Неполное Руководство по SQLite для пользователей Windows

Перевод: А.Г. Пискунов

21 августа 2014 г.

АННОТАЦИЯ

Текст документа является не совсем полным переводом некоторых глав монографии Grant Allen и Mike Owens 'The Definitive Guide to SQLite' http://www.apress.com/9781590596739. В частности, пропущены общие рассуждения во введениях к разделам и исторические ссылки, тонкости использования базы данных для пользователей Линукса (-ов).

Глава 1

НАЧАЛО

Текст документа является не совсем полным переводом четвертой и пятой глав монографии Grant Allen и Mike Owens 'The Definitive Guide to SQLite'

1.1 Получение исходных кодов примеров книги

Внизу страницы http://www.apress.com/9781590596739 найдите закладку Source Code/Download

Глава 2

ВВЕДЕНИЕ В SQLite

SQLite - это встраиваемая реляционная база данных, поставляемая с исходными кодами. Впервые выпущена в 2000 году, предназначена для предоставления привычных возможностей реляционных баз данных без присущих им накладных расходов. За время эксплуатации успела заслужить репутацию как переносимая, легкая в использовании, компактая, производительная и надежная база данных.

2.1 Встраиваемая база данных

Встраиваемость базы данных означает, что она существует не как процесс, отдельный от обслуживаемого процесса, а является его частью - частью некоторого прикладного приложения. Внешний наблюдатель не заметит, что прикладное приложение пользуется РСУБД. Тем не менее, приложение таки использует РСУБД. Это избавляет от необходимости сетевых настроек, никаких файерволов, никаких сетевых адресов, никаких пользователей и конфликтов их прав доступа. И клиент, и сервер работают в одном процессе. Это избавляет от проблем конфигурирования. Все, в чем нуждается программист уже скомпилировано в его приложении.

Взгляните на 2.1 , скрипт Perl, обычная программа на Си, скрипт PHP - все используют SQLite . В конце концов, каждый из трех процессов пользуется SQLite C API. Таким образом, SQLite встроен в адресное пространство каждого из них. Каждый из них становится независимым сервером базы данных. И, кроме того, хотя каждый процесс представляет независимый сервер, они могут выполнять операции на одном и том же файле (-ах) базы данных. SQLite позволяет им управляться с синхронизацией и блокировками.

На текущем рынке встроенных баз данных представлено много продуктов от различных производителей, но только один из них поставляется с открытыми исходниками, не требует лицензионных сборов и спроектирован ислючительно как встраиваемая БД - это SQLite .

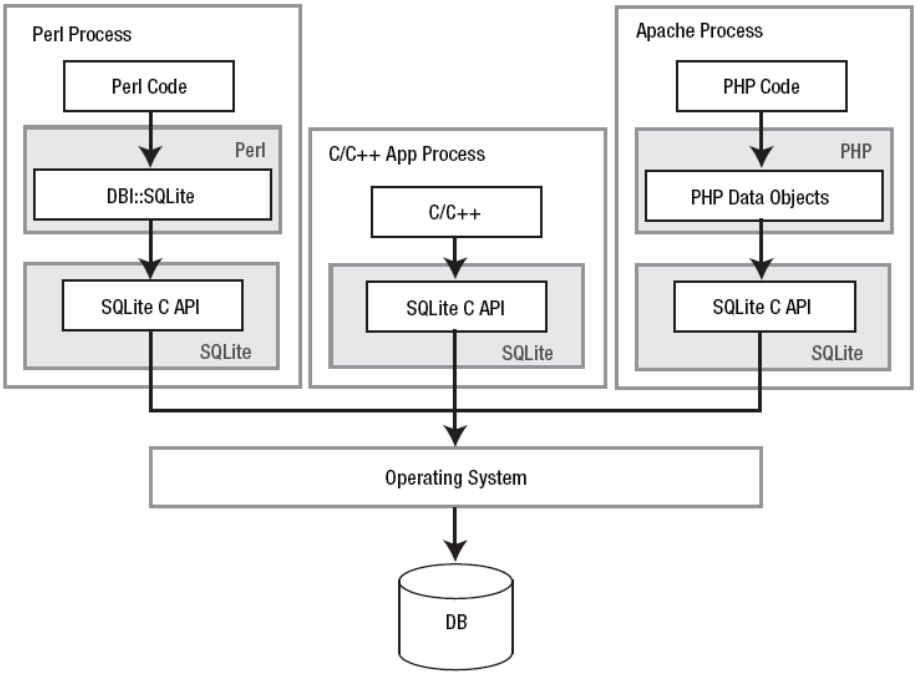


Рис. 2.1: SQLite встраивается в приложения

2.2 Архитектура

SQLite имеет элегантную модульную архитектуру, отображающую уникальные подходы к управлению реляционными базами данных. Восемь отдельных модулей сгруппированы в три главных подсистемы (см. 2.2). Они разделяют обработку запроса на отдельные задачи, которые работают подобно конвееру. Верхние модули компилируют запросы, средние выполняют их, а нижние управляются с диском и взаимодействуют с операционной системой.

2.2.1 Интерфейс

Инрерфейс является верхним модулем и состоит из м C API. Общение с SQLite производится через него.

2.3 Особенности и философия

Несмотря на маленький размер, SQLite предоставляет обескураживающий спектр особенностей и возможностей. Он поддерживает весьма полный набор стандарта ANSI SQL92 для особенностей языка SQL, а также такие особенности как триггера, индексы, столбцы с автоинкрементом, LIMIT/OFFSET особенности. Так же поддерживаются такие редкие свойства, как динамическая типизация и разрешение конфликтов.

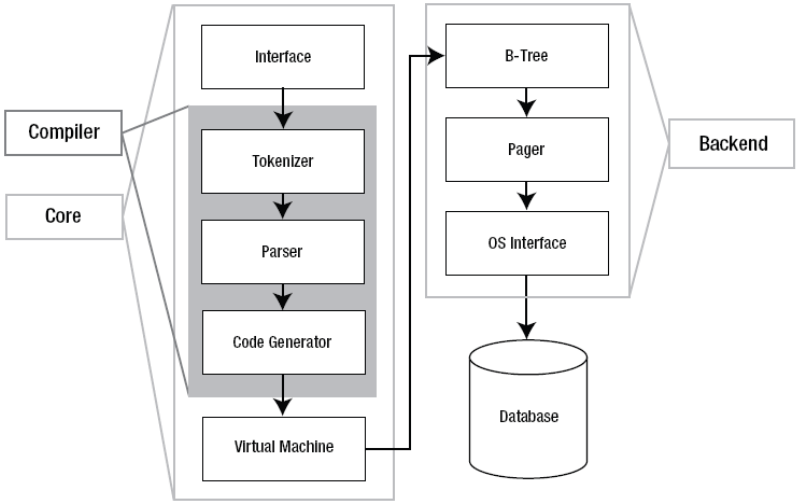


Рис. 2.2: архитектура SQLite

2.4 Примеры

База данных с примерами из этой книги доступна в интернете по адресу http://www.apress.com/9781590596739. Искать на второй закладке, на слове Source Code/DownLoads.

Не стесняйтесь и присылайте свои замечания или коментарии, или советы о книге и/или её примерах на мыло авторам sqlitebook@gmail.com (Michael) или grantondata@gmail.com (Grant).

Глава 3

НАЧИНАЕМ

3.1 Где брать SQLite ?

Как ни странно, но на сайте производителя - http://www.sqlite.org. Там можно найти готовые собранные библиотеки и утилиты для Windows в составе:

sqlite3 command-line program (CLP): отдельная самодостаточная утилиты для выполнения скриптов. Далее называется как CLP. Скачиваете её и немедленно выполняете SQL операторы, которые надо писать красивыми белыми буквами на замечательном черном фоне. Без всякой мышки, раздражающих иконок, инсталляций и перезагрузок операционной системы.

Динамически подгружаемыя библиотека SQLite (DLL): Это и есть сервер SQLite . Приложения для работы с SQLite должны динамически подгружать эту библиотеку.

Анализатор SQLite : утилита необходима для сбора статистики об использовании базы данных, полезна для увеличения производительности работы приложения.

3.2 SQLite под Windows

Собираетесь ли Вы использовать SQLite как пользователь, писать программы, или изучать теорию баз данных и SQL, SQLite устанавливается на Ваш Windows с минимумом суеты.

3.2.1 Загрузка CLP

Секция Download на сайте http://www.sqlite.org, далее - Precompiled Binaries For Windows - command-line shell. В загруженном архиве должен оказаться файл sqlite3.exe. В шелле cmd (Windows command shell - файл cmd.exe - красивое черное окно с белыми буквами), находясь в каталоге с файлом

sqlite3.exe, наберите sqlite3 и нажмите Enter. Преодолевший эти неимоверные трудности по запуску cmd, с некоторой вероятностью должен увидеть окно, похожее на первые три строки из окна на картинке 3.1 , Смело набирайте .exit. Ваша копия SQLite CLP работает.

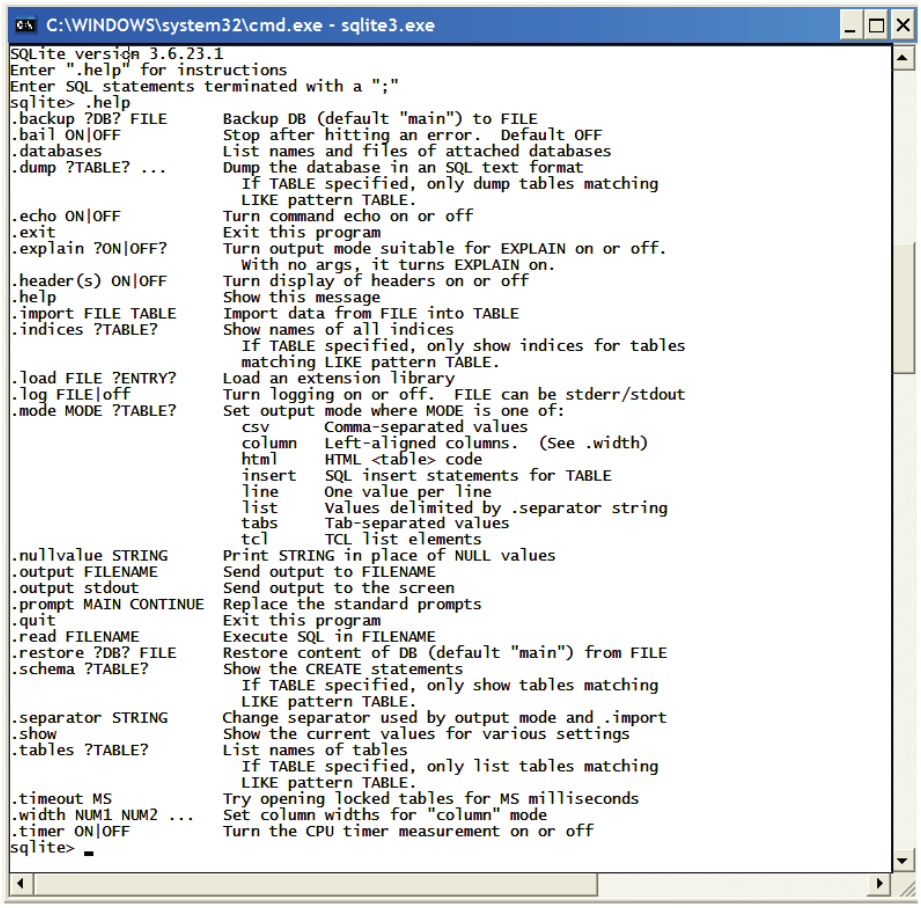


Рис. 3.1: шелл SQLite под Windows

3.2.2 Загрузка DLL

Точно также, как CLP.

3.3 Утилита CLP

Утилита CLP это наиболее общее средство для управления и работы с SQLite . Она может использоваться в интерактивного выполнения SQL операторов (как шелл) или в пакетном режиме.

3.3.1 Интерактивное использование CLP

Что бы использовать утилиту интерактивно (как шелл), наберите sqlite3 в окне Cmd, опционально можно указать название базы данных. Если не указать имени файла БД, SQLite будет использовать временную базу в оперативной памяти, содержание которой будет утеряно после выхода из CLP. Используя CLP как шелл Вы можете выполнять запросы, получать схему базы данных, импортировать и экспортировать данные, выполнять другие задачи администратора. Любое утверждение, которое не начинается на символ точки (.), воспринимается утилитой как запрос к базе данных. На символ точка (.) начинаются команды предназначенные исключительно для утилиты. Команда .help, например, приводит к появлению следующего экрана с подсказкой. 3.1 . Команды можно сокращать до одной буквы, то есть, вводить не .help, а .h, выйти из утилиты можно по команде .e - сокращение для .exit.

3.3.2 CLP в пакетном режиме

Можно использовать CLP в пакетном режиме, для таких задач как экспортирование/импортирование данных, выполнение пакетной обработки. Она идеальна для использования в скриптах шелла, чтобы автоматизировать администрирование БД. Что ознакомиться с предлагаемыми возможностями вызовите утилиту CLP с ключем -help, как показано ниже:

> sqlite3.exe -help

Ответ должен напоминать этот:

```
fuzzy@linux:/tmp$ sqlite3 -help
Usage: sqlite3 [OPTIONS] FILENAME [SQL]
FILENAME is the name of an SQLite database. A new database is created
if the file does not previously exist.
OPTIONS include:
   -help                show this message
   -init filename       read/process named file
   -echo                print commands before execution
   -[no]header          turn headers on or off
   -bail                stop after hitting an error
   -interactive         force interactive I/O
   -batch               force batch I/O
   -column              set output mode to 'column'
   -csv                 set output mode to 'csv'
   -html                set output mode to HTML
   -line                set output mode to 'line'
   -list                set output mode to 'list'
   -separator 'x'       set output field separator (|)
   -nullvalue 'text'    set text string for NULL values
   -version             show SQLite version   -init| filename       read/process named file
```

CLP в пакетном режиме принимает следующие аргументы:

- необязательный список ключей;
- имя файла БД;
- необязательная команда для выполнения.

Большая часть опций управляет форматированием вывода и только ключ init, за должно следовать имя файла с командами SQL для выполнения. Обязательным является имя файла БД. Заключительная команда является необязательной с небольшим предупреждением.

3.4 Администрирование

Наконец, похоже, утилита CLP может быть использована как интерактивно, так и пакетно. Взглянем на примеры использования утилиты для некоторых общих задач администратора. Начнем с создания файла базы данных.

3.4.1 Создаем файл базы данных

Создадим базу данных с именем test.db. Для этого в шелле cmd надо набрать следующее:

```
sqlite3 test.db
```

Несмотря на уже введенное имя файла, SQLite создат БД, только после того, как пользователь создаст в БД нечто вроде таблицы или обзора. До создания первой таблицы у пользователя есть возможность задать несколько постоянных параметров для БД. Такие постоянные параметры как размер страницы или таблица кодирования (UTF-8, UTF-16) не могут быть легко изменены после создания файла БД, поэтому до создания первой таблицы сохраняется возможность указать их. В этот раз продолжим работу с параметрами по умолчанию. Создаем таблицу следующей командой:

sqlite> create table test (id integer primary key, value text);

Теперь БД создана, называется test.db и содержит таблицу test. Таблица, как можно видеть, имеет два столбца:

- поле первичного ключа, называемая id, имеет тип integer primary key который дает возможность автоматически генерировать значения по умолчанию. То есть, если в операторе insert для такого поля не задано значение, то SQLite самостоятельно внесет в него следующее целое число.
- текстовое поле с названием value.

Добавим несколько записей в эту таблицу:

```
sqlite> insert into test (id, value) values(1, 'eenie');
sqlite> insert into test (id, value) values(2, 'meenie');
sqlite> insert into test (value) values('miny');
sqlite> insert into test (value) values('mo');
```

Теперь извлечем их:

```
sqlite> .mode column
sqlite> .headers on
sqlite> select * from test;
```

```
id          value
----------  ----------
1           eenie
2           meenie
3           miny
4           mo
```

Две команды, предшевствующие оператору select (.headers и .mode), были использованы чтобы немного улучшить форматирование результатов. На экране видны явно заданные значения для поля id, использованные в первых двух операторах. Далее видно, что SQLite внес последовательные значения 3 и 4, для записей в которых, поле id не было задано. Тут уместно будет упомянуть, что значение последнего сгенерированного значения можно получать при помощи функции last_insert_rowid():

```
sqlite> select last_insert_rowid();
```

```
last_insert_rowid()
-------------------
4
```

Перед выходом из утилиты, добавим индекс и обзор к БД. Для этого наберите следующие:

```
sqlite> create index test_idx on test (value);
sqlite> create view schema as select * from sqlite_master;
```

Для выхода из CLP наберите команду .exit:

```
sqlite> .exit
```

Под Windows прекратить работу CLP можно набрав комбинацию клавиш Ctrl+C.

3.4.2 Получение информации о внутренней схеме базы данных

Есть несколько команд для получения информации о содержимо БД. Можно получить список таблиц (равно как обзоров) используя команду .tables [pattern], где необязательный параметр [pattern] может быть регулярным выражением в смысле операции like языка SQL. В ответе окажутся таблицы и обзоры подходящие под регулярное выражение. Без регулярного выражение ответ будет содержать список всех таблиц и обзоров:

```
sqlite> .tables
```

```
schema test
```

В нашем ответе видно название таблицы test и обзора schema. Так же, список индексов для заданной таблицы можно получить набрав команду .indices [table name]:

```
sqlite> .indices test
```

```
test_idx
```

Ответ содержит название индекса test_idx, созданного ранее для таблицы test. Операторы языка определения данных (Data Definition Language) DDL которые использовались для создания таблицы или обзора можно получить используя команду .schema [table name]. Если имени таблицы не будет задано SQLite вернет определения всех объектов БД - таблиц, индексов, обзоров и триггеров:

```
sqlite> .schema test
```

```
CREATE TABLE test (id integer primary key, value text);
CREATE INDEX test_idx on test (value);
```

```
sqlite> .schema
```

```
CREATE TABLE test (id integer primary key, value text);
CREATE VIEW schema as select * from sqlite_master;
CREATE INDEX test_idx on test (value);
```

Более подробная информация о внутренней схеме БД может быть получена из главного системног обзора SQLite , который называется sqlite_master. Этот обзор является системным каталогом. Его поля описываются ниже.

- type - тип объекта (таблица, индекс, обзор, триггер);
- name - имя объекта;
- tbl_name - таблица, с которым ассоциирован объект;
- rootpage - индекс корневой страницы, с которой начинается объект;
- sql - определение объекта на языке DDL.

Опросив обзор sqlite_master для текущей тестовой базы, можно увидеть следующее (не забывайте использовать команды .mode и .headers, как показано выше, перед запросом):

```
sqlite> .mode column
sqlite> .headers on
sqlite> select type, name, tbl_name, sql from sqlite_master order by type;
```

```
type        name        tbl_name    sql
----------  ----------  ----------  -----------------------------------
index       test_idx    test        CREATE INDEX test_idx on test (value)
table       test        test        CREATE TABLE test (id integer primary
view        schema      schema      CREATE VIEW schema as select * from s
```

Ответ содержит полное описание всех объектов базы test.db: таблица, индекс, обзор и возле каждого есть соответствующий DDL - оператор.

Существует еще несколько дополнительных команд для получения информации о внутренней схеме БД: команда pragma, table_info, index_info и index_list. Они обсуждаются в разделе 5 .

Замечание

Не забывайте, что большинство утилит вроде SQLite CLP, хранит историю команд, введенных оператором. Что бы вернуть одну из предыдущих команд, надо нажать стрелку вверх несколько раз. Альтернативный доступ к истории дает нажатие клавиши F7.

3.4.3 Экспортирование данных

Команда утилиты CLP .dump позволяет экспортировать объекты базы данных в SQL формате. При отсутствии каких - либо аргументов .dump экспортирует все содержание базы данных как последовательность операторов языка DDL и операторов языка манипулирования данными DML (Data Manipulation Language). Этой последовательностью операторов можно пользоваться, для повторного создания объектов исходной базы данных, с таким же содержанием. Аргументы CLP трактует как имена таблиц или обзоров. Утилиты будет экспортировать таблицы или обзоры, соответствующие аргументам. Остальные - будут игнорироваться. В случае интерактивного использования вывод команды .dump по умолчанию направляется на экран. При необходимости сохранения его в файле, используйте команду .output [filename]. Данная команда перенаправит весь вывод в файл filename. Что бы восстановить вывод на экран надо набрать команду .output stdout. Таким образом, чтобы вывести содержимое рассматриваемой БД в файл file.sql, требуется выполнить следующее:

```
sqlite> .output file.sql
sqlite> .dump
sqlite> .output stdout
```

В текущей рабочей директории будет создан файл file.sql (если он уже не существовал ранее). Если файл с таким именем уже существовал, его содержимое будет перезаписано.

Комбинируя перенаправление вывода и различные опции форматирования (которые будут рассмотрены позже), можно управлять эспортом данных. Можно экспортировать различные подмножества таблиц и обзоров в различных форматах, с различными разделителями. Импортировать такие данные утилита CLP сможет по команде .import.

3.4.4 Импортирование данных

В зависимости от формата данных, приготовленных для импорта, импорт можно осуществить двумя путями. Если файл состоит из SQL операторов, то используется команда .read. Утилита выполнит операторы, содержащихся в файле. Если файл содержит значения разделенные запятой (или другим разделителем), так называемый CSV - формат, используется команда .import [file] [table]. Эта команда считывает строки из заданного файла file и пытается вставить их в указанную таблицу (импортирование таблицы). Для разделения строки файла на отдельные поля утилита использует символ, указанный в команде .separator, по умолчанию таким символом

является вертикальная черта (|). Естественно, число отдельных полей в строке должно совпадать с числом колонок в таблице. Команда .show показывает все значения, установленные для утилиты CLP, включая текущий сепаратор:

```
sqlite> .show
```

```
     echo: off
  explain: off
  headers: on
     mode: column
nullvalue: ""
   output: stdout
separator: "|"
    width:
```

Именно команда .read предоставляет возможность импортировать файлы, созданные командой .dump. В нижеследующимм примере удаляются два объекта БД (таблица *test* и обзор *schema*) и читается файл file.sql, записанный ранее командой .dump:

```
sqlite> drop table test;
sqlite> drop view schema;
sqlite> .read file.sql
```

3.4.5 Форматирование

Утилита CLP предоставляет несколько опций форматирования вывода (make your output neat and tidy - делают Ваш вывод аккуратным и опрятным). Наиболее простой командой является .echo, которая управляет повторением текста вводимой команды и .headers, которая включает имена полей в ответ. Текстовое представления значения $NULL$ - неизвестное значение - задается командой .nullvalue. Например, если требуется выводить значение $NULL$ как текстовую строку $NULL$, просто введите .nullvalue $NULL$. По умолчанию значение $NULL$ представляется пустой строкой.

Подсказка утилиты CLP меняется командой .prompt [value]:

```
sqlite> .prompt 'sqlite3> '
sqlite3>
```

Команда .mode помогает форматировать ответы на запросы. Она имеет опции csv, column, html, insert, line, list, tabs и tcl, каждая из которых полезна. Умолчательная опция - list. Например, режим вывода list оформляет результат ответа как поля, разделенные умолчательным сепаратором. То есть, для экспортирования таблицы в SCV формате требуется набрать:

```
sqlite3> .output file.csv

sqlite3> .separator ,
sqlite3> select * from test;
sqlite3> .output stdout
```

Файл file.csv будет содержать следующее:

```
1,eenie
2,meenie
3,miny
4,mo
```

На деле, такой же результат можно получить выбрав режим режим вывода CSV:

```
sqlite3> .output file.csv
sqlite3> .mode csv
sqlite3> select * from test;
sqlite3> .output stdout
```

Разница будет только в том, что во втором случае поля обрамляются двойными кавычками, а в первом - нет.

3.4.6 Эспортирование таблицы (Exporting Delimited Data)

Рассмотренные выше возможности для экспортирования, импортирования и форматирования данных позволяют экспортировать таблицы (как записи). Например, что бы экспортировать записи таблицы test, в которых поле value начинается на букву m в файл test.csv, проделайте следующее:

```
sqlite> .output text.csv
sqlite> .separator ,
sqlite> select * from test where value like 'm%';
sqlite> .output stdout
```

Если потребуется импортировать этот файл в таблицу со структурой похожей на структуру test (назовем её test2), проделайте следующее:

```
sqlite> create table test2(id integer primary key, value text);
sqlite> .import text.csv test2
```

3.4.7 Автоматизация обслуживания БД

До сих пор, утилита CLP использовалась интерактивно, то есть, как шелл, для создания баз данных и экспортирования данных. Однако, это скучно сидеть за компьютером, все время выполняя необходимые команды. Вместо этого, утилита может быть использована в пакетном режиме для выполнения команд CLP в командных файлах шелла Windows (бат файлы). Эти файлы можно использовать в шедулере операционной системы, который будет выполнять их по заданному графику.

Замечание

> Конечно, можно выполнять команды CLP интерактивно, но если есть последовательность команд, которую надо выполнять регулярно оказывается полезным использовать CLP в пакетном режиме.

