Python. Лекция 10. Работа с базой данных.

Основные понятия реляционной СУБД

Реляционная база данных - это набор таблиц с данными.

Таблица - это прямоугольная матрица, состоящая из строк и столбцов. Таблица задает отношение (relation).

Строка - запись, состоящая из полей - столбцов. В каждом поле может содержаться некоторое значение, либо специальное значение NULL (пусто). В таблице может быть произвольное количество строк. Для реляционной модели порядок расположения строк не определен и не важен.

Каждый **столбец** в таблице имеет собственное имя и тип.

Что такое DB-API 2

Вынесенная в заголовок аббревиатура объединяет два понятия: DB (Database, база данных) и API (Application Program Interface, интерфейс прикладной программы).

Таким образом, DB-API определяет интерфейс прикладной программы с базой данных. Этот интерфейс, описываемый ниже, должен реализовывать все модули расширения, которые служат для связи Python-программ с базами данных. Единый API (в настоящий момент его вторая версия) позволяет абстрагироваться от марки используемой базы данных, при необходимости довольно легко менять одну СУБД на другую, изучив всего один набор функций и методов.

DB-API 2.0 описан в PEP 249, и данное ниже описание основано именно на нем.

Описание DB API 2.0

DB API 2.0 регламентирует интерфейсы модуля расширения для работы с базой данных, методы объекта-соединения с базой, объекта-курсора текущей обрабатываемой записи, объектов различных для типов данных и их конструкторов, а также содержит рекомендации для разработчиков по реализации модулей. На сегодня Python поддерживает через модули расширения многие известные базы данных (уточнить можно на web-странице по этому адресу). Ниже рассматриваются почти все положения DB-API за исключением рекомендаций для разработчиков новых модулей.

Интерфейс модуля

Здесь необходимо сказать о том, что должен предоставлять модуль для удовлетворения требований DB-API 2.0.

Доступ к базе данных осуществляется с помощью **объекта-соединения** (connection object). DB-API-совместимый модуль должен предоставлять функцию-конструктор `connect()` для класса объектов-соединений. Конструктор должен иметь следующие именованные параметры:

- `dsn` Название источника данных в виде строки

- `user` Имя пользователя
- `password` Пароль
- `host` Адрес хоста, на котором работает СУБД
- `database` Имя базы данных.

Методы объекта-соединения будут рассмотрены чуть позже.

Модуль определяет константы, содержащие его основные характеристики:

- `apilevel` Версия DB-API ("1.0" или "2.0").
- `threadsafety` Целочисленная константа, описывающая возможности модуля при использовании потоков управления:
  - `0` Модуль не поддерживает потоки.
  - `1` Потоки могут совместно использовать модуль, но не соединения.
  - `2` Потоки могут совместно использовать модуль и соединения.
  - `3` Потоки могут совместно использовать модуль, соединения и курсоры. (Под совместным использованием здесь понимается возможность использования упомянутых ресурсов без применения семафоров).
- `paramstyle` Тип используемых пометок при подстановке параметров. Возможны следующие значения этой константы:
  - `"format"` Форматирование в стиле языка ANSI C (например, `"%s"`, `"%i"`).
  - `"pyformat"` Использование именованных спецификаторов формата в стиле Python (`"%(item)s"`)
  - `"qmark"` Использование знаков `"?"` для пометки мест подстановки параметров.
  - `"numeric"` Использование номеров позиций (`":1"`).
  - `"named"` Использование имен подставляемых параметров (`":name"`).

Модуль должен определять ряд исключений для обозначения типичных исключительных ситуаций: `Warning` (предупреждение), `Error` (ошибка), `InterfaceError` (ошибка интерфейса), `DatabaseError` (ошибка, относящаяся к базе данных). А также подклассы этого последнего исключения: `DataError` (ошибка обработки данных), `OperationalError` (ошибка в работе или сбой соединения с базой данных), `IntegrityError` (ошибка целостности базы данных), `InternalError` (внутренняя ошибка базы данных), `ProgrammingError` (программная ошибка, например, ошибка в синтаксисе SQL-запроса), `NotSupportedError` (при отсутствии поддержки запрошенного свойства).

