

## Лекция №12

### Работа с реляционными базами данных

- Базы данных: определения, свойства, требования
- Модели данных
- СУБД
- SQL
- SQLite
- Python DB-API
- ORM (SQLAlchemy)

# Определение

База данных – это систематизированный набор данных, отображающий атрибуты и взаимосвязь объектов предметной области и предназначенный для удовлетворения информационных потребностей пользователей.

Правильно читать, создавать и редактировать эти записи умеет система управления базами данных (СУБД) или сервер баз данных. СУБД выполняет манипуляции с данными в соответствии с командами, которые она получает. Сервер баз данных отвечает за целостность и сохранность данных, а также обеспечивает операции ввода-вывода при доступе клиента к информации.

Основное свойство базы данных – целостность – означает, что в базе данных содержится полная, непротиворечивая и адекватно отражающая предметную область информация.



VS



# Примеры нарушений

В целях обеспечения целостности к проектируемой базе данных предъявляется ряд требований, таких как полнота и неизбыточность данных, безопасность и контроль доступа к данным различных групп пользователей и т.д. К чему может привести нарушений этих требований?

ФИО	Должность	Оклад
Иванов И.И.	инженер	50000
Петров П.П.	старший инженер	51000
Сидоров С.С.	менеджер проекта	100000
Иванов И.И.	инженер	50000

**Нарушение полноты**  
информация неполная, т.к. записи не  
могут быть однозначно  
идентифицированы

ФИО	Должность	Оклад
Иванов И.И.	инженер	50000
Петров П.П.	старший инженер	51000
Сидоров С.С.	менеджер проекта	100000
Иванов И.И.	инженер	60000

**Нарушение неизбыточности**  
информация избыточная, т.к. оклад  
должен однозначно определяться  
должностью

ФИО	Должность	Оклад
Иванов И.И.	инженер	150000
Петров П.П.	старший инженер	51000
Сидоров С.С.	менеджер проекта	10000
Иванов И.И.	инженер	50000

**Нарушение безопасности**  
отсутствуют уровни доступа к  
информации, защита от изменения  
информации произвольным  
пользователем

# Модели данных

Чтоб обеспечивать целостность данных, очевидно, они должны быть упорядочены в соответствии с некоторой логической структурой. Такая структура и правила работы с ней описывается моделью данных, поддерживаемой в том числе на уровне СУБД. Существуют различные модели данных.

- иерархическая модель, в которой связи между данными можно описать с помощью упорядоченного графа (дерева)
- сетевая модель позволяет отображать разнообразные взаимосвязи элементов данных в виде произвольного графа, обобщая тем самым иерархическую модель данных
- многомерная модель представляется набором гиперкубов, применяется, как правило, в узкоспециализированных областях
- объектно-ориентированная модель рассматривает отдельные записи базы как свойства объектов, а связанные по смыслу совокупности таких записей – как сами объекты
- реляционная модель – совокупность отношений, содержащих информацию о предметной области, упрощенным представлением реляционной модели данных является набор таблиц
- постреляционная модель представляет из себя расширенную реляционную модель, в которой допускается вложенность таблиц
- и многие другие.

# Постановка задачи

Рассмотрим возможность применения различных моделей данных для решения следующей задачи. Необходимо спроектировать базу данных для некой организации, учитывая следующие факты:

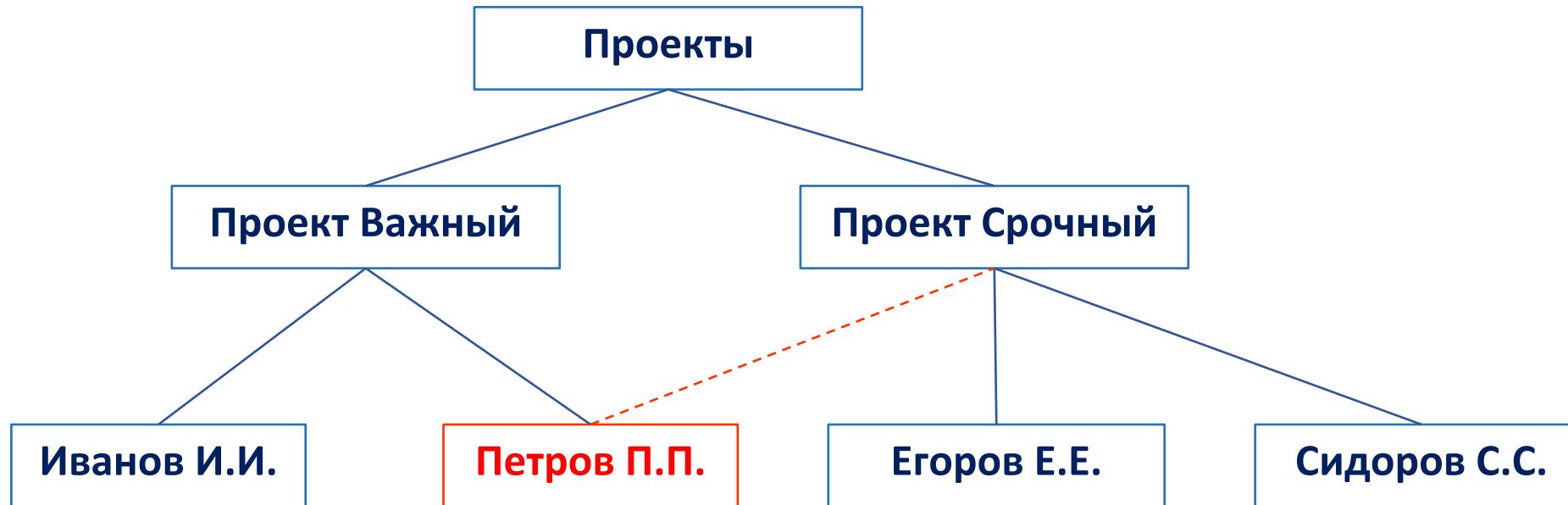
- организация состоит из сотрудников, у которых есть ФИО, должность и уникальный табельный номер
- сотрудники работают в различных проектах: один сотрудник может участвовать в нескольких проектах; в проекте, разумеется, может быть несколько сотрудников; бывают проекты без сотрудников (которые еще только планируются) и сотрудники без проектов (проект закрылся, но его сотрудников пока не сократили)
- у каждого проекта есть название и уникальный идентификатор
- в каждом проекте есть не более одного менеджера
- все сотрудники получают зарплату (как ни странно), которая складывается из должностного оклада и премии

# Иерархическая модель (простая, как дерево)

Структура – дерево.