Есть две возможности выполнения команд CLP в пакетном режиме. Любую команду SQL или утилиты CLP (такую, например, как .dump) можно задать в аргументах командной строки sqlite3.exe. CLP выполнит заданную команду, выведет результаты в стандартный вывод и прекратит работу. Например, что бы выполнить экспорт содержимого базы данных test.db, используя утилиту в пакетном режиме, наберите

```
sqlite3 test.db .dump
```

Следующая команда выведет содержимое стандартного вывода в файл test.sql:

```
sqlite3 test.db .dump > test.sql
```

После такой команды файл test.sql будет содержат вполне читабельные команды подъязыков DDL или DML для создания и наполнения базы данных test.db. Что бы извлечь все записи из таблицы test, выполните следующее:

```
sqlite3 test.db "select * from test"
```

Так же выполненить CLP в пакетном режиме можно читая команды со стандартного ввода. Например, чтобы создать новую базу данных с названием test2.db, используя дамп файл test.sql, наберите:

```
sqlite3 test2.db < test.sql
```

CLP выполнит все команды из файла test.sql, что бы создать еще одну копию тестовой базы данных в файле с именем test2.db.

Есть еще одна возможность - это передать имя файла с командами SQL как параметр аргумента -init:

```
sqlite3 -init test.sql test3.db
```

Утилита выполнит команды из файла, создаст еще одну копию test3.db и останется в интерактивном режиме. Почему? Текущий запрос не содержит ни одной команды из входного потока. Что бы обойти такую особенность нужно дописать любую команду в строку параметров. Например:

```
sqlite3 -init test.sql test3.db .exit
```

Команда .exit переведет утилиты в пакетный режим.

С таким же успехом, вместо команды утилиты .exit можно использовать пустой SQL оператор ; (символ точки с запятой).

3.4.8 Бекап базы данных

Есть несколько возможностей для выполнения бекапа. Бекап в виде SQL операторов является наиболее переносимой формой для хранения копий БД. Как показано выше, он выполняется командой .dump. Если утилита находится в интерактивном режиме, то выполняя следующую последовательность команд, можно добиться такого же результата:

```
sqlite> .output file.sql
sqlite> .dump
sqlite> .output stdout
sqlite> .exit
```

Наиболее легким способом выполнения обратной операции (то есть, импортирование) является чтение файла с дампом БД через стандартный ввод утилиты.

```
sqlite3 test.db < test.sql
```

Такая команда предполагает, что файла test.db в текущем каталоге не существует. Если он уже существует, то выполнение скрипта может пойти несколько неожиданным путем. Например, попытка создать уже существующую в БД таблицу, приведет к ошибке. Последующие операторы вставки записей вполне могут вызвать ошибки связанные с повторением значений первичных ключей (см. обсуждение опции PRAGMA в последующих секциях, опция может сделать поведение утилиты более приемлемым).

Еще одной возможностью для бекапа, является создание бинарной копии файла. Перед сохранением такой бинарной копии желательно выполнить упаковку файла. Она выполняется командой vacuum:

```
sqlite3 test.db vacuum
copy test.db test.backup
```

Еще надо помнить, что бинарные копии не являются такими же переносимыми, как SQL бекапы. Вообще говоря, SQLite не поддерживает совместимость файлов БД на различных платформах. Поэтому для долговременных бекапов выгоднее хранить SQL бекапы.

Замечание

> Не имеер значения сколько бекапов своей базы данных сделано, если восстановление из них не удается выполнить. Имееют значения только те, бекапы, из которых можно восстановить рабочую копию БД. Поэтому, всегда тестируйте восстановление из сделанного бекапа.

3.4.9 Получение информации о файле базы данных

Основным средстов для получения информации о логической структуре базы данных (внутренняя схема БД), является использование обзора sqlite_master, который содержит подробную информацию о всех объектах в текущей БД.

Если же требуется информация о физической структуре БД, необходимо придется SQLite Analyzer. Он может быть загружен с веб сайта производителя. SQLite Analyzer предоставит детальную техническую информацию о структуре файла БД, различную статистику о использовании диска, количестве таблиц, индексов, размере страниц файла, среднюю плотность использования страниц. В полном отчете содержатся объяснения всех терминов. Частичный отчет вывод утилиты sqlite_analyzer может выглядеть приблизительно так:

```
fuzzy@linux:/tmp$ sqlite3_analyzer test.db
```

```
/** Disk-Space Utilization Report For test.db
*** As of 2010-May-07 20:26:23

Page size in bytes.................... 1024
Pages in the whole file (measured).... 3
Pages in the whole file (calculated).. 3
Pages that store data................. 3          100.0%
Pages on the freelist (per header).... 0            0.0%
Pages on the freelist (calculated).... 0            0.0%
Pages of auto-vacuum overhead......... 0            0.0%
Number of tables in the database...... 2
Number of indices..................... 1
Number of named indices............... 1
Automatically generated indices....... 0
Size of the file in bytes............. 3072
Bytes of user payload stored.......... 26           0.85%

*** Page counts for all tables with their indices ********************

TEST.................................. 2           66.7%
SQLITE_MASTER......................... 1           33.3%


*** All tables and indices ***************************************

Percentage of total database.......... 100.0%
Number of entries..................... 11
Bytes of storage consumed............. 3072
Bytes of payload...................... 235          7.6%
Average payload per entry............. 21.36
Average unused bytes per entry........ 243.00
Maximum payload per entry............. 72
Entries that use overflow............. 0            0.0%
Primary pages used.................... 3
Overflow pages used................... 0


Total pages used...................... 3
Unused bytes on primary pages......... 2673        87.0%
Unused bytes on overflow pages........ 0
Unused bytes on all pages............. 2673        87.0%

*** Table TEST and all its indices **********************************

Percentage of total database..........  66.7%
Number of entries..................... 8
Bytes of storage consumed............. 2048
Bytes of payload...................... 60           2.9%
Average payload per entry............. 7.50
Average unused bytes per entry........ 243.00
Maximum payload per entry............. 10
Entries that use overflow............. 0            0.0%
Primary pages used.................... 2
Overflow pages used................... 0
Total pages used...................... 2
Unused bytes on primary pages......... 1944        94.9%
Unused bytes on overflow pages........ 0
Unused bytes on all pages............. 1944        94.9%
```

3.5 Дополнительные утилиты

Существует несколько бесплатных и коммерческих приложений. Из графических утилит можно выделить следующие:

- SQLite менеджер - http://www.sqliteman.com - не требует установки, приложение из нескольких файлов, наблюдается незначительные ошибки при выполнении команд;
- SQLite студия http://sqlitestudio.pl/ - один файл, не требует инсталляции, жуткое количество непонятных опций интерфейса, выглядит не очень;
- SQLite Эксперт Персонал - http://www.sqliteexpert.com/download.html - минимум инсталляции, хорошая функциональность;
- ODBC драйвер - http://www.ch-werner.de/sqliteodbc/sqliteodbc.exe - проверялся с Дельфи 7;
- ADO.Net провайдер - http://www.ch-werner.de/sqliteodbc/sqliteodbc.exe - проверялся с Дельфи 7;

3.6 Ado.Net провайдер

http://system.data.sqlite.org/index.html/doc/trunk/www/downloads.wiki;

3.6.1 Построение консольного приложения на C-Sharp

Текст программы:

---- File:./db/app.cs

```
//#define TEST

using System;
using System.Data;
using System.Data.Common;
using System.Threading;
using System.Data.SQLite;

struct Settings
{
  static public bool   dbg, verbose, tblFill;
  static  public  string Version ="0.05";
}


class application1
{

  static bool IsOneOf(string arg, params string[] vals)
  {
    string a = arg.Substring(1).ToLower();
    foreach(string v in vals) if(a == v) return true;
    return false;
  }
```

```
[STAThread]
static int Main(string[] args)
{
 string me_ = "Main";

 DateTime tm = DateTime.Now;
 for(int i = 0; i < args.Length; i++) {
   if(args[i][0] == '-' || args[i][0] == '/') {
     if(IsOneOf(args[i], "?", "h", "help")) {
       goto error;
     }
     else if(IsOneOf(args[i], "b", "batch"))  {
       Settings.tblFill = true;
     }
    else if (IsOneOf (args[i], "ver", "version"))  {
      Console.WriteLine("#" + me_ +
         "("+ Settings.Version +
         " for SQLite"
          +") ");
      return 0;
    }


   }
 }

//http://professorweb.ru/my/ADO_NET/base/level1/1_4.php

  using (SQLiteConnection cnn = new SQLiteConnection("Data Source=foods.db"))
  {
    cnn.Open();
    if (Settings.tblFill){        // использование DataSet
      DataSet ds = new DataSet();
    SQLiteDataAdapter da = new SQLiteDataAdapter( "select * from food_types", cnn);
      da.Fill(ds, "select");
      PrintTable(ds, "select",  true);
      ds.Dispose();
    }
    else {                        // использование датаридера
     SQLiteCommand cmd = new SQLiteCommand (cnn);
     cmd.CommandText = "select * from food_types";
     SQLiteDataReader  dr =  cmd.ExecuteReader();
     Console.WriteLine ("fields number: '{0}'", dr.FieldCount);

     if (dr.HasRows){
       while (dr.Read()){
         Console.WriteLine ("col1/name: '{0}'/'{2}', col2: '{1}', val column: '{3}'"
                  , dr.GetValue(0)
                  , dr.GetValue(1)
                  , dr.GetName(0)
                  , dr["id"]
                  );
       }
```

```
    }

    dr.Close();
   }
 }
 Console.WriteLine("\n after OnE {0}, total time {1}", DateTime.Now, DateTime.Now -tm);
  return 0;

error:
  string msg = "usage "+": app  [-b] -ver\n"
  +    "\t -b : to read all the table, opposite - record by record\n"
  +    "\t -ver : to print version\n";
     Console.WriteLine("sqllite example: \n {0}", msg);
 //   MessageBox.Show (msg, "sqllite example ");
  return 0;
}


public static void PrintTable(DataSet ds, string tbl,  bool verbose)
{
  string _me =  "::"+"PrintTable(DataSet ds, string tbl, bool verbose)";

  string sep = "; \t";

  if (true)
    Console.WriteLine("{0} is here. accuracy: {1}", _me,  verbose);
  if (ds == null)  return;
  DataTable iTbl = ds.Tables[tbl];
  if (iTbl == null)  {
    if (verbose)
      Console.WriteLine("#{0}: no such table: {1}", _me, tbl);
    return;
  }
  Console.WriteLine("#{0}", tbl);

  if(verbose) {
     for(int cc = 0; cc < iTbl.Columns.Count; cc++)
       Console.WriteLine("#{0}:{1}; ", cc, iTbl.Columns[cc].ColumnName.Trim());
  }

  for(int cr = 0; cr < iTbl.Rows.Count; cr++) {
   Console.Write( "#");
   for(int cc = 0; cc < iTbl.Columns.Count; cc++)
      Console.Write(  iTbl.Rows[cr][cc].ToString().Trim() + sep);

    Console.WriteLine();
  }
 }

}
```

---- End Of File:./db/app.cs

Пакетный файл для построения приложения:

---- File:./db/cs.cmd

```
echo off

SET N=app

echo on
call  params.%COMPUTERNAME%.cmd
echo off

SET FLGS=  /nologo /noconfig

rem
SET DEF= /define:TEST


rm  %1.exe .*
rm  "_.eRr._"

rem

ECHO ON
%X%\csc.exe %DEF% %S% %R% %FLGS%  /out:%1.exe   *.cs    >>"_.eRr._"

exit
```

---- End Of File:./db/cs.cmd

Динамически подгружаемые библиотеки и версия .NET:

---- File:./db/params.agp-x.cmd

```
SET X=c:\WINDOWS\Microsoft.NET\Framework\v4.0.30319
SET S=/r:%X%\System.Data.dll /r:%X%\System.dll  /r:%X%\System.Xml.dll /r:.\System.Data.SQLite.dll
```

---- End Of File:./db/params.agp-x.cmd

Результат работы приложения:

---- File:./db/.app.txt

```
fields number: '2'
col1/name: '1'/'id', col2: 'Bakery', val column: '1'
col1/name: '2'/'id', col2: 'Cereal', val column: '2'
col1/name: '3'/'id', col2: 'Chicken/Fowl', val column: '3'
col1/name: '4'/'id', col2: 'Condiments', val column: '4'
col1/name: '5'/'id', col2: 'Dairy', val column: '5'
col1/name: '6'/'id', col2: 'Dip', val column: '6'
col1/name: '7'/'id', col2: 'Drinks', val column: '7'
col1/name: '8'/'id', col2: 'Fruit', val column: '8'
```

col1/name: '9'/'id', col2: 'Junkfood', val column: '9'
col1/name: '10'/'id', col2: 'Meat', val column: '10'
col1/name: '11'/'id', col2: 'Rice/Pasta', val column: '11'
col1/name: '12'/'id', col2: 'Sandwiches', val column: '12'
col1/name: '13'/'id', col2: 'Seafood', val column: '13'
col1/name: '14'/'id', col2: 'Soup', val column: '14'
col1/name: '15'/'id', col2: 'Vegetables', val column: '15'

after OnE 23.02.2014 18:16:33, total time 00:00:00.3125000

---- End Of File:./db/.app.txt

Глава 4

ЯЗЫК SQL В SQLite

Эту главу можно рассматривать как введение в использование языка SQL для SQLite . Собственно, сам SQL занимает существенную долю обсуждений, посвященных базам данных и SQLite - не исключение. Информация из этой главы будет представлять интерес как для новичка, так и для опытного программиста. Её материал не должнет показаться слишком трудным, даже если Вы никогда ранее не использовали SQL. В текущей главе для обсуждения языка используется такой минимум понятий, чтобы вности необходимую ясность и избежать увязания в теории.

SQL является единственным и универсальным средством, позволяющим использовать реляционную базу данных. Это тягловая лошадь обработки информации. Язык спроектирован для структурирования, чтения, записи, сортирования, фильтрации, защиты, вычисления, генерации, группирования и общего управления информацией.

SQL является интуитивно понятным и дружелюбным языком. Он достаточно мощен и приятен в использовании. Достаточно забавным является наблюдение, что независимо от уровня профессионализма, любой человек, начав пользоваться языком SQL, продолжает искать новые возможности для решения своих задач.

В этой главе мы попробуем научить читателя как использовать SQL хорошо. Для этого продемонстрируем правильные техники и сопутствующие им хитрости. Как можно догадаться из оглавления, язык SQL для SQLite достаточно пространная тема, поэтому обсуждение разделено на две части. Суть оператора *select* излагается в первой части, во второй - обсуждаются остальные операторы. Закончив книгу, читатель будет готов работать с любой базой данных.

4.1 Пример базы данных

Перед изложением синтаксиса языка, познакомимся с базой данных для примеров. Используемая БД содержит названия блюд из каждого эпизода сериала Seinfeld. (If you've ever watched Seinfeld, you can't help but notice a slight preoccupation with food) Зритель этого сериала не может не ощутить

некоторую озабоченность едой. На протяжении ста восьмидесяти эпизодов упоминается более четырех сотен различных блюд. Это значит, что в каждом эпизоде появляется более двух новых блюд. Отняв время на рекламу, замечаем, что зритель знакомится с новым продуктом каждые десять минут. И такая озабоченность едой дает возможность сделать базу данных для демонстрации всех необходимых понятий языка SQL как такового и его диалекта для SQLite . На рисунке 4.1 показана схема БД.

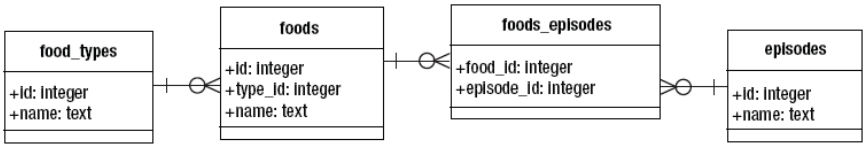


Рис. 4.1: Схема БД продуктов сериала

Далее следуют операторы SQL для создания внутренней схемы БД:

```
create table episodes (
  id integer primary key,
  season int,
  name text );

create table foods(
  id integer primary key,
  type_id integer,
  name text );

create table food_types(
  id integer primary key,
  name text );

create table foods_episodes(
  food_id integer,
  episode_id integer );
```

foods это основная таблица. Каждая её запись соответствует отдельному блюду, имя которого записывается в поле name. Поле *type_id* ссылается на таблицу *food_types*, в которой хранится классификация продуктов (то есть, фастфуд, напитки и так далее). Наконец, таблица *food_episodes* связывает продукты с сериями сериала.

4.1.1 Подготовка БД

В главе 1 объяснялось где взять скрипты для генерации БД. Распакуйте файл архива, в корневой директории архива найдите файл foods.sql, выполните следующую команду шелла:

```
sqlite3 foods.db < foods.sql
```

Как объяснялось в разделе 3.4 результатом команды будет создание файла БД с именем foods.db.

4.1.2 Выполнение примеров

Для удобства все примеры текущей главы собраны в файле с именем sql.sql, рядом с файлом foods.sql. Таким образом, можно не набирать пальцами

оператор, который хочется выполнить, а просто найти его в файле sql.sql.

Откройте указанный файл любимым текстовым редактором, скопируйте нужный оператор, сохраните его в отдельном файле, скажем, test.sql и выполните его из командной строки. Просто используйте тот же метод, который только что примерялся для создания файла базы данных по продуктам:

```
sqlite3 foods.db < test.sql
```

Результат будет выведен на экран. Так будет удобнее избегать бесконечного набирания и редактирования запросов, если захочется поэкспериментировать с ними. Делайте нужные изменение в Вашем редакторе, сохраняйте запрос в скрипте и выполняйте скрипт в шелле.

Следующие команды повысят читабельность вывода, их добавляют в начало скрипта:

```
.echo on
.mode column
.headers on
.nullvalue NULL
```

После таких команд утилита CLP

- повторяет текст введенной команды перед выполнением;
- печатает результат работы в аккуратные колонки;
- добавляет заголовки колонок;
- выводит отсутствующие значения как строку $NULL$.

Вывод всех примеров в этой главе форматирован с помощью этих команд. Есть еще полезная опция, которая задает ширину заданной колонки. Ширина меняется от примера к примеру.

```
sqlite> select *
   ...> from foods
   ...> where name='JujyFruit'
   ...> and type_id=9;

id          type_id     name
----------  ----------  ----------
244         9           JujyFruit
```

Иногда, как в предыдущем примере, будет добавляться лишняя строка между командой и выводом её результата, что бы улучшить читабельность. В случае сложных запросов, текст запроса будет отделяться серой линией от результата:

```
select f.name name, types.name type
from foods f
inner join (
  select *
  from food_types
  where id=6) types
on f.type_id=types.id;
```

name	type
Generic (as a meal)	Dip
Good Dip	Dip
Guacamole Dip	Dip
Hummus	Dip

4.2 Синтаксис

Синтаксис SQL декларативен и читается подобно естественным языкам. Утверждения выражаются в императивной форме, начиная с глагола, описывающего действие, далее следует субъект и предикат, как видно из рисунка 4.2

Заметно, что оператор читается как нормальное предложение. SQL был задуман, как легкий и понятный язык для работы непрограммистов.

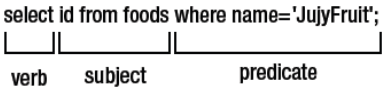


Рис. 4.2: Общий синтаксис языка SQL

Большая часть задумок была взята из декларативных языков, разработанных как альтернатива императивным языкам программирования, таких как C или Perl. Декларативными языками называются языки в которых описывается ЧТО требуется, а в императивных языках задаются КАК нужно получить результат. Например, задумайтесь над процессом заказа чисбургера. Обычно используется декларативный язык, чтобы выдать заказ. То есть, клиент объявляет официанту что он хочет:

Give me a double meat Whataburger with jalapenos and cheese, hold the mayo.

Дайте двойной мясной гамбугрег с перцом-халапеньо и сыром, майонез не надо.

Заказ передается на кухню, где повар готовит заказ, выполняя следующую программу записаную на императивном языке рецепта:

- взять говядину из третьего холодильника слева;
- сделать первую часть;
- жарить 3 минуты;

- перевернуть;
- жарить еще 3 минуты;
- повторить перечисленные шаги для второй части;
- положить горчицу на верхнюю булочку;
- положить обе пожаренные части на нижнюю булочку;
- положить сыр, салат, помидоры, лук и халапень (jalapenos) на булочку;
- добавляет заголовки колонок;
- сложить обе булочки и завернуть в желтую бумагу.

Заметно, что декларативные языки более краткие. В этом примере, декларативный язык заказа бургеров (DBL) за одно предложение материализует чисбургер, в то время как императивныому языку (ICL) требуется 10 шагов. Декларативные языки делают больше работы за меньше усилий. На деле, данный заказ записать на SQL не намного сложнее. Подходящий эквивалент заказа на SQL для нашего воображаемого DBL утверждения выглядит как то так:

```
select burger
from kitchen
where patties=2
and toppings='jalopenos'
and condiment != 'mayo'
limit 1;
```

Достаточно очевидно. Как уже отмечалось планировалось, что SQL будет дружелюбным языком. В давние дни предполагалось, что его будут использовать непрограммисты для выполнения разовых запросов и генерации отчетов, несмотря на то, что сейчас им пользуются исключительно разработчики и администраторы баз данных.

4.2.1 Операторы

Тест на языке SQL представляет собой последовательность операторов. Операторы обычно разделяются символом точка-с-запятой, который отмечает конец оператора. Например, следующий текст состоит из трех операторов:

```
select id, name from foods;
insert into foods values (null, 'Whataburger');
delete from foods where id=413;
```

Замечание

Точка-с-запятой используется в SQL как разделитель операторов (command terminator). Она отмечает конец оператора, который можно выполнять. Разделитель операторов ассоциируется с интерактивными программами (интерпретаторами) спроектированными для немедленного выполнения запросов в БД. Некоторые РСУБД в качестве разделителя операторов используют слово *go*.

Операторы, в свою очередь, составляются из серии лексем. Лексемы могут быть константами, ключевыми словами, идентификаторами, выражениями или специальными символами. Лексемы разделяются пробельными символами (пробел, табуляция, символ новой строки).

4.2.2 Константы

Константами, так же называемые литералами, явно записываются величины. Выделяется три типа констант: строки, числа и двоичные значения. Строка - это одна или несколько символов в одинарных кавычках. Например:

```
'Jerry'
'Newman'
'JujyFruit'
```

Хотя, собственно, SQLite позволяет записывать строки в двойных кавычках тоже, настоятельно рекомендуется использовать только одинарные, для языка SQL стандартом является использование одинарных кавычек. Привыкайте пользоваться стандартом, это избавит от затруднений в случае других диалектов SQL. Если одинарная кавычка является частью строки, надо вместо одной набрать две подряд:

```
'Kenny''s chicken'
```

Числа можно записывать как целые, с десятичной точкой и научной нотации. Примеры:

```
-1
3.142
6.0221415E23
```

Двоичные константы записываются как пары шестнадцатиричных цифр (0-9A-F) в одинарных кавычках с лидирующим символом x. Примеры:

```
x'01'
X'0fff'
x'0F0EFF'
X'0f0effab'
```

4.2.3 Ключевые слова и идентификаторы

Слова, имеющие в SQL специальный смысл, называются ключевыми. В частности, к ним относятся: *select*, *update*, *insert*, *create*, *drop*, *begin* и так далее. Идентификаторы (имена) указывают на специальные объекты в базе данных, такие как таблицы или индексы. Ключевые слова зарезервированы и не могут быть использованы как идентификаторы. SQL не чувствителен к регистру относительно имен и ключевых слов. Следующие два оператора является эквивалентными:

```
SELECT * from foo;
SeLeCt * FrOm FOO;
```

А в строках, по умолчанию, регистр букв различается, таким образом величины 'Mike' и 'mike' считаются различными.

4.2.4 Комментарии

Комментарии в SQLite обозначаются двумя последовательными минусами (–), которые коментируют остаток строки. Многострочные комментарии, как в языке C, обозначаются парами символов (/* */). Взгляните на пример:

```
-- This is a comment on one line
/* This is a comment spanning
   two lines */
```

И снова можно заметить, что без существенных причин не стоит использовать многострочные комментарии.

4.3 Создание базы данных

Таблицы являются естественной стартовой точкой, что бы начать эксплуатацию SQL в SQLite . Таблица это базовая единица информации в реляционной базе данных. Все вращается вокруг таблиц, которые состоят из колонок и записей. Все понятия, которые требуется рассмотреть после введения таблиц, не могут быть изложены в паре аккуратных параграфов. На деле, на это придется потратить всю текущую главу. А сейчас рассмотрим небольшой обзор, достаточный для того, чтобы сделать первую таблицы, или отказаться от этой идеи вообще. Остальные части главы будут улучшать понимание природы таблиц.

4.3.1 Создание таблиц

Как и реляционная модель вообще, SQL включает несколько частей. Структурная часть, предназначенная для создания и удаления объектов базы данных. Традиционно она называется языком определения данных (data definition language - DDL). Далее, функциональная часть предназначена для выполнения операций над этими объектами (например, выборки данных и вычисления над ними). Эта часть языка называется языком манипулирования данными (data manipulation language - DML).

