GeekBrains

Расширенные возможности SQL

SQLite 3.31.1

На этом уроке

1. Научимся работать с дополнительными возможностями языка SQL.
2. Рассмотрим варианты практического применения представлений, временных таблиц, транзакций и триггеров.
3. Узнаем, как анализировать сложные запросы.

Оглавление

Расширенные возможности SQL

Язык SQL не представляет собой язык программирования общего применения как, например, Python, Java или Ruby. Это специализированный язык запросов. Но в его инструментарий также заложены возможности, позволяющие выйти за рамки типичного применения команд для работы с данными, таких как вставка данных, редактирование или выборка.

Рассмотрим некоторые дополнительные структуры, которые реализуются в СУБД SQLite:

- представления;
- временные таблицы;
- транзакции;
- триггеры.

Представления

Представление проще всего определить как именованную выборку. Это срез данных, которому мы даём имя и используем в дальнейшем как виртуальную таблицу. Рассмотрим запрос на получение данных об оценках всех учеников, который рассматривали на прошлом занятии:

```
SELECT
  students.surname,
  students.name,
  courses.name AS course_name,
  streams.number AS stream_number,
  grades.grade
FROM students
  LEFT JOIN grades
    ON students.id = grades.student_id
  LEFT JOIN streams
    ON grades.stream_id = streams.id
  LEFT JOIN courses
    ON streams.course_id = courses.id;
```

Взяв этот запрос за основу, создадим представление, отображающее аналогичный срез данных. Базовая команда создания представления выглядит следующим образом:

```
CREATE VIEW 'Имя Представления' AS SELECT ...;
```

То есть нам надо дать представлению имя и после ключевого слова AS определить запрос, который хотим использовать:

```
CREATE VIEW students_info AS
SELECT
  students.surname,
  students.name,
  courses.name AS course_name,
  streams.number AS stream_number,
  grades.grade
FROM students
  LEFT JOIN grades
    ON students.id = grades.student_id
  LEFT JOIN streams
    ON grades.stream_id = streams.id
  LEFT JOIN courses
    ON streams.course_id = courses.id;
```

Выполним код:

```
sqlite> CREATE VIEW students_info AS
   ...> SELECT
   ...>   students.surname,
   ...>   students.name,
   ...>   courses.name AS course_name,
   ...>   streams.number AS stream_number,
   ...>   grades.grade
   ...> FROM students
   ...>   LEFT JOIN grades
   ...>     ON students.id = grades.student_id
   ...>   LEFT JOIN streams
   ...>     ON grades.stream_id = streams.id
   ...>   LEFT JOIN courses
   ...>     ON streams.course_id = courses.id;
sqlite>
```

Теперь у нас есть представление students_info. Его мы используем в запросах обычным образом:

```
SELECT * FROM students_info;
```

Или выберем только требуемые столбцы для отчёта:

```
SELECT surname, name, course_name, grade FROM students_info;
```

Проверим работу последнего варианта.

```
sqlite> .header on
sqlite> .mode column
sqlite> SELECT surname, name, course_name, grade FROM students_info;
surname     name        course_name             grade
----------  ----------  ----------------------  -------------
Иванов      Игорь       Linux. Рабочая станция  4.9
Иванов      Игорь       Основы Python           4.9
Иванов      Игорь       Базы данных             5
Павлова     Анастасия   Linux. Рабочая станция  5
Васильева   Ирина
sqlite>
```

Если представление надо удалить, применяем синтаксис DROP VIEW:

```
sqlite> DROP VIEW students_info;
```

Временные таблицы

Если в процессе работы требуется создать таблицу только для некоторых промежуточных данных, которые не надо хранить, создаём временную таблицу. Временная таблица создаётся в оперативной памяти системы, её данные не сохраняются на носителе информации. Такая таблица удаляется из памяти сразу после завершения работы текущей сессии пользователя в СУБД.