+ удобная, если сущностей мало и связи простые

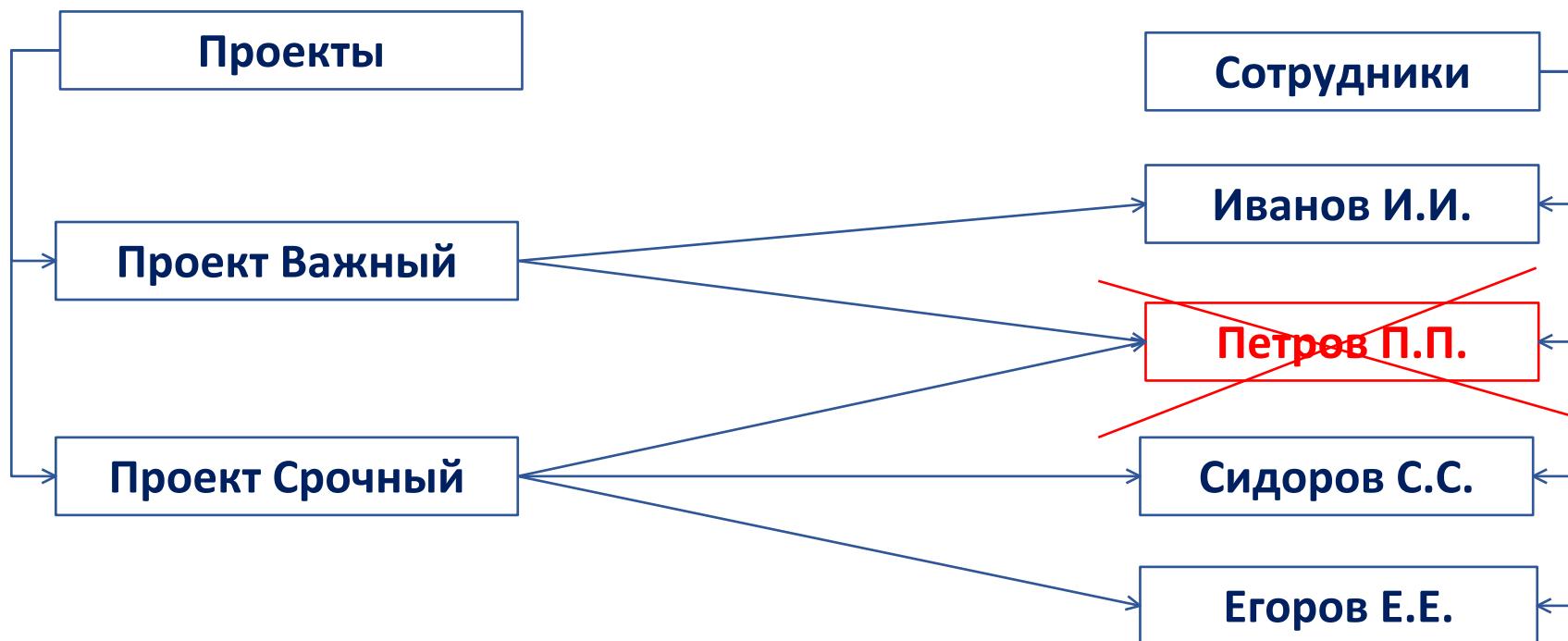
- отсутствие гибкости (как отразить возможность нахождения сотрудника сразу в нескольких проектах?)



# Сетевая модель (легко запутаться)

Структура – граф.

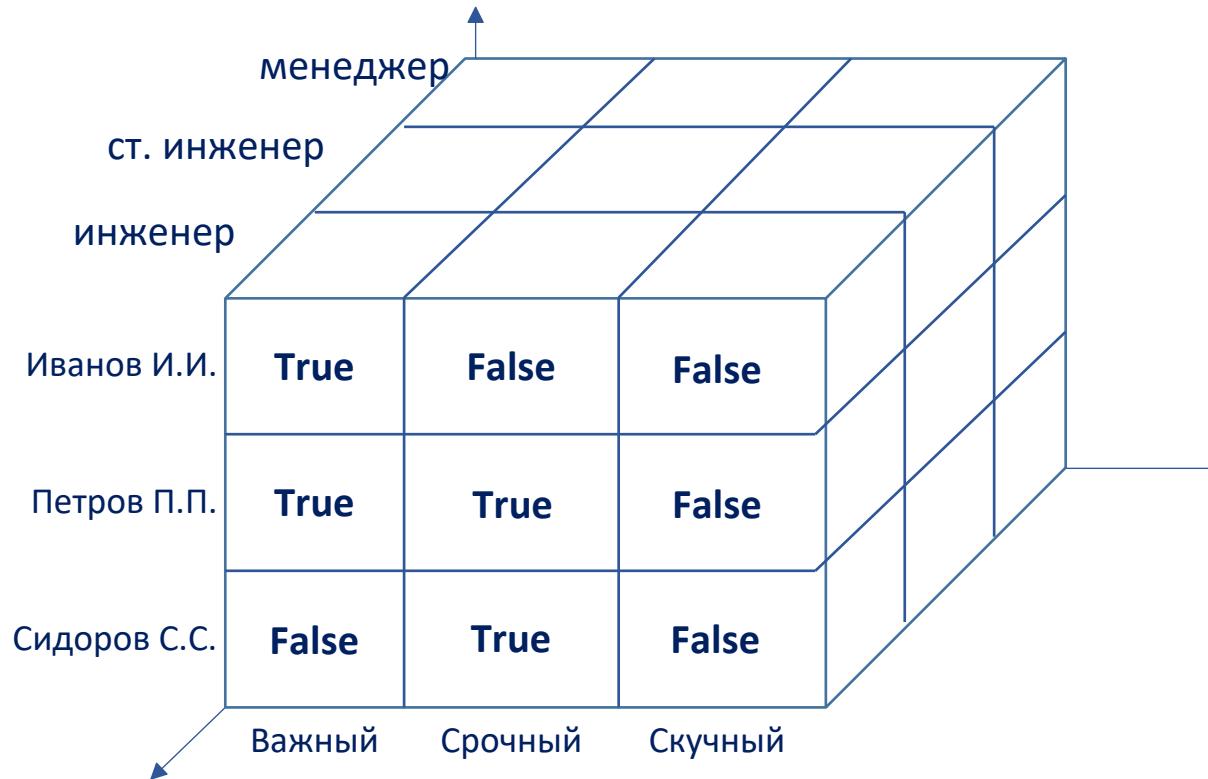
- + более гибкая, чем иерархическая
- сложно контролировать полноту и неизбыточность



# Многомерная модель (под специфические задачи)

Структура – гиперкуб.

- + удобная для аналитической обработки больших объемов данных (особенно, привязанных ко времени)
- громоздкая и неэффективная для оперативной обработки информации

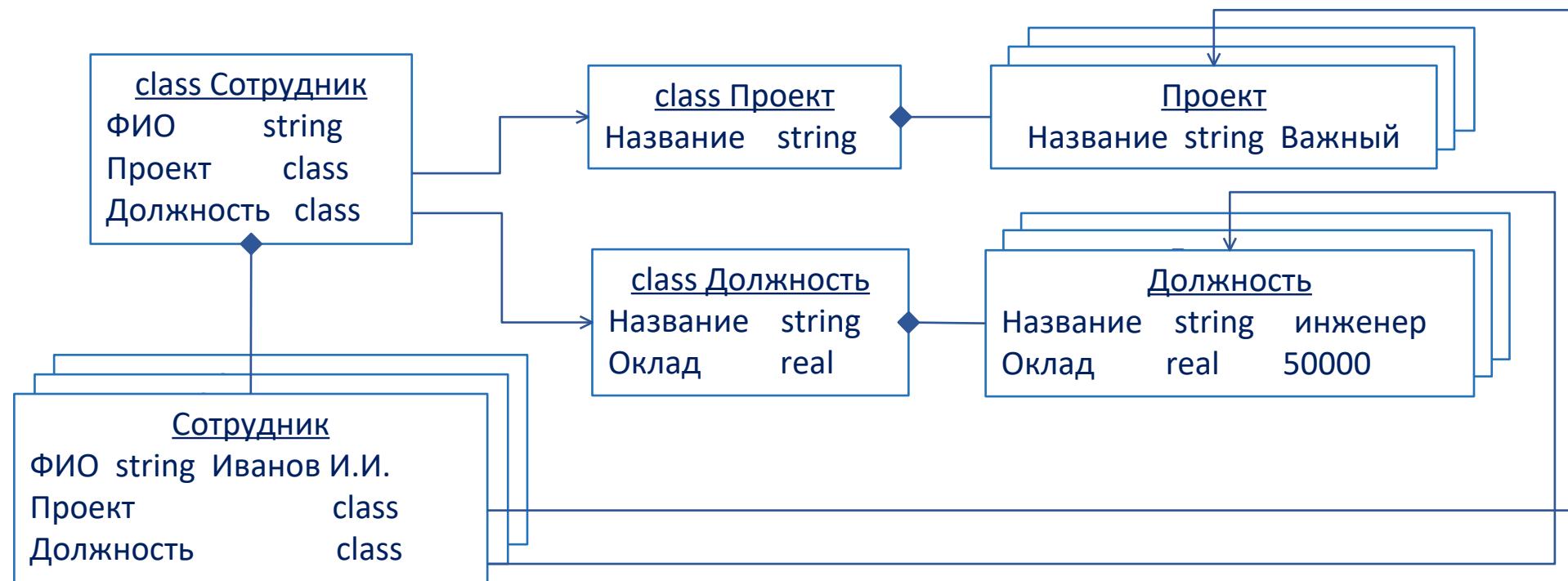


# Объектно-ориентированная модель (для ORM)

Структура – тоже дерево.

