# Технологии хранения данных на примере работы с базой данных SQLite

**Цель работы**: познакомится с принципами хранения и обработки данных в СУБД, изучить язык структурных запросов SQL, научится создавать приложения на языке Python, работающие с базами данных.

## ***Теоретическая часть***

В настоящее время имеются свободно распространяемые интерфейсы, позволяющие сценариям Python использовать не только свободные, но и коммерческие системы реляционных баз данных: MySQL, Oracle, Sybase, Informix, InterBase, PostgreSQL (Postgres), SQLite, ODBC и другие.

Начиная с версии Python 2.5, в стандартную библиотеку Python была включена встроенная поддержка системы реляционных баз данных SQLite. Поскольку эта система поддерживает переносимый API баз данных, она может работать практически без изменений с более полноценными базами данных, такими как MySQL или Oracle.

SQLite является стандартной частью Python. Механизм SQLite реализует законченную систему управления реляционными базами данных, однако он имеет вид библиотеки, подключаемой к процессу, а не сервера. В целом, такая организация больше подходит для создания хранилищ в программах, чем для удовлетворения потребностей хранения данных промышленного масштаба.

Однако благодаря переносимости API баз данных в языке Python другие пакеты популярных баз данных, таких как PostgreSQL, MySQL и Oracle, используются практически идентично – для сценариев, использующих стандартный программный код SQL, основные отличия будут заключаться в начальном вызове для подключения к базе данных, который для разных баз данных обычно требует передачи разных значений аргументов.

### Язык SQL

Информация в реляционной базе данных хранится в таблицах и в связях между таблицами. Таблицы двумерны, имеют фиксированное количество колонок (столбцов, полей) и произвольное количество строк (записей). Каждая колонка имеет наименование и содержит данные определенного типа. Для идентификации записи применяется т.н. первичный ключ − одно или несколько полей, которые содержат уникальные в пределах таблицы значения. Наибольшее распространение получил язык запросов [SQL](http://ru.wikipedia.org/wiki/Sql) (акроним от английского Structured Query Language − структурированный язык запросов). Пример простой команды на языке SQL, которая покажет все колонки и все строки из таблицы [CONTACT](http://gsbelarus.com/gs/wiki/index.php/GD_CONTACT)S:

SELECT \* FROM contacts

В свою очередь, язык SQL подразделяется на три подмножества: DDL (Data Definition Language), DML (Data Manipulation Language) и DCL (Data Control Language).

DDL определяет набор команд, с помощью которых в базе данных создаются структурные объекты, т.н. метаданные, − таблицы, домены, внешние ключи, индексы, хранимые процедуры и т.п.

На языке DML пишутся запросы на извлечение данных из базы, а так же на вставку, изменение или удаление записей в таблице.

И, наконец, язык DCL содержит набор операторов для разграничения доступа к данным для различных пользователей СУБД.

**Типы данных**. В большинстве движков баз данных SQL используется строгая статическая типизация. При статической типизации, тип данных того или иного значения определяется его контейнером — конкретным столбцом, в котором это значение хранится. SQLite использует более общую систему типизации — динамическую, когда тип данных значения связан с самим значением, а не с его контейнером. Динамическая система SQLite имеет обратную совместимость со статическими системами других СУБД. В том смысле, что SQL-запросы статически типизированных баз данных должны работать так же и с SQLite. Однако, динамическая типизация в SQLite позволяет выполнять операции, невозможные в традиционных жестко типизированных базах данных.

Каждое значение, хранящееся в базе данных SQLite (или обрабатываемое движком), имеет один из следующих классов хранения:

* NULL. Пустое значение в таблице базы.
* INTEGER. Целочисленное значение, хранящееся в 1, 2, 3, 4, 6 или 8 байтах, в зависимости от величины самого значения.
* REAL. Числовое значение с плавающей точкой. Хранится в формате 8-байтного числа IEEE с плавающей точкой.
* TEXT. Значение строки текста. Хранится с использованием кодировки базы данных (UTF-8, UTF-16BE или UTF-16LE).
* BLOB. Значение бинарных данных, хранящихся точно в том же виде, в каком были введены. Применяется для хранения в базе картинок, файлов, больших текстов, потоков данных и т.п.