Например, нам требуется таблица для ввода данных об оценках учеников за каждый урок. Посчитаем из этих данных среднюю оценку и сохраним её значение в записи таблицы оценок grades. Оценки по каждому уроку нам в долговременном плане не понадобятся, поэтому сформируем временную таблицу оценок за отдельные уроки lesson_scores:

```
CREATE TEMPORARY TABLE lesson_scores (
  Id INTEGER PRIMARY KEY AUTOINCREMENT,
  student_id INTEGER NOT NULL,
  stream_id INTEGER NOT NULL,
  lesson INTEGER NOT NULL,
  score REAL NOT NULL
);
```

Мы сможем работать с этой таблицей так же, как и с обычными таблицами — записывать данные, делать выборки. Но когда пользователь выйдет из соединения с базой данных, эта таблица не сохранится.

Важно! *Данные таблицы и её структура не сохранятся. Если мы захотим поработать с этой таблицей в следующий раз, понадобится создать её заново.*

Проверим работу временной таблицы:

```
sqlite> CREATE TEMPORARY TABLE lesson_scores (
   ...>   Id INTEGER PRIMARY KEY AUTOINCREMENT,
   ...>   student_id INTEGER NOT NULL,
   ...>   stream_id INTEGER NOT NULL,
   ...>   lesson INTEGER NOT NULL,
   ...>   score REAL NOT NULL
   ...> );
sqlite> INSERT INTO lesson_scores VALUES (1, 2, 3, 1, 4.9);
sqlite> SELECT * FROM lesson_scores;
Id          student_id  stream_id   lesson      score
----------  ----------  ----------  ----------  ----------
1           2           3           1           4.9
sqlite>
```

Если временная таблица больше не требуется, она удаляется обычным образом ещё до окончания текущей сессии.

```
sqlite> DROP TABLE lesson_scores;
sqlite> .schema lesson_scores
sqlite>
```

Транзакции

При работе с данными встречаются задачи, когда требуется выполнить несколько операций как одну неделимую логическую операцию. Например, для базы данных учеников рассмотрим ситуацию, когда надо удалить данные конкретного человека. Для выполнения этой задачи требуется удалить запись ученика из таблицы учеников students, а также все записи о его оценках из таблицы оценок, так как без ученика хранить их не имеет смысла.

Удаляем запись об ученике:

```
DELETE FROM students WHERE id = 1;
```

Удаляем все оценки ученика:

```
DELETE FROM grades WHERE student_id = 1;
```

Задание выглядит несложным, но может возникнуть проблема, когда одна из операций пройдёт с ошибкой. В этом случае выполним только часть всей задачи: удалим пользователя или его оценки. Желательно иметь механизм, гарантирующий только два варианта выполнения этих команд — либо все команды выполнены успешно, либо, в случае сбоя на любом из этапов, не выполнена ни одна из команд. Для решения такой задачи в SQL используются транзакции.

Транзакция — набор команд, которые выполняются только вместе. Если при выполнении любой команды возникает ошибка, то для всех уже выполненных команд выполняется откат, и данные остаются в исходном состоянии.

Для транзакций определяются четыре базовых свойства — атомарность, целостность, изолированность и устойчивость.

1. Атомарность — обеспечивает выполнение правила «всё или ничего». Выполняются все команды, или ни одна из них, третьего варианта нет.
2. Целостность — обеспечивает корректные изменения в данных при успешном выполнении транзакции и гарантирует отсутствие нарушений связности данных различных таблиц.
3. Изолированность — обеспечивает независимое выполнение транзакций, транзакции не влияют друг на друга.

4. Устойчивость — обеспечивает предсказуемость поведения транзакции, когда возникают сбои системы.

В литературе часто эти свойства определяются аббревиатурой ACID, сокращение от Atomicity, Consistency, Isolation, Durability.

Рассмотрим синтаксис транзакций. Начинается транзакция командой BEGIN TRANSACTION, завершается, в базовом варианте, командой выполнения COMMIT.

```
BEGIN TRANSACTION;
  Операция 1;
  Операция 2;
  ...
  Операция N;
COMMIT;
```

Для нашей задачи удаления ученика транзакция будет выглядеть следующим образом:

```
BEGIN TRANSACTION;
  DELETE FROM grades WHERE student_id = 1;
  DELETE FROM students WHERE id = 1;
COMMIT;
```

В теле транзакции у нас две команды — команда удаления оценок и команда удаления ученика. Если на любом этапе возникнет ошибка, то все уже выполненные команды отменятся (откатятся), и изменения в данных не произойдут.