+ удобная для работы с отдельными объектами, полноценно представляющими соответствующие сущности со всеми их связями

- сложность алгоритмов и низкая скорость выполнения запросов для обработки совокупности разнотипных объектов



# Реляционная модель (то, что надо!)

Структура – таблицы.

- + удобная для понимания, физической реализации и оперативной обработки данных
- в общем случае необходимо анализировать совокупность таблиц даже если модифицируются атрибуты отдельной сущности

Сотрудники		
ID	ФИО	Должность
1	Иванов И.И.	инженер
2	Петров П.П.	старший инженер
3	Сидоров С.С.	менеджер проекта
4	Егоров Е.Е.	инженер

Проекты	
ID	Название
1	Важный
2	Срочный
3	Скучный

Сотрудник-Проект	
ID сотрудника	ID проекта
1	1
2	1
2	2
3	2
4	2

Должность-Оклад	
Название	Оклад
инженер	50000
старший инженер	51000
менеджер проекта	100000

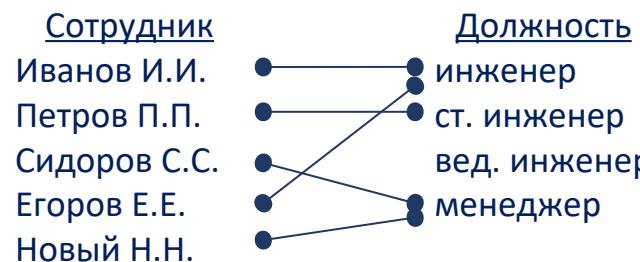
# Постреляционная модель (лучшее – враг хорошего )

Структура – таблицы с возможностью вложенности.

- + можно заменить совокупность связанных реляционных таблиц одной постреляционной
- сложно контролировать полноту и неизбыточность

Проекты		
ID	Название	ID сотрудника
1	Важный	1
		2
2	Срочный	2
		3
		4
3	Скучный	

# Метод ER-диаграмм для проектирования реляционной БД



# Метод ER-диаграмм для проектирования реляционной БД

<u>Должность</u>	<u>Оклад</u>
инженер	50000
ст. инженер	51000
вед. инженер	70000
менеджер	100000



<u>Сотрудник</u>	<u>Премия</u>
Иванов И.И.	30000
Петров П.П.	50000
Сидоров С.С.	30000
Егоров Е.Е.	20000
Новый Н.Н.	20000



<u>Сотрудник</u>	<u>Аккаунт</u>
Иванов И.И.	ivanovi
Петров П.П.	petrovvp
Сидоров С.С.	sidorovs
Егоров Е.Е.	egorove
Новый Н.Н.	novyin



# Метод ER-диаграмм: 6 правил

- [1]:[1] = одно отношение
- 1:[1] = два отношения
- 1:[N] = два отношения
- 1:1 = три отношения
- 1:N = три отношения
- N:N = три отношения

# Метод ER-диаграмм: результат проектирования

Сотрудники					
Таб. номер	ФИО	Должность	Премия	Логин	Пароль
1	Иванов И.И.	инженер	30000	ivanovi	ivanov123
2	Петров П.П.	старший инженер	50000	petrovpr	p1e2t3
3	Сидоров С.С.	менеджер проекта	30000	sidorovs	zayka88
4	Егоров Е.Е.	инженер	20000	egorove	qwerty
5	Новый Н.Н.	инженер	20000	novyin	a1111

Проекты	
ID	Название
1	Важный
2	Срочный
3	Скучный

Должность-Оклад	
Название	Оклад
инженер	50000
старший инженер	51000
ведущий инженер	70000
менеджер проекта	100000

Сотрудник-Проект	
Таб. номер	ID проекта
1	1
2	1
2	2
3	2
4	2

# Первичный ключ – уникальный идентификатор записи

Сотрудники					
Таб. номер	ФИО	Должность	Премия	Логин	Пароль
1	Иванов И.И.	инженер	30000	ivanovi	ivanov123
2	Петров П.П.	старший инженер	50000	petrovpr	p1e2t3
3	Сидоров С.С.	менеджер проекта	30000	sidorovs	zayka88
4	Егоров Е.Е.	инженер	20000	egorove	qwerty
5	Новый Н.Н.	инженер	20000	novyin	a1111

Проекты	
ID	Название
1	Важный
2	Срочный
3	Скучный

Должность-Оклад	
Название	Оклад
инженер	50000
старший инженер	51000
ведущий инженер	70000
менеджер проекта	100000

Сотрудник-Проект	
Таб. номер	ID проекта
1	1
2	1
2	2
3	2
4	2

# Внешний ключ – для связи двух отношений

Сотрудники					
Таб. номер	ФИО	Должность	Премия	Логин	Пароль
1	Иванов И.И.	инженер	30000	ivanovi	ivanov123
2	Петров П.П.	старший инженер	50000	petrovvp	p1e2t3
3	Сидоров С.С	менеджер проекта	30000	sidorovs	zayka88
4	Егоров Е.Е.	инженер	20000	egorove	qwerty
5	Новый Н.Н.	инженер	20000	novyin	a1111

Проекты	
ID	Название
1	Важный
2	Срочный
3	Скучный

Должность-Оклад	
Название	Оклад
инженер	50000
старший инженер	51000
ведущий инженер	70000
менеджер проекта	100000

Сотрудник-Проект	
Таб. номер	ID проекта
1	1
2	1
2	2
3	2
4	2

# SQL

Сервер реляционной базы данных, как правило, понимает команды на языке SQL, который де-факто стал стандартом управления данными в реляционных базах данных. SQL (Structured Query Language — язык структурированных запросов) позволяет осуществлять следующие операции:

- создание баз данных и отдельных таблиц с полным описанием их структуры
- выполнение основных операций манипулирования данными (такие как вставка, модификация и удаление данных из таблиц)
- выполнение простых и сложных запросов

В соответствии с этим SQL условно подразделяется на несколько подъязыков:

- DDL – Data Definition Language – определение данных (CREATE, ALTER, DROP)
- DML – Data Manipulation Language – манипулирование данными (SELECT, INSERT, UPDATE, DELETE)
- DCL – Data Control Language – управление доступом к данным (GRANT, REVOKE, DENY)
- TCL – Transaction Control Language – управление транзакциями (COMMIT, ROLLBACK, SAVEPOINT)

# Выбор реляционной СУБД



# SQLite

SQLite это библиотека, написанная на языке C, предоставляющая легковесную файловую базу данных, не требующую отдельного серверного процесса и понимающую обращения на одном из диалектов SQL. Приложения могут использовать SQLite для хранения внутренних данных. Также SQLite позволяет эффективно и быстро создавать прототипы приложений, использующих базы данных, а затем портировать код для работы с более сложными базами данных, такими как PostgreSQL или Oracle.

На данный момент актуальной является третья версия библиотеки – sqlite3.

