GeekBrains

Базы данных

Выборка и агрегация данных

SQLite version 3.31.1

© geekbrains.ru

На этом уроке

1. Узнаем, как строить запросы на получение информации.
2. Научимся использовать встроенные функции.
3. Попрактикуемся группировать данные и применять функции агрегирования.

Оглавление

Получение информации из базы данных

На предыдущем занятии мы рассмотрели все базовые операции по работе с данными — выборку, вставку, обновление и удаление. На практике чаще всего команды будут выполняться для получения того или иного среза информации, то есть наиболее частой задачей станет построение различных вариантов запросов SELECT.

Команда SELECT, базовый синтаксис

Рассмотрим задачу получения информации о пользователе с фамилией Иванов из нашей базы данных учеников students.db. На предыдущем занятии мы уже использовали команду SELECT, и знаем:

- в части FROM надо определить имя таблицы, к которой обращаемся;
- в части SELECT — перечислить имена столбцов, данные которых нас интересуют.

```
SELECT 'Столбец 1', 'Столбец 2', 'Столбец N' FROM 'Имя Таблицы';
```

Определение имени таблицы

Сначала определимся, в каких таблицах нам надо искать информацию о пользователе.

```
sqlite> .tables
courses   grades    streams   students
sqlite>
```

Информация, которая напрямую относится к студентам, находится в таблицах students и grades (ученики и оценки), а мы построим запросы к этим таблицам.

Список столбцов в запросе

Определим, какие именно данные у нас доступны для выборки из таблиц students и grades, посмотрим структуру таблиц командой .schema:.

```
sqlite> .schema students
CREATE TABLE students (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  surname TEXT NOT NULL,
  name TEXT NOT NULL
);
sqlite> .schema grades
CREATE TABLE grades (
  student_id INTEGER NOT NULL,
  stream_id INTEGER NOT NULL,
  grade REAL NOT NULL,
  PRIMARY KEY(student_id, stream_id),
  FOREIGN KEY (student_id) REFERENCES students(id),
  FOREIGN KEY (stream_id) REFERENCES streams(id)
);
sqlite>
```

Для таблицы students нам доступна выборка информации об идентификаторе пользователя, его имени и фамилии. В таблице оценок grades у нас есть доступ к идентификатору ученика и потока, а также к оценке ученика за курс.

Условие для получения среза данных

Найдём студента с фамилией Иванов. Для этого нам потребуется определить дополнительное условие в части WHERE:

```
sqlite> .header on
sqlite> .mode column
sqlite> SELECT * FROM students WHERE surname = 'Иванов';
id          surname     name
----------  ----------  ----------
1           Иванов      Игорь
sqlite>
```

В результате выполнения запроса мы получили идентификатор пользователя и его имя.

В таблице оценок grades нет фамилий учеников, но есть их идентификаторы (столбец student_id). В таком случае в условии WHERE определяем фильтр на значение student_id = 1. То, что идентификатор ученика Игоря Иванова равен единице, мы узнали из предыдущего запроса к таблице students.

```
sqlite> SELECT stream_id, grade FROM grades WHERE student_id = 1;
stream_id   grade
----------  -------------
1           4.9
2           4.9
3           5
sqlite>
```

В результате выполнения запроса мы получили идентификаторы потоков курсов, которые заканчивал Игорь Иванов, а также оценку за каждый курс.

Ограничение на вывод количества строк

Если все записи, возвращаемые запросом, нас не интересуют, мы можем ограничить количество строк результата посредством ключевого слова LIMIT. Например, нам надо узнать, закончил ли ученик хотя бы один из курсов. В этом случае в конце команды определяется ограничение LIMIT 1:

```
sqlite> SELECT stream_id, grade FROM grades WHERE student_id = 1 LIMIT 1;
stream_id   grade
----------  -------------
1           4.9
sqlite>
```

Сортировка результатов

Мы можем отсортировать вывод по одному или нескольким столбцам в прямом и обратном порядке, используя инструкцию ORDER BY. Для сортировки по возрастанию значений надо задать ASC, для сортировки в обратном порядке — DESC. В примере ниже отсортируем содержимое таблицы оценок по значению средней оценки в порядке возрастания значений:

```
sqlite> SELECT * FROM grades ORDER BY grade ASC;
student_id  stream_id   grade
----------  ----------  -------------
1           1           4.9
1           2           4.9
2           1           5
1           3           5
sqlite>
```