Отметим, что класс хранения — более широкое понятие, чем тип данных. К примеру, класс хранения INTEGER включает 6 различных типов целочисленных данных различной длины. На диске это записывается по-разному. Но как только целочисленные значения считываются с диска и поступают для обработки в оперативную память, они преобразуются в наиболее общий тип данных (8-байтное целое число). Следовательно, хранение по системе класса данных в практическом плане неотличимо от хранения по типу данных, и они могут быть взаимозаменяемыми.

Любой столбец базы данных SQLite версии 3, за исключением столбцов INTEGER PRIMARY KEY, может быть использован для хранения значений любого класса.

Все значения в инструкциях SQL имеют неявный класс хранения. В условиях, описанных ниже, движок базы данных может конвертировать значения между числовыми классами хранения (INTEGER и REAL) и TEXT во время выполнения запроса.

SQLite не имеет отдельного логического класса хранения. Вместо этого, логические значения хранятся как целые числа 0 (false) и 1 (true).

SQLite не имеют классов, предназначенных для хранения дат и/или времени. Вместо этого, встроенные функции даты и времени в SQLite способны работать с датами и временем, сохраненными в виде значений TEXT, REAL и INTEGER в следующих форматах:

* TEXT как строка формата ISO8601 ("YYYY-MM-DD HH:MM:SS.SSS").
* REAL как числа юлианского календаря. То есть число дней с полудня 24 ноября 4714 г. до н.э. по Гринвичу в соответствии с ранним григорианским календарём.
* INTEGER как время Unix, — количество секунд с 1970-01-01 00:00:00 UTC.

В приложениях следует выбирать, в каком из этих форматов хранить даты и время, а затем можно свободно конвертировать из одного формата в другой с помощью встроенных функций даты и времени.

CREATE TABLE

Новая таблица создаётся с помощью команды CREATE TABLE после которой указывается имя таблицы, а затем в круглых скобках указываются имена столбцов с параметрами.

Создать новую таблицу tbl\_info с четырьмя столбцами:

CREATE TABLE tbl\_info (

id INTEGER PRIMARY KEY AUTOINCREMENT,

name TEXT NOT NULL,

age INTEGER NOT NULL,

city TEXT NOT NULL

);

Ключевое слово NOT NULL требует обязательно присваивать значение столбцу при команде INSERT INTO.

Ключевое слово DEFAULT, позволяет вставить значение по умолчанию, например, city TEXT NOT NULL DEFAULT "Москва".

**Первичный ключ** - столбец таблицы, имеющий уникальное значение для каждой записи. Назначается при создании таблицы. Ключ не может содержать NULL, потому что теряется уникальность, ведь в других записях тоже может оказаться NULL. Значения первичного ключа должны оставаться неизменными.

Во многих случаях для этой цели создают новый столбец, который будет содержать уникальный номер. **id**.

Для указания первичного ключа используется ключевое слово PRIMARY KEY.

Чтобы база данных сама заботилась об уникальности первичного ключа, можно добавить к нему ключевое слово AUTOINCREMENT, которое будет автоматически увеличивать значение на единицу при вставке новой записи.

DROP TABLE

Удаление таблицы происходит с помощью команды DROP TABLE, затем указывается имя таблицы. При желании можно поставить проверочное условие IF EXISTS.

Удалить таблицу tbl\_info:

DROP TABLE IF EXISTS tbl\_info;

INSERT INTO VALUES

Для вставки новой записи в таблицу используется команда INSERT INTO, затем указывается имя таблицы, а в скобках имена столбцов. После них идёт ключевое слово VALUES, после которого в скобках идут вставляемые значения. Важно соблюдать количество столбцов с вставляемыми данными и их очерёдность при перечислении. Можно указывать только нужные столбцы для вставки данных.