Для установки sqlite3 консоли в Windows надо использовать инсталлятор, который можно скачать здесь: <https://www.sqlite.org/download.html>

В Linux (Ubuntu), если консоль не предустановлена, надо воспользоваться apt-get, выполнив команду:

```
$ sudo apt-get install sqlite3
```

Для установки sqlite3 браузера в Windows надо использовать инсталлятор, который можно скачать здесь: <https://sqlitebrowser.org/>

В Linux (Ubuntu) надо выполнить следующие команды:

```
$ sudo add-apt-repository -y ppa:linuxgndu/sqlitebrowser  
$ sudo apt-get update  
$ sudo apt-get install sqlitebrowser
```

# Конфигурирование таблиц

Прежде, чем начать работать непосредственно с данными, нам нужно сконфигурировать структуру их хранения (таблицы).

Создадим хотя бы одну таблицу с помощью оператора CREATE (в консоли sqlite3, .exit – для выхода):

```
sqlite> CREATE TABLE films (id INTEGER PRIMARY KEY NOT NULL, name CHAR(128) NOT NULL, desc TEXT);
```

Также таблицу можно создать (и просмотреть), используя SQLite браузер:

CREATE TABLE films (id INTEGER PRIMARY KEY NOT NULL, name CHAR(128) NOT NULL, desc TEXT)		
id	INTEGER	"id" INTEGER NOT NULL
name	CHAR(128)	"name" CHAR(128) NOT NULL
desc	TEXT	"desc" TEXT
Индексы (0)		
Представления (0)		
Тrigгеры (0)		

удалить таблицу (при необходимости) так же просто:

```
sqlite> DROP TABLE films;
```

# Операторы CRUD

После создания необходимых таблиц, мы можем вносить в них данные. Для обозначения основных действий с данными (записями) существует специальная аббревиатура — CRUD (create, read, update, delete — создать, прочесть, обновить, удалить) — акроним, обозначающий четыре базовые функции, используемые при работе с персистентными хранилищами данных. В соответствии с CRUD в SQL имеются следующие операторы:

- **INSERT** - оператор языка SQL, который позволяет добавить строку со значениями в таблицу

```
sqlite> INSERT INTO films (name, desc) VALUES ('Cool Film', 'SHORT LONG STORY');
```

- **SELECT** - оператор запроса в языке SQL, возвращающий набор данных (выборку) из базы данных.

```
sqlite> SELECT * FROM films;
```

# или:

```
sqlite> SELECT id, name FROM films WHERE id > 3 ORDER BY id DESC LIMIT 5;
```

- **UPDATE** — оператор языка SQL, позволяющий обновить значения в заданных столбцах таблицы.

```
sqlite> UPDATE films SET name = 'New Film Name' WHERE id = 1;
```

- **DELETE** — в языках, подобных SQL, операция удаления записей из таблицы. Критерий отбора записей для удаления определяется выражением **where**. В случае, если критерий отбора не определён, выполняется удаление всех записей:

```
sqlite> DELETE FROM films WHERE id = 6;
```

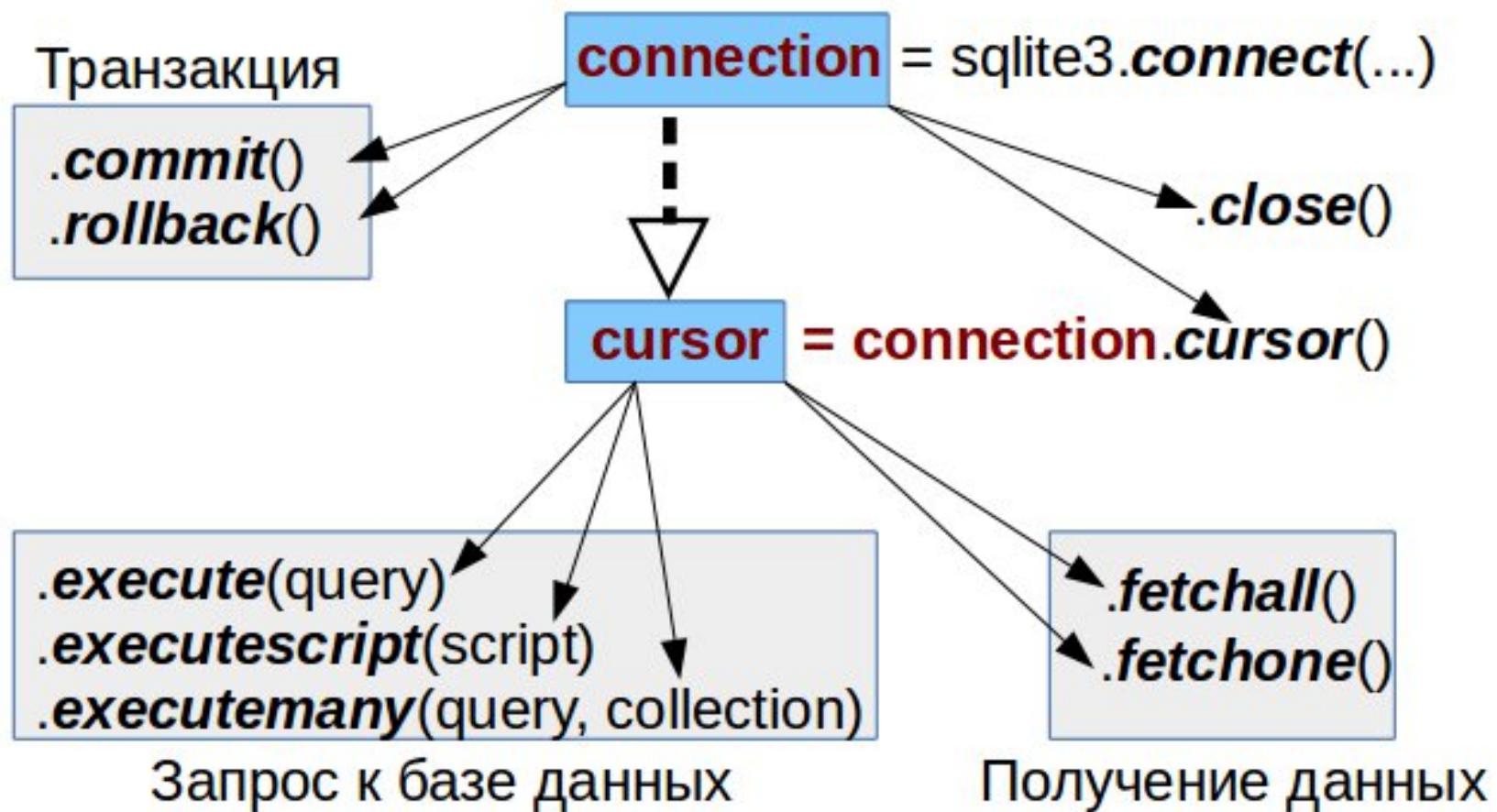
# или:

```
sqlite> DELETE FROM films;
```

# Python DB-API

PEP 249 определяет DB-API - набор методов и интерфейсов для работы с базами данных

## Python DB-API методы



# Работа с БД через Python DB-API

Для работы с SQLite в Python используется библиотека sqlite3. Рассмотрим общий порядок работы с этой библиотекой на примере задачи построения БД организации.

```
# Импортируем библиотеку, соответствующую типу нашей базы данных
import sqlite3

# Файл базы данных
# Если вместо файла указать :memory:, то база будет создана
# в оперативной памяти, а не в файле.
db_name = "example.db"

# Создаем соединение с нашей базой данных
# Если файл базы данных еще не создан, он создастся автоматически.
conn = sqlite3.connect(db_name)

# При необходимости меняем тип row_factory, чтобы в ответах
# базы данных отображались названия атрибутов.
conn.row_factory = sqlite3.Row

# РАБОТАЕМ С БАЗОЙ

# Не забываем закрыть соединение с базой данных после работы
conn.close()
```