Например, при выполнении нормально отработала первая команда по удалению оценок, но во время удаления ученика возникла ошибка. В этом случае подтверждение транзакции COMMIT не произойдёт, а все изменения в данных по первой команде отменятся, то есть данные вернутся в состояние, как перед выполнением транзакции. Если при выполнении команд в теле транзакции проблем не возникает, то выполняется подтверждение изменений COMMIT, и данные меняются окончательно.

```
sqlite> SELECT * FROM students;
id          surname     name
----------  ----------  ----------
1           Иванов      Игорь
2           Павлова     Анастасия
3           Васильева   Ирина
sqlite> SELECT * FROM grades;
student_id  stream_id   grade
----------  ----------  -------------
1           1           4.9
2           1           5
```

```
1           3           5
1           2           4.9
sqlite> BEGIN TRANSACTION;
sqlite>   DELETE FROM grades WHERE student_id = 1;
sqlite>   DELETE FROM students WHERE id = 1;
sqlite> COMMIT;
sqlite> SELECT * FROM students;
id          surname     name
----------  ----------  ----------
2           Павлова     Анастасия
3           Васильева   Ирина
sqlite> SELECT * FROM grades;
student_id  stream_id   grade
----------  ----------  -------------
2           1           5
sqlite>
```

Если мы не хотим сохранять изменения в данных по командам транзакции, то вместо подтверждения COMMIT произведём откат изменений ROLLBACK.

```
BEGIN TRANSACTION;
  DELETE FROM grades WHERE student_id = 2;
  DELETE FROM students WHERE id = 2;
ROLLBACK;
```

Выполним транзакцию с откатом для ученика с идентификатором 2:

```
sqlite> SELECT * FROM students;
id          surname     name
----------  ----------  ----------
2           Павлова     Анастасия
3           Васильева   Ирина
sqlite> SELECT * FROM grades;
student_id  stream_id   grade
----------  ----------  -------------
2           1           5
sqlite> BEGIN TRANSACTION;
sqlite>   DELETE FROM grades WHERE student_id = 2;
sqlite>   DELETE FROM students WHERE id = 2;
sqlite> ROLLBACK;
sqlite> SELECT * FROM students;
id          surname     name
----------  ----------  ----------
2           Павлова     Анастасия
3           Васильева   Ирина
sqlite> SELECT * FROM grades;
student_id  stream_id   grade
----------  ----------  -------------
```

```
2           1           5
sqlite>
```

Для команд транзакции выполнился откат, и данные остались в первоначальном виде.

Триггеры

Триггеры рассматриваются как задачи, которые выполняются автоматически при наступлении конкретного события — вставки строки, обновлении данных, удалении строки.

```
CREATE TRIGGER 'Имя триггера' [BEFORE|AFTER] 'Событие'
ON 'Таблица'
BEGIN
 -- Действия ....
END;
```

Посредством синтаксиса CREATE TRIGGER мы задаём имя триггера, определяем событие, по которому будет срабатывать триггер. После ключевого слова ON определяем таблицу, по ней будет отслеживаться событие, а в теле триггера BEGIN… END определяем необходимые действия.

Рассмотрим практическую задачу, которая решается посредством триггера. В таблице потоков streams хранится дата начала занятий для каждого потока в формате «год-месяц-день». Проблема в том, что СУБД SQLite никак не проверяет введенное значение на соответствие формату даты. Мы можем поместить в это поле дату в другом формате или вообще любую строку:

```
sqlite> INSERT INTO streams (number, course_id, start_date) VALUES (55, 1,
'01-03-21');
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
4           55          1           01-03-21
sqlite>
```

Очевидно, что было бы хорошо иметь механизм, который проверяет формат дат и вызывает ошибку с соответствующим сообщением пользователю при попытке вставить некорректное значение.