Объект-соединение

Объект-соединение, получаемый в результате успешного вызова функции `connect()`, должен иметь следующие методы:

- `close()` Закрывает соединение с базой данных.
- `commit()` Завершает транзакцию.
- `rollback()` Откатывает начатую транзакцию (восстанавливает исходное состояние). Закрытие соединения при незавершенной транзакции автоматически производит откат транзакции.
- `cursor()` Возвращает объект-курсор, использующий данное соединение. Если база данных не поддерживает курсоры, модуль сопряжения должен их имитировать.

Под **транзакцией** понимается группа из одной или нескольких операций, которые изменяют базу данных. Транзакция соответствует логически неделимой операции над базой данных, а частичное выполнение транзакции приводит к нарушению целостности

БД. Например, при переводе денег с одного счета на другой операции по уменьшению первого счета и увеличению второго являются транзакцией. Методы `commit()` и `rollback()` обозначают начало и конец транзакции в явном виде. Кстати, не все базы данных поддерживают механизм транзакций.

Следует отметить, что в зависимости от реализации DB-API 2.0 модуля, необходимо сохранять ссылку на объект-соединение в продолжение работы курсоров этого соединения. В частности, это означает, что нельзя сразу же получать объект-курсор, не привязывая объект-соединение к некоторому имени. Также нельзя оставлять объект-соединение в локальной переменной, возвращая из функции или метода объект-курсор.

Объект-курсор

Курсор (от англ. cursor - CURrrent Set Of Records, текущий набор записей) служит для работы с результатом запроса. Результатом запроса обычно является одна или несколько прямоугольных таблиц со столбцами-полями и строками-записями. Приложение может читать и обрабатывать полученные таблицы и записи в таблице по одной, поэтому в курсоре хранится информация о текущей таблице и записи. Конкретный курсор в любой момент времени связан с выполнением одной SQL-инструкции.

Атрибуты объекта-курсора тоже определены DB-API:

- `arraysize` Атрибут, равный количеству записей, возвращаемых методом `fetchmany()`. По умолчанию равен 1.
- `callproc(procname[, params])` Вызывает хранимую процедуру `procname` с параметрами из изменчивой последовательности `params`. Хранимая процедура может изменить значения некоторых параметров последовательности. Метод может возвратить результат, доступ к которому осуществляется через `fetch` - методы.
- `close()` Закрывает объект-курсор.
- `description` Этот доступный только для чтения атрибут является последовательностью из семиэлементных последовательностей. Каждая из этих последовательностей содержит информацию, описывающую один столбец результата:

```
(name, type_code, display_size, internal_size, precision, scale, null_ok)
```

Первые два элемента (имя и тип) обязательны, а вместо остальных (размер для вывода, внутренний размер, точность, масштаб, возможность задания пустого значения) может быть значение `None`. Этот атрибут может быть равным `None` для операций, не возвращающих значения.

- `execute(operation[, parameters])` Исполняет запрос к базе данных или команду СУБД. Параметры (`parameters`) могут быть представлены в принятой в базе данных нотации в соответствии с атрибутом `paramstyle`, описанным выше.
- `executemany(operation, seq_of_parameters)` Выполняет серию запросов или команд, подставляя параметры в заданный шаблон. Параметр `seq_of_parameters` задает последовательность наборов параметров.
- `fetchall()` Возвращает все (или все оставшиеся) записи результата запроса.
- `fetchmany([size])` Возвращает следующие несколько записей из результатов запроса в виде последовательности последовательностей. Пустая последовательность означает отсутствие данных. Необязательный параметр `size` указывает количество возвращаемых записей (реально возвращаемых записей

может быть меньше). По умолчанию `size` равен атрибуту `arraysize` объекта-курсора.

- `fetchone()` Возвращает следующую запись (в виде последовательности) из результата запроса или `None` при отсутствии данных.
- `nextset()` Переводит курсор к началу следующего набора данных, полученного в результате запроса (при этом часть записей в предыдущем наборе может остаться непрочитанной). Если наборов больше нет, возвращает `None`. Не все базы данных поддерживают возврат нескольких наборов результатов за одну операцию.
- `rowcount` Количество записей, полученных или затронутых в результате выполнения последнего запроса. В случае отсутствия execute-запросов или невозможности указать количество записей равен -1.
- `setinputsizes(sizes)` Предопределяет области памяти для параметров, используемых в операциях. Аргумент `sizes` задает последовательность, где каждый элемент соответствует одному входному параметру. Элемент может быть объектом-типом соответствующего параметра или целым числом, задающим длину строки. Он также может иметь значение `None`, если о размере входного параметра ничего нельзя сказать заранее или он предполагается очень большим. Метод должен быть вызван до execute-методов.
- `setoutputsize(size[, column])` Устанавливает размер буфера для выходного параметра из столбца с номером `column`. Если `column` не задан, метод устанавливает размер для всех больших выходных параметров. Может использоваться, например, для получения **больших бинарных объектов** (**B** inary **L** arge **O** bject, **BLOB**).