# Скрипт конфигурирования БД: CREATE

```
# Конфигурирование базы данных (если необходимо выполнить в скрипте)
def configure_db(conn):
    cur = conn.cursor()

    # Создаем таблицу Employees
    cur.execute("CREATE TABLE Employees"
               "    (Id          INTEGER      PRIMARY KEY AUTOINCREMENT,"
               "     Name        CHAR(128)    NOT NULL,"
               "     Position    CHAR(64)     NOT NULL,"
               "     Bonus       INTEGER      DEFAULT 0,"
               "     Login       CHAR(16)     NOT NULL,"
               "     Password    CHAR(16)     NOT NULL)")

    # Создаем таблицу Projects
    cur.execute("CREATE TABLE Projects"
               "    (Id          INTEGER      PRIMARY KEY AUTOINCREMENT,"
               "     Name        CHAR(128)    NOT NULL)")

    # Создаем таблицу PositionSalary
    cur.execute("CREATE TABLE PositionSalary"
               "    (Position    CHAR(64)     PRIMARY KEY NOT NULL,"
               "     Salary      INTEGER      NOT NULL)")

    # Создаем таблицу EmployeeProject
    cur.execute("CREATE TABLE EmployeeProject"
               "    (EmployeeId  INTEGER,"
               "     ProjectId   INTEGER,"
               "     PRIMARY KEY (EmployeeId, ProjectId))")
```

# Добавление записей: INSERT

```
# Добавление записей в таблицу Проекты
def insert_project(conn, name):
    # Создаем курсор - специальный объект,
    # который делает запросы и получает их результаты
    cur = conn.cursor()
    # Делаем INSERT запрос к базе данных, используя обычный SQL-синтаксис
    cur.execute("INSERT INTO Projects (Name) VALUES (:name)",
               {'name': name})
    # Если мы не просто читаем, но и вносим изменения в базу данных
    # - необходимо сохранить транзакцию
    conn.commit()
```

```
# Добавление записей в таблицу ДолжностьОклад
def insert_position(conn, position, salary):
    cur = conn.cursor()
    cur.execute("INSERT INTO PositionSalary (Position, Salary)"
               " VALUES (:position, :salary)",
               {'position': position, 'salary': salary})
    conn.commit()
```

# Добавление записей: INSERT

```
# Добавление записей в таблицу Сотрудники
def insert_employee(conn, name, position, bonus, login, pwd):
    cur = conn.cursor()
    cur.execute("INSERT INTO Employees (Name, Position, Bonus, Login, Password)"
               " VALUES (:name, :position, :bonus, :login, :pwd)",
               {'name': name, 'position': position, 'bonus': bonus,
                'login': login, 'pwd': pwd})
    conn.commit()
```

```
# Добавление записей в таблицу СотрудникиПроекты
def add_employee_to_project(conn, employee_id, project_id):
    cur = conn.cursor()
    cur.execute("INSERT INTO EmployeeProject (EmployeeId, ProjectId)"
               " VALUES (:employeeId, :projectId)",
               {'employeeId': employee_id, 'projectId': project_id})
    conn.commit()
```

# Создание и начальное наполнение БД

```
db_name = "example.db"
db_exists = os.path.exists(db_name)

conn = sqlite3.connect(db_name)
conn.row_factory = sqlite3.Row

if not db_exists:
    configure_db(conn)

    insert_project(conn, "Важный")
    insert_project(conn, "Срочный")

    insert_position(conn, "инженер", 50000)
    insert_position(conn, "старший инженер", 51000)
    insert_position(conn, "менеджер проекта", 100000)

    insert_employee(conn, "Иванов И.И.", "инженер", 30000,
                    "ivanovi", "ivanov123")
    insert_employee(conn, "Петров П.П.", "старший инженер", 50000,
                    "petrovp", "ple2t3")
    insert_employee(conn, "Сидоров С.С.", "менеджер проекта", 30000,
                    "sidorovs", "zayka88")

    add_employee_to_project(conn, 1, 1)
    add_employee_to_project(conn, 2, 1)
    add_employee_to_project(conn, 2, 2)
    add_employee_to_project(conn, 3, 2)
```

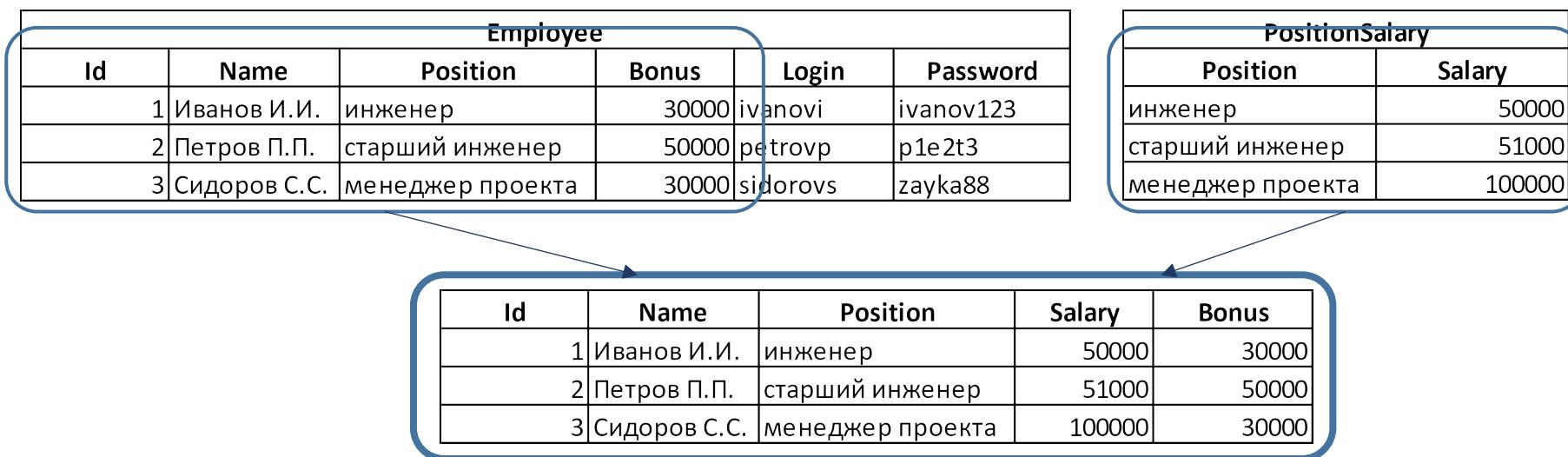
# Чтение данных: SELECT

```
# Проверка наличия пользователя в базе данных
# с указанным логином/паролем
def authentication(conn, login, pwd):
    cur = conn.cursor()
    # Делаем SELECT запрос к базе данных, используя обычный SQL-синтаксис
    cur.execute("SELECT E.Id, E.Name, E.Position, EP.ProjectId"
               " FROM Employees AS E, EmployeeProject AS EP"
               " WHERE E.Id = EP.EmployeeId"
               " AND E.Login = :login AND E.Password = :pwd",
               {'login': login, 'pwd': pwd})
    # Получаем результат сделанного запроса
    return cur.fetchone()

# Проверка наличия указанного сотрудника в указанном проекте
def is_employee_in_project(conn, employee_id, project_id):
    cur = conn.cursor()
    cur.execute("SELECT EP.ProjectId"
               " FROM EmployeeProject AS EP"
               " WHERE EP.EmployeeId = :employee_id"
               " AND EP.ProjectId = :project_id",
               {'employee_id': employee_id, 'project_id': project_id})
    return bool(cur.fetchone())
```