Создание таблиц относится к структурной части, к DDL.

```
create [temp] table table_name (column_definitions [, constraints]);
```

Оператор создает временную таблицу при использовании ключевое слово *temp* или *temporary*. Такие таблицы временные, они исчезнут после окончания текущей сессии, то есть, как так только пользователь отсоединится от базы данных (если он не удалит их вручную). Квадратные скобки возле слова *temp* означает, что это необязательная часть оператора. В дальнейшем, любой текст в квадратных скобках является необязательным. Символ вертикальной черты (|) означает альтернативу (аналог слова или). Таки образом, следующий текст:

```
create [temp|temporary] table … ;
```

означает возможность использования любого из ключевых слов: temp или

temporary. Результат применения любого из операторов будет тем же самым.

В противоположном случае, оператор *create table* создает базовую таблицу. Термин базовая таблица означает обозначение именованной, постоянной таблицы в базе данных. Вообще говоря, этот термин использоваться, что бы отличать таблицы созданные оператором *create table* от системных таблиц или других объектов, подобных таблицам (обзоров).

Абсолютный минимум информации, необходимый для создания таблицы содержит имя таблицы и имя колонки (поля). Имя таблицы задается текстом table_name, оно должно быть уникальным среди всех других имен объектов базы данных. Далее, текст *column_definitions* является списком разделенных запятыми определений колонок (полей). Любое определение поля состоит из имени, домена и списка ограничений целостности, относящихся к определяемому полю (column constraint). Ограничения тоже перечисляются через запятые. Домен или, иначе, тип поля является аналогом типа данных в языках программирования. Он задает какие именно значения можно хранить в данном поле.

В SQLite существует пять встроенных типов: *integer*, *real*, *text*, *blob*, *NULL*. Каждый из этих типов будет описан позже в следующей главе, в секции 'Классы памяти' 5.2.3 каждый из этих типов будет описан подробнее. А раздел 'Целостность данных' 5.2 посвящен описанию ограничений целостности.

Оператор создания таблицы *create table* можно дополнять списком дополнительных ограничений целостности, как в следующем примере:

```
create table contacts ( id integer primary key,
                        name text not null collate nocase,
                        phone text not null default 'UNKNOWN',
                        unique (name,phone) );
```

Тут объявлено, что поле *id* относится к типу целых и имеет ограничение *primary key*, так сложилось, что комбинация этого типа и ограничения целостности имеет специальной смысл в SQLite . Такие поля приобретают свойство автоприращения (*autoincrement*), позже это свойство будет описано подробно. Поле *name* - текстовое и имеет два ограничения: *not NULL* и *collate nocase*. Поле *phone* - тоже текстовое и два ограничения. Далее следует ограничение целостности, которое выражает требование не к отдельным полям, а ко всей записи. Тут требуется уникальность пары name и phone. Таблица не может содержать две записи, которые имеют одинаковые значения в этих полях.

Уже изложено достаточно много информации, что бы воспринять её всю сразу, но уже можно самостоятельно оценить сложность понятия таблица. Все изложенное в дальнейшем будет объясняться еще раз, а сейчас важно только понимание общего формата оператора *create table*.

4.3.2 Обновление таблиц

Стуктуру таблицы частично можно изменить оператором *alter table*. Версия этого оператора в SQLite может переименовать таблицу и/или добавить поле. Общая форма операторы выглядит так:

```
alter table table { rename to name | add column column_def }
```

Обратите внимание на новые символы - { и }. Эти скобки включают список нескольких опций, одна из которых обязательно должна появиться в операторе. В этом случае требуется обязательно написать либо *alter table rename ...* либо *alter table add column ...* То есть, необходимо либо переименовать таблицу, либо добавить поле в таблицу. Чтобы добавить переименовать таблицу просто напишите новое имя.

Если добавляется поле, то определение поля, обозначаемое символами *column_def*, записывается как в операторе создания таблицы. Определение поля включает имя поля, за которым следует домен и список ограничений целостности. Пример:

```
sqlite> alter table contacts
        add column email text not null default '' collate nocase;
sqlite> .schema contacts

create table contacts ( id integer primary key,
                        name text not null collate nocase,
                        phone text not null default 'UNKNOWN',
                        email text not null default '' collate nocase,
                        unique (name,phone) );
```

Что бы увидеть определение таблицы, находясь в шелле CLP, используйте команду шелла .schema, за которой идет имя таблицы.

Таблицы можно создавать при помощи результатов выполнения оператора *select*. В этом случае создается структура таблицы, а сама таблица наполняется данными. В разделе 'Вставка записей' 5.1.1 обсуждается конкретное использование такой версии оператора *create table*.

4.4 Запросы к базе данных

Все усилия и проектировщиков баз данных, и кодировщиков сконцентрированы на единственной цели: использовании данных. Именно для этого используется подъязык DML. При этом ядром языка манипуляции данными является оператор *select*. Этот оператор имеет честь быть единственным оператором для выполнения запросов к базе данных. При этом он является наиболее сложным оператором в SQL. Большая часть его возможностей есть следствием реляционной алгебры, он просто содержит большую часть её. Возможности и сложность оператора *select* обширны даже в такой упрощенной среды как SQLite . Но не стоит терять энтузиазм. Подход SQLite вполне логичен и, как у всех реляционных систем управления базами данных (РСУБД), основан на прочном теоретическом фундаменте реляционной алгебры отношений.

4.4.1 Операции реляционной алгебры

Полезно было бы поразмышлять что можно сделать при помощи оператора *select* и почему это можно сделать с теоретической точки зрения. В большинстве реализаций языка SQL, включая рассматриваемую, оператор

select реализует несколько операций реляционной алгебры, при помощи которых связываются, сравниваются и отбираются данные. Эти операции реляционной алгебры обычно делятся на три категории.

Основные операции

- ограничение - выборка
- проекция
- прямое произведение
- объединение
- разность
- переименование

Дополнительные операции

- пересечение
- естественное соединение
- присвоение

Расширенные операции

- обобщенная проекция
- внешнее левое соединение
- внешнее правое соединение
- полное внешнее соединение

Истоки основных реляционных операции (кроме переименования) кроются в теории множеств. Дополнительные операции добавлены для удобства, каждая из них является сокращением для часто используемых комбинаций основных операций. Например, пересечение можно представить как объединение двух множеств из которого удалили объединение двух разностей исходных множеств. Расширенные операции расширяют возможности фундаментальных и расширенных операций. Например, операция обобщенной проекции добавляет арифметические выражения и возможности группировния к операции основной проекции. Внешнее соединение расширяет возможности операции соединения и позволяет извлекать доплнительную информации из базы данных.

В стандарте ANSI SQL оператор *select* может выполнять каждую из операций реляционной алгебры. Они восходят к исходным реляционным операциям определенным Коддом в его работе по теории реляционных баз данных (исключение составляет операция divide). SQLite может выполнять почти все реляционные операции определенные в ANSI SQL, исключение

составляют только правое и полное внешнее соединение. Их можно заменить комбинациями других реляционных операций, так что их отсутствие не являет большим недостатком.

Все операции определены в терминах отношений, которые, вообще говоря, называются таблицами. Каждая из операций выполняется над отношениями и, в результате её применения опять получаем отношение. Это позволяет соединять операции в реляционных выражениях. Сложность реляционных выражений может быть произвольной. Например, выход одной операции *select* может быть входом другой операции *select*, как показано ниже:

```
select name from (select name, type_id from (select * from foods));
```

Тут выход самого внутреннего оператора *select* передается на вход следующему, выход которого в свою очередь передается на вход самому внешнему. Это все остается просто реляционным выражением. Любой, кто знаком с организацией конвейеров команд в Линуксе, Юниксе или Виндовс, оценит по достоинству такие возможности. Вывод каждой операции *select* можно использовать в качестве ввода всех остальных операций.

4.4.2 *select* и конвейер операций

Оператор *select* включает реляционные операции при помощи серии фраз (предложений - clause). Каждая фраза соответствует своей реляционной операции. В SQLite почти все фразы не обязательны. Пользователь SQLite может использовать только те, операции в которых он нуждается.

Самая общая форма *select* в SQLite , без некоторых отвлекающих подробностей, может быть представлена как

```
select [distinct] heading
from tables
where predicate
group by columns
having predicate
order by columns
limit count,offset;
```

Каждое ключевое слово - *from*, *where*, *having* и так далее - начинает отдельную фразу. Фраза начинается с ключевого слова, за которым следуют выделенные курсивом аргументы. Далее мы будем ссылаться на фразы просто используя это ключевое слово. Вместо текста 'фраза *where*' будет просто использовано слово *where*.

Лучший способ думать об операторе *select* - это представить его как конвейер, который обрабатывает отношения. На конвейере есть необязательные процессы, выполнение которых можно пропускать. Независимо от того, используются или не используются конкретные операции (процессы), конвейер всегда работает одинаково. На рисунке 4.3 можно посмотреть порядок выполнения.

Выполнение оператора *select* начинается с фразы *from*, которая принимает одно или более отношений и соединяет их в одно составное отношение и, затем, передает последовательной цепочке операций.

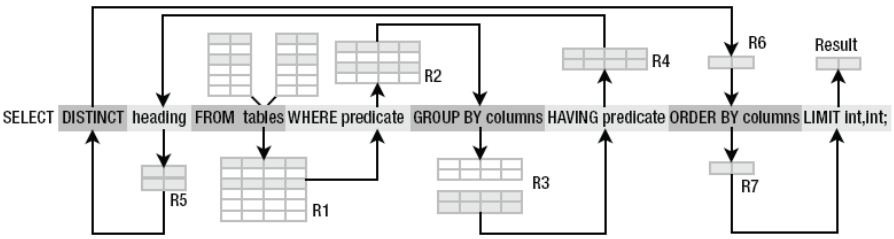


Рис. 4.3: Фразы оператора *select* в SQLite

Все фразы кроме самой операции *select* являются необязательными. Фраза *select* является обязательной. Далее, большинство операторов состоит из трех фраз: *select*, *from* и *where*. Это основной синтаксис и связанные с ним фразы выглядят как:

```
select heading from tables where predicate;
```

Фраза from содержит список нескольких таблиц, обзоров и подзапросов (представленных переменной *tables* на рис. 4.3), разделенных запятыми. Более чем одна таблица (обзор или подзапрос) будет соединяться в одно отношение, которое на том же рис. 4.3 представляется именем R1. Различные объекты в одно отношение соединяются операцией join. Результирующее отношение, которое производится фразой from является начальным материалом для дальнейшей работы. Все последующие операции будут работать либо с этим отношением, либо с теми, которые из него будут получаться.

Фраза *where* отбирает требуемые записи из R1. За ключевым словом *where* следует предикат, иначе логическое выражение, которое определяет критерий отбора записей из R1, которые должны быть включены в следующее отношение. Отобранные записи образуют новое отношение R2, как показано на рис. 4.3 выше.

В нашем примере R2 отношение передается по конвейеру операций практически неизменной, пока не достигает фразы *select*. Как отображено на рис. 4.4 Фраза отбирает столбцы отношения. Аргументом фразы является список названий полей или выражений, разделенных запятыми. Этот список и определяет результат, это называется список проекции.

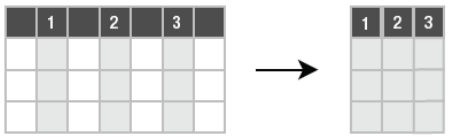


Рис. 4.4: Проекция

Далее, следует конкретный пример запроса из нашей базы данных:

```
sqlite> select id, name from food_types;
id          name
----------  ----------
1           Bakery
2           Cereal
3           Chicken/Fowl
4           Condiments
5           Dairy
6           Dip
7           Drinks
8           Fruit
9           Junkfood
10          Meat
11          Rice/Pasta
12          Sandwiches
13          Seafood
14          Soup
15          Vegetables
```

В этом запросе отсутствует фраза *where* для отбора записей, таким образом будут возвращаться все записи из таблицы $food\_types$. Во фразе *select* указаны все поля таблицы, а фраза $from$ не соединяет таблицы. Результатом будет точная копия таблицы $food\_types$. Как и в большинстве реализаций языка SQL, SQLite в качестве сокращения для выбора всех полей предлагает символ звездочку - (*). Значит, предыдущий запрос можно переписать более легким способом:

```
select * from food_types;
```

Как показывает рис. 4.5 фраза *select* в SQLite соединяет все данные, рассмотренные во фразе $from$, отбирает записи (ограничивает) их во фразе *where* и отбирает поля (проектирует) во фразе *select*.

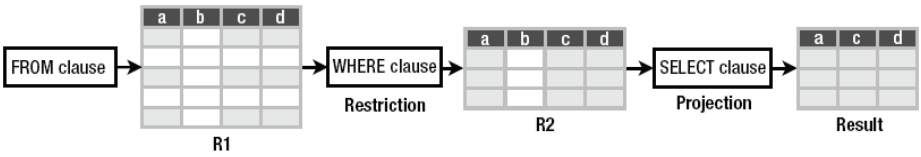


Рис. 4.5: Ограничение и проекция во фразе *select*

После этого простого примера, становится понятно, что языки запросов вообще, и язык SQL, в частности, в конечном счете оказываются реляционными операциями. За декларациями о просторе оказалась настоящая математика.

4.4.3 Выборка

Как оператор *select* является наиболее сложным из операторов SQL, так и фраза *where* оказалась наиболее сложной фразой оператора. Эта фраза выполняет большую часть работы. Ясное понимание его работы увеличит пользу от ежедневного использования SQL.

SQLite применяет фразу *where* к каждой записи отношения R1, выработанного фразой clause. Как замечалось выше, фраза *where* - выборка

- выполняет функции фильтра. Аргументом фразы является логическая функция - предикат. Предикат, в самом простом смысле, является просто суждением о чем - либо. Рассмотрем следующее предложение:

The dog (subject) is purple and has a toothy grin (predicate).

Фиолетовая собака с зубастым оскалом

Собака это подлежащее, а предикат состоит из двух суждений: её цвет фиолетовый и оскал зубастый. В зависимости от собаки, предложение может оказаться правильным или нет.

Во фразе *where* подлежащим оказывается запись из таблицы. Фраза *where* оказывается суждением (предикатом). Записи, для которых суждение оказывается, правдой попадают (выбираются) в результирующее отношение R2. Записи для которых суждение оказывается неправдой - исключаются. Таким образом, заявление о собаке можно рассматривать как эквивалент следующего оператора:

```
select * from dogs where color='purple' and grin='toothy';
```

В ответ на это, система управления базой данных берет записи из таблицы dogs (подлежащее) и применяет фразу *where* чтобы образовать логическое суждение:

```
This row has color='purple' and grin='toothy'.
```

В случае, если поле color в записи row содержит фиолетовый цвет, а поле grin признак зубатости, то строка включается в отношение - результат. Фраза *where* напоминает мощный фильтр. Он предоставляет замечательную степень контроля за процедурой включения (исключения) записей в (из) отношение(я) - результата.

4.4.3.1 Значения

Значения представляют собой данные из реального мира. Их разделяют на типы, такие как численные значения (1, 2 и так далее) и строки или строчные значения ("JujiFruit"). Значения можно записывать как литералы (которые явно представляют собой эти значения), переменные (обычно в виде полей таблицы вроде foods.name), выражения из литералов и переменных (3+2/5), выражения, в которых используются функции (count(foods.name)) - будут рассмотрены позже.

4.4.3.2 Операции

Операции принимают на вход одно или больше значений и вырабатывают значение, которое рассматривается в качестве выхода. Бинарные операции имеют два входных значения, тернарные операции имеют три, унарные - одно.

Операции можно записывать в одном выражении, при этом выход одной операции трактуется как вход другой (рис. 4.6).

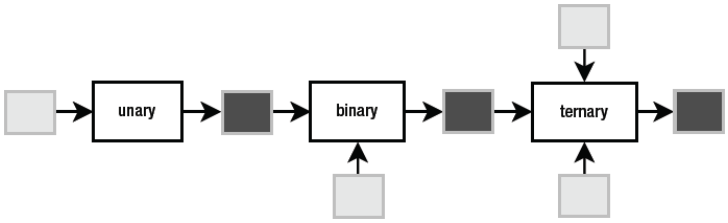


Рис. 4.6: Унарные, бинарные и тернарные операции могут образовавать конвейер

Используя вместе несколько операций, переменных и значений можно записывать выражения произвольной сложности. Например:

```
x = count(episodes.name)
y = count(foods.name)
z = y/x * 11
```

4.4.3.3 Бинарные операции

Самую большую группу операций в SQLite составляют бинарные операции. Таблица 4.1 содержит список бинарных операций, упорядоченных по приоритету, от высшего к низшему.

Таблица 4.1: Таблица бинарных операций

Операция	Действие
$\|\|$	конкатенация строк
$*$	умножение
$/$	деление
$+$	сложение
$-$	вычитание
$<<$	сдвиг битов вправо
$>>$	сдвиг битов влево
$\&$	логическое и
$\|$	логическое или
$<$	меньше
$<=$	меньше равно
$>$	больше
$>=$	больше равно
$=$	равно
$==$	равно
$<>$	не равно
$!=$	не равно
IN	принадлежит
AND	логическое и
OR	логическое или
IS	логическая эквивалентность
$LIKE$	подобие строк
$GLOB$	подобие имен файлов

Арифметические операции (например, сложение, вычитание, деление) это бинарные операции, которые из числовых величин получают числовую величину. Операции сравнения (например, меньше, больше, равно) бинарные операции, которые сравнивают величины или выражения и возвращают логические величины, которые только две: true и false. Операции сравнения образуют логические выражения, такие как:

```
x > 5
1 < 2
```

Логическим выражением называется выражение, которое возвращает логическое выражение. В SQLite 0 трактуется как $false$, а любое число отличное от 0 трактуется как $true$. Например:

```
sqlite> SELECT 1 > 2;

1 > 2
----------
0

sqlite> SELECT 1 < 2;

1 < 2
----------
1

sqlite> SELECT 1 = 2;

1 = 2
----------
0

sqlite> SELECT -1 AND 1;

-1 AND 1
----------
1
```

4.4.3.4 Логические операции

Логические операции (AND, OR, NOT, IN) это бинарные операции для вычислений на логических значениях и логических выражениях. Они вычисляют логическое выражение в зависимости от входных данных. Их можно использовать для более сложных логических выражений из простых, например, как ниже:

```
(x > 5) AND (x != 3)
(y < 2) OR (y > 4) AND NOT (y = 0)
(color='purple') AND (grin='toothy')
```

Логические операции выполняются согласно обычным правилам логики, но в SQL есть дополнительная загогулина, которая касается использования неопределенных значений. Эта особенность будет обсуждаться позже, сейчас её касаться не будем.

```
sqlite> select * from foods where name='JujyFruit' and type_id=9;

id          type_id     name
----------  ----------  ----------
244         9           JujyFruit
```

Ограничения этого примера выполняются согласно выражению ($name =' JujyFruit'$) and ($type\_id = 9$), которое состоит из двух логических выражений соединенных логическим и. Оба из условий должны выполнится для любой записи, выбранной в ответ.

4.4.3.5 Операция $LIKE$ и $GLOB$

Особенно полезна условная операция $like$. Операция подобна операции эквивалентности (=), но используется для подбора текстовых значений похожих на образец (или шаблон). Например, что бы выбрать все записи в таблице $foods$, у которых в поле name текст начинается на букву J, требуется использовать следующий оператор:

```
sqlite> select id, name from foods where name like 'J%';

id     name
-----  --------------------
156    Juice box
236    Juicy Fruit Gum
243    Jello with Bananas
244    JujyFruit
245    Junior Mints
370    Jambalaya
```

Символ процента (Символ подчерка (_) в образце означает любой одиночный символ (далее, спецсимволы). Алгоритм, подбирающий подстрочку для символа процента, является 'жадным', то есть, из нескольких подходящих вариантов выбирается масимально длинный.

```
sqlite> select id, name from foods where name like '%ac%P%';

id     name
-----  --------------------
127    Guacamole Dip
168    Peach Schnapps
198    Mackinaw Peaches
```

Так же можно использовать отрицание NOT с образцом:

```
sqlite> select id, name from foods
        where name like '%ac%P%' and name not like '%Sch%'

id     name
-----  --------------------
 38    Pie (Blackberry) Pie
127    Guacamole Dip
198    Mackinaw peaches
```

Поведение операции *glob* очень похоже на поведение *like*. Ключевое отличие заключается в том, что операция похожа на операции поиска файлов в операционных системах Unix или Linux. В операции в качестве спецсимволов используются символы звездочка (*) и подчерка (_). Следующий пример показывает использование операции *glob*:

```
sqlite> select id, name from foods
   ...> where name glob 'Pine*';

id          name
----------  ----------
205         Pineapple
258         Pineapple
```

SQLite поддерживает предикаты *match* и *regexp*, хотя в текущей версии у них нет собственной реализации. Чтобы их использовать требуется разработать реализацию с помощью функции $sqlite\_create\_function()$. Её использование обсуждается ниже, в соответствующей главе.

4.4.4 Ограничение и упорядочение

Можно указывать размер и границы подмножества записей из отношения - результата. Для этого используются ключевые слова *limit* и $offset$. *limit* задает максимальное число записей, которое может содержать отношение - результат. $offset$ задает количество записей, которое требуется пропустить и не включать в отношение - результат. Например, следующий оператор получает вторую запись из таблицы $food\_types$:

```
select * from food_types order by id limit 1 offset 1;
```

Фраза $offset$ пропускает первую запись, а фраза *limit* позволяет выбрать только одну следующую, то есть, вторую запись из отношения - результата.

что означает текст *order by*? Эта фраза приводит к сортировке результата - отношения по заданному (-ым) полю (-ям) перед окончательной выдачей. Это важно для нашего примера, так как нет никаких гарантий что есть определенный порядок записей в результате - отношении – стандарт SQL гарантирует отсутствие определенного порядка. Это означает, что при необходимости полагаться на какое либо упрядочение записей необходимо использовать фразу *order by*. Фраза записывается подобно фразе *select*: это список имен полей, разделенных запятыми. За именем каждого поля может следовать ключевое слово для указания возрастающего (*asc*) или убывающего (*desc*) порядка сортировки. Например:

```
sqlite> select * from foods where name like 'B%'
        order by type_id desc, name limit 10;

id     type_id   name
-----  --------  --------------------
382    15        Baked Beans
383    15        Baked Potato w/Sour
384    15        Big Salad
385    15        Broccoli
362    14        Bouillabaisse
328    12        BLT
327    12        Bacon Club (no turke
326    12        Bologna
329    12        Brisket Sandwich
274    10        Bacon
```

Представляется полезным, для сортировки отношения по нескольким полям, первым указывать поле, имеющее много повторяющихся значений. В примере поле type_id используется для группировки записей с последующим упорядочением по названию внутри таких групп.

Замечание

> Ключевые слова *limit* и $offset$ не входят в ANSI стандарт SQL. Многие СУБД имеют эквивалентные возможности, но используют другой синтаксис.

При желании использовать оба ограничения (и размер отношения - результата и смещение), можно использовать альтернативный способ записи, при котором не используется ключевое слово $offset$. Например, следующий оператор:

```
select * from foods where name like 'B%'
order by type_id desc, name limit 1 offset 2;
```

можно записать в следующей форме:

```
sqlite> select * from foods where name like 'B%'
        order by type_id desc, name limit 2,1;

id     type_id   name
-----  --------  --------------------
384    15        Big Salad
```

В случае использования краткого способа записи (без ключевого слова $offset$), смещение требуется ставить перед максимальным размером отношения - результата. В примере значения, следующие за ключевым словом $limit$, означают смещение 2 и максимальный размер 1. Заметьте, что можно использовать ключевое слово $limit$ без ключевого слова $offset$, но нельзя поступать наоборот.

Так же, заметьте, что эти ключевые слова выполняются последними на конвейере операций. Не надо ожидать, что использование $limit/offset$ ускорит получение ответа при помощи ограничения количества записей с которыми работает фраза $where$. Это не так. Для того что бы фраза $order\ by$ получила правильный результат, требуется получить все записи перед началом сортировки. Небольшое повышение производительности все таки есть, но не настолько заметное, как ожидают некоторые.

> There is a small performance boost, "in that SQLite only needs to keep track of the order of the 10 biggest values at any point". - текст в кавычках был опущен.