Объекты-типы

DB-API 2.0 предусматривает названия для объектов-типов, используемых для описания полей базы данных:

Объект	Тип
`STRING`	Строка и символ
`BINARY`	Бинарный объект
`NUMBER`	Число
`DATETIME`	Дата и время
`ROWID`	Идентификатор записи
`None`	NULL-значение (отсутствующее значение)

С каждым типом данных (в реальности это - классы) связан конструктор. Совместимый с DB-API модуль должен определять следующие конструкторы:

- `Date(год, месяц, день)` Дата.
- `Time(час, минута, секунда)` Время.
- `Timestamp(год, месяц, день, час, минута, секунда)` Дата-время.
- `DateFromTicks(secs)` Дата в виде числа секунд `secs` от начала эпохи (1 января 1970 года).
- `TimeFromTicks(secs)` Время, то же.
- `TimestampFromTicks(secs)` Дата-время, то же.
- `Binary(string)` Большой бинарный объект на основании строки `string`.

Работа с базой данных из Python-приложения

Далее в лекции на конкретных примерах будет показано, как работать с базой данных из программы на языке Python. Нужно отметить, что здесь не ставится цели постичь премудрости языка запросов (это тема отдельного курса). Простые примеры

позволят понять, что при программировании на Python доступ к базе данных не сложнее доступа к другим источникам данных (файлам, сетевым объектам).

Именно поэтому для демонстрации выбрана СУБД SQLite, работающая как под Unix, так и под Windows. Кроме установки собственно SQLite и модуля сопряжения с Python, каких-либо дополнительных настроек проводить не требуется, так как SQLite хранит данные базы в отдельном файле: сразу приступать к созданию таблиц, занесению в них данных и произведению запросов нельзя. Выбранная СУБД (в силу своей "легкости") имеет одну существенную особенность: за одним небольшим исключением, СУБД SQLite не обращает внимания на типы данных (она хранит все данные в виде строк), поэтому модуль расширения `sqlite` для Python проделывает дополнительную работу по преобразованию типов. Кроме того, СУБД SQLite поддерживает достаточно большое подмножество свойств стандарта SQL92, оставаясь при этом небольшой и быстрой, что немаловажно, например, для web-приложений. Достаточно сказать, что SQLite поддерживает даже транзакции.

Еще раз стоит повторить, что выбор учебной базы данных не влияет на синтаксис использованных средств, так как модуль `sqlite`, который будет использоваться, поддерживает DB-API 2.0, а значит, переход на любую другую СУБД потребует минимальных изменений в вызове функции `connect()` и, возможно, использования более удачных типов данных, свойственных целевой СУБД.

Схематично работа с базой данных может выглядеть примерно так:

- Подключение к базе данных (вызов `connect()` с получением объекта-соединения).
- Создание одного или нескольких курсоров (вызов метода объекта-соединения `cursor()` с получением объекта-курсора).
- Исполнение команды или запроса (вызов метода `execute()` или его вариантов).
- Получение результатов запроса (вызов метода `fetchone()` или его вариантов).
- Завершение транзакции или ее откат (вызов метода объекта-соединения `commit()` или `rollback()`).
- Когда все необходимые транзакции произведены, подключение закрывается вызовом метода `close()` объекта-соединения.

Знакомство с СУБД

Допустим, программное обеспечение установлено правильно, и можно работать с модулем `sqlite`. Стоит посмотреть, чему будут равны константы:

```
>>> import sqlite
                >>> sqlite.apilevel
                '2.0'
                >>> sqlite.paramstyle
                'pyformat'
                >>> sqlite.threadsafety
                1
```

Отсюда следует, что `sqlite` поддерживает DB-API 2.0, подстановка параметров выполняется в стиле строки форматирования языка Python, а соединения нельзя совместно использовать из различных потоков управления (без блокировок).