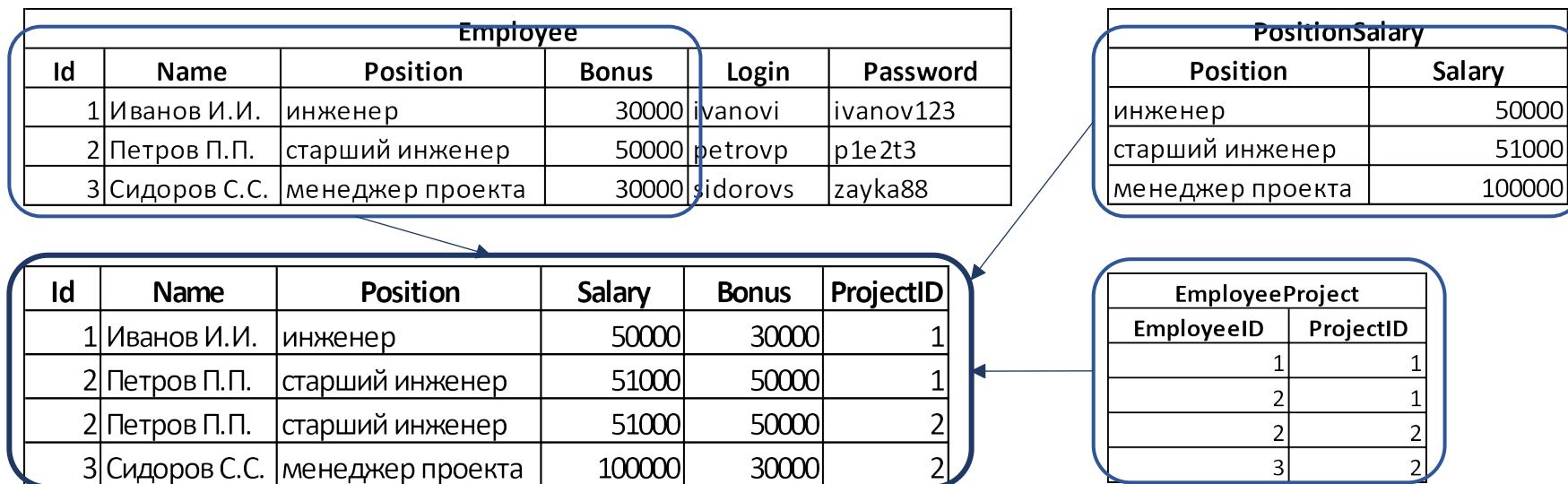
# Чтение данных: SELECT

```
# Вывод информации для сотрудника
# Соединяем таблицы Employees, PositionSalary
def show_employee_info(conn, employee_id):
    cur = conn.cursor()
    cur.execute("SELECT E.Id, E.Name, P.Salary + E.Bonus As Pay"
               " FROM Employees AS E, PositionSalary AS P"
               " WHERE E.Position = P.Position"
               " AND E.Id = :employee_id",
               {'employee_id': employee_id})
    print("Информация для сотрудника:")
    for row in cur.fetchall():
        print(dict(row))
```



# Чтение данных: SELECT

```
# Вывод информации для менеджера проекта
# Соединяем таблицы Employees, PositionSalary, EmployeeProject
def show_manager_info(conn, project_id):
    cur = conn.cursor()
    cur.execute("SELECT E.Id, E.Name, P.Salary + E.Bonus As Pay"
               " FROM Employees AS E, PositionSalary AS P, "
               " EmployeeProject AS EP"
               " WHERE E.Position = P.Position"
               " AND E.Id = EP.EmployeeId"
               " AND EP.ProjectId = :project_id",
               {'project_id': project_id})
    print("Информация для менеджера:")
    for row in cur.fetchall():
        print(dict(row))
```



# Изменение данных: UPDATE и DELETE

```
# Изменение премии сотрудника
def update_employee_bonus(conn, employee_id, new_bonus):
    cur = conn.cursor()
    # Делаем UPDATE запрос к базе данных, используя обычный SQL-синтаксис
    cur.execute("UPDATE Employees"
                " SET Bonus = :new_bonus"
                " WHERE Id = :employee_id",
                {'employee_id': employee_id, 'new_bonus': new_bonus})
    conn.commit()

# Удаление сотрудника из проекта (но не из базы данных)
def delete_employee_from_project(conn, employee_id, project_id):
    cur = conn.cursor()
    # Делаем DELETE запрос к базе данных, используя обычный SQL-синтаксис
    cur.execute("DELETE FROM EmployeeProject"
                " WHERE EmployeeId = :employee_id"
                " AND ProjectId = :project_id",
                {'employee_id': employee_id, 'project_id': project_id})
    conn.commit()
```

# Решение задачи (бета-версия)

```
login = input("Логин: ")
pwd = input("Пароль: ")

res = authentication(conn, login, pwd)
if res:
    user = dict(res)
    print(f"Здравствуйте, {user['Name']}")

    if user['Position'] == "менеджер проекта":
        show_manager_info(conn, user['ProjectId'])

    id_upd = int(input("Изменение премии. ID сотрудника (0 - отмена): "))
    if id_upd:
        if (id_upd != user['Id'] and
            is_employee_in_project(conn, id_upd, user['ProjectId'])):
            new_bonus = input("Новая премия: ")
            update_employee_bonus(conn, id_upd, new_bonus)
        else:
            print("Невозможно изменить премию для данного сотрудника")

    id_del = int(input("Удаление сотрудника. ID сотрудника (0 - отмена): "))
    if id_del:
        if id_del != user['Id']:
            delete_employee_from_project(conn, id_del, user['ProjectId'])
        else:
            print("Невозможно удалить данного сотрудника из проекта")
    else:
        show_employee_info(conn, user['Id'])
else:
    print("Доступ запрещен")
```

# Тестирование бета-версии

---

```
Логин: sidoroovs
Пароль: zayka88
Здравствуйте, Сидоров С.С.
Информация для менеджера:
{'Id': 2, 'Name': 'Петров П.П.', 'Pay': 101000}
{'Id': 3, 'Name': 'Сидоров С.С.', 'Pay': 130000}
Изменение премии. ID сотрудника (0 - отмена): 2
Новая премия: 60000
Удаление сотрудника. ID сотрудника (0 - отмена): 0
```

---

```
Логин: sidoroovs
Пароль: zayka88
Здравствуйте, Сидоров С.С.
Информация для менеджера:
{'Id': 2, 'Name': 'Петров П.П.', 'Pay': 111000}
{'Id': 3, 'Name': 'Сидоров С.С.', 'Pay': 130000}
Изменение премии. ID сотрудника (0 - отмена): 0
Удаление сотрудника. ID сотрудника (0 - отмена): 2
```

---

```
Логин: sidoroovs
Пароль: 123
Доступ запрещен
```

# SQL-инъекции: уязвимый код

```
def bad_authentication(conn, login, pwd):  
    cur = conn.cursor()  
    cur.execute("SELECT E.Id, E.Name, E.Position, EP.ProjectId"  
               " FROM Employees AS E, EmployeeProject AS EP"  
               " WHERE E.Id = EP.EmployeeId"  
               " AND E.Login = '{login}' AND E.Password = '{pwd}'".  
               format(login=login, pwd=pwd))  
    return cur.fetchone()
```

---

Логин: ivanovi  
Пароль: ivanov123  
Здравствуйте, Иванов И.И.  
Информация для сотрудника:  
{'Id': 1, 'Name': 'Иванов И.И.', 'Pay': 80000}

---

Логин: ivanovi  
Пароль: 123  
Доступ запрещен

---

Логин: ivanovi  
Пароль: 123' OR 'a'='a  
Здравствуйте, Иванов И.И.  
Информация для сотрудника:  
{'Id': 1, 'Name': 'Иванов И.И.', 'Pay': 80000}