В решении такой задачи используется триггер, но сначала удалим строку с неправильной датой.

```
DELETE FROM streams WHERE id = 4;
```

Если потребуется проверить формат даты при создании записи, надо, чтобы триггер срабатывал перед вставкой записи (BEFORE INSERT) в таблице потоков streams. В теле триггера проверим формат даты.

```
CREATE TRIGGER check_start_date_format BEFORE INSERT
ON streams
BEGIN
 -- Проверяем формат даты 'ГГГГ-ММ-ДД'
END;
```

В тело триггера добавим логику, которая проверяет введенное значение даты на соответствие формату ‘ГГГГ-ММ-ДД’:

```
CREATE TRIGGER check_start_date_format BEFORE INSERT
ON streams
BEGIN
  SELECT CASE
  WHEN
    (NEW.start_date NOT LIKE '____-__-__')
    OR (CAST(SUBSTR(NEW.start_date, 1, 4) AS INTEGER) NOT BETWEEN 2021 AND 2022)
    OR (CAST(SUBSTR(NEW.start_date, 6, 2) AS INTEGER) NOT BETWEEN 1 AND 12)
    OR (CAST(SUBSTR(NEW.start_date, 9, 2) AS INTEGER) NOT BETWEEN 1 AND 31)
  THEN
    RAISE(ABORT, 'Wrong format for start_date!')
  END;
END;
```

Рассмотрим логику работы этого триггера более подробно. Перед вставкой новой строки в таблицу потоков срабатывает триггер check_start_date_format. Значение даты (NEW.start_date) передаётся в конструкцию CASE, где в части WHEN выполняется проверка соответствия формата по четырём признакам. Если хоть одна из проверок не проходит, то выполняется команда в части THEN, которая прерывает операцию вставки строки, и пользователь получает ошибку с сообщением, что формат ввода даты неправильный. А если с форматом даты всё в порядке, то операция вставки завершается успешно.

Создадим триггер:

```
sqlite> CREATE TRIGGER check_start_date_format BEFORE INSERT
   ...> ON streams
   ...> BEGIN
   ...>   SELECT CASE
   ...>   WHEN
   ...>     (NEW.start_date NOT LIKE '____-__-__')
   ...>     OR (CAST(SUBSTR(NEW.start_date, 1, 4) AS INTEGER) NOT BETWEEN 2021
AND 2022)
```

```
   ...>     OR (CAST(SUBSTR(NEW.start_date, 6, 2) AS INTEGER) NOT BETWEEN 1 AND
12)
   ...>     OR (CAST(SUBSTR(NEW.start_date, 9, 2) AS INTEGER) NOT BETWEEN 1 AND
31)
   ...>   THEN
   ...>     RAISE(ABORT, 'Wrong format for start_date!')
   ...>   END;
   ...> END;
sqlite>
```

Для проверки правильности работы триггера вставим две строки — одну с неправильным форматом даты начала занятий, а вторую с корректным форматом. В первом случае нам надо получить ошибку, во втором — операция должна пройти успешно.

```
sqlite> INSERT INTO streams (number, course_id, start_date) VALUES (55, 1,
'01-03-21');
Error: Wrong format for start_date!
sqlite>
```

Мы получили сообщение об ошибке, всё верно, так и должно быть. Исправим дату и попробуем ещё раз:

```
sqlite> INSERT INTO streams (number, course_id, start_date) VALUES (55, 1,
'2021-03-21');
sqlite> SELECT * FROM streams WHERE number = 55;
id          number      course_id   start_date
----------  ----------  ----------  ----------
9           55          1           2021-03-21
sqlite>
```

На этот раз добавление строки прошло успешно, триггер работает, как ожидается.