Вставить запись в таблицу **tbl\_info**:

INSERT INTO tbl\_info(name, age, city) VALUES ("Сергей", 28, "Москва");

Существует укороченная запись без перечисления столбцов. В этом случае нужно указывать все значения и в том порядке, в котором создавались соответствующие столбцы.

INSERT INTO tbl\_info VALUES (3, "Иван", 25, "Питер");

При наличии автонумерации, после такой вставки, нумерация продолжится с с 4, т.е. могут возникать пропуски.

SELECT \* FROM

Для чтения данных используется команда SELECT с условием, затем ключевое слово FROM с указанием имени таблицы. Чтобы просмотреть все записи, для условия SELECT используется звёздочка (\*).

SELECT \* FROM tbl\_info;

UPDATE

Обновить запись в таблице tbl\_info:

UPDATE tbl\_info SET age=20 WHERE \_id=1;

После UPDATE указываете таблицу, после SET - в каком столбце нужно внести изменения и указывается новое значение, а затем указывается условие.

Можно обновлять группу столбцов, указывая их через запятую.

UPDATE ваша\_таблица SET первый\_столбец="новое значение", второй\_столбец="новое значение";

Команда UPDATE заменяет собой пару команд INSERT/DELETE. Обновить данные в нужном столбце:

UPDATE table\_name SET имя\_столбца = новое\_значение WHERE имя\_столбца = старое\_значение

Также можно производить математические действия: прибавлять, отнимать, умножать, делить. Увеличим возраст на единицу.

UPDATE tbl\_info SET age=age+1 WHERE name="Иван";

DELETE

Удалить запись из таблицы tbl\_info:

DELETE FROM tbl\_info WHERE \_id=1;

Вам не надо перечислять все столбцы, достаточно указать в условии нужный столбец. Условие WHERE работает аналогично как в команде SELECT и позволяет использовать ключевые слова LIKE, BETWEEN и т.д.

Удалить все записи:

DELETE FROM tbl\_info;

SELECT

Рассмотрим подробнее команду SELECT.

Чтобы не искать все записи, можно ограничить поиск условием WHERE, после которого идёт имя столбца и условие равенства. Показать всех людей, чей возраст больше 20.

SELECT \* FROM tbl\_info WHERE age > 20;

Вместо звёздочки можно указать столбцы, которые вам нужны. Например, нам нужны только имена с этим же условием.

SELECT name FROM tbl\_info WHERE age > 20;

Столбцы указываются через запятую. Если нам нужны имена и адреса с этим же условием:

SELECT name, city FROM tbl\_info WHERE age > 20;

Условие WHERE можно объединять с помощью ключевого слова AND. Список людей старше 20 лет и проживающих в Москве.

SELECT name, city FROM tbl\_info WHERE age > 20 AND city="Москва";

Также доступны слова OR (ИЛИ):

SELECT name, city FROM tbl\_info WHERE age > 20 OR city="Москва";

Слов AND и OR может быть несколько в одном запросе.

Проверку на NULL (поле не было заполнено) можно сделать с помощью ключевого слова IS NULL (если столбец таблицы создавался без NOT NULL)

SELECT \* FROM tbl\_info WHERE age IS NULL;

Значения колонок можно сравнивать с константами, значениями других колонок, результатами выражений или подзапросов.

При этом используются следующие операторы:

=, <> или !=, >, <, >=, <=;

BETWEEN;

IN (список констант или подзапрос) - проверяет, принадлежит ли значение поля указанному множеству;

LIKE - для строковых полей проверяет соответствие поля указанной маске. В тексте маски можно применять символы \_ и %. Первый заменяет один произвольный символ в указанной позиции, а второй произвольное число символов;

Ключевое слово LIKE позволяет сократить множество операторов OR. Например, мы хотим узнать имена людей, которые заканчиваются на "ей":

SELECT name FROM tbl\_info WHERE name LIKE '%ей'

Символ % в строке указывает на любое слово с нужным окончанием (представляет любое количество неизвестных символов).