Сортировка по возрастанию значений применяется по умолчанию, поэтому ASC указывать необязательно:

```
sqlite> SELECT * FROM grades ORDER BY grade;
student_id  stream_id   grade
----------  ----------  -------------
1           1           4.9
1           2           4.9
2           1           5
1           3           5
sqlite>
```

Если мы хотим отсортировать данные в обратном порядке, то потребуется указать ключевое слово DESC:

```
sqlite> SELECT * FROM grades ORDER BY grade DESC;
student_id  stream_id   grade
----------  ----------  -------------
2           1           5
1           3           5
1           1           4.9
1           2           4.9
sqlite>
```

Получение только уникальных значений

Например, нам надо найти идентификаторы всех учеников, кто окончил хотя бы один курс. Для этого воспользуемся простым запросом ниже:

```
sqlite> SELECT student_id FROM grades;
student_id
----------
1
1
1
2
sqlite>
```

Проблема такого решения в том, что мы получим и все дублирующие значения идентификаторов, которые сейчас нам не требуются. Чтобы избавиться от дубликатов, применяется ключевое слово DISTINCT, которое даёт указание СУБД вернуть только уникальные значения из всей выборки:

```
sqlite> SELECT DISTINCT(student_id) FROM grades;
student_id
----------
1
2
sqlite>
```

Расширенные варианты определения условия

Выше мы рассмотрели базовые варианты построения запросов, но в реальных проектах часто требуется применить более сложную логику, чтобы получить данные.

Условие неравенства

Рассмотрим задачу, где надо определить курсы, количество занятий которых не равно 12. Для решения воспользуемся двумя вариантами синтаксиса — «!=» и «<>»:

```
sqlite> SELECT name FROM courses WHERE lessons_amount != 12;
name
----------------------
```

```
Linux. Рабочая станция
Основы Python
sqlite> SELECT name FROM courses WHERE lessons_amount <> 12;
name
----------------------
Linux. Рабочая станция
Основы Python
sqlite>
```

Условие «больше или меньше»

Для определения граничных значений используется синтаксис «больше или меньше»:

```
sqlite> SELECT name FROM courses WHERE lessons_amount < 12;
name
----------------------
Linux. Рабочая станция
Основы Python
sqlite> SELECT name FROM courses WHERE lessons_amount > 8;
name
-----------
Базы данных
sqlite>
```

Обратите внимание, что в части WHERE мы используем значения столбца количества уроков lessons_amount, но само это значение в части SELECT не выводим. Выводить значения тех столбцов, которые используются в условии, необязательно. Необходимость этого определяется из постановки задачи.

Условие по нескольким критериям

Иногда надо определить условие по нескольким критериям. Например, требуется выбрать все потоки с номерами больше 40 и меньше 50. В этом случае для комбинации двух условий используется логическое «И» (AND):

```
sqlite> SELECT * FROM streams WHERE number > 40 AND number < 50;
id          number      course_id   start_date
----------  ----------  ----------  ----------
1           45          2           2020-08-18
2           48          3           2020-10-02
sqlite>
```

Если потребуется выполнить поиск по нескольким значениям, то применяется логическое «ИЛИ» (OR). Например, найдём все потоки с идентификатором курса 1 или 3:

```
sqlite> SELECT * FROM streams WHERE course_id = 1 OR course_id = 3;
id          number      course_id   start_date
----------  ----------  ----------  ----------
2           48          3           2020-10-02
3           54          1           2020-11-12
sqlite>
```

Поиск по диапазону значений, BETWEEN

Для поиска по диапазону значений используется ключевое слово BETWEEN, для которого определяются граничные значения через AND. Найдём потоки с номерами, лежащими в диапазоне значений между 45 и 50 включительно:

```
sqlite> SELECT number, start_date FROM streams WHERE number BETWEEN 45 AND 50;
number      start_date
----------  ----------
45          2020-08-18
48          2020-10-02
sqlite>
```

Команду с BETWEEN всегда можно переписать с операторами сравнения и логическим «И» в условии. Ту же логику мы получим, подкорректировав один из предыдущих примеров:

```
sqlite> SELECT number, start_date FROM streams WHERE number >= 45 AND number <=
50;
number      start_date
----------  ----------
45          2020-08-18
48          2020-10-02
sqlite>
```

Оба варианта могут использоваться на практике, но синтаксис с BETWEEN выглядит более компактным и наглядным.