4.4.5 Функции и операции агрегирования

В SQLite есть некоторое количество встроенных функций и операций агрегирования, которые могут быть использованы в различных фразах. SQL содержит функции различного типа, как математические, такие как $abs()$, вычисляющие модуль числа, так и функции форматирования текста, вроде $upper()$ или $lower()$, которые заменяют буквы текста на заглавные или прописные. Например:

```
sqlite> select upper('hello newman'), length('hello newman'), abs(-12);

upper('hello newman')  length('hello newman') abs(-12)
---------------------  --------------------   ----------
HELLO NEWMAN           12                     12
```

Обратите внимание, что имена функций - не чувствительны к регистру. $upper()$ и $UPPER()$ означают одну и туже функцию. В качестве входных парметров функции могут принимать названия полей:

```
sqlite> select id, upper(name), length(name) from foods
        where type_id=1 limit 10;

id     upper(name)               length(name)
-----  ------------------------  ------------
1      BAGELS                    6
2      BAGELS, RAISIN            14
3      BAVARIAN CREAM PIE        18
4      BEAR CLAWS                10
5      BLACK AND WHITE COOKIES   23
6      BREAD (WITH NUTS)         17
7      BUTTERFINGERS             13
8      CARROT CAKE               11
9      CHIPS AHOY COOKIES        18
10     CHOCOLATE BOBKA           15
```

Их можно использовать в любых выражених и, значит, их можно использовать в выражениях фразы $where$:

```
sqlite> select id, upper(name), length(name) from foods
        where length(name) < 5 limit 5;

id     upper(name)          length(name)
-----  -------------------  -------------------
36     PIE                  3
48     BRAN                 4
56     KIX                  3
57     LIFE                 4
80     DUCK                 4
```

Операции агрегирования являются специальным подмножеством функций (групповые функции), которые вычисляют значения на группе записей. К стандартным групповым функциям относятся $sum()$, $avg()$, $count()$, $min()$ и $max()$. Например, что бы посчитать количество записей таблицы $foods, 1$(baked goods, type_id=1), count(), :

```
sqlite> select count(*) from foods where type_id=1;

count
-----
47
```

Операция *count* возвращает количество записей в отношении. Вообще говоря, каждый раз, когда используется операция агрегирования, надо думать: "для каждой записи в таблицы сделать то-то".

Групповые функции можно применять не только к именам полей, а и к любым выражениям – включая функции. Например, что бы посчитать среднюю длину поля с названием еды, нужно применить операцию avg к выражению $length(name)$:

```
sqlite> select avg(length(name)) from foods;

avg(length(name))
-----------------
12.58
```

Операции агрегирования используются во фразе *select*. Они вычисляют значения на записях, выбранных фразой *where*, но не на всех записях выбранных фразой $from$. Сначала производится выборка и, лишь затем, применяется операция агрегирования.

SQLite включает стандартный для языка SQL набор функций и групповых функцй, но хорошо бы помнить, что SQLite C API позволяет создавать пользовательские функции и операции агрегирования.

4.4.6 Группировки

С операциями агрегирования могут использоваться группировки. То есть, вместо вычисления, например, средних величин для всего отношения - результата, можно поделить отношение на группы записей и вычислять средние для каждой группы отдельно. Для этого используется фраза group by. Пример:

```
sqlite> select type_id from foods group by type_id;
```

```
type_id
----------
1
2
3
.
.
.
15
```

Фраза слегка отличается от остальных фраз оператора *select*. На конвейере операций её действия выполняются после фразы *where* и до фразы *select*. Отношение - результат после фразы *where* разделяется на группы записей, содержащих одинаковые значения в указанных полях. Далее, эти группы передаются фразе *select*. В рассмотренном примере существует 15 различных типов продуктов (поле type_id). После фразы group by в отношении - результате записи будут рассортированы по 15 группам записей, в зависимости от значения поля type_id. Далее, фраза *select* из каждой группы выбирает общее значение поля type_id и образует из него отдельную запись. Таким образом, в окончательном отношении - результате оказывается 15 записей из одного поля.

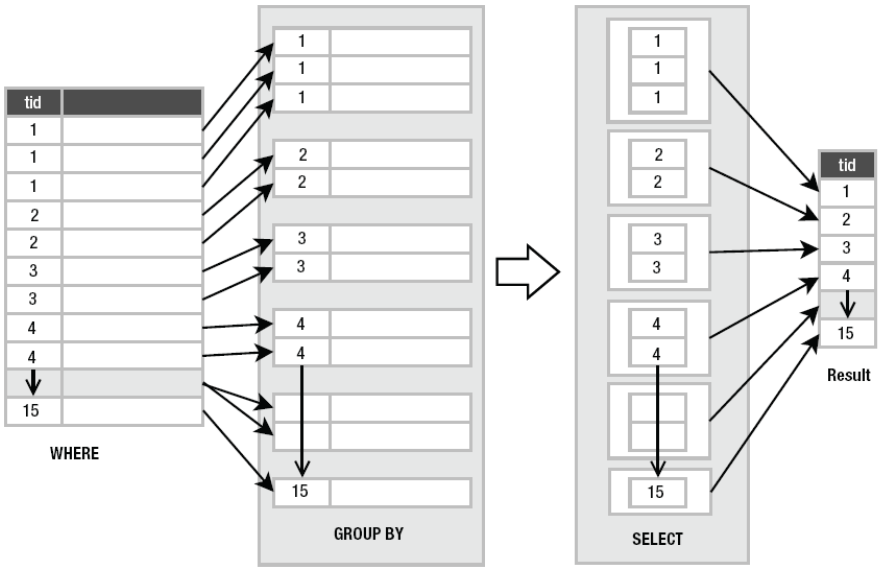


Рис. 4.7: Работа фразы group by

В случае, когда используется group by операции агрегирования применяются к каждой группе отдельно (собственно поэтому и называются групповыми функциями), при этом каждая группа, как бы, собираются в отдельную запись отношения-результата. Применим функцию count для подсчета количества записей в каждой группе предыдущего примера.

```
sqlite> select type_id, count(*) from foods group by type_id;

type_id     count(*)
----------  ----------
1           47
2           15
3           23

4           22
5           17
6           4
7           60
8           23
9           61
10          36
11          16
12          23
13          14
14          19
15          32
```

В последнем примере функция *count*() применяется 15 раз, один раз для каждой группы, как показано на рисунке 4.8 , причем на рисунке не отображено, собственно, правильное количество записей в каждой группе (например, не нарисовано, что группа с $type\_id = 1$ имеет 47 записей).

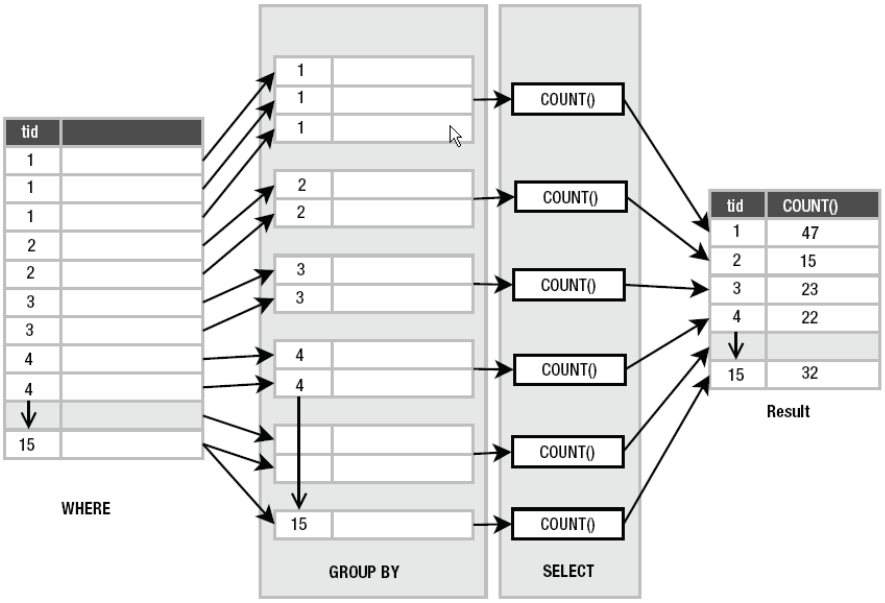


Рис. 4.8: Операции агрегирования и group by

Количество записей для группы $type\_id = 1$ (Baked Goods, хлебобулочные изделия) равно 47. Число записей в группе с $type\_id = 2$ (Cereal, крупы) – 15. Для следующей - $type\_id = 3$ (Chiken/Fowl, птицы) – 23, и так далее. Эту же информацию можно извлечь по-другому, выполнив пятнадцать запросов:

```
select count(*) from foods where type_id=1;
select count(*) from foods where type_id=2;
select count(*) from foods where type_id=3;
.
.
.
select count(*) from foods where type_id=15;
```

Или же, используюя *group by*, одним запросом:

```
select type_id, count(*) from foods group by type_id;
```

Осталось немного уточнить. Поскольку *group by* должен создавать группы записей, используя одинаковые значения в заданных полях, кажется логичным иметь возможность выборки перед окончательной передачей отношения фразе *select*. Этим занимается фраза *having*. Этот предикат можно применять к результату работы фразы *group by*. Фраза *having* работает с отношением - результатом фразы *group by*, точно так же, как фраза *where* выбирает записи из отношения - результата после фразы $from$. Разница заключается в том, что предикат *where* выполняется над значениями отдельных записей, а предикат *having* - над значениями групповых функций.

Попробуем из предыдущего примера извлечь группы, имеющие меньше чем 20 различных наименований продуктов:

```
sqlite> select type_id, count(*) from foods
        group by type_id having count(*) < 20;

type_id     count(*)
----------  ----------
2           15
5           17
6           4
11          16
13          14
14          19
```

Предикат $count(*) < 20$ применяется ко всем группам. Все группы, имеющие более 19 наименований продуктов не передаются фразе *select*. На рисунке 4.9 проиллюстрирован весь процесс:

Третий серый столбец на рисунке показывает группы записей, упорядоченные по полю *type_id*. Числа показывают значения поля *type_id*. На рисунке не показано настоящее число записей в каждой группе, а только по две, что бы дать возможность эту группу увидеть.

Значит, фразы *group by* и *having* предоставляют дополнительные возможности для выборок. *group by* делит результат - отношение фразы *where* на группы записей с общим значением в заданных полях. having применяет свой предикат, чтобы выбрать некоторые из получившихся групп. Выбранные группы, передаются фразе *select* для выполнения операции прекции.

Внимание

Некоторые СУБД, включая SQLite позволяют одновременно использовать в операторе *select* поля исходной таблицы и поля отношения - результата фразы *group by*. Например, в SQLite можно выполнить следующий оператор:

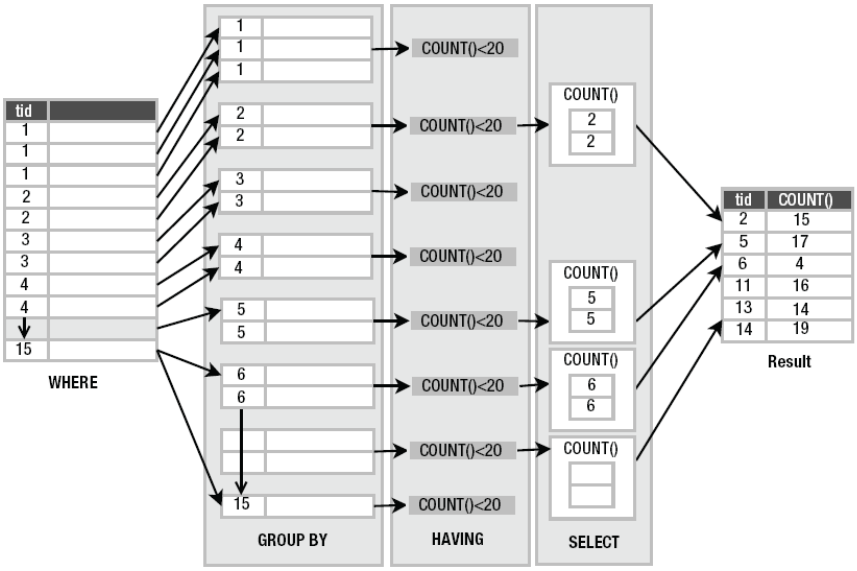


Рис. 4.9: Выборка над группами при помощи having

```
select type_id, count(*) from foods
```

Операция агрегирования *count* уменьшает (вследствие группировки) количество записей входной таблицы, что приводит к расхождению в количестве записей, передаваемых операции выборка. В этом случае *count* вернет одну запись, но нет указания для SQLite , что делать с полями *type_id*. Тем не менее, какой - то ответ будет получен. Как не печельно, но он не имеет смысла. Все поля фразы *select*, не используемые в операции агрегирования, необходимо перечислять во фразе *group by*. И только такие SQL - оператор нужно использовать.

4.4.7 Удаление повторяющихся записей

Фраза *distinct* удаляет все повторяющиеся записи из отношения - результата фразы *select*. Её можно использовать, что бы извлечь все неповторяющиеся значения поля *type_id* из таблицы продуктов *foods*:

```
sqlite> select distinct type_id from foods;

type_id
----------
1
2
3
.
.
.
15
```

В этом случае, конвейер операций выполнит следующее: фраза *where* вернет общее число записей таблицы foods (все 412 записей). После фраза

select в отношении - результате останется только поле type_id, и, наконец, distinct удалит все повторяющиеся записи, сокращая их число с 412 до 15 уникальных.

4.4.8 Соединение таблиц

Без операция соеднинения невозможно представить работу с несколькими таблицами (или отношениям), с неё начинает свою работу оператор *select*.

Посмотрим пример, чтобы понять соединение в SQLite . Таблица foods имеет поле type_id. Как можно выяснить из БД, значения в этом поле соответствуют значениям поля id в таблице food_types. Значит, между этими двумя таблицами существует определенная логическая связь. Для любого значения в поле foods.type_id должно найтись такое же значение в поле foods_types.id, которое называется первичным ключом (primary key) таблицы foods_types. Поле foods.type_id, вследствие связи между таблицами, называется внешним ключом (foreign key), оно содержит (или ссылается на) значения из первичного ключа в другой таблицы. Такая логическая связь называется отношением внешнего ключа.

Такая связь позволяет соединять таблицу foods и foods_types по этим двум полям, чтобы образовать новое отношение. Его можно дополнить новой информацией, например, добавить поле foods_types.name для каждого продукта в таблице foods. Следующий оператор SQL показывает как это делается:

```
sqlite> select foods.name, food_types.name
        from foods, food_types
        where foods.type_id=food_types.id limit 10;

name                      name
------------------------  ---------------
Bagels                    Bakery
Bagels, raisin            Bakery
Bavarian Cream Pie        Bakery
Bear Claws                Bakery
Black and White cookies   Bakery
Bread (with nuts)         Bakery
Butterfingers             Bakery
Carrot Cake               Bakery
Chips Ahoy Cookies        Bakery
Chocolate Bobka           Bakery
```

Как можно видеть из примера, за первым полем $foods.name$ следует поле $food\_types.name$. Как можно увидеть на рис. 4.10 каждая запись из таблицы $foods$ связана с записью с таблицей $food\_types$ при помощи связи $foods.type\_id \rightarrow foods\_types.id$:

Замечание

В нашем примере появился новый способ идентификации полей в операторе *select*. Так как в операторе *select* используется несколько таблиц, то приходится использовать нотацию имя_таблицы.имя_поля, вместо того, чтобы просто писать имя_поля. Пока имя поля уникально средит всех таблиц оператора *select*, то СУБД разберется из какой таблицы поле. В противном случае СУБД не сможет понять какое именно поле подразумевается

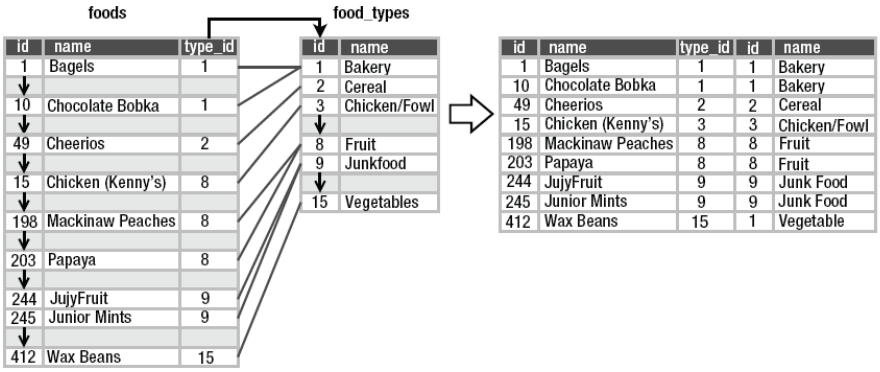


Рис. 4.10: Соединение таблиц foods и food_types

и вернет сообщение об ошибке. На практике, используйте обсуждаемую нотацию в любом случае, если соединяете таблицы. Смотрите раздел 4.4.9 для более подробной информации.

Чтобы соединить таблицы СУБД подберет такие соответствующие записи. Для каждой записи из первой таблицы СУБД найдет все записи из второй таблицы, которые имеют одинаковые значения в упомянутых полях и вернет полученное отношение. В текущем примере фраза from строит составное отношение, соединяя записи обеих таблиц.

Последующие фразы (*where*, group by и так далее) работают как и раньше. Изменения коснулись только начального соединения таблиц. Как будет ясно позже SQLite поддерживает шесть различных типов соединений. Только что обсужденное соединение называется внутренним и используется наиболее часто.

4.4.8.1 Внутреннее соединение

Внутреннее соединение между двумя таблицами имеет смысл использовать тогда, когда существует отношение внешнего ключа между двумя полями этих таблиц, как было показано в предыдущем примере. Это соединение является широко используемым (и, возможно, самым полезным) типом соединений.

Внутреннее соединение использует еще одну операцию над множествами, которая называется пересечение, что бы выбрать элементы, принадлежащими обеим множествам. Рисунок 4.11 иллюстрирует сказанное. Результатом пересечения множества $1, 2, 8, 9$ и множества $1, 3, 5, 8$ является множество $1, 8$ Рисунок представляет операцию пересечения при помощи диаграммы, показывающей общие элементы для обоих множеств.

Именно так выполняется внутреннее соединение, каждая запись из его результата имеет только те значения в общих (для обеих таблиц) полях, которые принадлежат обеим таблицам. Пусть левое множество рисунка

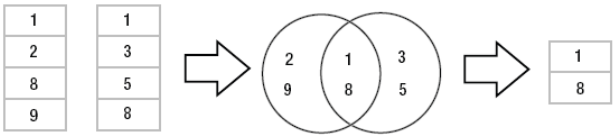


Рис. 4.11: Пересечение множеств

4.11 представляет значений поля $foods.type\_id$, а правое множество представляет значения поля $food\_type.id$. Получив значения этих полей, внутреннее соединение подбирает записи из обеих таблиц, содержащие одинаковые значения, и соединяет их в одну, чтобы получить таблицу - результат. Надо обратить внимание, что на рисунке 4.12 предполагается наличие по четыре записи в каждой таблице.

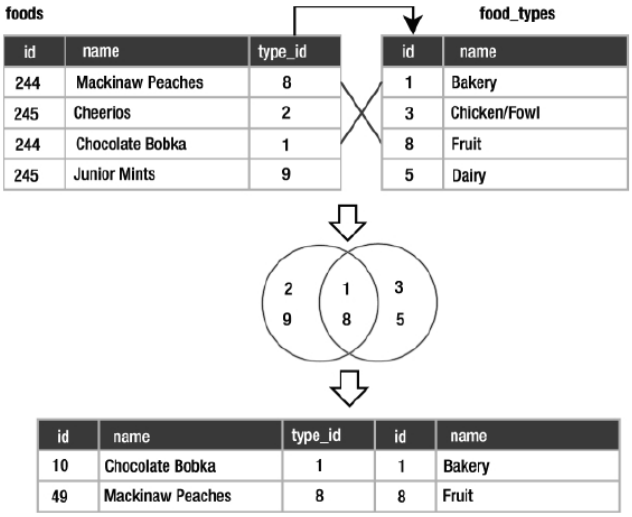


Рис. 4.12: Пересечение и соединение таблиц

Таблица-результат внутреннего соединения содежит записи, удовлетворяющие данному отношению полей. Результирующая таблица являются ответом на вопрос:
Какие записи таблицы B соответствуют записям таблицы A в данном отношении?
Покажем оператор языка SQL, который на практике выполняет наши теоретические рассуждения.

```
Select *
From foods inner join food_types on foods.id = food_types.id
```

Синтаксис оператора становится вполне понятным после просмотра содержимого таблицы - результата.

4.4.8.2 Прямое произведение

Представьте на мгновение, что между таблицами нет никакого соединяющего условия. Что можно сделать в таком случае? В случае, когда между

таблицами вообще не существует никаких связей, оператор *select* может выполнить более абстрактный вид соединнения, который называется прямым произведением (или декартовым произведением). Прямое произведение создает кажущиеся бессмысленными комбинации всех записей из первой таблицы и всех записей из второй.

На языке SQL прямое произведение таблицы $foods$ и $food\_types$ записывается следующим образом:

```
select * from foods, food_types;
```

По причине отсутствия чего либо еще, производится прямое произведение. Результат показан ниже. Каждая запись из таблицы $foods$ образует (и это важно) с _каждой_ записью таблицы $food\_types$ новую запись для результата. Нет никакой связи между исходными записями, нет никакого условия для соединения их, просто одна за другой перебираются все возможные комбинации, которые записываются вместе.

```
1           1           Bagels      1           Bakery
1           1           Bagels      2           Cereal
1           1           Bagels      3           Chicken/Fowl
…
(6174 rows omitted to save paper)
…
412         15          Wax Beans   13          Seafood
412         15          Wax Beans   14          Soup
412         15          Wax Beans   15          Vegetables
```

Можно спросить себя какова цель такого соединения. Вообще говоря, этот вопрос надо задавать каждый раз перед его использованием. Требуется ли на самом деле результат, состоящий из сочетаний каждой записи одной исходной таблицы с другой. Ответом на такой вопрос почти всегда является нет.

4.4.8.3 Внешнее соединение

Три из оставшихся четырех соединений называются внешними соединениями. Внутреннее соединение выбирает записи соединяемых таблиц соответствующие данному отношению. Внешнее соединение точно так же выбирает все записи, которые выбирает внутреннее соединение, и еще те, которые не удовлетворяют заданному отношению. Три внешних соединения называются левым, правым и полным. Левое внешнее соединение полностью выбирает все записи из "левой таблицы"оператора SQL. Например, в операторе:

```
select *
from foods left outer join foods_episodes on foods.id=foods_episodes.food_id;
```

левой таблицей является $foods$. Левое внешнее соединение (как и внутреннее соединение) для каждой записи таблицы $foods$ подберет
записи таблицы $foods\_episodes$ согласно условию соединения
($foods.id = foods\_episodes.food\_id$). Однако, если существуют некие названия блюд, которые никогда не появлялись ни в одном эпизоде сериала, то в $foods$ окажутся записи, которые не соответствуют записями $foods\_episodes$. Тем не менее, левое внешнее соединение все равно добавит эти строки в таблицу - результат, причем для них поля из таблицы $foods\_episodes$ будут содержать значение $NULL$.

4.4.8.4 Естественное соединение

Последнее соединение из списка операций называется натуральное соединение. Под этим термином скрывается знакомое внутреннее произведение, с немного измененным синтаксисом и дополнительным удобством. Натуральное соединение соединяет две таблицы при помощи общих имен полей. Таким образом при использовании естественного произведения получается внутреннее произведение тех же таблиц, без добавления условия соединения.

При естественном соединении в условие попадают все поля с одинаковыми названиями в обеих таблицах. Простое добавление или удаление полей в-из таблицы может коренным образом изменить результат запроса с естественным соединением. То есть, изменения в таблицах могут приводить к непредсказуемым результатам. Всегда лучше явно указывать условия соединения, полагаясь на смысл схемы таблиц.

4.4.8.5 Предпочтительный синтаксис

Синтаксис языка SQL предлагает много возможностей чтобы задать соединение. Пример внутреннее соединения таблиц $foods$ и $foods\_types$ иллюстрирует неявное выполнение соединения при помощи фразы $where$:

```
select * from foods, food_types where foods.id=food_types.food_id;
```

Как только РСУБД встречает список из более чем одной таблицы, она начинает выполнять соединение (по крайней мере, прямое произведение). фраза $where$ в этом примере приводит к выполнению внутреннего произведения.

Такая неявная форма оператора, хотя является вполне понятной, является устарелой. Желательно её избегать. Чтобы политически корректно записать оператор желательно использоват ключевое слово $join$. Общий вид оператора приводится ниже:

```
select heading from left_table join_type right_table on join_condition;
```

Такая явная форма оператора может быть использована для всех типов соединений. Например:

```
select * from foods inner join food_types on foods.id=food_types.food_id;
select * from foods left outer join food_types on foods.id=food_types.food_id;
select * from foods cross join food_types;
```

Некоторые типы соединений можно задать только при использовании явной формы оператора. Например, различные виды внешнего соедниения – левое, правое или полное. И это оказывается наиболее важной причиной для использования синтаксиса с ключевым словом $join$.

4.4.9 Имена и алиасы

Таблицы с одинаковыми именами полей приводят к неоднозначности в момент соединения. Если каждая из соединяемых таблиц имеют поле id, на которые ссылается фраза $selectid$, какое поле должен возвращать SQLite ?

Для явного указания требуется использовать имена полей вместе с именем таблицы, как уже писалось ранее.

