из которых является execute. Он передает СУБД указание выполнить запрос, написанный на языке SQL. Фактически, все взаимодействие с БД происходит посредством этого языка. Для его изучения понадобится менеджер баз данных DBeaver.

### Раздаточный материал № 5

execute(SQL)

SQL (Structured Query Language)

DBeaver - <https://dbeaver.io/>

### Подключение к БД, создание таблиц в БД

Для добавления возможности использования СУБД SQLite в программе на Python необходимо импортировать модульsqlite3. После чего станут доступны API-функции этого расширения.

## Раздаточный материал № 6

```
import sqlite3 as sq

with sq.connect('saper.db') as con:
    cur = con.cursor()
#cur.execute("DROP TABLE IF EXISTS users")
cur.execute("""CREATE TABLE IF NOT EXISTS users (
    user_id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    sex INTEGER NOT NULL DEFAULT 1,
    old INTEGER,
    score INTEGER
) """)
```

В первую очередь необходимо вызвать метод connect, чтобы установить связь с определенной БД. В данном случае – это файл saper.db, который должен располагаться в том же каталоге, что и файл программы на Питоне. В качестве расширений этого файла, обычно, используют следующие:

## Раздаточный материал № 7

\*.db, \*.db3, \*.sqlite и \*.sqlite3

При выполнении команды connect файл saper.db либо будет открыт, либо будет создан, если он не существует. В результате создается (или открывается) БД с именем saper.db.

При успешном соединении с БД метод connect возвращает экземпляр объекта Connection, на который ссылается переменная con.

Далее необходимо использовать объект Cursor для взаимодействия с БД и выполнения SQL-запросов. Например, это можно сделать с помощью метода execute, которому в качестве аргумента как раз и передается строка с SQL-запросом.

Рекомендуется соединяться с БД через менеджер контекста: он автоматически сохраняет данные в БД (вызывает метод commit()) и закрывает БД (метод close()) даже при возникновении ошибочных ситуаций.

## Создание таблицы

Создадим первую таблицу со структурой:

user\_id – первичный ключ  
name – строка с именем игрока;  
sex – число, пол игрока (1 – мужской; 2 – женский);  
old – число, возраст игрока;  
score – суммарное число набранных очков за все игры.

Полный список типов полей:

## Раздаточный материал № 8

NULL – значение NULL;  
INTEGER – целочисленный тип (занимает от 1 до 8 байт);  
REAL – вещественный тип (8 байт в формате IEEE);  
TEXT – строковый тип (в кодировке данных базы, обычно UTF-8);  
BLOB (двоичные данные, хранятся «как есть», например, для небольших изображений).

SQL-запрос «CREATE TABLE IF NOT EXISTS users» создает таблицу только если она не существует.

Ограничитель PRIMARY KEY (первичный ключ) означает, что поле user\_id должно содержать уникальные значения, а ограничитель AUTOINCREMENT указывает СУБД автоматически увеличивать значение user\_id при добавлении новой записи.

Ограничитель NOT NULL означает, что поле обязательно должно содержать какие-либо данные, а для задания значения по умолчанию – ограничитель DEFAULT.

SQL-запрос "DROP TABLE IF EXISTS users" удаляет таблицу.

Заполнение БД и SQL-выполнение запросов к БД можно выполнить в менеджере баз данных DBeaver.

## Команда INSERT

Предназначена для добавления записей в БД.

### Раздаточный материал № 9

INSERT INTO <table\_name> (<column\_name1>, <column\_name2>, ...) VALUES (<value1>, <value2>, ...)

Или так:

INSERT INTO <table\_name> VALUES (<value1>, <value2>, ...)

Здесь table\_name – имя таблицы, за которым в круглых скобках указываются столбцы (поля), в которые будет происходить добавление информации при создании новой записи. Остальные столбцы будут принимать или значение NULL, или значение по умолчанию, если в структуре поля был указан ограничитель DEFAULT. Во втором варианте можно не перечислять поля таблицы, тогда предполагается, что после ключевого слова VALUES будут указаны данные для каждого поля по порядку, начиная с первого и до последнего.