Создание базы данных

Для создания базы данных нужно установить, какие таблицы (и другие объекты, например индексы) в ней будут храниться, а также определить структуры таблиц (имена и типы полей).

Задача - создание базы данных, в которой будет храниться телепрограмма. В этой базе будет таблица со следующими полями:

- `tvdate`,
- `tvweekday`,
- `tvchannel`,
- `tvtime1`,
- `tvtime2`,
- `prname`,
- `prgenre`.

Здесь `tvdate` - дата, `tvchannel` - канал, `tvtime1` и `tvtime2` - время начала и конца передачи, `prname` - название, `prgenre` - жанр. Конечно, в этой таблице есть функциональная зависимость (`tvweekday` вычисляется на основе `tvdate` и `tvtime1`), но из практических соображений БД к нормальным формам приводиться не будет. Кроме того, будет создана таблица с названиями дней недели (устанавливает соответствие между номером дня и днем недели):

- `weekday`,
- `wdname`.

Следующий сценарий создаст таблицу в базе данных (в случае с SQLite заботиться о создании базы данных не нужно: файл создастся автоматически. Для других баз данных необходимо перед этим создать базу данных, например, SQL-инструкцией `CREATE DATABASE`):

```
import sqlite as db

c = db.connect(database="tvprogram")
cu = c.cursor()

try:
  cu.execute("""
    CREATE TABLE tv (
      tvdate DATE,
      tvweekday INTEGER,
      tvchannel VARCHAR(30),
      tvtime1 TIME,
      tvtime2 TIME,
      prname VARCHAR(150),
      prgenre VARCHAR(40)
    );
""")
except db.DatabaseError, x:
  print "Ошибка: ", x
c.commit()

try:
  cu.execute("""
    CREATE TABLE wd (
      weekday INTEGER,
      wdname VARCHAR(11)
    );
""")
except db.DatabaseError, x:
  print "Ошибка: ", x
```

```
c.commit()
c.close()
```

Здесь просто исполняются SQL-инструкции, и обрабатывается ошибка базы данных, если таковая случится (например, при попытке создать таблицу с уже существующим именем). Для того чтобы таблицы создавались независимо, используется `commit()`.

Кстати, удалить таблицы из базы данных можно следующим образом:

```
import sqlite as db

c = db.connect(database="tvprogram")
cu = c.cursor()

try:
  cu.execute("""DROP TABLE tv;""")
except db.DatabaseError, x:
  print "Ошибка: ", x
c.commit()

try:
  cu.execute("""DROP TABLE wd;""")
except db.DatabaseError, x:
  print "Ошибка: ", x
c.commit()
c.close()
```

Наполнение базы данных

Теперь можно наполнить таблицы значениями. Следует начать с расшифровки числовых значений для дней недели:

```
weekdays = ["Воскресенье", "Понедельник", "Вторник", "Среда",
                          "Четверг",        "Пятница",        "Суббота",
"Воскресенье"]

import sqlite as db

c = db.connect(database="tvprogram")
cu = c.cursor()
cu.execute("""DELETE FROM wd;""")
cu.executemany("""INSERT INTO wd VALUES (%s, %s);""",
           enumerate(weekdays))
c.commit()
c.close()
```

Стоит напомнить, что встроенная функция `enumerate()` создает список пар номер-значение, например:

```
>>> print [i for i in enumerate(['a', 'b', 'c'])]
[(0, 'a'), (1, 'b'), (2, 'c')]
```

Из приведенного примера ясно, что метод `executemany()` объекта-курсора использует второй параметр - последовательность - для массового ввода данных с помощью SQL-инструкции `INSERT`.