Также можно использовать спецсимвол **\_** для одного символа. Ищем Ивана.

select name from tbl\_info where name like '\_ван'

С помощью ключевого слова BETWEEN можно быстро и удобно задать диапазон.

SELECT name FROM tbl\_info WHERE age BETWEEN 20 and 30;

Сравните с более длинной записью

SELECT name FROM tbl\_info WHERE age >= 20 and age <=30;

С помощью условия IN за которыми в скобках идут нужные значения, для которых выполняется выборка.

SELECT name FROM tbl\_info WHERE age IN (20, 30);

Это короче, чем

SELECT name FROM tbl\_info WHERE age = 20 OR age = 30;

Ключевое слово NOT IN выполняет обратную задачу и позволяет получить записи, которые не входят в данное условие.

SELECT name FROM tbl\_info WHERE age NOT IN (20);

Ключевое слово NOT можно использовать не только с IN, но и с BETWEEN, LIKE.

Узнать число записей можно через функцию COUNT. Если запись содержит NULL, то она не учитывается.

SELECT COUNT(name) FROM table\_info;

Для показа минимального или максимального значения используются функции MIN() или MAX(:

SELECT name, MAX(age) FROM table\_info;

Среднее значение и сумма по числовым колонкам вычисляются с помощью функции AVG() и SUM().

Если нам нужно вывести только определённое количество записей, то используйте ключевое слово LIMIT с указанием значения.

SELECT \* FROM table\_info LIMIT 3;

Если мы хотим придать результирующему набору записей определенный порядок, то должны непосредственно сказать об этом серверу, прописав секцию ORDER BY в запросе.

SELECT name, age FROM table\_info ORDER BY name;

SELECT name, age FROM table\_info ORDER BY name ASC;

SELECT name, age FROM table\_info ORDER BY age DESC;

Вместо имени колонки можно указать ее порядковый номер (начиная с 1).

ALTER

Добавить новый столбец в таблицу можно с помощью необязательного ключевого слова ALTER, за которым идёт название столбца в таблице.

ALTER TABLE tbl\_info ADD COLUMN weight INTEGER

RENAME TO **-** переименовать саму таблицу.

ALTER TABLE tbl\_info RENAME TO table\_info;

UPPER

Преобразовать текст из указанного столбца в верхний регистр.

UPDATE table\_info SET name = UPPER(name);

**Внешние ключи**

Внешние ключи в SQL используются для принудительного поддержания связей между таблицами в актуальном состоянии. Рассмотрим, например, схему базы данных, созданную следующими командами SQL:

CREATE TABLE artist(

artistid INTEGER PRIMARY KEY NOT NULL,

artistname TEXT

);

CREATE TABLE track(

trackid INTEGER NOT NULL,

trackname TEXT,

trackartist INTEGER NOT NULL **--Должен ссылаться на artist.artistid!**

);

В приложениях, использующих эту базу данных, приходится следить за тем, чтобы для каждой строки таблицы «track» существовала соответствующая строка в таблице «artist». Именно об этом сообщает комментарий в декларации таблицы.

К сожалению, если пользователь редактирует базу данных с использованием внешних инструментов, или если в приложении имеется баг, появляется возможность вставлять в таблицу «track» строки, которые не имеют соответствующих строк в таблице «artist». Или можно удалить строки из таблицы «artist» тем самым создавая «висячие» строки в таблице «track», для которых не остается соответствующих строк в таблице «artist».

Такая ситуация может приводить к логическим ошибкам в приложениях, программным сбоям или сделает разработку приложений более сложной.