SQL-запрос:

### Раздаточный материал № 10

INSERT INTO users VALUES (1, 'Михаил', 1, 19, 1000)

Команда в Python:

### Раздаточный материал № 11

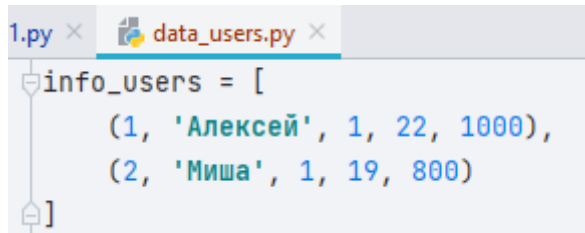
```
cur.execute("INSERT INTO users VALUES (1, 'Алексей', 1, 22, 1000)")
```

Как правило, информация для внесения в БД поступает из внешнего файла, содержащего значительный объем данных, поэтому необходимо организовать работу через цикл или воспользоваться методом executemany():

## Раздаточный материал № 12

```
cur.executemany("INSERT INTO users VALUES (?, ?, ?, ?, ?)", info_users)
```

Содержимое info\_users



```
1.py x data_users.py x  
info_users = [  
    (1, 'Алексей', 1, 22, 1000),  
    (2, 'Миша', 1, 19, 800)  
]
```

Метод принимает два параметра: SQL-запрос и ссылку на итерируемый объект - в данном случае, это список кортежей со значениями полей. Вместо знаков вопроса будут подставлены соответствующие данные из кортежей списка.

## Команда SELECT

Отвечает за выборку данных из таблицы. Формат:

## Раздаточный материал № 13

```
SELECT col1, col2, ... FROM <table_name>
```

Например:

```
SELECT name, old, score FROM users
```

На выходе получим выборку из трех столбцов и набора всех записей из таблицы users.

Если нужно выбрать все столбцы, то вместо их перебора можно просто записать звездочку:

```
SELECT * FROM users
```

Если нужно добавить фильтр для выбираемых записей, то это делается с помощью ключевого слова WHERE, которое записывается после имени таблицы:

## Раздаточный материал № 14

```
SELECT col1, col2, ... FROM <table_name> WHERE <условие>
```

Например, отберем все записи со значением очков меньше 1000:

```
SELECT * FROM users WHERE score < 1000
```

После слова WHERE записывается логическое выражение и в качестве сравнения можно использовать следующие операторы:

## Раздаточный материал № 15

= или ==, >, <, >=, <=, !=, BETWEEN

Например:

```
SELECT * FROM users WHERE score BETWEEN 500 AND 1000
```

Будут выбраны все записи с числом очков в диапазоне от 500 до 1000.

Часто при описании фильтра требуется учитывать значения сразу нескольких столбцов. Например, выбрать всех игроков старше 20 лет и с числом очков более 300. Здесь уже нужно использовать составное условие. Для этого дополнительно используются следующие ключевые слова:

### Раздаточный материал № 16

AND – условное И: `expr1 AND expr2`. Истинно, если одновременно истинны `expr1` и `expr2`.

OR – условное ИЛИ: `expr1 OR expr2`. Истинно, если истинно `expr1` или `expr2` или оба выражения.

NOT – условное НЕ: `NOT expr`. Преобразует ложное условие в истинное и, наоборот, истинное – в ложное.

IN – вхождение во множество значений: `col IN (val1, val2, ...)`

NOT IN – не вхождение во множество значений: `col NOT IN (val1, val2, ...)`

Самый высокий приоритет имеет операция NOT. Приоритет у операции AND выше, чем у OR.

Например:

```
SELECT * FROM users WHERE old > 20 AND score < 1000
```

Выбирает игроков возрастом более 20 лет и с числом очков менее 1000. Или, так:

```
SELECT * FROM users WHERE old IN(19, 32) AND score < 1000
```

Создается выборка из игроков возрастом 19 или 32 года и числом очков менее 1000.

```
SELECT * FROM users WHERE old IN(19, 32) AND score > 300 OR sex = 1
```

```
SELECT * FROM users WHERE (old IN(19, 32) OR sex = 1) AND score > 300
```

После условия в команде SELECT можно дополнительно указывать сортировку записей по определенному столбцу. Предположим, что мы хотим выбрать всех игроков с числом очков менее 1000 (поле `score`) и отсортировать их по возрастанию возраста (поле `old`).