Еще одна полезная возможность называется алиасы или синонимы. Если имя таблицы достаточно длинное и не хочется писать его перед каждым полем, то можно использовать синоним. Использование синонимов на деле является основной реляционной операцией, которая называется переименование. Операция переименования просто назначает новое имя таблице. Рассмотрим, например, следующий оператор:

```
select foods.name, food_types.name
from foods, food_types
where foods.type_id = food_types.id
limit 10;
```

Тут набирать приходится достаточно много. Но можно дать синоним во фразе $from$ при помощи включения нового имени непосредственно за именем таблицы, как ниже:

```
select f.name, t.name
from foods f, food_types t
where f.type_id = t.id
limit 10;
```

В примере таблице $foods$ дается синоним f, а таблице $food\_types$ дается синоним t. Далее любые ссылки на переименованные таблицы должны выполняться при помощи синонимов. Алиасы можно использовать при соединении таблицы с собой. Например, требуется выяснить какие продукты четвертого сезона упоминались в других сезонах. Тогда придется получить список всех эпизодов и список продуктов четвертого сезона при помощи соединения $episodes$ и $episodes\_foods$. Затем придется получать похожие списки для всех продуктов всех сезонов кроме четвертого. И, наконец, получить два списка, основываясь на общих продутах. Следующий запрос выполняет соединение таблицы с собой:

```
select f.name as food, e1.name, e1.season, e2.name, e2.season
from episodes e1, foods_episodes fe1, foods f,
     episodes e2, foods_episodes fe2
where
  -- Get foods in season 4
  (e1.id = fe1.episode_id and e1.season = 4) and fe1.food_id = f.id
  -- Link foods with all other episodes
  and (fe1.food_id = fe2.food_id)
  -- Link with their respective episodes and filter out e1's season
  and (fe2.episode_id = e2.id AND e2.season != e1.season)
order by f.name;
```

```
food                name             season  name                    season
------------------  ---------------  ------  ----------------------  ------
Bouillabaisse       The Shoes        4       The Stake Out           1
Decaf Cappuccino    The Pitch        4       The Good Samaritan      3
Decaf Cappuccino    The Ticket       4       The Good Samaritan      3
Egg Salad           The Trip 1       4       Male Unbonding          1
Egg Salad           The Trip 1       4       The Stock Tip           1
Mints               The Trip 1       4       The Cartoon             9
Snapple             The Virgin       4       The Abstinence          8
Tic Tacs            The Trip 1       4       The Merv Griffin Show   9
Tic Tacs            The Contest      4       The Merv Griffin Show   9
Tuna                The Trip 1       4       The Stall               5
Turkey Club         The Bubble Boy   4       The Soup                6
Turkey Club         The Bubble Boy   4       The Wizard              9
```

В оператор добавлены комментарии для лучшего понимания что происходит. В примере используется два соединения двух таблиц при помощи фразы *where*. Там есть по два экземпляра таблиц *episodes* и *foods_episodes*, которые трактуются как отдельные таблицы. Запрос соединяет *foods_episodes* саму с собоей, которые связываются с двумя экземплярами *episodes*. На этих двух экземплярах *episodes* выполняется отношение неравенства, чтобы гарантировать продукты из разных сезонов. Точно также можно давать синонимы именам полей и синонимы для выражений. Общий синтаксис для SQLite одинаков для всех фраз.

```
Select base-name [[as] alias] …
```

Использование ключевого слова *as* не обязательно, но много людей предпочитают его использовать, так как переименование становится более читабельным и и уменьшает вероятность перепутывания синонимов с именами таблиц и выражений.

4.4.10 Подзапросы

Подзапросом называется оператор *statement* внутри другого оператора *statement*. Еще один англоязычный синоним - *subselect*. Подзапросы используются достаточно широко. Везде, где могут быть записаны обычные выражения, можно подставлять подзапросы, то есть их можно использовать во множестве мест не только в *select*, но и в других операторах.

Фраза *where* наиболее часто используемым местом для применения подзапроса, особенно вместе с операцией *in*. Операндами бинарная операция *in* являются выражение и список величин. Она возвращает true если значение выражения встречается в списке или false - в противном случае. Например:

```
sqlite> select 1 in (1,2,3);
1
sqlite> select 2 in (3,4,5);
0
sqlite> select count(*)
   ...> from foods
   ...> where type_id in (1,2);
62
```

Используя подзапросы, можно переписать последний оператор в терминах названий продуктов (блюд) используя таблицу *food_types*:

```
sqlite> select count(*)
   ...> from foods
   ...> where type_id in
   ...>  (select id
   ...>   from food_types
   ...>   where name='Bakery' or name='Cereal');
62
```

Кроме этого, подзапросы во фразе *select* можно использовать для добавления дополнительных данных из другой таблицы к получаемому результату. Например, количество эпизодов в которых появлялся данный продукт, можно получить из таблицы *foods_episodes* выполняя подзапрос внутри фразы *select*:

```
sqlite> select name,
        (select count(id) from foods_episodes where food_id=f.id) count
        from foods f order by count desc limit 10;
```

```
name            count
----------      ----------
Hot Dog         5
Pizza           4
Ketchup         4
Kasha           4
Shrimp          3
Lobster         3
Turkey Sandwich 3
Turkey Club     3
Egg Salad       3
Tic Tacs        3
```

Фразы *order by* и *limit* используются в этом операторе, чтобы создать список из 10 блюд. Заметьте, что подзапрос ссылается на таблицу из внешнего оператора при помощи сравнения $food\_id = f.id$. Имя $f.id$ существует во внешнем запросе. Подзапрос в этом примера называтся коррелирующим подзапросом (соотнесенным подзапросом) так как, он ссылается (или коррелирует) на имя во внешнем запросе.

Подзапросы могут быть использованы во фразе *order by*. Следующий оператор группирует продукты-блюда по величине соответствующих групп по убыванию:

```
select * from foods f
order by (select count(type_id)
from foods where type_id=f.type_id) desc;
```

Фраза *order by* в этом случае не ссылается ни на какое конкретное поле в результате. Как тогда выполняется запрос в этом случае? Подзапрос во фразе *order by* выполняется для каждой записи и результат ассоциируется с данными записями. На подзапрос можно смотреть как на воображаемое поле в множестве - результате, которое используется для сортировки записей.

Наконец, рассмотрим фразу $from$. Может появиться необходимость использовать результаты запроса в операции соединения. Это иллюстрируется следующим оператором:

```
select f.name, types.name from foods f
inner join (select * from food_types where id=6) types
on f.type_id=types.id;
```

```
name                        name
--------------------------  -----
Generic (as a meal)         Dip
Good Dip                    Dip
Guacamole Dip               Dip
Hummus                      Dip
```

Надо заметить, что использование подзапроса во фразе $from$ требует обязательную операцию переименования. В этом случае подзапрос получил синоним *types*. Такой подзапрос, который создает отношение для фразы $from$ часто называется встроенным обзором или производной таблицей.

Подзапрос может быть использован везде, где может быть использовано реляционное выражение. Хороший способ для изучения как, где и когда

можно использовать подзапросы это просто пробовать их использовать и изучать результаты. В языке SQL очень часто решить какую либо задачу можно несколькими путями. Выбрать более эффективный способ решения можно позже, после понимания общей картины.

4.4.11 Составные запросы

Составным запросом называется запрос, который при помощи объединения, пересечения и разности обрабатывает результаты нескольких запросов. Для записи таких запросов в SQLite предназначены следующие ключевые слова: *union*, *intersect*, *except* соответственно. Необходимо помнить следующее:

- вовлеченные в запрос отношения должны иметь одинаковое количество полей;
- может быть только одна фраза *order by*, которая встречается в самом конце составного запроса и применяется ко всему результату.

Кроме того, отношения в составных запросах обрабатываются слева направо.

Операция *union* два различных отношения A и B соединяет в одно, содержащее различные записи из обоих. Фраза *union* в SQL объединяет результаты двух различных операторов *select*. По умолчанию, повторяющиеся записи удаляются. При необходимости иметь все записи в отношении - результате потребуется использовать фразу *union all*. Например, следующий оператор находит наиболее часто и наиболее редко упоминаемые блюда:

```
select f.*, top_foods.count from foods f
inner join
  (select food_id, count(food_id) as count from foods_episodes
     group by food_id
     order by count(food_id) desc limit 1) top_foods
  on f.id=top_foods.food_id
union
select f.*, bottom_foods.count from foods f
inner join
  (select food_id, count(food_id) as count from foods_episodes
     group by food_id
     order by count(food_id) limit 1) bottom_foods
  on f.id=bottom_foods.food_id
order by top_foods.count desc;
```

```
id     type_id  name       top_foods.count
-----  -------  ---------  ---------------
288    10       Hot Dog    5
1      1        Bagels     1
```

Оба запроса возвращают по одной записи. Разница заключается только в порядке сортировки результатов. Фраза *union* просто соединяет обе записи в одно отношение.

Два операнда есть у операции *intersect* - отношения A и B. Операция извлекает все записи из отношения A, которые принадлежат отношению B. Следующий пример показывает как при помощи фразы *intersect* можно получить первый десяток блюд, появлявшихся в 3, 4 и 5-ом сезонах.

```
select f.* from foods f
inner join
  (select food_id, count(food_id) as count
     from foods_episodes
     group by food_id
     order by count(food_id) desc limit 10) top_foods
  on f.id=top_foods.food_id

intersect
select f.* from foods f
  inner join foods_episodes fe on f.id = fe.food_id
  inner join episodes e on fe.episode_id = e.id
  where e.season between 3 and 5
order by f.name;
```

```
id     type_id  name
-----  -------  --------------------
4      1        Bear Claws
146    7        Decaf Cappuccino
153    7        Hennigen's
55     2        Kasha
94     4        Ketchup
164    7        Naya Water
317    11       Pizza
```

Первый оператор *select* использует фразу *order by*, что бы получить десятку наиболее часто появлявшихся блюд. Так как *order by* может появиться только в конце составного запроса, то его приходится оформлять как вложенный запрос внутреннего соединения для вычисления первой десятки наиболее используемых блюд. Сначала создается отношение с ними, а затем второй запрос возвращает отношение, содержащие блюда появившиеся в эпизодах с третьего по пятый включительно. Операция *intersect* из первой десятки удаляет записи, отсутствующие во втором запросе.

Операция *except* принимает два отношения A и B и находит все записи первого A, которые отсутствуют во втором - B. Заменяя фразу *intersect* на фразу *except* в предыдущем примере, можно получить какие блюда из первой десятки не появлялись в сезонах с третьего по пятый включительно:

```
select f.* from foods f
inner join
  (select food_id, count(food_id) as count from foods_episodes
     group by food_id
     order by count(food_id) desc limit 10) top_foods
  on f.id=top_foods.food_id

except
select f.* from foods f
  inner join foods_episodes fe on f.id = fe.food_id
  inner join episodes e on fe.episode_id = e.id
  where e.season between 3 and 5
order by f.name;
```

```
id     type_id  name
-----  -------  --------
192    8        Banana
133    7        Bosco
288    10       Hot Dog
```

Как уже упоминалось выше эта операция SQL соответствует операции разности в реляционной алгебре.

Составные запросы очень полезны, если требуется обработать похожие наборы данных несколькими способами. В случаях, когда не удается выра-

зить что либо меньше чем двумя операторами *select*, используются составные запросы для обработки двух или более полученных отношения.

4.4.12 Условные выражения

Выражение *case* позволяет обрабатывать различные условия внутри оператора *statement*. Существует две формы выражения. Первая и более простая принимает статическое значение и набор различных пар значение - возвращаемое значение в ветках ветвления:

```
case value
  when x then value_x
  when y then value_y
  when z then value_z
  else default_value
end
```

Далее, пример использования:

```
select name || case type_id
                 when 7  then ' is a drink'
                 when 8  then ' is a fruit'
                 when 9  then ' is junkfood'
                 when 13 then ' is seafood'
                 else null
               end description
from foods
where description is not null
order by name
limit 10;
```

```
description
---------------------------------------------
All Day Sucker is junkfood
Almond Joy is junkfood
Apple is a fruit
Apple Cider is a drink
Apple Pie is a fruit
Arabian Mocha Java (beans) is a drink
Avocado is a fruit
Banana is a fruit
Beaujolais is a drink
Beer is a drink
```

В этом примере выражение *case* получает различные значения поля *type_id* и подбирает строку, подходящую для каждого из значений. Полученная строка конкатенируется операцией (||) с названием блюда *name* образуя законченное предложение. Получаемое поле принимает имя *description*, что указывается после ключевого слова *end*. Для тех значений поля *type_id*, для которых не найдется подходящего условия *when*, выражение вернет значение $NULL$, что приводит к значению $NULL$ после конкатенации. Оператор *select* отфильтровывает $NULL$ значения во фразе *where*, таким образом, возвращаеются только те записи, которые должным образом обрабатываются выражением *expression*.

Вторая форма выражения *case* позволяет использовать выражения в ветках *when*. Общая форма:

```
case
  when condition1 then value1
  when condition2 then value2
  when condition3 then value3
  else default_value
end
```

Выражение работает подобно операциям агрегирования. Следующий оператор подбирает частоту упоминания блюд:

```
select name,(select
               case
                 when count(*) > 4 then 'Very High'
                 when count(*) = 4 then 'High'
                 when count(*) in (2,3) then 'Moderate'
                 else 'Low'
               end
             from foods_episodes
             where food_id=f.id) frequency
from foods f
where frequency like '%High'
```

```
name       frequency
---------  ----------
Kasha      High
Ketchup    High
Hot Dog    Very High
Pizza      High
```

Запрос выполнят подзапросы для каждой записи в таблице $foods$, так чтобы разделить блюда на группы по числу эпизодов, в которых они появлялись. Работа этих подзапросов образует поле с названием $frequency$. Затем фраза $where$ фильтрует значения в поле $frequency$, выбирая записи имеющие текст 'High'.

Выражение может вернуть результат работы только одной ветка $when$. Если более, чем одно условие во фразе $when$ может быть удовлетворено, выражение возвращает результат для первого из них. В случае, когда не подошло ни одно из условий, выражение $case$ вернет значение $NULL$.

4.4.13 Обработка значений $NULL$ в SQLite

Большинство реляционных баз данных используют понятие неизвестного значения, для которого заведено специальное ключевое слово $NULL$. На деле оно не является, собственно, значением и обозначает отсутствие какого либо значения. Используется чтобы обозначить места для отсутствующей информации, $NULL$ не означает ничего, $NULL$ не означает чтото, $NULL$ не есть $true$, $NULL$ не есть $false$, $NULL$ не является нулем, $NULL$ не является пустой строкой. $NULL$ означает исключительно самого себя, хотя некоторые могут не соглашаться с этим утверждением. Приведем несколько основных правил и идей, которые необходимо узнать, чтобы научиться пользоваться этим термином.

Первое, для использования значения $NULL$ в логических выражения, в SQL применяется трехзначная логика, при которой $NULL$ считается одним из логических значений. Следующая таблица показывает правила вычисления логических операций:

x	y	x AND y	x OR y
true	true	true	true
true	false	false	true
true	NULL	NULL	true
false	false	false	false
false	NULL	false	NULL
NULL	NULL	NULL	NULL

Попробуйте выполнить несколько простых операторов *select*, чтобы самостоятельно убедиться в вычислениях.

Второе, для определения наличия или отсутствия значения $NULL$ используются операции $isNULL$ и $isnotNULL$. Попытка использовать любую другую операцию, такую как равно, не равно, больше чем и так далее, может вызвать удивлению, которое подводит к третьему правилу.

Третье и последнее правило напоминает, что $NULL$ не равен ничему, даже самому себе. Нельзя сравнивать $NULL$ с любым значением, так как $NULL$ не может быть больше или меньше любого другого значения $NULL$. Непонимание правила часто подводит опрометчивых, так как операторы похожие на следующий пример не возвращают никаких записей:

```
select *
from mytable
where myvalue = null;
```

Ни одно из значений не может равняться значению $NULL$, таким образом, этот оператор SQLite не вернет ни одной строки. Теперь, поскольку основы уже освоены, можно познакомиться с дополнительными функциями работы с $NULL$, предоставляемые SQLite . Первая из них используется тогда, когда нужно отказаться от очень жесткого ограничения "ничего не может равняться значению $NULL$". Начиная с версии 3.6.19 в SQLite появился оператор *is*, который используется для сравнения одного значения $NULL$ с другим. Наиболее просто выполнить следующий оператор:

```
sqlite› select NULL is NULL;
1
```

Как уже упоминалось выше, любое значение отличное от нуля является правдой, то есть, значением *true*. Значит в SQLite считается что один экземпляр $NULL$ точно такой же, что и другой экземпляр. Но не следует слишком полагаться на эту особенность языка. Трехзначная логика SQL может быть и неудобна, но это стандарт, так что использование оператора *is* может повлечь проблемы связанные с использованием других систем и языков программирования.

Функция *coalesce* входит в стандарт SQL99, она принимает на вход список значений и возвращает первое не неизвестное значение в списке. Рассмотрим следующий пример:

```
select coalesce(null, 7, null, 4)
```

Из этого списка функция в качестве не непустого значения вернет 7. Функция очень полезна при вычислениях, с её помощью удобно вместо неизвестного значения подставлять что либо осмысленное, например 0.

Обратная функция называется *nullif*. Она принимает два параметра и возвращает неизвестное значения в случае если они эквивалентны и первый аргумент в противном случае:

```
sqlite> select nullif(1,1);
null
sqlite> select nullif(1,2);
1
```

При использовании неизвестного значения, требуется особоя тщательность в составлении запросов, ссылающихся на поля, в которых может появляться значение *NULL*. В противном случае, *NULL* может существенно изменить число записей в ответе.

4.5 Итого

Наконец, знакомство с оператором *select* в реализации SQLite закончено. Помимо знакомства с работой оператора немного обсуждалась реляционная теория. Приводились различные примеры использования оператора *select* для получения данных, соединения, агрегирования, объединения и так далее.

Дальнейшее обсуждение языка SQL будет продолжено в следующих разделах. В них познакомимся с остальными операторами языка манипулирования данными DML, равно как с операторами языка определения данных - DDL.

Глава 5

ПРОДОЛЖАЕМ ИЗУЧАТЬ SQL В SQLite

На протяжении главы 4 рассматривался оператор *select*, теперь пришло время чтобы познакомиться с остальными операторами текущего диалекта SQL. В настоящей главе обсуждаются операторы, которыми редактируются данные - *insert*, *update* и *delete*; ограничения целостности и более сложные темы, такие как создание таблиц и новых типов данных.

5.1 Изменение данных

По сравнению с оператором *select*, операторы для редактирования данных кажутся достаточно простыми для понимания и использования. Существует три оператора языка DML (Языка Манипулирования Данными) для редактирования - *insert*, *update* и *delete* - они вполне соответствуют своим названиям.

5.1.1 Вставка записей

Что бы вставлять записи в таблицу требуется использовать оператор *insert*. Этим оператором можно вставлять в единственную таблицу либо одну запись, либо несколько. Во втором случае придется использовать оператор *select*. Общая форма оператора *insert* следующая:

```
insert into table (column_list) values (value_list);
```

Текст *table* указывает имя таблицы (целевой таблицы), в которую вставляются записи. Текст *column_list* является списком имен полей, разделенных запятыми. Каждое из этих полей должно присутствовать в целевой таблице. Текст *value_list* является списком выражений, разделенных запятыми, которые соответствуют именам полей, упомянутых в *column_list*. Порядок выражений должен соответствовать порядку полей.

5.1.1.1 Вставка одной записи

Следующий оператор можно выполнить для вставки записи в таблицу *foods*:

```
sqlite> insert into foods (name, type_id) values ('Cinnamon Bobka', 1);
```

Этот оператор вставляет одну запись, указывая два значения. *'Cinnamon Bobka'* - первое значение в списке выражений - соответствует полю *name*, которое является первым полем в списке полей. Соответственно, значение 1 соответствует полю *type_id*, которое записано вторым. При этом поле *id* не упоминается вовсе, это означает, что для вставки записи СУБД будет использовать значение по умолчанию. Так как *id* объявлено как *integer primary key*, СУБД самостоятельно сгенерирует новое значение, чтобы вставить его в эту запись. В разделе 5.2.1 это будет обсуждаться подробнее. Результат выполнения оператора *insert* можно проверить при помощи следующего оператора *select*:

```
sqlite> select * from foods where name='Cinnamon Bobka';

id          type_id     name
----------  ----------  --------------
413         1           Cinnamon Bobka
```

Чтобы узнать, какое значение использовалось для поля *id*, можно выполнить следующие операторы:

```
sqlite> select max(id) from foods;

MAX(id)
----------
413

sqlite> select last_insert_rowid();

last_insert_rowid()
-------------------
413
```

Понятно, что значение 413, которое СУБД сгенероровало автоматически для поля *id*, является наибольшей величиной в этом поле таблицы. То есть, генерирует монотонно возрастающие числа. Как показано выше, функцию *last_inserted_rowid*(), можно использовать что бы увидеть, какое именно значение было использовано для последней вставке.

Если в списке выражений оператора *insert* есть выражения для каждого поля таблицы, то список полей может быть опущен. В этом случае СУБД предполагает, что список выражений соответствует порядку полей, в котором они перечислялись в оператора создания таблицы *create table*. Далее, пример:

```
sqlite> insert into foods values(NULL, 1, 'Blueberry Bobka');
sqlite> select * from foods where name like '%Bobka';

id          type_id     name
----------  ----------  ---------------
10          1           Chocolate Bobka
413         1           Cinnamon Bobka
414         1           Blueberry Bobka
```

Строка *'Blueberry Bobka'* появляется в операторе после числа 1, так как это соответствует порядку упоминания полей при создании таблицы. Что бы пересмотреть определение таблицы, просто наберите *.schema foods*:

```
sqlite> .schema foods
CREATE TABLE foods(
  id integer primary key,
  type_id integer,
  name text );
CREATE INDEX foods_name_idx on foods (name COLLATE NOCASE);
```

Первое поле - *id*, за которым следует *type_id*, за которым следует *name*. Это именно тот порядок, которого необходимо придерживаться, что бы вставлять записи в таблицу *foods*. Почему оператор *insert* начинается со значения *NULL*? SQLite знает, что поле *id* в таблице *foods* является полем с автоприращением, а значение *NULL* есть способом не указывать конкретное значение. Отсутствие значения вызовет автоматическое генерирование нового ключа. Это просто удобный способ записи. За этой хитростью не кроется никакого теоретического обоснования.

5.1.1.2 Вставка множества записей

В операторе *insert* можно использовать подзапросы двумя способами: как элемент списка выражений и как замену списка выражений. Есои подзапрос используется как замену списка, то в таблицу вставляется множество записей. Все записи, которые возвращает подзапрос вставляются в таблицу. Ниже приводится пример вставки множества из одной записи:

```
insert into foods
values (null,
       (select id from food_types where name='Bakery'),
       'Blackberry Bobka');
select * from foods where name like '%Bobka';
```

id	type_id	name
10	1	Chocolate Bobka
413	1	Cinnamon Bobka
414	1	Blueberry Bobka
415	1	Blackberry Bobka

В этом примере используется SQLite , вместо того, чтобы руками вводить правильное значение для поля *type_id*. Еще один пример:

```
insert into foods
select last_insert_rowid()+1, type_id, name from foods
where name='Chocolate Bobka';
select * from foods where name like '%Bobka';
```

id	type_id	name
10	1	Chocolate Bobka
413	1	Cinnamon Bobka
414	1	Blueberry Bobka
415	1	Blackberry Bobks
416	1	Chocolate Bobka

В этом операторе список выражений полностью заменен подзапросом *select*. Пока число полей в подзапросе *select* равняется числу полей в таблице (или числу полей в списке полей, если оператор *insert* его содержит), вставка будет работать отлично. В этом примере в таблицу добавляется еще одна булочка с шоколадом, для поля *id* используется выражение $last\_insert\_rowid() + 1$. Вместо выражения можно использовать *NULL*.

На практике это даже лучше, так как функция *last_insert_rowid*() может вернуть 0, если в текущей сессии вообще не выполнялось каких - либо вставок записей.

5.1.1.3 Опять о вставке множества записей

Нет никаких причин, из-за которых нельзя было бы использовать специальную форму оператора *insert* для вставки одним оператором SQL всего множества записей, извлекаемого подзапросом *select*. Если количество полей подзапроса совпадает с количеством полей таблицы, этот оператор вставит в неё каждую запись подзапроса. Пример:

```
sqlite> create table foods2 (id int, type_id int, name text);
sqlite> insert into foods2 select * from foods;
sqlite> select count(*) from foods2;

count(*)
--------------------
418
```

В примере создается новая таблица $foods2$, после чего в неё вставляются все записи из таблицы $foods$.

Но, собственно, сам оператор *create table* тоже имеет специальную форму для создания таблицы с помощью подзапроса *select*. Как видно из следующего, еще более простого примера:

```
sqlite> create table foods2 as select * from foods;
sqlite> select count(*) from list;

count(*)
--------
418
```

Такое оператор выполняет оба шага одним махом. Такая форма особенно полезна для создания временных таблиц:

```
create temp table list as
select f.name food, t.name name,
       (select count(episode_id)
        from foods_episodes where food_id=f.id) episodes
from foods f, food_types t
where f.type_id=t.id;
select * from list;
```

Food	Name	Episodes
Bagels	Bakery	1
Bagels, raisin	Bakery	2
Bavarian Cream Pie	Bakery	1
Bear Claws	Bakery	3
Black and White cook	Bakery	2
Bread (with nuts)	Bakery	1
Butterfingers	Bakery	1
Carrot Cake	Bakery	1
Chips Ahoy Cookies	Bakery	1
Chocolate Bobka	Bakery	1

Надо отметить, что при использовании этой сокращенной формы оператора *create table* все ограничения целостности исходной таблицы опускаются, их нет в новой таблице. В ней отсутствуют поля с автоприращением,

индексы, ограничения уникальности и так далее. Многие обозначают этот подход аббревиатурой CTAS, что означает Create Table As Select.