Одним из решений является добавление к схеме базы данных внешнего ключа SQL, который бы принудительно обеспечивал сохранение связей между таблицами «artist» и «track». Чтобы это сделать, нужно модифицировать декларацию таблицы «track» добавив к нему определение внешнего ключа следующим образом:

--включение поддержки ограничений на внешние ключи (для текущей сессии)

PRAGMA FOREIGN\_KEYS=ON

CREATE TABLE track(

trackid INTEGER,

trackname TEXT,

trackartist INTEGER,

**FOREIGN KEY(trackartist) REFERENCES artist(artistid)**

);

Таким образом устанавливается ограничение для SQLite. Теперь попытки вставить строку в таблицу «track», для которой нет соответствующей строки в таблице «artist», будут неуспешны, равно как и удаление строки из таблицы «artist», имеющей зависимые строки в таблице «track».

Следующий вывод иллюстрирует результат действия внешнего ключа, добавленного к таблице «track»:

SELECT \* FROM artist;

artistid artistname

-------- -----------------

1 Dean Martin

2 Frank Sinatra

SELECT \* FROM track;

trackid trackname trackartist

------- ----------------- -----------

11 That's Amore 1

12 Christmas Blues 1

13 My Way 2

*-- Следующий запрос приведет к ошибке, поскольку значению,*

*-- вставляемому в столбец trackartist (3), не соответствует*

*-- записи в таблице artist.*

INSERT INTO track VALUES(14, 'Mr. Bojangles', 3);

**IntegrityError: foreign key constraint failed**

*-- Необходимо вначале добавить исполнителя в таблицу artist.*

INSERT INTO artist VALUES(3, 'Sammy Davis Jr.');

INSERT INTO track VALUES(14, 'Mr. Bojangles', 3);

sqlite> INSERT INTO track VALUES(15, 'Boogie Woogie', 3);

Также можно убедиться, что невозможно удалять и изменять строки таблицы «artist», так чтобы нарушалось ограничение внешнего ключа:

*-- Попытка удалить данные об артисте "Frank Sinatra"*

*-- приведет к ошибке, поскольку таблица track содержит*

*-- запись, ссылающуюся на артиста.*

DELETE FROM artist WHERE artistname = 'Frank Sinatra';

**IntegrityError: foreign key constraint failed**

*-- Удаляем все записи из таблицы track, которые ссылаются на*

*-- артиста "Frank Sinatra". Только после этого становится*

*-- возможным удалить этого артиста.*

DELETE FROM track WHERE trackname = 'My Way';

DELETE FROM artist WHERE artistname = 'Frank Sinatra';

*-- Попытка изменить artistid строки из таблицы artist*

*-- заканчивается неудачей, если в таблице track имеются*

*-- записи, ссылающиеся на данную строку из artist.*

UPDATE artist SET artistid=4 WHERE artistname = 'Dean Martin';

**IntegrityError: foreign key constraint failed**

Извлечь данные сразу из обеих таблиц, например, список всех треков и соответствующих им артистов, можно с помощью следующего запроса:

SELECT track.trackname, artist.artistname FROM artist, track WHERE track.trackartist = artist.artistid

В SQLite используется следующая терминология:

* **Родительская таблица** — это таблица, на которую ссылается внешний ключ. В рассмотренном примере родительской таблицей является таблица «artist».
* **Дочерняя таблица** — это таблица, к которой применяется ограничение внешнего ключа. Это та таблица, объявление которой содержит выражение REFERENCES. В данном примере используется дочерняя таблица «track».
* **Родительский ключ** — это один или несколько столбцов родительской таблицы, на которые ссылается внешний ключ. Как правило, но не обязательно, это есть первичный ключ родительской таблицы.
* **Дочерний ключ** — это один или несколько столбцов дочерней таблицы, значения которых ограничиваются внешним ключом и которые перечисляются в выражении REFERENCES.