Анализ сложных запросов

Во время работы над существующими проектами будет возникать необходимость разбирать код SQL, который написан ранее. Рассмотрим вариант анализа на примере запроса ниже. Цель анализа — полное понимание логики работы запроса.

```
SELECT st.surname AS student_surname, st.name AS student_name, c.name AS
course_name, sr.number AS stream_number, sr.start_date
FROM students st LEFT JOIN grades g ON st.id = g.student_id LEFT JOIN streams sr
ON stream_id = sr.id LEFT JOIN courses c ON course_id = c.id
WHERE g.grade = 5 ORDER BY sr.start_date;
```

Выполним этот запрос, убедимся, что он рабочий, и попытаемся сделать некоторые догадки о том, для чего он нужен, по выводимым данным.

```
sqlite> SELECT st.surname AS student_surname, st.name AS student_name, c.name AS
course_name, sr.number AS stream_number, sr.start_date
   ...> FROM students st LEFT JOIN grades g ON st.id = g.student_id LEFT JOIN
streams sr ON stream_id = sr.id LEFT JOIN courses c ON course_id = c.id
   ...> WHERE g.grade = 5 ORDER BY sr.start_date;
student_surname  student_name  course_name             stream_number  start_date
---------------  ------------  ----------------------  -------------  ----------
Павлова          Анастасия     Linux. Рабочая станция  45             2020-08-18
Иванов           Игорь         Базы данных             54             2020-11-12
sqlite>
```

Запрос выводит некоторые данные об учениках, сказать точнее пока ничего нельзя. Надо иметь в виду, что не всегда есть возможность проверить запрос на данных. Поэтому потребуется научиться понимать назначение запроса без его выполнения. Первое, что надо сделать — поработать над стилем оформления кода и привести его структуру в порядок, используя отступы и переносы строк.

```
SELECT
  st.surname AS student_surname,
  st.name AS student_name,
  c.name AS course_name,
  sr.number AS stream_number,
  sr.start_date
  FROM students st
    LEFT JOIN grades g
      ON st.id = g.student_id
    LEFT JOIN streams sr
      ON stream_id = sr.id
    LEFT JOIN courses c
      ON course_id = c.id
  WHERE g.grade = 5
  ORDER BY sr.start_date;
```

В таком виде запрос выглядит значительно лучше, но новичков могут путать сокращённые алиасы (синонимы) таблиц. Трудно сориентироваться, на какую таблицу ссылается столбец (c.name, sr.number). Для более наглядного представления уберём все алиасы таблиц и заменим их полными именами. Заметим, что алиасы столбцов отчёта мы не убираем.

```
SELECT
  students.surname AS student_surname,
  students.name AS student_name,
  courses.name AS course_name,
  streams.number AS stream_number,
  streams.start_date
```

```
  FROM students
    LEFT JOIN grades
      ON students.id = grades.student_id
    LEFT JOIN streams
      ON stream_id = streams.id
    LEFT JOIN courses
      ON course_id = courses.id
  WHERE grades.grade = 5
  ORDER BY streams.start_date;
```

Код запроса становится понятнее, но в условиях объединения ON stream_id = streams.id и ON course_id = courses.id пока неясно, на какие таблицы ссылаются столбцы stream_id и course_id. Требуется провести небольшое расследование и определить, в каких таблицах из запроса они находятся. Мы видим, что это таблицы grades и streams соответственно.

```
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> .schema streams
CREATE TABLE streams (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  number INTEGER NOT NULL UNIQUE,
  course_id INTEGER NOT NULL,
  start_date TEXT NOT NULL,
  FOREIGN KEY (course_id) REFERENCES courses(id)
);
sqlite>
```

Укажем недостающие имена таблиц в запросе.

```
SELECT
  students.surname AS student_surname,
  students.name AS student_name,
  courses.name AS course_name,
  streams.number AS stream_number,
  streams.start_date
  FROM students
    LEFT JOIN grades
      ON students.id = grades.student_id
    LEFT JOIN streams
      ON grades.stream_id = streams.id
    LEFT JOIN courses
```

```
    ON streams.course_id = courses.id
WHERE grades.grade = 5
ORDER BY streams.start_date;
```

Когда все проблемы в оформлении кода решены, приступаем к анализу его функциональных частей. Удобно анализировать сложные запросы в таком порядке:

- сначала определяем, откуда берутся данные (часть запроса FROM), а также группировку (если есть);
- затем смотрим, что выводится в отчёт (часть запроса SELECT);
- в конце смотрим фильтр (WHERE) и сортировку (ORDER BY).