# SQL-инъекции: защищенный код

```
def authentication(conn, login, pwd):
    cur = conn.cursor()
    cur.execute("SELECT E.Id, E.Name, E.Position, EP.ProjectId"
                " FROM Employees AS E, EmployeeProject AS EP"
                " WHERE E.Id = EP.EmployeeId"
                " AND E.Login = :login AND E.Password = :pwd",
                {'login': login, 'pwd': pwd})
    return cur.fetchone()

def authentication2(conn, login, pwd):
    cur = conn.cursor()
    cur.execute("SELECT E.Id, E.Name, E.Position, EP.ProjectId"
                " FROM Employees AS E, EmployeeProject AS EP"
                " WHERE E.Id = EP.EmployeeId"
                " AND E.Login = ? AND E.Password = ?",
                (login, pwd))
    return cur.fetchone()
```

---

Логин: ivanovi

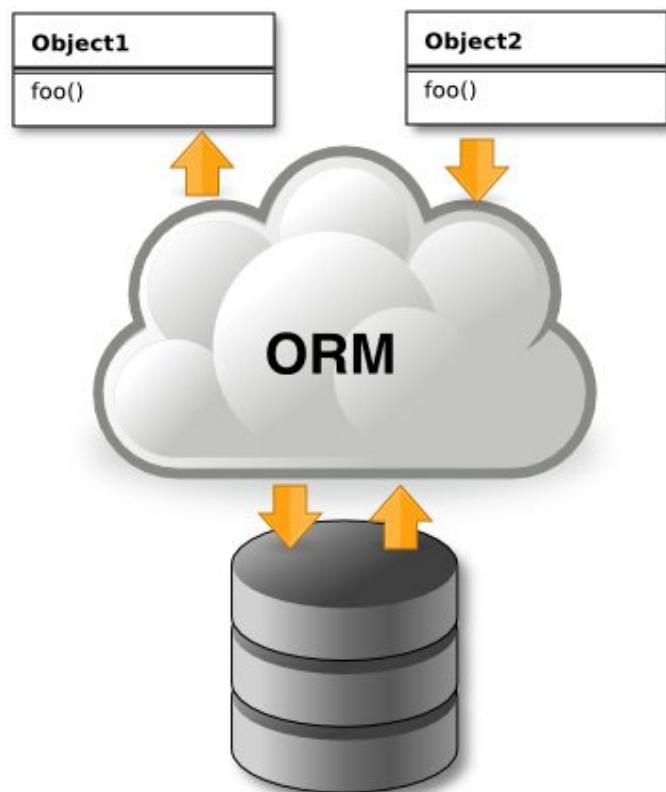
Пароль: 123' OR 'a'='a

Доступ запрещен

# ORM

ORM (Object-Relational Mapping – объектно-реляционное преобразование) – технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая "виртуальную объектную базу данных" (Wiki).

Известные ORM в Python: SQLAlchemy, DjangoORM, peewee, PonyORM, SQLObject, Storm, quick\_orm и другие.



# Классы для таблиц

```
class PositionSalary(Base):  
    __tablename__ = 'position_salary'  
    position = Column(String, primary_key=True)  
    salary = Column(Integer, nullable=False)
```

PositionSalary	
Position	Salary
инженер	50000
старший инженер	51000
менеджер проекта	100000

```
class Employee(Base):  
    __tablename__ = 'employee'  
    id = Column(Integer, primary_key=True)  
    name = Column(String, nullable=False)  
    position = Column(String, ForeignKey('position_salary.position'), nullable=False)  
    bonus = Column(Integer, default=0)  
    login = Column(String, nullable=False, unique=True)  
    password = Column(String, nullable=False)
```

Employee					
Id	Name	Position	Bonus	Login	Password
1	Иванов И.И.	инженер	30000	ivanovi	ivanov123
2	Петров П.П.	старший инженер	50000	petrovvp	p1e2t3
3	Сидоров С.С.	менеджер проекта	30000	sidorovs	zayka88

# Преимущества и недостатки ORM

## Преимущества

- Сокращение кода
- Единая парадигма программирования
- Независимость от диалекта SQL



## Что это дает

- Ускорение разработки
- Простота понимания всего кода
- Универсальность методов отладки
- Кросс-СУБД код

## Недостатки

- Медленнее чистого SQL
- Требует больше памяти
- Уступает в полноте и гибкости



## Однако

Программист при необходимости может сам задать код SQL-запросов, который будет использоваться при тех или иных действиях

# SQLAlchemy

SQLAlchemy – библиотека Python для работы с базами данных по технологии ORM. Она позволяет ассоциировать пользовательские классы Python с таблицами баз данных, и объекты этих классов с записями в соответствующих таблицах. Она включает в себя систему, прозрачно синхронизирующую все изменения в состояниях между объектами и соответствующими строками, равно как и систему для выполнения запросов к базе данных в терминах пользовательских классов и с учетом взаимосвязей этих классов.

Таким образом, SQLAlchemy предоставляет:

- ORM уровень
- обобщенный API для работы с различными СУБД
- интерфейс, достаточно близкий по полноте к чистому SQL
- возможность использования прямых SQL-запросов

SQLAlchemy

# SQLAlchemy vs DB-API

```
# Добавление нового сотрудника с привязкой к проекту средствами SQLAlchemy
def add_new_employee_to_project(self, name, position, bonus, login, password, project_id):
    e = Employee(name=name, position=position, bonus=bonus, login=login, password=password)
    self._session.add(e)
    self._session.commit()
    self._session.add(EmployeeProject(employee_id=e.id, project_id=project_id))
    self._session.commit()

# Добавление нового сотрудника с привязкой к проекту средствами DB-API
def add_new_employee_to_project(conn, name, position, bonus, login, pwd, project_id):
    cur = conn.cursor()
    cur.execute("INSERT INTO Employees (Name, Position, Bonus, Login, Password)"
               " VALUES (:name, :position, :bonus, :login, :pwd)",
               {'name': name, 'position': position, 'bonus': bonus,
                'login': login, 'pwd': pwd})
    conn.commit()
    cur.execute("SELECT E.Id FROM Employees AS E "
               "WHERE E.Login = :login AND E.Password = :pwd",
               {'login': login, 'pwd': pwd})
    employee_id = dict(cur.fetchone())['Id']
    cur.execute("INSERT INTO EmployeeProject (EmployeeId, ProjectId)"
               " VALUES (:employeeId, :projectId)",
               {'employeeId': employee_id, 'projectId': project_id})
    conn.commit()
```

# Описание классов для БД организации

```
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer, String, ForeignKey

Base = declarative_base()

class Employee(Base):
    __tablename__ = 'employee'
    id = Column(Integer, primary_key=True)
    name = Column(String, nullable=False)
    position = Column(String, ForeignKey('position_salary.position'), nullable=False)
    bonus = Column(Integer, default=0)
    login = Column(String, nullable=False, unique=True)
    password = Column(String, nullable=False)

class Project(Base):
    __tablename__ = 'project'
    id = Column(Integer, primary_key=True)
    name = Column(String, nullable=False)

class PositionSalary(Base):
    __tablename__ = 'position_salary'
    position = Column(String, primary_key=True)
    salary = Column(Integer, nullable=False)

class EmployeeProject(Base):
    __tablename__ = 'employee_project'
    employee_id = Column(Integer, ForeignKey('employee.id'), primary_key=True)
    project_id = Column(Integer, ForeignKey('project.id'), primary_key=True)
```

# Подключение к БД через engine

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

class DBClient:
    def __init__(self, dbtype='sqlite', dbname='/example.db', username=None, password=None):
        self._engine = self._get_engine(dbtype, dbname, username, password)

    def __enter__(self):
        self._session = sessionmaker(bind=self._engine)()
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        self._session.close_all()