## Практическая часть

В Python базы данных SQL основываются на нескольких понятиях:

Объекты соединений. Представляют соединение с базой данных, служат интерфейсом для операций отмены и подтверждения, предоставляют доступ к реализации пакета и создают объекты курсоров.

Объекты курсоров.Представляют одну команду SQL, посылаемую в виде строки, и могут использоваться для доступа к результатам, возвращаемым командой SQL.

Результаты запроса команды SQL **select**. Возвращаются в сценарии в виде последовательностей (например, списков кортежей), представляющих таблицы записей баз данных. Внутри этих последовательностей записей значения полей являются обычными объектами Python, такими как строки, целые и вещественные числа (например, [('bob',48), ('emily',47)]). Значения полей могут также быть объектами специального типа, хранящими такие значения, как дата и время, а значения NULL в базе данных возвращаются в виде объекта None.

**Задание 1**. **Соединение с базой данных**.

Первое, что необходимо сделать, – это открыть соединение с базой данных и создать таблицу для хранения записей:

import sqlite3

conn = sqlite3.connect('dbase1') #можно указать полный путь к файлу

При необходимости этот вызов создаст новый файл или откроет существующий – интерфейсу SQLite также можно передать специальную строку «:memory:», чтобы создать временную базу данных в памяти.

Если сценарий придерживается использования стандартного программного кода SQL, аргументы метода connect – это единственное, что будет отличаться при использовании разных систем баз данных. Например, в интерфейсе MySQL этот метод принимает доменное имя сетевого хоста, имя пользователя и пароль, которые передаются в виде именованных аргументов.

**Задание 2. Создание таблиц.**

Далее, создадим курсор для отправки инструкций SQL серверу баз данных и создадим с его помощью первую таблицу:

cur = conn.cursor()

sql = '''CREATE TABLE people (name TEXT,

job TEXT,

pay INTEGER)'''

cur.execute(sql)

В этом примере создается в базе данных таблица с именем «people» – идентификаторы name, job и pay определяют столбцы в этой таблице и их типы – две строки и целое число. В интерфейсе SQLite роль базы данных играет файл, поэтому здесь отсутствует понятие создания и выбора базы данных внутри него, как в некоторых других системах.

**Задание 3. Удаление таблицы**

Повторный запуск рассмотренного скрипта приведет к ошибке, поскольку такая таблица уже будет существовать. Поэтому таблицу предварительно можно удалять при запуске (естественно, что и все данные будут также утеряны):

cur.execute('DROP TABLE IF EXISTS people')

sql = '''CREATE TABLE people (name TEXT,

job TEXT,

pay INTEGER)'''

cur.execute(sql)

**Задание 4. Добавление записей.**

Существует три основных способа: вставлять записи по одной, вставить сразу несколько записей в одной инструкции или задействовать цикл Python.

**А)** Команда insert в языке SQL добавляет в таблицу единственную запись. После вызова метода execute в атрибуте rowcount курсора возвращается количество записей, созданных или затронутых последней выполненной инструкцией:

cur.execute('INSERT INTO people VALUES (?, ?, ?)', ('Bob','Dev', 5000))

print(cur.rowcount)

print(sqlite3.paramstyle)

1

qmark

Значения параметров для подстановки в инструкцию SQL обычно передаются в виде последовательности (например, в виде списка или кортежа). Обратите внимание на вызов метода paramstyle, который сообщает стиль обозначения параметров в строке с инструкцией. В данном случае qmark означает, что параметры внутри инструкций обозначаются знаками вопроса ?.

Другие модули баз данных могут использовать такие стили, как format (параметры обозначаются, как %s), числовые индексы или именованные параметры, – за дополнительной информацией обращайтесь к описанию API базы данных.