Поиск по списку значений, IN

Если есть список значений для поиска, применяется логическое «ИЛИ», как мы делали в примере выше. Найдём количество уроков по курсам «Linux. Рабочая станция» и «Основы Python»:

```
sqlite> SELECT lessons_amount FROM courses WHERE name = 'Linux. Рабочая станция'
OR name = 'Основы Python';
lessons_amount
--------------
8
8
sqlite>
```

Но со списками удобнее работать, используя ключевое слово IN, для которого в скобках мы перечисляем все искомые значения:

```
sqlite> SELECT lessons_amount FROM courses WHERE name IN('Linux. Рабочая
станция', 'Основы Python');
lessons_amount
--------------
8
8
sqlite>
```

Условия равенства для значения NULL

Вспомним, что NULL означает отсутствие значения, поэтому сравнивать значения с NULL обычным образом нельзя. Например, выражение ниже будет некорректным, хотя в SQLite ошибку мы не получим:

```
sqlite> SELECT name, lessons_amount FROM courses WHERE lessons_amount = NULL;
```

Правильнее использовать синтаксис IS NULL для проверки на пустое значение, и IS NOT NULL — для проверки на непустое значение. В первом выражении ниже мы ищем записи в курсах с незаполненным значением количества уроков (таких строк нет). Во втором выражении выводим все курсы с заполненными значениями:

```
sqlite> SELECT name, lessons_amount FROM courses WHERE lessons_amount IS NULL;
sqlite> SELECT name, lessons_amount FROM courses WHERE lessons_amount IS NOT
NULL;
name          lessons_amount
------------  --------------
Базы данных   12
Linux. Рабо   8
Основы Pyth   8
sqlite>
```

Арифметические операции

В SQL используются основные арифметические операции сложения, вычитания, умножения, деления, а также определение модуля числа.

```
sqlite> SELECT 3 + 5;
3 + 5
----------
8
sqlite> SELECT 23 - 7;
23 - 7
----------
```

```
16
sqlite> SELECT 3 * 12;
3 * 12
----------
36
sqlite> SELECT 30 / 5;
30 / 5
----------
6
sqlite> SELECT 5 % 2;
5 % 2
----------
1
```

Обратите внимание, что в примерах выше для вывода значений мы используем оператор SELECT, также как делаем это при запросе данных из таблиц базы данных. Хотя обращения к данным не происходит, СУБД рассчитывает выражение и возвращает результат.

Использование встроенных функций

Во всех современных СУБД, включая SQLite, применяется много готовых (встроенных) функций, которые участвуют в решении самых разных задач. На этом занятии мы рассмотрим функцию поиска по шаблону LIKE, некоторые функции работы с датой, а также функции агрегирования. Полный список встроенных функций с описанием их использования — в официальной документации.

Поиск по шаблону, LIKE

Предположим, мы знаем, что в имени курса есть слово Linux, но точное название курса нам неизвестно. В этом случае воспользуемся функцией сопоставления с шаблоном LIKE. Передадим этой функции параметром искомую строку, заключённую с двух сторон символами процента. Символ процента — это знак подстановки, который говорит, что в этой части строки находится любое количество разных символов:

```
sqlite> SELECT name, lessons_amount FROM courses WHERE name LIKE('%Linux%');
name                    lessons_amount
----------------------  --------------
Linux. Рабочая станция  8
sqlite>
```

На практике для этой функции скобки часто не используются. Тогда наш запрос примет следующий вид:

```
sqlite> SELECT name, lessons_amount FROM courses WHERE name LIKE '%Linux%';
name                    lessons_amount
----------------------  --------------
Linux. Рабочая станция  8
sqlite>
```

Функции для работы с датой и временем

Хотя SQLite не имеет специального типа данных для даты и времени, воспользуемся некоторыми функциями для выполнения задач с хранением и выборкой таких значений.

Чтобы получить текущее значение даты и времени, применяется функция CURRENT_TIMESTAMP:

```
sqlite> SELECT CURRENT_TIMESTAMP;
2021-01-05 17:21:20
sqlite>
```

В результате мы получим дату и время выполнения запроса в формате «год-месяц-число часы:минуты:секунды». Такое представление часто используется в реляционных базах данных, так как оно называется «формат баз данных».