Так же стоит упомянуть, что требуется помнить про ограничения уникальности при вставке записей. SQLite пресекает попытки добавить записи с дублирующимися значениями в полях, отмеченных ограничением *unique*:

```
sqlite> select max(id) from foods;

max(id)
-------
416
sqlite> insert into foods values (416, 1, 'Chocolate Bobka');
SQL error: PRIMARY KEY must be unique
```

5.1.2 Обновление записей

Для редактирования записей используется оператор *update*. Этот оператор может менять одно или более полей одной или более записей в таблице. Общая форма оператора следующая:

```
update table set update_list where predicate;
```

Список обновления *update_list* содержит одну или более пар имя поле и выражение в форме $column\_value = value$. Фраза *where* работает точно также, как в операторе *select*. На деле половина оператора *update* является оператором *select*. Фраза *where* отбирает записи для редактирования. Затем список обновления будет применяться к этим записям. Далее пример:

```
update foods set name='CHOCOLATE BOBKA'
where name='Chocolate Bobka';
select * from foods where name like 'CHOCOLATE%';
```

id	type_	name
10	1	CHOCOLATE BOBKA
11	1	Chocolate Eclairs
12	1	Chocolate Cream Pie
222	9	Chocolates, box of
223	9	Chocolate Chip Mint
224	9	Chocolate Covered Cherries

Оператор *update* очень простой и понятный. Как и при применении оператора *insert* необходимо учитывать ограничения целостности. Попытка нарушить *unique* приведет к следующему результату:

```
sqlite> update foods set id=11 where name='CHOCOLATE BOBKA';
SQL error: PRIMARY KEY must be unique
```

5.1.3 Удаление записей

Для удаления записей используется оператор *delete*. Общая форма оператора следующая:

```
delete from table where predicate;
```

Синтаксически *delete* является упрощенным оператором *update*. Удаление фразы *set* из *update* приводит в точности к *delete*. Фраза *where* применяется точно как в операторах *select* и *update*. Отличие только в том, что отобранные записи удаляются. Далее пример:

```
delete from foods where name='CHOCOLATE BOBKA';
```

5.2 Целостность данных

Целостность данных - это понятие связанное с определением и выполнением некоторых взаимоотношений данных внутри и между таблицами. Выделяется четыре различных вида: доменная целостность, сущностная целостность, ссылочная целостность, целостность, задаваемая пользователем. При помощи доменной целостности контролируются значения в поле. Сущностная целостность используется для контроля записей в одной таблице. Ссылочная целостность используется для контроля записей в нескольких таблицах – под этим подразумевается отношение внешнего ключа. Все остальные ограничения целостности считаются целостностью, задаваемой пользователем.

Для определения требуемых взаимоотношений применяются ограничения целостности. Естественно, ограничения целостности ограничивают множество значений, которые могут храниться в данном поле или записи. СУБД может обеспечить выполнение всех четырех видов целостности просто отслеживая значения в полях таблицы. В SQLite , кроме того, ограничения целостности используются для разрешения конфликтов. Разрешение конфликтов будет обсуждаться позднее в этой главе.

Давайте ненадолго отвлечемся от нашей таблицы *foods* и рассмотрим таблицу *contacts*, которая создавалась следующим оператором:

```
create table contacts (
id integer primary key,
name text not null collate nocase,
phone text not null default 'UNKNOWN',
unique (name,phone) );
```

Как уже объяснялось, ограничения целостности широко используются при определении таблиц. Ограничения могут ассоциироваться с определении поля или вводится независимо, в теле определения таблицы. Ограничения полей используют такие ключевые слова как *not NULL*, *unique*, *primary key*, *foreign key*, *check* и *collate*. Ограничения для всей таблицы используют *primary key*, *unique* и *check*. Далее ограничения будут обсуждаются в контексте видов целостности.

Смысл многих ограничений должен быть интуитивно понятен, поскольку уже обсуждались операторы *update*, *insert* и *delete*. Ограничения целостности выполняют точно такие же дейстия над данными как и операторы изменения данных, чтобы гарантировать соблюдение требований к данным, определенных в таблице.

5.2.1 Целостность сущности

Теория реляционных БД – как она внедрена в большинство СУБД, включая SQLite , требует, чтобы система управления предоставляла однозначный доступ к любому полю любой записи БД. Отсюда следует предоставление однозначного доступа к любой соответствующей записи. Значит каждая

запись должна быть уникальна в некоторм смысле. Эта задача решается при помощи первичного ключа.

Первичный ключ состоит, по крайней мере, из одного поля или группы полей, которые удовлетворяют ограничению уникальности, которое,как можно будет вскоре убедится, просто означает что каждые отдельные значения в этом поле (или группы полей) должны быть различные. Это означает, что первичный ключ гарантирует, что каждая запись должна быть каким либо способом отличима от остальных записей в таблице, и это, в конечном счете означает, что каждое поле является адресуемым. Целостность сущности позволяет организовать данные в виде таблиц. И на последок, насколько нужны данные, если их нельзя найти?

5.2.1.1 Ограничение уникальности (unique)

Начнем с ограничения уникальности, поскольку первичные ключи основаны на нем. Это ограничение требует, что все величины в поле или (n-ка величин, которая является величиной) в группе полей должны быть уникальными. При попытке добавить дублирующее значение или попытке заменить некоторую величину на уже существующую в поле(-ях) СУБД должна сигнализировать о нарушении ограничения и прекращать выполнение операции. Ограничение уникальности может быть определено для поля или всей таблицы. Если ограничение задается для всей таблицы, ограничение уникальности может применяется к нескольким полям. В этом случае, каждая комбинация значений полей (n-ка) должна быть уникальной. В таблице *contacts* существует ограничение уникальности на комбинации полей *name* и *phone*. Посмотрим, что произойдет при попытке ввести еще одну запись для значения 'Jerry' в поле *name* и значения 'UNKNOWN' в поле *phone*:

```
sqlite> insert into contacts (name,phone) values ('Jerry','UNKNOWN');
SQL error: columns name, phone are not unique

sqlite> insert into contacts (name) values ('Jerry');
SQL error: columns name, phone are not unique

sqlite> insert into contacts (name,phone) values ('Jerry', '555-1212');
```

В первом операторе явно указаны поля *name* и *phone*. Значения в этих полях совпадают с значениями в уже существующей записи, таким образом, ограничение уникальности не позволяет СУБД выполнить оператор. Третий оператор показывает, что ограничение применяется к паре полей, а не к одному из них. Запись со значением 'Jerry' в поле *name* можно вставлять и это не приводит к ошибке, так как значения в этом поле не обязаны быть уникальными.

Уникальность и $NULL$

Сколько значений $NULL$ можно занести в поле, объявленное с помощью ограничения *unique*? Теоретические размышления, основанные на информации из раздела 4.4.13 , подсказывают, что правильный ответ – столько, сколько нужно. Достаточно вспомнить, что $NULL$ не равняется ничему, в том числе, другому такому значению $NULL$. И это правило выполняется в SQLite .

Такие СУБД как Oracle и PostgreSQL похожим образом решают этот вопрос. Но это мнение не является единственным в среде разработчиков баз данных. Например, пользователи Informix, Sybase и MS SQL могут иметь только одно такое значение в поле с ограничением уникальности. А DB2 вообще запрещает значения *NULL* в этих полях.

5.2.1.2 Ограничение первичного ключа

Независимо от того, определяется в первичный ключ или нет, SQLite всегда создает поле первичного ключа. Это поле является шестидесятичетырехбитным целым значением и называется *rowid*. Так же, на это поле можно ссылаться при помощи двух синонимов: *_rowia_* и *oid*. SQLite всегда автоматически генерирует правильные значения для этого поля.

SQLite предоставляет возможность автоприращения для первичных ключей. При объявлении поля таблицы как *integer primary key*, СУБД добавит умолчательное значение в этом поле, которое будет гарантированно уникальным. На деле, такое поле будет еще одним синонимом для *rowid*. Так как SQLite использует шестидесятичетырехбитные целые со знаком, то максимально значение для такого поля равно 9,223,372,036,854,775,807.

Даже если удасться достичь этой границы, SQLite просто начнет подбирать уникальные значения, которые еще отсутствуют в таблицы. Как следствие, получаем, что такие значения не обязаны быть упорядоченны в возрастающем порядке.

Внимание

Уже обсуждалось, что нельзя считать что данные в реляционных СУБД каким то образом упорядоченны. Это еще один повод никогда не полагаться на какой - либо определенный порядок в таблицах SQLite , даже если покажется, что он существует.

В рассмотренных выше примерах было вставлено два записи в таблицу *contscts*. Нужно заметить ни разу не указывалось значение для поля *id*. Как уже объяснялось, это стало возможным, так как *id* объявлено как *integer primary key*. Как можно убедиться, SQLite самостоятельно предоставит целочисленные значения для каждого оператора *insert*:

```
sqlite› select * from contacts;

id   name   phone
---  -----  --------
1    Jerry  UNKNOWN
2    Jerry  555-1212
```

При этом, в дополнению к полю *id*, доступны все формы синонимов:

```
sqlite› select rowid, oid,_rowid_,id, name, phone from contacts;

id  id  id  id  name   phone
--  --  --  --  ----   -----
1   1   1   1   Jerry  UNKNOWN
2   2   2   2   Jerry  555-1212
```

Есть еще одна возможность. После текста *integer primary key*, можно добавлять ключевое слово *autoincrement*. Это слегка изменит алгоритм генерации значений для такого поля. Если таблица создана с использованием ограничения целостности *autoincrement*, то SQLite будет запоминать максимальное значение *rowid* в системной таблице называемой *sqlite_sequence* и для новых вставок будут использоваться только значения большие, чем в *sqlite_sequence*. После достижения абсолюбного максимума, SQLite вернет ошибку SQLITE_FULL для следующего оператора *insert*.

```
sqlite> create table maxed_out(id integer primary key autoincrement, x text);
sqlite> insert into maxed_out values (9223372036854775807, 'last one');
sqlite> select * from sqlite_sequence;

name        seq
----------  -------------------
maxed_out   9223372036854775807

sqlite> insert into maxed_out values (null, 'will not work');
SQL error: database is full
```

Сначала в таблицу примера, в поле *id* вставляется максимальное шестидесятичетырехбитное целое число. Следующая вставка требует увеличения умолчательного значения на единицу. Происходит переполнение и оно становится равным 0. SQLite генерирует ошибку SQLITE_FULL.

Хотя SQLite и отслеживает максимальное значение полей с ограничением *autoincrement* в таблице *sqlite_sequence*, но позволяет вносить явно заданное значение в операторе *insert*. Естественно, такое значение должно оставаться уникальным:

```
sqlite> drop table maxed_out;
sqlite> create table maxed_out(id integer primary key autoincrement, x text);
sqlite> insert into maxed_out values(10, 'works');
sqlite> select * from sqlite_sequence;


name        seq
----------  ----------
maxed_out   10

sqlite> insert into maxed_out values(9, 'works');
sqlite> select * from sqlite_sequence;

name        seq
----------  ----------
maxed_out   10

sqlite> insert into maxed_out values (9, 'fails');
SQL error: PRIMARY KEY must be unique

sqlite> insert into maxed_out values (null, 'should be 11');
sqlite> select * from maxed_out;

id          x
----------  ------------
9           works
10          works
11          should be 11

sqlite> select * from sqlite_sequence;

name        seq
----------  ----------
maxed_out   11
```

В примере удаляется и вновь создается таблица *maxed_out*, затем в неё вставляется запись с явно заданным первичным ключем 10. Первая попытка вставить запись с первичным ключем равным 9 выполняется удачно. Вторая такая попытка завершается отказом, в связи с нарушением уникальности. Наконец, попытка вставить еще одну запись с умолчательным значением первичного ключа завершается успешно, значение первичного ключа для этой записи оказывается равным 11.

В заключение напомним, что при использовании ключевого слова *autoincremet* для описания поля первичного ключа SQLite останавливается при достижении максимального шестидесятичетырехбитного целого со знаком и прекращает генерацию следующих значений. Такая особенность требовалась для некоторых специфических приложений. Если она не нужна, то для полей с автоприращением лучше просто использовать *integer primary key*.

Подобно ограничению уникальности, ограничение первичного ключа можно задавать для нескольких полей. Если проектировщик БД для какой либо таблицы намеревается использовать составной первичный ключ, SQLite все равно будет поддерживать поле *rowid* для собственных целей, наряду с использованием заданного ограничения уникальности. Рассмотрим пример:

```
sqlite> create table pkey(x text, y text, primary key(x,y));
sqlite> insert into pkey values ('x','y');
sqlite> insert into pkey values ('x','x');
sqlite> select rowid, x, y from pkey;

rowid       x           y
----------  ----------  ----------
1           x           y
2           x           x

sqlite> insert into pkey values ('x','x');
SQL error: columns x, y are not unique
```

С точки зрения техники, первичный ключ здесь просто ограничение уникальности, заданное на двух полях, так как SQLite все равно поддерживает внутреннее поле *rowid*. Хотя многие эксперты проектирования БД призывают к использованию реальных полей для первичных ключей, представляет целесообразным делать это только тогда, когда это имеет смысл.

5.2.2 Доменная целостность

По-простому, доменная целостность означает, что значения поля таблицы соответствуют связанному с ней домену. То есть, каждая величина поля должна принадлежить домену, определяемому полем. Но, все таки, термин определен не достаточно точно. Домены часто сравниваются с типами в языках программирования, такими как строки или плавающие. И на деле, хотя это не самая плохая аналогия, доменная целостность существенно шире чем типы.

Доменные ограничения позволяют, начиная с простых типов (например, такие как целые), добавлять новые требованияя, уменьшающие множество допустимых значений. К примеру, сначала создается поле целого типа, после к нему добавляется требование, что приемлимыми для данного поля будут только три целых: -1. 0. 1. В этом случае изменяется (от всех целых до указанных трех) множество допустимых величин, но не меняется

тип данных как таковой. Тут приходится иметь дело с двумя понятиями: типом данных и множеством допустимых значений.

Рассмотрим еще один пример: поле *name* в таблице *contacts*. Оно было объявлено следующим образом:

```
name text not null collate nocase
```

Домен *text* определяет тип и первоначальную область допустимых значений. Все что идет далее, уточняет первоначальные требования и, следовательно, ограничивает область. Таким образом, поле *text* объявляется доменом всех текстовых величин, не содержащим значение NULL, в котором заглавные и строчные буквы не различаются. Значения в поле остаются текстовыми, над ними допустимы обычные текстовыми операции, но область допустимых значений меньше, чем множество всех текстов.

Можно утверждать, что домены поля не являются типами. Более того, домен является комбинацией типа и дополнительных ограничений. Типы поля определяют представления величин в поле и операции, которые над этими величинами можно выполнять – сортировку, поиск, сложение, вычитание и так далее. Дополнительные ограничения уточняют и ограничивают множество допустимых величин, которые предполагается хранить в поле, и обычно такое множество не совпадает с типом поля. Как можно видеть, множество величин в домене обычно меньше, чем множество величин типа, вследствие дополнительных ограничений. С практической точки зрения, можно считать, что домены поля являются типами с навешанными дополнительными ограничениями.

Поэтому, в доменной целостности есть две существенные стороны: проверка типа и проверка допустимости значений. Хотя в SQLite поддерживается много обычных ограничений доменной целостности (*not NULL*, *check* и так далее), но общий подход к проверке типа несколько отличается от других СУБД. На деле, подход, принятый в SQLite является одним из наиболее противоречивых, труднопонимаемых и спорных. Сейчас будут описываться основные черты, а более тонкие подробности будут обсуждаться в главе [1, гл.11].

ППрямо сейчас будут обсуждаться легкие вопросы: значения по умолчанию, ограничение *not NULL*, ограничение *check* и сортирующие последовательности.

5.2.2.1 Значения по умолчанию

Ключевое слово *default* задает умолчательное значение для поля, если никакого значения не было задано в операторе *insert*. *default* является ограничением в том смысле, что оно вводит заданное значение в поле в случае необходимости. Тем не менее, оно все равно считается ограничением целостности и включено в раздел "Доменная целостность так как помогает задать политику для обработки нулевых значений поля. В случае отсутствия умолчательного значения и значения в операторе *insert*, SQLite вставит в подразумеваемое поле значение *NULL*. Предположим, например, для поля *contacts.phone* задано умолчательное значение 'UNKNOWN', теперь рассмотрим следующий пример:

```
sqlite> insert into contacts (name) values ('Jerry');
sqlite> select * from contacts;

id          name        phone
----------  ----------  ----------
1           Jerry       UNKNOWN
```

В операторе *insert* задано значение для поля *name* и не задано для поля *phone*. Как можно видеть из примера, в результирующей строке появилось умолчательное значение 'UNKNOWN'. Без заданного умолчательного значения для поле *phone*, в нем оказалось бы значение $NULL$.

Стандарт ANSI/ISO позволяет использовать три ключевых слова для генерации умолчательной даты и времени вместе с $default$. $current\_time$ приводит к генерации локального текущего времени в формате HH:MM:SS (Стандарт ANSI/ISO-8601). $current\_date$ генерирует текущую дату в фомате YYYY-MM-DD, $current\_timestamp$ является комбинацией предыдущих двух. Посмотрим на пример:

```
create table times ( id int,
  date not null default current_date,
  time not null default current_time,
  timestamp not null default current_timestamp );
insert into times (id) values (1);
insert into times (id) values (2);
select * from times;
```

```
id  date        time        timestamp
---  ----------  ----------  -------------------
1    2010-06-15  23:30:25    2010-06-15 23:30:25
2    2010-06-15  23:30:40    2010-06-15 23:30:40
```

Эти три ключевые слова вполне полезны для использования в таблицах, в которых требуется журналировать события.

5.2.2.2 Ограничение *not NULL*

Если Вы относитесь к людям, которым не нравится значение $NULL$, тогда ограничение *not NULL* создано для Вас. Это ограничение гарантирует, что в такое поле никогда не попадет значение $NULL$. Ни оператор *insert* не может добавить $NULL$ в поле, ни оператор *update* не может заменить существующее значение на $NULL$. Зачастую можно видеть как *not NULL* поднимает свою уродливую голову (так было в первоисточнике :))) при выполнении оператора *insert*. А именно, ограничение *not NULL* без заданного умолчательного значения означает, что выполнить оператор *insert* без явно заданного значения выполнять нельзя (ибо таким значением является $NULL$). В предыдущем примере ограничение *not NULL* для поля *name* требует, что бы любой оператор *insert* явно задавал значения для указанного поля. Например:

```
sqlite> insert into contacts (phone) values ('555-1212');
SQL error: contacts.name may not be NULL
```

Этот оператор *insert* задает значение для поля *phone*, но не для поля *name*, поэтому его нельзя выполнить в связи с ограничением *not NULL*, заданным на поле *name*.

Прагматический подход для использования неизвестных значений и ограничения *not NULL* влечет ограничение *default*. В рассмотренных случаях оно использовалось для поля *phone*, кроме того, это поле имеет ограничение *not NULL*. При попытке выполнить оператор *insert* без явно заданного телефона значение из *default* не являющееся *NULL* и будет введено в запись. Люди часто используют эти два ограничения вместе, не допуская попадания значения *NULL* в поле.

5.2.2.3 Ограничение *check*

Это ограничение позволяет задать выражение, которое вычисляется после вставки записи или обновления поля. Значения поля, не удовлетворяющие условию заданному в выражении, СУБД воспримет как нарушение ограничения. Таким образом, это позволяет определять дополнительные ограничения целостности кроме рассмотренных *unique* и *NOTNULL*. В примере приводится ограничение, требующее не менее семи символов для номера телефона. Ограничение можно добавлять при определении поля *phone* или как отдельное ограничение в определении таблицы:

```
create table contacts
( id integer primary key,
name text not null collate nocase,
phone text not null default 'UNKNOWN',
unique (name,phone),
check (length(phone)>=7) );
```

Теперь попытки вставить слишком короткую строчку или укоротить существующее значение в поле *phone* будут вызывать нарушение ограничения целостности. Любое выражение, допустимое для фразы *where* (за исключением подзапросов), может использоваться в ограничении *check*. Пусть определена таблица *foo* как показано ниже:

```
create table foo
( x integer,
y integer check (y>x),
z integer check (z>abs(y)) );
```

В этой таблице каждое значение поля z должно быть больше, чем модуль значения в y, которое в свою очередь должно быть больше x. Попробуем выполнить следующие операторы:

```
insert into foo values (-2, -1, 2);
insert into foo values (-2, -1, 1);
SQL error: constraint failed

update foo set y=-3 where x=-3;
SQL error: constraint failed
```

Выражение из ограничения *check* вычисляется перед обновлением поля. Чтобы обновление поля завершилось, результат всех выражений должен быть *true*. Можно использовать триггера для проверки целостности данных. Причем триггера имеют больше возможностей. Если окажется невозможным записать нужную функциональность ограничением *check*, попробуйте ипользовать триггер. Они обсуждаются Позднее в этой главе, в разделе 5.2.6 они будут обсуждены.

5.2.2.4 Внешние ключи

SQLite поддерживает понятие такое понятие реляционной теории БД как ссылочная целостность

примечание

При определении четырх видов целостности выше, использовался термин "referencial integrity а сейчас – "relational integrity".

Как и большинство СУБД SQLite для этого использует механизм внешних ключей. Ссылочная целостность гарантирует, что ключи в одной таблице логически ссылаются на записи в другой таблице, то есть, записи в другой таблице реально существуют. Классический примеры для применения ссылочной целостности это связи дети - родители, связь между заказом и товарами в заказе, связь между эпизодами и упомянутыми блюдами.

SQLite для создания внешнего ключа предоставляет следующий синтаксис оператора *createtable* (несколько упрощен для легкого чтения):

```
create table table_name
( column_definition references foreign_table (column_name)
  on {delete|update} integrity_action
  [not] deferrable [initially {deferred|immediate}, ]
… );
```

Текст выглядит сложновато, но может быть разделен на три основные части. Используем БД блюд и напитков. В ней таблицы $foods$ и $food\_types$ создаются следующим образом:

```
CREATE TABLE food_types(
  id integer primary key,
  name text );

CREATE TABLE foods(
  id integer primary key,
  type_id integer,
  name text );
```

Известно, что каждая запись из $food\_types$ имеет первичный ключ id который её идентифицирует. Таблица $foods$ использует поле $type\_id$ для ссылок на таблицу $food\_types$. Если есть необходимость использовать ссылочную целостность, что бы любое значение поля $type\_id$ всегда определяло существующий тип блюда, то оператор *createtable* для таблицы $foods$ должен быть подобен следующему:

```
create table foods(
  id integer primary key,
  type_id integer references food_types(id)
  on delete restrict
  deferrable initially deferred,
  name text );
```

Отличия подчеркнуты жирным шрифтом и вполне понятны, если их рассматривать по очереди. Первая строка сообщает SQLite -ту, что поле $type\_id$ ссылается на поле id таблицы $food\_types$. Далее, переходим к фразе действия. В примере используется опция *on delete restrict*. Она запрещает удалять запись из таблицы $food\_types$ если существует хотя бы

запись в таблице $foods$, которая ссылается на ключ id удаляемой записи. $restrict$ это одно из пяти возможных действий, которые можно задавать. Рассмотрим их ниже:

set NULL

любое значение из детской таблицы устанавливается в $NULL$, если значение в родительской таблице удаляется.

set default

любое значение из детской таблицы устанавливается в умолчательное, если значение в родительской таблице удаляется.

cascade

Если изменяется значение ключа в родительской таблице, то должны быть так же изменены значения внешних ключей в детской таблице. Если удаляются записи в родительской таблице, то должны удаляться записи детской таблицы, ссылающиеся на удаляемые. Не стоит злоупотреблять этой опцией, так как результаты каскадного удаления могут неприятно удивить в неожиданный момент.

restrict

Не позволять удалять или изменять значения первичного ключа родительской таблицы, если на него ссылается детская.

no action

ничего не делать, кроме наблюдения за пролетающими изменениями. Ошибка возникнет в конце выполнения оператора или целой транзакции.

Третья строка, содержащая фразу $deferrable$, позволяет существовать нарушениям внешней целостности до завершения тразакции.

5.2.2.5 Сортирующие последовательности (*collation*)

Сортирующие последовательности задают способ сравнения текстов. Различные сортирующие последовательности приведут к различным результатом при сравнении. Например, одна сортирующая последовательность может быть нечувствительна к регистру, и значит строки $'JujyFruit'$ и $'JUJYFRUIT'$ будут неразличимы. А другая может быть чувствительна к регистру, в этом случае упомянутые строки будут считаться различными.

В SQLite есть три встроенные сортирующие последовательности. По умолчанию используется двоичная - $binary$, в этом случае тексты сравниваются по байтно при помощи функции $memcmp()$. Такой способ оказывается вполне удовлетворительным для многих западных языков вроде английского. Сортирующая последовательность $nocase$ не различает маленькие и большие буквы из латинского алфавита. Наконец, сортирующая последовательность $reverse$ является обратной для $binary$. Она редко применяется для целей отличных от целей тестирования и иллюстрации.