**Б)** Для добавления в одной инструкции сразу несколько записей используется метод executemany и последовательность записей (например, список списков). По своему действию вызов этого метода напоминает вызов метода execute для каждой записи в аргументе:

cur.executemany('INSERT INTO people VALUES (?, ?, ?)',

[ ('Dan', 'mgr', '70000'),

('Ann', 'mgr', '60000')])

print(cur.rowcount)

2

**С)** Добиться того же результата можно вставляя по одной записи в цикле:

rows = [ ['Tom', 'adm', 50000],

['Kim', 'adm', 40000],

['Pat', 'dev', 60000]]

for row in rows:

cur.execute('INSERT INTO people VALUES (? , ?, ?)', row)

conn.commit()

conn.close()

Обратите внимание на последнюю команду – чтобы сохранить изменения в базе данных, необходимо всегда вызывать метод commit объекта соединения. В противном случае, когда соединение будет закрыто, изменения будут потеряны. В действительности, пока не будет вызван метод commit, ни одна из добавленных записей не будет видна из других соединений с базой данных.

Технически спецификация API требует, чтобы при закрытии (вручную, вызовом метода close или автоматически в момент утилизации объекта сборщиком мусора) объект соединения автоматически вызывал метод rollback с целью отменить неподтвержденные изменения. Для систем баз данных, не поддерживающих операции подтверждения и отмены транзакций, эти методы могут не выполнять никаких действий. SQLite реализует оба метода, commit и rollback; последний из них откатывает любые изменения до момента последнего вызова метода commit.

**Задание 4. Выполнение запросов**

К настоящему моменту таблица базы данных заполнена. Выполним запрос SQL, чтобы посмотреть, что получилось:

cur.execute('SELECT \* FROM people')

print(cur.fetchall())

[('Bob', 'Dev', 5000), ('Dan', 'mgr', 70000), ('Ann', 'mgr', 60000), ('Tom', 'adm', 50000), ('Kim', 'adm', 40000), ('Pat', 'dev', 60000)]

Здесь с помощью объекта курсора выполняется инструкция SQL select, которая отбирает все записи, и вызывается метод fetchall курсора, чтобы извлечь их. Записи возвращаются сценарию в виде последовательности последовательностей. В данном модуле это список кортежей – внешний список представляет таблицу результатов, вложенные кортежи представляют записи (строки), а содержимое вложенных кортежей – столбцы данных.

Поскольку все эти данные являются обычными данными Python, после получения результатов запроса их можно обрабатывать с помощью обычного программного кода Python. Например, чтобы сделать вывод более удобочитаемым, выполним цикл по результатам:

cur.execute('SELECT \* FROM people')

for row in cur.fetchall():

print(row)

('Bob', 'Dev', 5000)

('Dan', 'mgr', 70000)

('Ann', 'mgr', 60000)

('Tom', 'adm', 50000)

('Kim', 'adm', 40000)

('Pat', 'dev', 60000)

Поскольку результатом запроса является последовательность, для ее обработки можно использовать мощные операции над последовательностями и инструменты итераций, имеющиеся в Python. Например, чтобы отобрать значения только из столбца name, можно выполнить более специализированный запрос SQL и получить список кортежей:

cur.execute('SELECT name FROM people')

print(cur.fetchall())

[('Bob',), ('Dan',), ('Ann',), ('Tom',), ('Kim',), ('Pat',)]

А можно использовать для выборки желаемых полей программный код Python, например, генератор списков:

cur.execute('SELECT \* FROM people')

names = [rec[0] for rec in cur.fetchall()]

print(names)

['Bob', 'Dan', 'Ann', 'Tom', 'Kim', 'Pat']

Использовавшийся до сих пор метод fetchall извлекает сразу все результаты запроса в виде единой последовательности (в случае отсутствия результатов возвращается пустая последовательность).

Это удобно, но такой способ может оказаться достаточно медленным, чтобы временно заблокировать вызывающую программу при большом объеме результатов или необходимости передавать значительные объемы данных по сети, когда взаимодействие выполняется с удаленным сервером.