Предположим, что телепрограмма задана в файле tv.csv в формате CSV (он уже обсуждался):

```
10.02.2003 9.00|ОРТ|Новости|Новости|9.15
10.02.2003 9.15|ОРТ|"НЕЖНЫЙ ЯД"|Сериал|10.15
10.02.2003           10.15|ОРТ|"Маски-шоу"|Юмористическая
программа|10.45
```

```
                    10.02.2003 10.45|ОРТ|"Человек и закон"||11.30
                    10.02.2003          11.30|ОРТ|"НОВЫЕ          ПРИКЛЮЧЕНИЯ
СИНДБАДА"|Сериал|12.00
```

Следующая программа разбирает CSV-файл и записывает данные в таблицу `tv`:

```
    import calendar, csv
                    import sqlite as db
                    from sqlite.main import Time, Date  ## Только для
                    db.Date, db.Time = Date, Time       ## sqlite

                    c = db.connect(database="tvprogram")
                    cu = c.cursor()

                    input_file = open("tv.csv", "rb")
                    rdr = csv.DictReader(input_file,
                              fieldnames=['begt',     'channel',     'prname',
'prgenre', 'endt'])
                    for rec in rdr:
                        bd, bt = rec['begt'].split()
                        bdd, bdm, bdy = map(int, bd.split('.'))
                        bth, btm = map(int, bt.split('.'))
                        eth, etm = map(int, rec['endt'].split('.'))
                        rec['wd'] = calendar.weekday(bdy, bdm, bdd)
                        rec['begd'] = db.Date(bdy, bdm, bdd)
                        rec['begt'] = db.Time(bth, btm, 0)
                        rec['endt'] = db.Time(eth, etm, 0)

                        cu.execute("""INSERT INTO tv
                        (tvdate,   tvweekday,   tvchannel,   tvtime1,   tvtime2,
prname, prgenre)
                        VALUES (
                        %(begd)s, %(wd)s, %(channel)s, %(begt)s, %(endt)s,
                        %(prname)s, %(prgenre)s);""", rec)
                    input_file.close()
                    c.commit()
```

Большая часть преобразований связана с получением дат и времен (приходится разбивать строки на части в соответствии с форматом даты и времени). День недели получен с помощью функции из модуля `calendar`.

Примечание:

Из-за небольшой ошибки в пакете `sqlite` конструкторы `Date`, `Time` и т.д. не попадают из модуля `sqlite.main` при импорте из `sqlite`, поэтому пришлось добавить две строки, специфичные для `sqlite`, в универсальный "модуль" с именем `db`.

В этом же примере было продемонстрировано использование словаря для вставки значений в таблицу базы данных. Следует заметить, что подстановка выполняется внутри вызова `execute()` в соответствии с типами переданных значений. SQL-инструкция INSERT была бы некорректной при попытке выполнить подстановку самостоятельно, например, операцией форматирования `%`.

Выборки из базы данных

Базы данных создаются для удобства хранения и извлечения больших объемов. Следующий нехитрый пример позволяет проверить, правильно ли были введены в таблицу дни недели:

```
import sqlite as db

                c = db.connect(database="tvprogram")
```

```
cu = c.cursor()
cu.execute("SELECT weekday, wdname FROM wd ORDER BY
weekday;")
for i, n in cu.fetchall():
  print i, n
```

Если все было сделано правильно, получится:

```
0 Воскресенье
1 Понедельник
2 Вторник
3 Среда
4 Четверг
5 Пятница
6 Суббота
7 Воскресенье
```

Несложно догадаться, как сделать выборку телепрограммы:

```
import sqlite as db

c = db.connect(database="tvprogram")
cu = c.cursor()
cu.execute("""
SELECT tvdate, tvtime1, wd.wdname, tvchannel, prname, prgenre
FROM tv, wd
WHERE wd.weekday = tvweekday
ORDER BY tvdate, tvtime1;""")
for rec in cu.fetchall():
  dt = rec[0] + rec[1]
  weekday = rec[2]
  channel = rec[3]
  name = rec[4]
  genre = rec[5]
  print "%s, %02i.%02i.%04i %s %02i:%02i %s (%s)" % (
         weekday, dt.day, dt.month, dt.year, channel,
         dt.hour, dt.minute, name, genre)
```

В этом примере в качестве типа для даты и времени используется тип из `mx.DateTime`. Именно поэтому стало возможным получить год, месяц, день, час и минуту обращением к атрибуту. Кстати, `datetime` -объект стандартного модуля `datetime` имеет те же атрибуты. В общем случае для даты и времени может использоваться другой тип, поэтому если получаемые из базы даты будут проходить более глубокую обработку, их следует переводить во внутреннее представление сразу после получения по запросу. Тем самым тип даты из модуля DB-API не будет влиять на другие части программы.