Чтобы работать с текущей датой и временем, ещё используются функции DATE, TIME, DATETIME, которые ссылаются на текущее время посредством now:

```
sqlite> SELECT DATE('now');
DATE('now')
-----------
2021-01-10
sqlite> SELECT TIME('now');
TIME('now')
-----------
09:04:57
sqlite> SELECT DATETIME('now');
DATETIME('now')
-------------------
2021-01-10 09:05:02
sqlite>
```

При использовании этих функций с CURRENT_TIMESTAMP мы получим аналогичный результат:

```
sqlite> SELECT DATE(CURRENT_TIMESTAMP);
DATE(CURRENT_TIMESTAMP)
-----------------------
2021-01-10
sqlite> SELECT TIME(CURRENT_TIMESTAMP);
TIME(CURRENT_TIMESTAMP)
-----------------------
09:17:37
sqlite> SELECT DATETIME(CURRENT_TIMESTAMP);
DATETIME(CURRENT_TIMESTAMP)
---------------------------
2021-01-10 09:17:46
sqlite>
```

На практике функции CURRENT_TIMESTAMP и now взаимозаменяемые. Используйте тот вариант, который считаете удобным.

Дата и время также выделяются из строки:

```
sqlite> SELECT DATE('2021-01-05 17:21:20');
DATE('2021-01-05 17:21:20')
---------------------------
2021-01-05
sqlite> SELECT TIME('2021-01-05 17:21:20');
TIME('2021-01-05 17:21:20')
---------------------------
17:21:20
sqlite>
```

Если посмотрим текущий формат даты начала обучения в таблице по потокам, то увидим, что он не соответствует стандартному формату.

```
sqlite> SELECT start_date FROM streams;
start_date
----------
18.08.2020
02.10.2020
12.11.2020
sqlite>
```

Привести данные к подходящему виду позволит функция SUBSTRING (сокращённо — SUBSTR), вырезающая часть строки.

Чтобы понять, как работает эта функция, рассмотрим сначала простой пример. Пусть из текущей даты, которая получена функцией CURRENT_TIMESTAMP, нам надо получить только год.

```
sqlite> SELECT CURRENT_TIMESTAMP;
CURRENT_TIMESTAMP
-------------------
2021-01-10 09:39:37
sqlite>
```

Определим, в каких позициях символов находится значение года — год начинается с позиции 1, и вырезать потребуется 4 символа. Тогда для получения подходящего значения надо воспользоваться выражением SUBSTR(CURRENT_TIMESTAMP, 1, 4):

```
sqlite> SELECT SUBSTR(CURRENT_TIMESTAMP, 1, 4);
SUBSTR(CURRENT_TIMESTAMP, 1, 4)
-------------------------------
2021
sqlite>
```

То есть первым аргументом функции SUBSTR мы задаём строку, которую хотим обработать, вторым — позицию первого символа требуемой нам подстроки. Третьим аргументом определяем количество символов.

Применим аналогичный подход, чтобы преобразовать даты начала занятий таблицы, относящейся к потокам так, как нам надо. Вырежем:

- сначала год — начинается с символа 7 длиной 4 символа;
- затем месяц — начинается с символа 4 длиной 2 символа;
- и дату — начинается с символа 1 длиной 2 символа.

Чтобы объединить все части даты нового формата в одну строку, применяем в качестве оператора конкатенации две вертикальных черты ||:

```
sqlite> SELECT SUBSTR(start_date, 7, 4) || '-' || SUBSTR(start_date, 4, 2) ||
'-' || SUBSTR(start_date, 1, 2) FROM streams;
2020-08-18
2020-10-02
2020-11-12
sqlite>
```

Пример выше покажет даты в требуемом формате на экране, но в таблице формат её хранения не изменится. Чтобы внести изменения, скомпонуем команду на обновление UPDATE, используя то же выражение для замены значения в столбце start_date:

```
sqlite> UPDATE streams SET start_date = SUBSTR(start_date, 7, 4) || '-' ||
SUBSTR(start_date, 4, 2) || '-' || SUBSTR(start_date, 1, 2);
sqlite> SELECT * FROM streams;
id          number      course_id   start_date
----------  ----------  ----------  ----------
1           45          2           2020-08-18
2           48          3           2020-10-02
3           54          1           2020-11-12
sqlite>
```

Агрегирование данных

При работе с данными часто возникают задачи по обработке набора некоторых значений, такие как подсчёт количества, суммы, среднего значения и т. д. Операции такого типа называются агрегированием. Во время работы с базами данных под агрегированием понимается выполнение действий над значениями в различных строках таблицы посредством специальных функций, которые называются функциями агрегирования, или просто агрегаторами.