    @staticmethod
    def _get_engine(dbtype, dbname, username=None, password=None):
        if username:
            if password:
                login = f'{username}:{password}'
            else:
                login = username
                dbstr = f'{login}@{dbname}'
        else:
            dbstr = dbname
        engine = create_engine(f'{dbtype}:///{dbstr}')
        return engine
```

# Создание схемы данных для ORM

```
def create_schema(self):
    # Создаем схему
    Base.metadata.create_all(self._engine)

def delete_schema(self):
    # Удаляем схему
    Base.metadata.drop_all(self._engine)
```

# Добавление записей через ORM

```
# Добавление записей в таблицу ДолжностьОклад
def insert_position(self, position, salary):
    self._session.add(PositionSalary(position=position, salary=salary))
    self._session.commit()

# Добавление записей в таблицу Проекты
def insert_project(self, name):
    p = Project(name=name)
    self._session.add(p)
    self._session.commit()
    return p.id

# Добавление записей в таблицу Сотрудники
def insert_employee(self, name, position, bonus, login, password):
    e = Employee(name=name, position=position, bonus=bonus, login=login, password=password)
    self._session.add(e)
    self._session.commit()
    return e.id

# Добавление записей в таблицу СотрудникиПроекты
def add_employee_to_project(self, employee_id, project_id):
    self._session.add(EmployeeProject(employee_id=employee_id, project_id=project_id))
    self._session.commit()
```

# Создание и первичное наполнение БД через ORM

```
db_type = "sqlite"
db_name = "example.db"
db_exists = os.path.exists(db_name)

if not db_exists:
    with DBClient(db_type, db_name) as dbc:
        dbc.create_schema()

        dbc.insert_position("инженер", 50000)
        dbc.insert_position("старший инженер", 51000)
        dbc.insert_position("менеджер проекта", 100000)

    pid = dbc.insert_project("Важный")
    eid = dbc.insert_employee("Иванов И.И.", "инженер", 30000, "ivanovi", "ivanov123")
    dbc.add_employee_to_project(eid, pid)

    eid = dbc.insert_employee("Петров П.П.", "старший инженер", 50000, "petrovvp", "ple2t3")
    dbc.add_employee_to_project(eid, pid)

    pid = dbc.insert_project("Срочный")
    dbc.add_employee_to_project(eid, pid)

    eid = dbc.insert_employee("Сидоров С.С.", "менеджер проекта", 30000, "sidorovs", "zayka88")
    dbc.add_employee_to_project(eid, pid)
```

# Чтение данных через ORM

```
# Проверка наличия пользователя в базе данных с указанным логином/паролем
def authentication(self, login, password):
    try:
        res = self._session.query(Employee.id, Employee.name, Employee.position, EmployeeProject.project_id).\
            join(EmployeeProject, EmployeeProject.employee_id == Employee.id).\
            filter(and_(Employee.login == login, Employee.password == password)).\
            one()
        return res
    except MultipleResultsFound:
        print("Multiple Results Found")
    except NoResultFound:
        print("No Result Found")
    return None

# Проверка наличия указанного сотрудника в указанном проекте
def is_employee_in_project(self, employee_id, project_id):
    try:
        res = self._session.query(EmployeeProject.project_id).\
            filter(and_(EmployeeProject.employee_id == employee_id, EmployeeProject.project_id == project_id)).\
            one()
        return True
    except MultipleResultsFound:
        print("Multiple Results Found")
    except NoResultFound:
        print("No Result Found")
    return None
```

# Чтение данных через ORM

```
# Вывод информации по сотруднику (соединяем таблицы Employees, PositionSalary)
def show_employee_info(self, employee_id):
    res = self._session.query(Employee.id, Employee.name,
                             (PositionSalary.salary + Employee.bonus).label("Pay")).\
        filter(and_(Employee.position == PositionSalary.position, Employee.id == employee_id)).\
        all()
    print("Информация для сотрудника:")
    for row in res:
        print(row)
    return res

# Вывод информации по проекту (соединяем таблицы Employees, PositionSalary, EmployeeProject)
def show_manager_info(self, project_id):
    res = self._session.query(Employee.id, Employee.name, (PositionSalary.salary + Employee.bonus).label("Pay")).\
        filter(and_(Employee.position == PositionSalary.position,
                   Employee.id == EmployeeProject.employee_id,
                   EmployeeProject.project_id == project_id)).\
        all()
    print("Информация для менеджера:")
    for row in res:
        print(row)
    return res
```

# Изменение данных через ORM

```
# Изменение премии сотрудника
def update_employee_bonus(self, employee_id, new_bonus):
    e = self._session.query(Employee).get(employee_id)
    if e:
        e.bonus = new_bonus
        self._session.add(e)
        self._session.commit()

# Удаление сотрудника из проекта (но не из базы данных)
def delete_employee_from_project(self, employee_id, project_id):
    ep = self._session.query(EmployeeProject).get((employee_id, project_id))
    if ep:
        self._session.delete(ep)
        self._session.commit()
```

# Решение задачи через ORM (альтернативная версия)

```
with DBClient(db_type, db_name) as dbc:
    res = dbc.authentication(login, pwd)
    if res:
        user = res._asdict()
        print(f"Здравствуйте, {user['name']}!")
        if user['position'] == "менеджер проекта":
            dbc.show_manager_info(user['project_id'])
            id_upd = int(input("Изменение премии. ID сотрудника (0 - отмена): "))
            if id_upd:
                if (id_upd != user['id'] and dbc.is_employee_in_project(id_upd, user['project_id'])):
                    new_bonus = input("Новая премия: ")
                    dbc.update_employee_bonus(id_upd, new_bonus)
                    dbc.show_manager_info(user['project_id'])
                else:
                    print("Невозможно изменить премию для данного сотрудника")
            id_del = int(input("Удаление сотрудника. ID сотрудника (0 - отмена): "))
            if id_del:
                if id_del != user['id']:
                    dbc.delete_employee_from_project(id_del, user['project_id'])
                    dbc.show_manager_info(user['project_id'])
                else:
                    print("Невозможно удалить данного сотрудника из проекта")
            else:
                dbc.show_employee_info(user['id'])
        else:
            print("Доступ запрещен")
```

# Тестирование альтернативной версии

---

```
Логин: sidorovs
Пароль: zayka88
Здравствуйте, Сидоров С.С.
Информация для менеджера:
(2, 'Петров П.П.', 101000)
(3, 'Сидоров С.С.', 130000)
Изменение премии. ID сотрудника (0 - отмена): 2
Новая премия: 60000
Информация для менеджера:
(2, 'Петров П.П.', 111000)
(3, 'Сидоров С.С.', 130000)
Удаление сотрудника. ID сотрудника (0 - отмена): 0
```

---

```
Логин: sidorovs
Пароль: zayka88
Здравствуйте, Сидоров С.С.
Информация для менеджера:
(2, 'Петров П.П.', 111000)
(3, 'Сидоров С.С.', 130000)
Изменение премии. ID сотрудника (0 - отмена): 0
Удаление сотрудника. ID сотрудника (0 - отмена): 2
Информация для менеджера:
(3, 'Сидоров С.С.', 130000)
```

# Практика

1. Написать класс-обертку над SQLite (с возможностями менеджера контекста), которая может на вход принимать строки SQL запросов и возвращать данные в формате json. Класс должен иметь, как минимум, методы select и execute.
2. \* Написать скрипт, работающий под SQLite/MySQL/PostgreSQL, который создает 3 сущности: производители, покупатели, товары. Необходимо добавить демо-данные и выполнить следующие выборки:
  - вывести все товары с указанием информации о производителе
  - найти все товары, которые никто не покупал