Другие СУБД и Python

Модуль `sqlite` дает прекрасные возможности для построения небольших и быстрых баз данных, однако для полноты изложения предлагается обзор модулей расширения Python для других СУБД.

Выше везде импортировался модуль `sqlite`, с изменением его имени на `db`. Это было сделано не случайно. Дело в том, что подобные модули, поддерживающие DB-API 2.0, есть и для других СУБД, и даже не в единственном числе. Согласно информации на сайте www.python.org DB-API 2.0-совместимые модули для Python имеют следующие СУБД или протоколы доступа к БД:

- `zxJDBC` Доступ по JDBC.

- `MySQL` Для СУБД MySQL.
- `mxODBC` Доступ по ODBC, продается фирмой eGenix.
- `DCOracle2, cx_Oracle` Для СУБД Oracle.
- `PyGresQL, psycopg, pyPgSQL` Для СУБД PostgreSQL.
- `Sybase` Для Sybase.
- `sapdbapi` Для СУБД SAP.
- `KInterbasDB` Для СУБД Firebird (это потомок Interbase).
- `PyADO` Адаптер к Microsoft ActiveX Data Objects (только под Windows).

Примечание:

Для СУБД PostgreSQL нужно взять не PyGreSQL, а psycopg, так как в первом есть небольшие проблемы с типом для даты и времени при вставке параметров в методе `execute()`. Кроме того, `psycopg` оптимизирован для скорости и многопоточности (`psycopg.threadsafety=2`).

Таким образом, в примерах, используемых в этой лекции, вместо `sqlite` можно применять, например, `psycopg`: результат должен быть тем же, если, конечно, соответствующий модуль был установлен.

Однако в общем случае при переходе с одной СУБД на другую могут возникать нестыковки, даже, несмотря на поддержку одной версии DB-API. Например, у модулей могут различаться `paramstyle`. В этом случае придется немного переделать параметры к вызову `execute()`. Могут быть и другие причины, поэтому переход на другую СУБД следует тщательно тестировать.

Иметь интерфейс DB-API могут не только базы данных. Например, разработчики проекта `fssdb` стремятся построить DB-API 2.0 интерфейс к... файловой системе.

Несмотря на достаточно хорошие теоретические основы и стабильные реализации, реляционная модель - не единственная из успешно используемых сегодня. К примеру, уже рассматривался язык XML и интерфейсы для работы с ним в Python. Древовидная модель данных XML для многих задач является более естественной, и в настоящее время идут исследования, результаты которых позволят работать с XML так же легко и стабильно, как с реляционными СУБД. Язык программирования Python - один из полигонов этих исследований.

Решая конкретную задачу, разработчик программного обеспечения должен сделать выбор средств, наиболее подходящих для решения задачи. Очень многие подходят к этому выбору с предвзятостью, выбирая неоптимальную (для данной задачи или подзадачи) модель данных. В результате данные, которые по своей природе легче представить другой моделью, приходится хранить и обрабатывать в выбранной модели, зачастую невольно моделируя более естественные структуры доступа и хранения. Так, XML можно хранить в реляционной БД, а табличные данные - в XML, однако это неестественно. Из-за этого сложность и подверженность ошибкам программного продукта возрастают, даже если использованные инструменты высокого качества.

Заключение

В рамках данной лекции были рассмотрены возможности связи Python с системами управления реляционными базами данных. Для Python разработан стандарт, называемый DB-API (версия 2.0), которого должны придерживаться все разработчики модулей сопряжения с реляционными базами данных. Благодаря этому API код прикладной программы становится менее зависимым от марки используемой базы данных, его могут понять разработчики, использующие другие базы данных. Фактически DB-API 2.0 описывает имена функций и классов, которые должен содержать модуль сопряжения с базой данных, и их семантику. Модуль сопряжения

должен содержать класс объектов-соединений с базой данных и класс для курсоров - специальных объектов, через которые происходит коммуникация с СУБД на прикладном уровне.

Здесь была использована СУБД SQLite и соответствующий модуль расширения Python для сопряжения с этой СУБД - `sqlite`, так как он поддерживает DB-API 2.0 и достаточно прост в установке. С его помощью были продемонстрированы основные приемы работы с базой данных: создание и наполнение таблиц, выполнение выборок и анализ полученных данных.

В конце лекции дан список других пакетов и модулей, которые позволяют Python-программе работать со многими современными СУБД.