Рассмотрим типичные задачи, которые требуют применения функций агрегирования:

- определение среднего значения;

- подсчёт количества;
- определение максимального и минимального значения;
- подсчёт суммы.

Понятие группировки, нахождение среднего значения

В общем случае применяется функция агрегирования к данным всей таблицы. Например, найдём среднюю оценку всех учеников. Для этого воспользуемся функцией нахождения среднего значения AVG():

```
sqlite> SELECT AVG(grade) FROM grades;
AVG(grade)
------------------
4.95
sqlite>
```

Но как быть, если нам надо найти среднюю оценку по всем курсам для каждого ученика? В этом случае потребуется группировка. То есть надо явным образом указать, по какому набору строк мы хотим подсчитать результат. Если нам требуются данные в разрезе учеников, то логично в качестве значения для группировки указать идентификатор ученика:

```
sqlite> SELECT student_id, AVG(grade) FROM grades GROUP BY student_id;
student_id  AVG(grade)
----------  ------------------
1           4.93333333333333
2           5.0
sqlite>
```

Проще говоря, в выражении GROUP BY нам надо определить группы, к значениям которых применится функция агрегирования.

В общем виде команда агрегирования выглядит так:

```
SELECT 'Функция Агрегирования' FROM 'Имя Таблицы' GROUP BY 'Столбец
Группировки';
```

Подсчёт количества

Для подсчёта количества значений или строк применяется функция COUNT().

Найдём количество записей потоков для каждого курса. Чтобы подсчитать строки, определим параметром функции COUNT знак подстановки «звёздочка» и сгруппируем данные в разрезе идентификаторов курсов course_id:

```
sqlite> SELECT course_id, COUNT(*) FROM streams GROUP BY course_id;
course_id   COUNT(*)
----------  ----------
1           1
2           1
3           1
sqlite>
```

Другой вид с именами столбцов получается посредством режима вывода line:

```
sqlite> .mode line
sqlite> SELECT course_id, COUNT(*) FROM streams GROUP BY course_id;
course_id = 1
 COUNT(*) = 1

course_id = 2
 COUNT(*) = 1

course_id = 3
 COUNT(*) = 1
sqlite>
```

Чтобы улучшить вывод, надо дать расчётному столбцу COUNT(*) синоним (алиас), воспользовавшись ключевым словом AS:

```
sqlite> .mode line
sqlite> SELECT course_id, COUNT(*) AS 'streams_total' FROM streams GROUP BY
course_id;
    course_id = 1
streams_total = 1

    course_id = 2
streams_total = 1

    course_id = 3
streams_total = 1
sqlite>
```

Поиск максимального и минимального значения

Для поиска максимального и минимального значения используются функции MAX и MIN соответственно. Определим максимальную и минимальную оценки ученика по всем курсам. Так как нам надо получить данные в разрезе учеников, то группируем по идентификатору ученика:

```
sqlite> .header on
sqlite> .mode column
sqlite> SELECT student_id, MAX(grade) AS 'max_grade' FROM grades GROUP BY
student_id;
student_id  max_grade
```

```
----------  ----------
1           5
2           5
sqlite>
sqlite> SELECT student_id, MIN(grade) AS 'min_grade' FROM grades GROUP BY
student_id;
student_id  min_grade
----------  ----------
1           4.9
2           5
sqlite>
```

Получение суммы

Функция SUM() позволит найти сумму значений столбца. Подсчитаем общее количество занятий по всем курсам. Обратите внимание, что в этом случае группировка не требуется:

```
sqlite> SELECT SUM(lessons_amount) AS 'lessons_total' FROM courses;
lessons_total
-------------
28
sqlite>
```