Начинаем с части запроса FROM.

```
FROM students
  LEFT JOIN grades
    ON students.id = grades.student_id
  LEFT JOIN streams
    ON grades.stream_id = streams.id
  LEFT JOIN courses
    ON streams.course_id = courses.id
```

Мы видим объединение четырёх таблиц — учеников (students), оценок (grades), потоков (streams) и курсов (courses). Важно, что таблица учеников в объединении самая левая, а все остальные таблицы присоединены посредством левого внешнего объединения. По логике запроса нам надо получить в отчёте всех учеников, независимо от того, есть для них данные в остальных трёх таблицах или нет. Объединение таблиц происходит по значениям столбцов связи grades.student_id, grades.stream_id и streams.course_id.

Рассмотрим часть запроса SELECT.

```
SELECT
  students.surname AS student_surname,
  students.name AS student_name,
  courses.name AS course_name,
  streams.number AS stream_number,
  streams.start_date
```

Когда у нас есть полные имена таблиц, которым принадлежат столбцы, понять, что выбирает запрос несложно. Нам надо получить фамилию ученика, имя ученика, название курса, номер и дату начала потока.

На этом этапе анализа мы понимаем, что выбирается и откуда. Следующий шаг — анализ условия (фильтра) выборки.

```
WHERE grades.grade = 5
```

Мы видим, что из всех записей объединения в результат попадут только те, где оценка ученика за курс равна пяти.

```
ORDER BY streams.start_date
```

Осталась сортировка — строки отчёта требуется отсортировать по дате начала потока.

Подведём итог анализа. Рассмотренный запрос выбирает учеников, которые завершили курсы на отлично. В отчёт выводится имя ученика, фамилия, название курса, номер потока, дата начала потока. Строки отсортируются по дате начала потока.

Практическое задание

Работаем с базой данных учителей teachers.db. Для каждого задания надо сдать только код, который выполняется для получения результата, в текстовом файле.

1. Создайте представление, которое для каждого курса выводит название, номер последнего потока, дату начала обучения последнего потока и среднюю успеваемость курса по всем потокам.

2. Удалите из базы данных всю информацию, которая относится к преподавателю с идентификатором, равным 3. Используйте транзакцию.

3. Создайте триггер для таблицы успеваемости, который проверяет значение успеваемости на соответствие диапазону чисел от 0 до 5 включительно.

4. **Дополнительное задание.** Создайте триггер для таблицы потоков, который проверяет, что дата начала потока больше текущей даты, а номер потока имеет наибольшее значение среди существующих номеров. При невыполнении условий необходимо вызвать ошибку с информативным сообщением.

Глоссарий

Представление — именованный срез данных.

Временная таблица — таблица, которая существует только на время текущей сессии пользователя, структура и данные временной таблицы не сохраняются на носитель.

Транзакция — набор команд, которые выполняются как одна операция. В случае возникновения проблем на любом этапе выполнения, изменения по всем уже выполненным командам отменяются.

Атомарность — свойство транзакции, которое обеспечивает выполнение правила «всё или ничего». Выполняются либо все команды, либо ни одна из них, третьего варианта нет.

Целостность — свойство транзакции, которое обеспечивает корректные изменения в данных при успешном выполнении транзакции и гарантирует отсутствие нарушений связности данных различных таблиц.

Изолированность — свойство транзакции, которое обеспечивает независимое выполнение транзакций, транзакции не влияют друг на друга.

Устойчивость — свойство транзакции, которое обеспечивает предсказуемость поведения транзакции, когда происходят сбои системы.

Триггер — задача, которая выполняется автоматически при наступлении некоторого события.

Дополнительные материалы

1. Статья «Unetway. Представления».
2. Статья «Unetway. Транзакции».
3. Статья «Unetway. Триггеры».

Используемые источники

1. Документация SQLite, Views.
2. Документация SQLite, Temporary tables.
3. Документация SQLite, Transactions.
4. Документация SQLite, Triggers.