Через программный интерфейс языка Си SQLite предоставляет возможность создавать свои собственные сортирующие последовательности. Она позволяет разработчикам писать программы для языков, для которых не подходит умолчательная сортирующая последовательность. В главе 8 про это можно почитать больше.

Ключевое слово *collate* задает сортирующую последовательность для колонки. Например, для поля *contacts.name* задана сортирующая последовательность *nocase*, которая не различает большие и маленькие буквы.

Примечание

Напомним, что в разделе 1.1 рассказано как взять все примеры, рассматриваемые в документе. В частности, рассматриваемая таблица задается как

```
CREATE TABLE contacts
  ( id INTEGER PRIMARY KEY,
  name TEXT NOT NULL COLLATE NOCASE,
  phone TEXT NOT NULL DEFAULT 'UNKNOWN',
  UNIQUE (name,phone)
);
```

Значит, попытка добавить еще одну запись, содержащую в поле *name* строчку $'JERRY'$, а в поле *phone* - $'555-1212'$, должна закончиться неудачей:

```
sqlite> insert into contacts (name,phone) values ('JERRY','555-1212');
SQL error: columns name, phone are not unique
```

В соответствии с заданной сортирующей последовательностью строчка $'JERRY'$ не отличается от $'Jerry'$ и в таблице уже существуют запись с таким значением. Значит, новая запись дублирует уже существующую. По умолчанию, сортирующая последовательность SQLite чувствительна к регистру. Если специально не задать последовательность *nocase* для этого поля оператор из примера будет выполняться хорошо.

5.2.3 Классы памяти

Как уже упоминалось выше, SQLite с типами данных обращается не так, как другие СУБД. Отличия есть как в самих типах, так и в хранении, сравнении, принудительном приведении и присваивании. В этом разделе будут изложены основы классов памяти в SQLite , так чтобы получить хорошие практические знания. В главе 9 будут обсуждены большое количество внутренних тонкостей существенно неординарного и неожиданно гибкого способа использования типов данных, принятого в SQLite .

Пять примитивных типов данных, которые есть в SQLite будут называться классами памяти. Термин класс памяти ссылается на формат в котором данные хранятся на диске. Независимо от этого, используется синоним тип данных. Классы памяти описаны в следующей таблице

integer

используется для хранения целых чисел (положительных и отрицательных). Размер в байтах может меняться от 1 до 8 байт. Целые

изменяются от -9223372036854775808 до 9223372036854775807. Размер целого в байтах выбирается автоматически в зависимости от величины целого.

real

используется для хранения десятичных вещественных чисел. В SQLite для этого предназначен 8-байтовый double (в оригинале - float).

text

используется для текстовых данных. SQLite поддерживает различный кодировоки, включая UTF-8 и UTF-16(big и little endian). Длина максимальной строки в SQLite выбирается во время компиляции и во время выполнения, по умолчанию равна 1 000 000 000 байт.

blob

используется для хранения больших двоичных объектов (binary large object), то есть, данных любой структуры. Длина максимального блоба в SQLite выбирается во время компиляции и во время выполнения, по умолчанию равна 1 000 000 000 байт.

NULL

значение представляет собой отсутствующую информацию. SQLite предоставляет полную поддержку при обработке таких значений.

SQLite выводит тип значения из его представления по следующим правилам вывода:

- заключенному в одинарные или двойные кавычки значению в SQL операторе приписывается класс памяти *text*;
- последовательность цифр без десятичной точки и экспоненты будет относиться к целому классу памяти;
- последовательность чифир с десятичной точкой и/или экспонентой будет относиться к классу памяти *real*;
- последовательности символов *NULL* приписывается класс памяти *NULL*;
- последовательности шестнадцатиричных чисел, заключенных в кавычки, перед которыми находится символ x, независимо от регистра приписывается класс памяти *blob*.

Для определения класса памяти существует функция *typeof*(). Можно проиллюстрировать правила вывода на практике, с её помощью, как в следующем примере:

```
sqlite> select typeof(3.14), typeof('3.14'),
       typeof(314), typeof(x'3142'), typeof(NULL);

typeof(3.14)  typeof('3.14')  typeof(314)  typeof(x'3142')  typeof(NULL)
------------  --------------  -----------  ---------------  ------------
real          text            integer      blob             null
```

Используя указанное представление данных, примере показывает все пять классов памяти. Значение 3.14 выглядит как плавающие и, поэтому, относится к классу *real*. $'3.14'$ выглядит как текст и относится к классу *text* и так далее.

Поле в таблице SQLite может содержать значения принадлежащие к различным классам памяти. Это первое существенное отличие обработки данных принятой в SQLite . Узнав это, пользователи знакомые с другими СУБД могут от неожиданности сесть и спросить: "Чего-чего?"Просмотрим следующий пример:

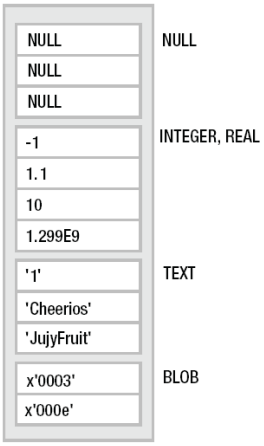
```
sqlite> drop table domain;
sqlite> create table domain(x);
sqlite> insert into domain values (3.142);
sqlite> insert into domain values ('3.142');
sqlite> insert into domain values (3142);
sqlite> insert into domain values (x'3142');
sqlite> insert into domain values (null);
sqlite> select rowid, x, typeof(x) from domain;

rowid       x           typeof(x)
----------  ----------  ----------
1           3.142       real
2           3.142       text
3           3142        integer
4           1B          blob
5           NULL        null
```

Пример влечет несколько вопросов. Каким образом величины в полях сортируются и сравниваются? Как сортируются поля с целыми, вещественными, текстовыми, блобами и отсутствующими значнеями? Как сравнивается целое и блоб? Что из них больше? Могут ли они быть одинаковыми?

Так уж получается, что значения в полях с различными классами памяти могут быть отсортированы. А отсортированы они могут быть, так как они оказываются сравнимы. Для этого в SQLite реализованы понятные правила. Классы памяти сортируются в соответствии с соответствующими значениями класса, которые задаются следующим образом:

- Класс памяти $NULL$ имеет самое низкое значение класса. Значение из класса памяти $NULL$ считается меньше, чем любое другое (включая другое значение с таким же классом памяти $NULL$). Между различными значениями $NULL$ не задается никакого порядка сортировки;
- Классы памяти *integer* и *real* считаются больше чем $NULL$ и имет одинаковое значение класса. Величины из *integer* и *real* сравниваются численно.
- Значения класса памяти *text* считается больше чем у *integer* и *real*. Любое значение из *integer* и *real* считается меньше чем любое значение из класса *text*. Для сравнения двух значения из класса *text* используется сравнение, задаваемое сортирующей последовательностью.
- Наивысшее значение имеет класс памяти *blob*. Любое значение из любого предыдущего класса памяти меньше чем значение из класса *blob*. Два различных значения из класса *blob* сравниваются при помощи функции Си *memcmp*().

Таким образом, когда SQLite сортирует таблицу по колонке, то сначала записи группируются по классам памяти - сначала $NULL$, затем *integer* и *real*, затем *text* и, наконец, *blob*. После чего, записи сортируются внутри каждой группы. Первая группа из $NULL$ значений не сортируется вообще. Числа сравниваются численно, строки согласно сортирующей последовательности, и блобы - побитово с помощью "memcmp()". Следующая фигура иллюстрирует сортировку воображаемой таблицы в возрастающем порядке:



Наверно, имеет смысл вернуться к этой секции еще раз, что бы попрактиковаться в использовании классов памяти SQLite , чтобы закрепить обсужденные аспекты SQLite . В главе 9 эта тема будет обсуждаться еще раз, и в ней можно будет углубиться в технические тонкости классов памяти, манифесты типизации, аффинити и остальные темы, родственные типам и классам памяти.

5.2.4 Обзоры

Обзоры (view) являются виртуальными таблицами. Их еще называются производными таблицами, в связи с тем, что их содержимое является результатом выполнения запросов к другим таблицам. На деле, хотя обзоры похожи на настоящие таблицы, они ими не являются. Содержимое настоящих таблиц представляет собой настоящие данные, в то время, как содержимое обзора динамически генерируется во время обращения к обзору. Синтаксис для создания обзора следующий:

```
create view name as select-stmt;
```

Название обзора задается текстом *name*, а определяется оператором *select − stmt*. В результате обзор будет выглядеть, как таблица с названием *name*. Представьте, что существует запрос, который требуется часто выполнять. Обзор - это средство, которое позволяет избегать постоянного набирания такого запроса. Пусть запрос выглядит как следующий:

```
select f.name, ft.name, e.name
from foods f
inner join food_types ft on f.type_id=ft.id
inner join foods_episodes fe on f.id=fe.food_id
inner join episodes e on fe.episode_id=e.id;
```

Он вернет имя каждого блюда, его тип и каждый эпизод в котором это блюдо упоминалось. Результат запроса является одной большой таблицей с информацией про каждый случай употребления блюда. Вместо того, чтобы каждый раз перезаписывать или запоминать такой запрос всегда, когда потребуются эти данные, лучше записать его в виде обзора. Пусть он будет называться *details*:

```
create view details as
select f.name as fd, ft.name as tp, e.name as ep, e.season as ssn
from foods f
inner join food_types ft on f.type_id=ft.id
inner join foods_episodes fe on f.id=fe.food_id
inner join episodes e on fe.episode_id=e.id;
```

Теперь можно опрашивать обзор *details* как таблицу. Например, так:

```
sqlite> select fd as Food, ep as Episode
        from details where ssn=7 and tp like 'Drinks';

Food                 Episode
-------------------  -------------------
Apple Cider          The Bottle Deposit 1
Bosco                The Secret Code
Cafe Latte           The Postponement
Cafe Latte           The Maestro
Champagne Coolies    The Wig Master
Cider                The Bottle Deposit 2
Hershey's            The Secret Code
Hot Coffee           The Maestro
Latte                The Maestro
Mellow Yellow soda   The Bottle Deposit 1
Merlot               The Rye
Orange Juice         The Wink
Tea                  The Hot Tub
Wild Turkey          The Hot Tub
```

Содержимое обзоров генерируется данамически. Таким обрабом, каждый раз, при обращении к *details*, выполняется SQL оператор ассоциированный с обзором, который создает результат, основываясь на данных в БД на момент времени выполнения. Некоторые особенности обзоров связанные с безопасностью, доступные в других СУБД, вообще говоря, отсутствуютв SQLite . А некоторые будут обсуждаться в следующих главах.

Наконец, для удаления обзора, существует команда *drop view*:

```
drop view name;
```

Название удаляемого обзора задается текстом *name*;

Обновляемые обзоры

Реляционная модель требует обновляемые обзоры. Это такие обзоры, данные в которых могут быть изменены. Например, выполнение операторов *insert* или *update* на обзоре, должно приводить к соответствующим изменениям в содержании таблиц. Они не поддерживаются в SQLite . Однако, используя триггеры, можно создать нечто, похожее на обновляемые обзоры. В секции 5.2.6 будут обсуждаться технические тонкости для таких триггеров.

5.2.5 Индексы

Индексы это средство, спроектированное для ускорения запросов в специальных случаях. Рассмотрим следующий запрос:

```
SELECT * FROM foods WHERE name='JujyFruit';
```

По умолчанию СУБД выбирает подходящие записи при помощи последовательного перебора. СУБД просматривает каждую запись в таблице, чтобы сравнить поле *name* со строкой *JujyFruit*.

Однако для часто выполняемого запроса и большой таблице имеет смысл использовать индексы. Как и в многих других СУБД, в SQLite реализованы индексы, основаные на B-деревьях.

Индексы увеличивает размер БД, так как в них хранится копии всех индексируемых полей. если создать индекс для каждого поля в таблице, то её размер увеличится более чем в двое. Также необходимо помнить, что индексы требуется обновлять. После каждой вставки, удаления или обновления записей кроме изменения собственно таблицы, выполняется изменение каждого индекса таблицы. Таким образом индекс может ускорить запрос, но замедляет выполнение операторов изменения таблицы.

Общий синтаксис команды создания индекса следующий:

```
create index [unique] index_name on table_name (columns)
```

На место текста *index_name* надо подставлять имя индекса, на место *table_name* имя таблиц с полями, для которых будет строится индекс. Текст *columns* является либо полем, либо списком полей, разделенных запятыми.

Если используется ключевое слово *unique*, то к индексу добавляется дополнительное ограничение уникальности каждой п-ки из упомянутых полей. Каждая комбинация значений всех полей из индекса должна быть уникальна. Далее, пример:

```
sqlite> create table foo(a text, b text);
sqlite> create unique index foo_idx on foo(a,b);
sqlite> insert into foo values ('unique', 'value');

sqlite> insert into foo values ('unique', 'value2');
sqlite> insert into foo values ('unique', 'value');
SQL error: columns a, b are not unique
```

Как видно, требование уникальности применяется к паре полей, но не к одному полю. Не стоит забывать, что сортирующие последовательности играют важную роль для требования уникальности.

Удаление индекса выполняется оператором *drop index*, его синтаксис следующий:

```
drop index index_name;
```

5.2.5.1 Сортирующие последовательности

Каждое поле индекса может иметь свою собственную сортирующую последовательность. Например, если бы для создания индекса, не чувствительного к регистру, был использован следующий оператор:

```
create index foods_name_idx on foods (name collate nocase);
```

Это бы означало, что величины в поле *name* сортируются без учета регистра. Список индексов таблицы в утилите CLP от SQLite можно получить при помощи команды *.indices*. Например:

```
sqlite> .indices foods
foods_name_idx
For more information, you can use the .schema shell command as well:
sqlite> .schema foods
CREATE TABLE foods(
  id integer primary key,
  type_id integer,
  name text );
CREATE INDEX foods_name_idx on foods (name COLLATE NOCASE);
```

Эту же информацию можно получить запросами к таблице *sqlite_master*, которая будет описана позже в этом разделе.

5.2.5.2 Применение индексов

Важно понимать, когда СУБД использует, а когда - не использует индексы. SQLite начнет пользоваться индексами в совершенно конкретных условиях. Например, индекс из одного поля, если он доступен, будет использоваться при наличии следующих выражений во фразе *WHERE*:

```
column {=|>|>=|<=|<} expression
expression {=|>|>=|<=|<} column
column IN (expression-list)
column IN (subquery)
```

Индексы из двух и более полей будут использоваться еще в более специфических условиях. Возможно, лучше проиллюстрировать на примере. Пусть существует таблица:

```
create table foo (a,b,c,d);
```

Затем был создан индекс с несколькими полями:

```
create index foo_idx on foo (a,b,c,d);
```

Поля из индекса можно использовать только слева - направо. Значит, следующий запрос:

```
select * from foo where a=1 and b=2 and d=3
```

в нем, только первое и второе поле использует индекс. Отсутствие поля *c* в выражении есть причина неиспользовании третьего выражения с полем *d*. В сущности, при использовании индекса из нескольких полей, SQLite подбирает поля слева - направо. СУБД начинает с левого поля и ищет выражение с ним. Затем переходит ко второму полю и так далее. Процесс прекращается либо когда не находится выражение с очередным полем во фразе *where* либо заканчиваются поля в индексе.

Кроме этого, существуют дополнительные ограничения. SQLite использует индекс из нескольких полей, если все выражения с полями индекса сравниваются либо по условию равно (=), либо при помощи операции *in*, кроме наиболее правого выражения. Для последнего поля, можно использовать не более двух неравенств, чтобы определить верхнюю и нижнюю границы. Рассмотрим пример:

```
select * from foo where a>1 and b=2 and c=3 and d=4
```

В таком случае, SQLite использует сканирование индекса только для поля a. Выражение $a > 1$ считается наиболее правым полем индекса, в связи с неравенством. В следующем примере

```
select * from foo where a=1 and b>2 and c=3 and d=4
```

СУБД из индекса используются поля a b, так как выражение $b > 2$ делает b наиболее правым допустимым полем индекса.

Напоследок, не создавайте индекс без причины. Должнен быть конкретный выигрыш производительности от его введения. Хорошо выбранные индексы являются полезным средством, но бессмысленно разбросанные там и сям дарят напрасные надежды и просто бесполезны.

5.2.6 Триггеры

Триггером называются несколько операторов SQL, которые выполняются при выполнении некоторого действия СУБД на таблицах. Общий синтаксис триггера выглядит так:

```
create [temp|temporary] trigger name
[before|after] [insert|delete|update|update of columns] on table
action
```

Как видно из синтаксиса, триггер задается именем, SQL операторами и таблицей. Операторы составляют тело триггера (*action*). Триггер выполняется (по английски зажигается - fire) до и или после (ключевые слова $before$ и $after$ соответственно) попытки редактировать заданную таблицу. К действиям СУБД на таблице относятся выполнение SQL операторов *insert*, *delete* и *update* на указанной таблице. Триггеры можно использовать для создания ограничений целостности, логгирования изменений, обновления других таблиц и многого другого. Они ограничиваются только тем, что можно выразить на языке SQL.

5.2.6.1 Триггер при обновлении

В отличие от триггера для *insert* или *delete*, триггер для *update* можно задать для конкретного поля в таблице. Общий синтаксис для определения такого триггера следующий:

```
create trigger name
[before|after] update of column on table
action
```

Следующий SQL скрипт содержит пример с применением такого триггера:

```
create temp table log(x);

create temp trigger foods_update_log update of name on foods
begin
  insert into log values('updated foods: new name=' || new.name);
end;
```

```
begin;
update foods set name='JUJYFRUIT' where name='JujyFruit';
select * from log;
rollback;
```

Скрипт создает временную таблицу *log* и временный триггер на таблице $foods.name$, который вставляет одну запись в таблицу *log* во время своей работы. Это событие происходит в течении транзакции. Первым делом обновляется поле *name* записи содержащей строчку $'JUJYFRUIT'$. Это обновление вызывает триггер. Во время своей работы триггер выполняет вставку записи в таблицу *log*. Далее, транзакция читает эту таблицу, показывая, что триггер на самом деле отработал. Затем откатываются изменения и сессия заканчивается. Временная таблица и временный триггер удаляются. Выполнение скрипта приводит к следующим результатам:

```
# sqlite3 foods.db < trigger.sql
```

```
create temp table log(x);

create temp trigger foods_update_log after update of name on foods
begin
  insert into log values('updated foods: new name=' || new.name);
end;

begin;
update foods set name='JUJYFRUIT' where name='JujyFruit';
SELECT * FROM LOG;
x
--------------------------------------------------
updated foods: new name=JUJYFRUIT
rollback;
```

В триггере SQLite предоставляет доступ к обеим записям: исходной (не неё ссылаются при помощи ключевого слова *old*) и обновленной (*new*). Надо обратить внимание, что в примере триггер используется текст $new.name$. После символа . можно записывать имя любого поля, например, $new.type\_id$ или $old.id$.

5.2.6.2 Обработка ошибок

Триггер, выполняемый перед некоторы событием, дает возможность его исследовать, быть может, изменить свое мнение по поводу события и даже отказаться от него. $before\ after$ (ключевые слова в определении триггера) позволяют внедрять новые ограничения целостности. Кроме того, SQLite предоставляет специальную функция SQL - $raise()$, которая позволяет создавать событие внутри тела триггера. *raise* имеет следующий прототип:

```
raise(resolution, error_message);
```

Первый аргумент - это политика разрешения конфликта (*abort*, *fail*, *ignore*, *rollback* и так далее). Второй - сообщение об ошибке. Если в качестве первого аргумента используется *ignore*, то оставшаяся часть триггера вместе с оператором SQL, который инициировал триггер, равно как и остальные триггера, которы должны были бы начать выполняться - прекращаются. Если SQL оператор - инициатор триггера сам по себе является частью другого триггера, то этот триггер приостанавливает выполнение на своем следующем операторе.

5.2.6.3 Обновляемые обзоры

Как упоминалось выше, при помощи триггеров можно создавать нечто, вроде обновляемых обзоров. Идея состоит в том, что для обзора создается триггер на обновление. В SQLite триггеры для обзоров могут использовать ключевое слово *instead* в своем определении. Создадим обзор, соединяющий таблицы $foods$ и $food\_types$:

```
create view foods_view as
  select f.id fid, f.name fname, t.id tid, t.name tname
  from foods f, food_types t;
```

В нем таблицы соединяются в соответствии с внешним ключем. Нужно заметить, что для всех полей обзора использовались синонимы. Это требуется, чтобы в теле будущего триггера различать поля *id* и *name* из обеих таблиц. Далее, запишем триггер для обзора:

```
create trigger on_update_foods_view
instead of update on foods_view
for each row
begin
   update foods set name=new.fname where id=new.fid;
   update food_types set name=new.tname where id=new.tid;
end;
```

Триггер начнет работу при попытке обновить $foods\_nm$. Он возмет значения из оператора *update* и использует их, чтобы обновить таблицы $foods$ и $food\_types$. Посмотрим, как он работает:

```
.echo on
-- Update the view within a transaction
begin;
update foods_view set fname='Whataburger', tname='Fast Food' where fid=413;
-- Now view the underlying rows in the base tables:
select * from foods f, food_types t where f.type_id=t.id and f.id=413;
-- Roll it back
rollback;
-- Now look at the original record:
select * from foods f, food_types t where f.type_id=t.id and f.id=413;
```

```
begin;
update foods_view set fname='Whataburger', tname='Fast Food' where fid=413;
select * from foods f, food_types t where f.type_id=t.id and f.id=413;
id   type_id  name            id   name
---  -------  --------------  ---  ---------
413  1        Whataburger     1    Fast Food

rollback;

select * from foods f, food_types t where f.type_id=t.id and f.id=413;
id   type_id  name            id   name
---  -------  --------------  ---  -------
413  1        Cinnamon Bobka  1    Bakery
```

Точно также можно добавить триггера для операторов *insert* и *delete* и этим позволить все основные манипуляции с данными через обзор.

5.3 Транзакции

Транзакции определяют группу операторов SQL, которые либо выполняются успешно все вместе, либо не выполняется ни один из них. Обычно это

понимается как атомарность целостности данных. Классическим примером иллюстрирующим зачем нужны транзакции является перевод денег. Предположи банковская программа переводит деньги с одного счета на другой. Программа может это сделать двумя способами: сначала добавить деньги на один счет, потом снимет деньги со второго либо сначала снимет деньги со второго счета, а потом добавит на первый. В любом случае перевод денег выполняется в два шага: уменьшение, а пото добавление либо добавление, а потом уменьшение.

А что произойдет, если СУБД не сможет выполнить вторую операцию, например в результате исчезновения напряжения в сети? В первом сценарии появятся лишние деньги, во втором - деньги безвовратно пропадут и целостность данных будет нарушена. В любом случае такого не должно случаться. Вывод из этого - либо обе операции должны завершиться успешно, либо ни одна из них. В этом сущность транзакций.

5.3.1 Область видимости транзакций

Для описания транзакций используется три оператора: *begin*, *commit* и *rollback*. *begin* начинает транзакцию. Любой оператор, следующий за *begin* окажется не выполненным, если перед завершением сессии не окажется не окажется оператора *commit*. Оператор *commit* завершает выполнение всех операторов с начала транзакции. Подобно ему *rollback* отменяет действие всех операторов с начала транзакции. Область видимости транзакции это несколько операторов SQL, которые вмести либо завершаются и фиксируются в БД (коммитятся) либо откатываются (Начинаются с *degin*, заканчивается *commit*/*rollback*). Далее пример:

```
sqlite> begin;
sqlite> delete from foods;
sqlite> rollback;
sqlite> select count(*) from foods;

count(*)
--------
412
```

В примере начинается транзакция, удаляются все записи из таблицы *foods*, затем выполняется оператор *rollback*. Оператор *select* показывает что ничего не изменилось.

По умолчанию, каждый SQL оператор в SQLite выполняется под своей собственной тразакцией. То есть, если в явной форме не определяется область видимости транзакции *begin...commit*/*rollback*, SQLite будет просто заворачивать в операторы *begin...commit*/*rollback* каждый отдельный оператор SQL. В таком случае каждый успешно выполненный оператор фиксируется (коммитится). Каждый оператор завершенный с ошибкой - откатывается. Этот режим работы СУБД (неявные транзакции) называется режим автоматической фиксации (autocommit mode). Каждый успешно выполненный оператор SQL автоматически фиксируется.

В SQLite существует еще два оператора: *savepoint* и *release*. Они позволят расширить гибкость транзакций. Точку сохранения можно указать среди несколько SQL операторов такоим образом, что откат будте выполнятся

не до начала транзакции, а до указанной точки. Как показано в следующем примере, чтобы создать точку сохранения требуется просто написать оператор *savepoint* с произвольны названием по своему вкусу:

```
savepoint justincase;
```

Позднее, если обнаружится что обработку данных надо отменить, вместо отката на начало транзакции можно выполнить откат на заданную точку сохранения:

```
rollback [transaction] to justincase;
```

Тут использован текст *justincase* как название точки сохранения. Вместо него можно пользоваться любым другим наименованием по своему вкусу.