Применение фильтра к расчётным значениям, HAVING

Разберём один из предыдущих примеров по нахождению минимальной оценки. Как быть, если нам требуется информация только по средним оценкам, значения которых меньше пяти? Первая идея — применить условия WHERE, но в этом случае мы получим ошибку:

```
sqlite> SELECT student_id, MIN(grade) AS min_grade FROM grades GROUP BY
student_id WHERE min_grade < 5;
Error: near "WHERE": syntax error
sqlite>
```

Определить условие через WHERE мы можем только для данных, которые уже есть в таблице. В рассмотренном примере значение min_grade рассчитывается посредством функции, то есть получаем результат во время выполнения запроса. Мы получили ошибку, потому что условие WHERE нельзя применять для рассчитываемых значений.

Для определения условия в таких случаях вместо WHERE применяется ключевое слово HAVING, которое имеет тот же смысл — определяет фильтр для вывода результата:

```
sqlite> SELECT student_id, MIN(grade) AS min_grade FROM grades GROUP BY
student_id HAVING min_grade < 5;
student_id  min_grade
----------  ----------
```

```
1           4.9
sqlite>
```

На основе приёмов, рассмотренных в этом занятии, получится сформировать более сложные запросы, которые решат реальные практические задачи. Например, найдём, сколько в программе курсов, которые относятся к языку программирования Python, сгруппированных по количеству уроков, а также выведем в отчёте текущий год:

```
SELECT
  SUBSTR(DATE(), 1, 4) AS year,
  lessons_amount,
  COUNT(*) AS courses_amount
FROM courses
  WHERE name LIKE '%Python%'
  GROUP BY lessons_amount;
```

Обратите внимание, что в этом случае мы используем переносы строки и табуляцию для представления более сложной логики запроса в наглядном виде.

Выполним запрос и проверим результат:

```
sqlite> SELECT
   ...>   SUBSTR(DATE(), 1, 4) AS year,
   ...>   lessons_amount,
   ...>   COUNT(*) AS courses_amount
   ...> FROM courses
   ...>   WHERE name LIKE '%Python%'
   ...>   GROUP BY lessons_amount;
year        lessons_amount  courses_amount
----------  --------------  --------------
2021        8               1
sqlite>
```

Практическое задание

Работаем с базой данных учителей teachers.db. Для каждого задания надо создать запрос, сдать нужно только код запросов в текстовом файле.

1. Преобразовать дату начала потока в таблице потоков к виду год-месяц-день. Используйте команду UPDATE.
2. Получите идентификатор и номер потока, запланированного на самую позднюю дату.
3. Покажите уникальные значения года по датам начала потоков обучения.

4. Найдите количество преподавателей в базе данных. Выведите искомое значение в столбец с именем total_teachers.
5. Покажите даты начала двух последних по времени потоков.
6. Найдите среднюю успеваемости учеников по всем потокам преподавателя с идентификатором равным 1.
7. Дополнительное задание (выполняется по желанию): найдите идентификаторы преподавателей, у которых средняя успеваемость по всем потокам меньше 4.8.

Глоссарий

Агрегирование данных — выполнение действий над набором данных. При работе с базами данных обрабатывается набор данных конкретного столбца таблицы.

Функции агрегирования, агрегаторы — функции, применяемые для обработки набора данных. Например, подсчёт количества, нахождение суммы или среднего значения и т. д.

Группировка — определение признака для разбивки всех строк на группы для последующего применения над ними некоторой функции агрегирования.

Дополнительные материалы

1. Документация SQLite, команда SELECT.
2. Документация SQLite, операторы.
3. Документация SQLite, выражения.
4. Документация SQLite, оператор WHERE.
5. Документация SQLite, операторы AND и OR.
6. Документация SQLite, оператор LIKE.
7. Документация SQLite, оператор LIMIT.
8. Документация SQLite, оператор ORDER BY.
9. Документация SQLite, оператор GROUP BY.
10. Документация SQLite, оператор HAVING.
11. Документация SQLite, оператор DISTINCT.
12. Документация SQLite, синтаксис ALIAS.

13. Документация SQLite, дата и время.
14. Документация SQLite, функции.

Используемые источники

1. Документация SQLite, оператор SELECT.
2. Документация SQLite, выражения.
3. Документация SQLite, встроенные функции.
4. Документация SQLite, функции даты и времени.
5. Документация SQLite, функции агрегирования.