### Раздаточный материал № 17

```
SELECT * FROM users WHERE score < 1000 ORDER BY old
```

По умолчанию сортировка делается по возрастанию (в явном виде - флаг ASC). Если нужно отсортировать данные по убыванию, то после имени поля следует указать флаг DESC:

### Раздаточный материал № 18

```
SELECT * FROM users WHERE score < 1000 ORDER BY old DESC
```

### Работа с выборкой в программе на Python

Все приведенные SQL-запросы можно выполнять непосредственно из программы:

### Раздаточный материал № 19

```
with sq.connect('saper.db') as con:
    cur = con.cursor()
```

```
cur.execute("SELECT * FROM users WHERE score > 1000 ORDER BY score DESC")
result = cur.fetchall()
print(result)
```

```
#результат: [(4, 'Мария', 2, 18, 1500), (5, 'Александр', 1, 20, 1100)]
```

Здесь используется метод `fetchall` для получения результатов отбора SQL-запроса. В результате, `result` будет ссылаться на упорядоченный список, состоящий из кортежей с данными таблицы.

Или, перебрать последовательно, используя `Cursor` в качестве итерируемого объекта:

## Раздаточный материал № 20

```
cur.execute("SELECT * FROM users WHERE score > 1000 ORDER BY score DESC")
for result in cur:
    print(result)
```

Этот вариант более предпочтителен, когда число выбираемых записей может быть велико. Тогда не формируется список, а последовательно выбираются записи из БД и тут же обрабатываются. Такой подход существенно экономит память.

## Раздаточный материал № 21

Есть еще два метода, которые выдают результат выборки из таблицы:

`fetchmany(size)` – возвращает число записей не более `size`;

`fetchone()` – возвращает первую запись.

Например

```
cur.execute("SELECT * FROM users")
result1 = cur.fetchone()
result2 = cur.fetchmany(2)
print(result1)
print(result2)

#результат (1, 'Алексей', 1, 22, 1000)
#[ (2, 'Миша', 1, 19, 800), (3, 'Сергей', 1, 19, 900) ]
```

## Команда UPDATE

Изменение данных в записях.

## Раздаточный материал № 22

UPDATE имя\_таблицы SET имя\_столбца = новое\_значение WHERE условие

Например,

можно менять значения сразу несколько столбцов записи, перечисляя их через запятую:

UPDATE users SET score = 700, old = 45 WHERE old > 40

всем игрокам женского пола увеличить число очков на 500:

UPDATE users SET score = score+500 WHERE sex = 2

обратиться к игрокам по имени и указать им определенное число очков:

UPDATE users SET score = 1500 WHERE name LIKE 'Федор'

Здесь ключевое слово `LIKE` возвращает `True`, если поле `name` содержит имя «Федор». В этой строке можно использовать специальные символы, т.е. создавать шаблоны для сравнения:

### Раздаточный материал № 23

% - любое продолжение строки;

\_ - любой символ;

Например:

```
UPDATE users SET score = score+100 WHERE name LIKE 'M%'
```

Однако, на практике злоупотреблять сравнением строк не стоит, т.к. это относительно ресурсоемкая операция - проще выполнить сравнение чисел.

Команда в Питоне:

### Раздаточный материал № 24

```
cur.execute("UPDATE users SET score = score+100 WHERE name LIKE 'M%'")
```

Команда DELETE

удаление записи из БД

### Раздаточный материал № 25

DELETE FROM имя\_таблицы WHERE условие

Например:

```
DELETE FROM users WHERE user_id = 5
```

Указывается имя таблицы, из которой осуществляется удаление и, затем, условия для выбора удаляемых записей. Обычно, в качестве фильтра задается строгое условие, чтобы случайно не удалить «лишние» данные.

Команда в Питоне:

### Раздаточный материал № 25

```
cur.execute("DELETE FROM users WHERE user_id = 5")
```