Чтобы избежать этого, можно извлекать данные по одной записи или пакетами записей с помощью методов fetchone и fetchmany. Метод fetchone возвращает следующую запись из результатов или None по достижении конца таблицы:

cur.execute('SELECT \* FROM people')

while True:

row = cur.fetchone()

if not row: break

print(row)

('Bob', 'Dev', 5000)

('Dan', 'mgr', 70000)

('Ann', 'mgr', 60000)

('Tom', 'adm', 50000)

('Kim', 'adm', 40000)

('Pat', 'dev', 60000)

Метод fetchmany возвращает последовательность записей из результатов, но не всю таблицу – можно явно указать количество записей, извлекаемых при каждом обращении, или положиться на значение по умолчанию, которое определяется атрибутом arraysize курсора. Каждый вызов возвращает не более указанного числа записей из результатов, или пустую последовательность по достижении конца таблицы:

cur.execute('SELECT \* FROM people')

while True:

rows = cur.fetchmany() #size=N необязательный аргумент

if not rows: break

for row in rows:

print(row)

('Bob', 'Dev', 5000)

('Dan', 'mgr', 70000)

('Ann', 'mgr', 60000)

('Tom', 'adm', 50000)

('Kim', 'adm', 40000)

('Pat', 'dev', 60000)

Как только таблица результатов будет исчерпана, метод fetchone или fetchmany вернет значение False.

Спецификация API баз данных также требует, чтобы метод fetchall возвращал «все оставшиеся записи», поэтому перед извлечением новых данных, как правило, необходимо снова вызвать метод execute, чтобы получить результаты. Иначе получается следующее:

print(cur.fetchall())

[]

**Задание 5. Внесение изменений**

Инструкция SQL update изменяет записи – следующий пример запишет новое значение 55 000 столбца pay в трех записях (Bob, Tom и Kim), потому что зарплата этих сотрудников не превышает $60 000. Как обычно, атрибут rowcount содержит количество изменившихся записей:

cur.execute('UPDATE people SET pay=? where pay <= ?',

[55000, 50000])

print(cur.rowcount)

3

cur.execute('SELECT name, pay FROM people WHERE pay == 55000')

print(cur.fetchall())

[('Bob', 55000), ('Tom', 55000), ('Kim', 55000)]

Инструкция SQL delete удаляет записи в соответствии с необязательным условием (чтобы удалить все записи, достаточно опустить условие). Следующий пример удалит запись о сотруднике с именем Bob:

cur.execute('DELETE FROM people WHERE name = ?', ['Bob'])

cur.execute('SELECT \* FROM people')

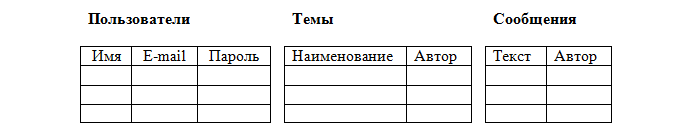
print(cur.fetchall())

**Задание 5.**

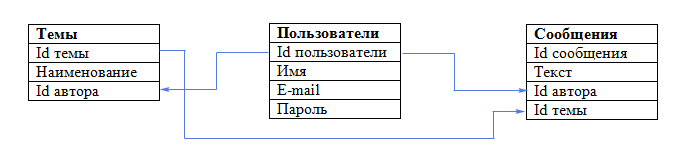
Рассмотреть работу остальных конструкций языка SQL из теоретической части. В том числе и пример с созданием внешних ключей.

**Задание 6.**

Создать базу данных интернет форума. В базе должна храниться информация о пользователях (имя, e-mail и пароль), созданных ими темах и сообщениях. Базу данных удобно представить в виде 3-х таблиц:



Связь между таблицами организовать с использованием внешних ключей:



Продемонстрировать работу с базой данных с помощью различных запросов SELECT/INSERT/UPDATE/DELETE. Продемонстрировать ограничения, накладываемые внешними ключами.