5.3.2 Политики разрешения конфликтов

Что случается когда выполнение оператора прерывается в середине серии обновлений базы данных? В большинстве СУБД отменяются все изменения. Так в них спроектированы обработка нарушений целостности - окончание всего.

Однако, в SQLite используется специальная возможность, которая позволяет указать другой способ обработки нарушения целостности. Эта возожность называется разрешением конфликтов. Возьмем, к примеру, следующий *update*:

```
sqlite> update foods set id=800-id;
SQL error: PRIMARY KEY must be unique
```

В нем нарушение целостности *unique* возникнет как только оператор *update* достигает 388 запись, при попытке её поле *id* приниает значение $800-388=412$. Так как запись со значением 412 в поле *id* уже есть, выполнение оператора прерывается. Но, перед тем как возникло нарушение, SQLite уже обновило первые 387 записей. Что делать с ними? Поведение по умолчанию прерывает выполнение оператора и отменяет все уже сделанные изменения.

Однако, если все таки требуется сохранить все эти 387 изменений, несмотря на нарушение целостности? Если это надо, то возможность их оставить есть. Требуется подобрать подходящий способ разрешения конфликтов. Всего есть пять возможностей (политик) : *replace*, *ignore*, $fail$, *abort* и *rollback*. Эти возможности задает весь спектр чувствительности к ошибкам: от наиболее терпимого способа - *replace*, до наиболее жесткого - *rollback*. Способы разрешения конфликтов определяются в следующе порядке:

- *replace*: В случае нарушения целостности *unique*, SQLite удаляет запись (или записи) из-за которых возникло нарушение, и заменяет их новыми (к появлению которых приводит операторы *insert* или *update*). Следующие SQL операторы продолжают выполняться без ошибок. В случае нарушения целостности *notnull*, значения *null* заменяются умолчательными значениями, заданными для поля. Если умолчательного значения для поля не задано, то SQLite переходит

к политике *abort*. Заметим, что удаление записей происходит вследствие политики разрешения конфликтов, что бы удовлетворить ограничениям целостности, то для этих записей не вызываются триггера на удаление. Возможно, в будущем это поведение СУБД SQLite будет изменено.

- *ignore*: в случае нарушения ограничений целостности, SQLite выполняет SQL- оператор, оставляя записи, из-за которых возникли нарушения без изменений. Все остальные записи, изменяемые и до и после возникновения нарушения целостности таки изменяются. Таким образом, записи, влекущие нарушения целостности, просто остаются в исходном состоянии, при этом считается что SQL - оператор выполнялся без ошибок.

- *fail*: в случае нарушения ограничений целостности, SQLite прекращает выполнение оператора, но не отменяет изменения, которые уже были сделаны. То есть, все изменения базе данных, которые SQL - оператор успел сделать до возникновения нарушения, остаются несмотря на отмену оператора. Например, если выполнение оператора *update* привело к нарушению целостности при изменении сотой записи, то изменения первых девяносто девяти записей остаются. Записи, которые должны были бы быть изменены после сотой не меняются, вследствие прекращения выполнения оператора.

- *abort*: в случае нарушения ограничений целостности, SQLite восстанавливает все изменения, сделанные оператором и прекращает его выполнение. *abort* - это умолчательная политика разрешения конфликтов для всех операторов в SQLite и в стандарте языка SQL. Дополнительно можно отметить, что это дорогая политика, которая требует больших вычислительных ресурсов, даже если конфликта не возникло.

- *rollback*: в случае нарушения ограничений целостности, SQLite выполняет операцию отката *rollback* - прекращает выполнять текущий оператор и текущую транзакцию. Все изменения в БД сделанные в транзакции откатываются. Это наиболее радикальная политика при разрешении конфликтов, при которой даже одиночное нарушение влечет полный отказ от всего, сделанного в транзакции.

Политика разрешения конфликтов можно задавать в операторе SQL, равно как при определении таблицы или индекса. Точнее, указывать политику разрешения конфликтов можно в *insert*, *update*, *createtable* и *createindex*. Более того, ее можно указывать внутри триггеров. Синтаксис для разрешения конфликтов в операторах *insert* и *update* следующий:

```
insert or resolution into table (column_list) values (value_list);
update or resolution table set (value_list) where predicate;
```

Ключевые слова для политики появляются справа после слова *insert* или *update* и начинается со слова *or*. Так же, текст *insert or replace* можно сокращать до простого *replace*. Это очень похоже на поведение "*merge*" или "*upset*" в других СУБД.

В предыдущем примере с *update*, в котором изменения были выполнены до 387 записи, был произведен откат, так как по умолчанию применяется политика для разрешения конфликтов *abort*. Если требуется оставить изменения, политику надо поменять на $fail$. Для иллюстрации в следующем примере скопируем содержимое таблицы $foods$ в новую таблицу *test*. Добавим в *test* дополнительное поле с названием $modified$, со значением по умолчанию - *no*. В операторе *update* будем изменять его на *yes*, чтобы запомнить, какие записи окажутся измененными перед возникновением нарушения ограничения целостности. Используя политику $fail$, которая оставит эти записи в измененном состоянии, проверим сколько записей окажутся отредактированными.

```
create table test as select * from foods;
create unique index test_idx on test(id);
alter table test add column modified text not null default 'no';
select count(*) from test where modified='no';

count(*)
--------------------
412


update or fail test set id=800-id, modified='yes';
SQL error: column id is not unique

select count(*) from test where modified='yes';

count(*)
--------------------
387

drop table test;
```

Внимание

Желательно помнить про следующее, связанное с политикой $fail$. Порядок записей в таблице, вообще говоря, не определен. Таким образом, нельзя полагаться на определенный порядок записей в таблице, и на то, в какой последовательности SQLite будет обрабатывать эти записи. Можно предположить, что они упорядочены по полю *rowid*, однако это не безопасно. В документации ничего не сказано по этому поводу. Еще раз, никогда не делайте допущений по поводу порядка записей в таблицах любой СУБД. Во многих случаях гораздо лучше использовать политику *ignore*, чем $fail$. Она завершит работу и поменяет все записи, которые сможет и не остановится на первом нарушении целостности.

В случае явного использовании политики разрешения конфликтов, она задается для каждого отдельного поля. Далее, пример:

```
sqlite> create temp table cast(name text unique on conflict rollback);
sqlite> insert into cast values ('Jerry');
sqlite> insert into cast values ('Elaine');
sqlite> insert into cast values ('Kramer');
```

В таблице *cast* есть одно поле с названием *name*, на поле задано ограничение целостности *unique*, для разрешения конфликтов выбрано *rollback*.

Любые операторы *insert* или *update*, которые приводят к нарушению целостности, завершатся откатом всей транзакции, вследствие политики *rollback*:

```
sqlite> begin;
sqlite> insert into cast values('Jerry');
SQL error: uniqueness constraint failed

sqlite> commit;
SQL error: cannot commit - no transaction is active
```

commit не выполняется, так как был откат транзакции. Для оператора *create index* будут годится теже правила. Разрешение конфликтов для таблиц и индексов изменяет умолчательное поведение СУБД с политики *abort* на ту, которая задана для указанного поля, приведшего к нарушению целостности.

Политика, заданная в операторах языка DML (уровень операторов), перекрывает политику заданную на операторах языка DDL (уровень объектов).

Еще один пример:

```
sqlite> begin;
sqlite> insert or replace into cast values('Jerry');
sqlite> commit;
```

В этом случае применяется политика *replace*, заданная в операторе *insert*, а не политика *insert*, заданная на поле *cast.name*.

5.3.3 Блокировки в базе данных

Блокировки напрямую ассоциируются с транзакциями в SQLite . Поэтому для эффективного использования транзакций надо представлять как БД выполняет блокировки.

Блокировки SQLite относятся к грубым (coarse-grained locking). Если одна сессия выполняет запись в БД, все остальные сессии блокируются от записи, до тех пор, пока уже пишущая не закончит свою транзакцию. SQLite старается оттянуть пишущие блокировки как можно дольше, с целью улучшить конкурентность.

Будущее SQLite

> Начиная с версии 3.7.0 в SQLite будет использоваться регистрация записи с упреждением (write-ahead logging (WAL)). Придется изменить поведение транзакций и блокировок что бы освободить SQLite от описанной в этом разделе стратегии. в главе 9 будут описаны будущие изменения в SQLite .

В SQLite применяется политика укрупнения блокировки, при которой соединения получают полный доступ к БД чтобы выполнить операции записи. В SQLite ыделяется пять различных состояний блокировок: unlocked - незаблокированое, shared - разделяемое, reserved - резервирования, pending - отложенное and exclusive - исключительное. Каждая сессия (или соединение с) СУБД в любой момент времени может быть только в одном из

этих состояний. Более того, для всех состояний, кроме unlocked, существует своя собственная блокировка. В незаблокрованом состоянии блокировки не требуется.

Незаблокрованное состояние считается основным, с него и начнем. В этом состоянии нет сессии, которая бы имела доступ к БД. Это состояние возникает сразу после соединения с БД или непосредственно после начала транзакции.

Следующее за ним состояние - разделяемое (shared). Каждая сессия, которая читает из БД, но не пишет в БД, сначала переходит в разделяемое состояние, требующее разделяемую блокировку (shared lock). Несколько сессий могут одновременно находится в таком состоянии в любой момент времени. Значит, несколько сессий могут одновременно читать из общей БД в любой время. При этом, ни одна из сессий не может выполнять запись в БД - до тех пор, пока остаются хотя бы одна разделяемая блокировка.

Если сессии требуется выполнить запись в БД, она должна сначала затребовать блокировку резервирования. Только одна такая блокировка может быть в одной БД в каждый момент времени. Разделяемых блокировок может несколько. Блокировка резервирования является первой фазой записи в БД. Она не блокирует сессии с разделяемыми блокировками от чтения и позволяет сессиям запрашивать новые разделяемые блокировки.

После получения блокировки резервирования, сессия может начинать процесс обновления БД, правда эти обновления кешируются и, на деле, не пишутся на физический носитель. Все изменения сохраняются в оперативной памяти (для дополнительной информации по этому вопросу хорошо бы посмотреть обсуждение прагмы cache_size в секции 5.4.3 ниже в этой главе).

Когда сессия собирается переместить изменения из кэша в БД, она начинает процесс перехода от блокировки резервирования в исключительную блокировку. Что бы получить исключительную блокировку сессия сначала переходит в отложенную. Блокировка резервирования начинает процесс отсеивания, не позволяющий другим сессиям получать новые разделяемые блокировки. Таким образом, уже читающие сессии могут продолжать читать как и раньше, а новые сессии не смогут получать разделяемые блокировки. С этого момента сессия с отложенной блокировкой ожидает пока читающие сессии не освободят свои разделяемые блокировки.

Как только все разделяемые блокировки освобождаются, сессия в отложенном состоянии может запрашивать исключительную блокировку. И только после этого начинается реальная запись изменений в БД.

5.3.4 Мертвые блокировки

Может недавнее обсуждение блокировок и могло показаться интересным, но оно не отменяет вопроса зачем это надо? Зачем вообще о них думать? Затем, что в случае не понимания своих действий можно получить мертвую блокировку.

Рассмотрим события из таблицы 5.1 . Две сессии, A и B - полностью игнорирующие друг друга - пользуются одной и той же БД в одно время.

Сессия A выполняет первую команду, сессия B - вторую и третью, A - четвертую, и так далее.

Session A	Session B
`sqlite> begin;`	
	`sqlite> begin;`
	`sqlite> insert into foo values ('x');`
`sqlite> select * from foo;`	
	`sqlite> commit;`
	`SQL error: database is locked`
`sqlite> insert into foo values ('x');`	
`SQL error: database is locked`	

Рис. 5.1: Мертвые блокировки

В итоге, обе сессии завершаются в мертвой блокировке. Сессия B первой попыталось записать в БД и начинает отложенную блокировку. Её попытка выполнить *insert* закончится неудачей, при смене разделяемой блокировки на блокировку резервирования.

Без лишних разговоров, видно, что сессия A решила подождать, пока СУБД позволит изменять БД. Этим же занимается сессия B. В этот момент, все остальные сессии будут заблокированы. Новая открытая сессию даже не сможет читать из этой БД, по той причине, что B начала отложенную блокировку, которая не позволяет остальным сессиям запрашивать разделяемую блокировку. Таким образом, в мертвую блокировку попали не только A и B, в заблокированными оказались все, что связано с этой БД.

Каким образом можно избежать мертвой блокировки? Эти сессии не могут организовать встречу, на которой бы их представители обсудили возникшую проблему. Сессии даже не знают о существовании друг друга. Ответом является выбор правильного типа транзакции для выполнения работы.

5.3.5 Типы транзакций

В SQLite существует три типа транзакций, которые начинаются с различных состояний. Транзакция могут начинать работу как *deferred* - отложенная, *immediate* - немедленная, *exclusive* - исключительная. Тип указывается после ключевого слова *begin*:

```
begin [ deferred | immediate | exclusive ] transaction;
```

Отложенная транзакция не требует никаких блокировок пока это действительно не понадобится. То есть, оператор *begin* на деле ничего не означает, он начинает незаблокированное состояние. По умолчанию, если после

begin нет ключевого слова, то это отложенная транзакция. Несколько сессий могут одновременно начинать отложенную транзацию без каких либо блокировок. В этом случае, первый читающий оператор повлечет разделяемую блокировку, первый пишущий - пытается запросить блокировку резервирования.

Немедленная транзакция пытается получить блокировку резервирования сразу после команды *begin immediate*;. Если это удалось, транзакция гарантирует, что остальные существующие сессии не смогут писать в БД, только читать. Новые сессии не смогут даже читать. Еще одним следствием будет то, что существующие сессии не смогут начинать немедленные или исключительные транзакции. Ответом SQLite на такие попытки будет ошибка $SQLITE\_BUSY$. Транзакция может делать изменения в БД, но, возможно, не сможет и выполнить оператор *commit*, получая ошибку $SQLITE\_BUSY$. Получение ошибки означает, что существуют активные читающие сессии, как в приведенном примере. Когда они закончат чтение, транзакция может завершаться *commit*-ом.

Исключительная транзакция получает исключительную блокировку на БД. Она выполняется подобно немедленной, но будучи успешной начатой гарантирует отсутствие других активных сессий и можно беззаботно читать из и писать в БД.

Главная проблема в рассмотренном примере это то, что обе сессии одновременно начали писать в БД, и не делали попыток ослабить блокировки. В конечном счете, корнем проблемы оказались разделяемые блокировки. Если бы обе сессии начали транзакцию с *begin immediate* то мертвой блокировки не возникло бы. В этом случае первая из сессий начала бы писать в БД, а вторая - должна была бы ждать. Ожидающая транзакция может продолжать попытки, будучи уверенной, что дождется. Транзацкии *begin immediate* и *begin exclusive* используемые всеми сессиями, желающими изменять содержимое БД, предоставляют механизм синхронизации, который предотвращает мертвые блокировки. Но при этом, требуется помнить правила их использования.

Если БД используется одной сессией, тогда достаточно просто *begin*. Если БД используется несколькими сессиями и в БД пишет больше одной, тогда транзакции надо начинать с *begin immediate* или *begin exclusive*. Оба работают хорошо. Далее транзации и блокировки будут обсуждаться в главе 6

5.4 Администрирование БД

К администрирование БД, вообще говоря, относится управление операциями с БД. Большинство задач администратора могут быть выполнены при помощи операторов SQL. Кроме того, SQLite имеет присущие только ему административные возможности, такие как присоединение нескольких баз в одной сессии или специальные параметры, именуемые прагма, которые используются для конфигурирования БД.

5.4.1 Присоединение БД

SQLite позволяет присоединить несколько баз данных в текущий сессии используя оператор *attach*. После присоединения БД, все её содержимое становится доступным в глобальной области видимости. Оператор имеет следующий синтаксис:

```
attach [database] filename as database_name;
```

В операторе слово $filename$ обозначает путь к файлу и имя файла с БД. $database\_name$ обозначает логическое имя, при помощи которого можно ссылаться на БД и её объекты. Первой присоединенной БД автоматически присваивается имя $main$. Временные объекты, которые возможно требуются в сессии, создаются в базе данных с именем $temp$ (Эти объекты можно просмотреть используя прагму $database\_list$). Логическое имя используют, чтобы ссылаться на объекты внутри заданной БД. Если несколько таблиц (или других объектов) в присоединенных БД имеют одинаковые имена, то логическое имя требуется для идентификации таких объектов. Например, если в двух базах существуют таблицы с именем foo, а логическое имя одной из присоединенных баз $db2$, то для запроса к таблице foo в $db2$ требуется использовать полное имя $sb2.foo$ как показано ниже:

```
sqlite> attach database '/tmp/db' as db2;
sqlite> select * from db2.foo;

x
----------
bar
```

Если требуется сделать запрос к таблице в первой присоединенной БД, то надо использовать имя $name$:

```
sqlite> select * from main.foods limit 2;

id          type_id     name
----------  ----------  --------------
1           1           Bagels
2           1           Bagels, raisin
```

Так же поступают со временной базой:

```
sqlite> create temp table foo as select * from food_types limit 3;
sqlite> select * from temp.foo;

id   name
---  -------------
1    Bakery
2    Cereal
3    Chicken/Fowl
```

Для отсоединения от БД, используется оператор *detach*, определяемый ниже:

```
detach [database] database_name;
```

Оператор при помощи логического имени $database\_name$ отсоединяет SQLite от присоединенного ранее файла БД. Как показно в разделе 5.4.3 список всех присоединенных баз можно посмотреть используя прагму $database\_list$.

5.4.2 Уплотнение БД

В SQLite существует два оператора спроектированных для уплотнения БД - *reindex* и *vacuum*. *reindex* используется для перестройки индесов. У него есть две формы:

```
reindex collation_name;
reindex table_name|index_name;
```

Оператор в первой форме перестраивает все индексы, использующие сортирующую последовательность *collation_name*. Такая форма оператора используется, если поменялась поведение заданной пользователем последовательности. Все индексы заданной таблицы или заданный индекс можно перестроить, используя вторую форму оператора.

Оператор *vacuum* освобождает все неиспользуемое пространство в базе данных, перестраивая файл базы. *vacuum* не может быть выполнен, если есть открытые транзакции. Альтернативой ручному использования оператора является возможность автоосвобождения ненужного пространства при помощи прагмы *auto_vacuum*, описанной в следующем разделе.

5.4.3 Конфигурирование БД

В SQLite отсутствует файл конфигурации. Все параметры для СУБД задаются через так называемые прагмы. Они разделяются на неколько групп. Некоторыми можно пользоаться подобно обычным переменным, некоторые - напоминают операцию *like*, некоторые требуют параметры и, поэтому, похожи на функции. Различные особенности касательно информации времени исполнения, схемы БД, версий, формата файла, использования оперативной памяти и отладки управляются через прагмы. Обычно для прагм существует временная и постоянная форма. Временная форма актуальна только на время текущей сессии. Прагма в постоянной форме запоминается в файле БД и будет влиять на работу СУБД в последующих сессиях. К таким относится прагма, задающая размер кэша для соединения.

В этой секции описываются наиболее часто используемые прагмы.

5.4.3.1 Размер кеша для соединения

Эта прагма влияет на то, как много страниц из файла БД, СУБД может держать в оперативной памяти. Что бы задать размер доступной памяти для текущей сессии используется прагма *cache_size*:

```
sqlite> pragma cache_size;

cache_size
---------------
2000

sqlite> pragma cache_size=10000;
sqlite> pragma cache_size;

cache_size
---------------
10000
```

Чтобы изменить размер кеша для всех сессий, используется

прагма $default\_cache\_size$. Это значение будет запомнено в БД и окажет влияние на сессии, созданные после изменения, но не на текущую.

Кэш используется также для того, что бы хранить изменения передаваемые в БД. Как описывалось в разделе 5.3.3 сессия в этот момент находится в состоянии резервирования (и иметь блокировку резервирования). Сессия, переполнившая кеш, не сможет далее изменять данные до получения исключительной блокировки. Это означает, что она будет вынуждена ждать окончания работы читающих сессий.

Увеличение размера кеша может помочь при многих сессиях интенсивно меняющих базу данных. Чем больше его размер, тем больше изменений может выполнить сессия перед получением исключительной блокировки. Увеличение размера позволяет не только сделать больше работы, но оно также укорачивает время исключительной блокировки, так как все изменения уже находятся в кеше и готовы для передачи в файл БД. Глава 6 содержит некоторые тонкости для правильной настройки размера кеша.

5.4.3.2 Получение информации о базе данных

Следующие прагмы используются для получения информации о БД:

- $database\_list$ - список присоединенных баз данных;
- $index\_info$ - информация о полях внутри индексов, принимает имя индекса в качестве аргумента;
- $index\_list$ - информация о индексах в таблице, принимает имя таблицы в качестве аргумента;
- $table\_info$ - информация о полях таблицы.

Следующие примеры иллюстрируют сказанное:

```
sqlite> pragma database_list;

seq   name     file
----  -------  --------------------
0     main     /tmp/foods.db
2     db2      /tmp/db

sqlite> create index foods_name_type_idx on foods(name,type_id);
sqlite> pragma index_info(foods_name_type_idx);

seqn  cid      name
----  -------  --------------------
0     2        name
1     1        type_id

sqlite> pragma index_list(foods);

seq    name                unique
-----  ------------------- ---------------
0      foods_name_type_idx 0
```

```
sqlite> pragma table_info(foods);

cid    name            type            notn  dflt  pk
-----  --------------  --------------  ----  ----  ----------
0      id              integer         0           1
1      type_id         integer         0           0
2      name            text            0           0
```

5.4.3.3 Перенос изменений в файл БД

Обычно SQLite переносит (commit) все изменения на жесткий диск в специальные (критичекие) моменты времени, добиваясь надежности транзакций. В других СУБД похожим образом ведет себя функциональность контрольных точек (checkpoint). Но существует возможность отключить это свойство, если требуется повысить производительность SQLite . Для этого используетя прагма *synhronous*. Она может принимать три значения: $full$, *normal* и off. Их смысл определяется ниже:

- $full$ - SQLite останавливается в критические моменты времени, чтобы гарантировать реальное перенос новых данных на жесткий диск перед продолжением работы. При этом крах операционной системы или пропадание напряжения не приведет к нарушению условий целостности. Полная синхронизация очень безопасна, зато достаточно медленна;
- *normal* - SQLite будет останавливаться в наиболее критические моменты, но реже чем в предыдущем режиме. Существует маленькая, но не нулевая возможность искажения БД после пропадания напряжения. На практике, однако, более вероятно, что БД будет испорчена в результате катастрофического отказа диска или других похожих проблем с компьютером. ;
- off - SQLite работает без каких либо перерывов со скоростью передачи данных операционной системе. Такой режим может ускорить некоторые операции в 50 и более раз. База данных будет удовлетворять условиям целостности после краха СУБД. Но она может быть испорчена в случае краха операционной системы или потери напряжения.

У этой прагмы нет постоянной формы. Глава 6 содержит объяснения как работает и насколько критическую роль играет прагма *synhronous* для надежности транзакций.

5.4.3.4 Хранилище временных объектов

Хранилище временных объектов это место, где SQLite хранит промежуточные данные, такие как временные таблицы, индексы и другие объекты. По умолчанию расположение хранилища временных объектов задается на этапе компиляции SQLite , которое меняется в зависимости от операционной системы. Две прагмы - *temp_store* и *temp_store_directory* управляют расположением хранилища. Первая задает будет ли SQLite использовать

оперативную память или жесткий диск для временных объектов. Она может принимать три значения: $default$, $file$, $memory$. $default$ используется место заданное при компиляции SQLite , $file$ - используется файл операционной системы, $memory$ - используется оперативная память. В случае значения $file$, можно использовать вторую прагму - $temp\_store\_directory$, что бы задать расположение этого файла.

5.4.3.5 Размер страницы БД, кодировка и уплотнение

Размер страницы, кодировка и автоуплотнение файла БД (autovacuuming) надо задавать перед созданием файла БД. То есть, чтобы изменить умолчательные значения требуется задать эти прагмы перед созданием любого объекта в новой БД. Размер страницы по умолчанию задается в зависимости от нескольких, зависимых от компьютера и операционной системы, характеристик. К ним относится размер сектора на диске и кодирование. SQLite поддерживает размер страницы от 512 до 32786 байт (задавать надо степени двойки). Кодировки могут выбираться UTF-8, UTF-16le (ittle-endian) и UTF-16be (big-endian).

Что бы автоматически поддерживать минимальный размер файла базы данных используется прагма $auto\_vacuum$. Обычно при окончании транзакции, удалившей данные из БД, размер файла не изменяется. При включенной прагме $auto\_vacuum$, при удалении данных файл будет укорачиваться. Для поддержания такой функциональности SQLite придется хранить дополнительную информацию, что приведет к небольшому увеличению файла БД. Оператор $vacuum$ не окажет никакого влияния на БД, использующей $auto\_vacuum$.

5.4.3.6 Отладка

Еще четыре прагмы используются для различных целей отладки. $integrity\_check$ следит за неупорядоченными и некорректными записями, пропущенными страницами, некорректными индексами. При наличии каких либо проблем, возвращается текст с их описанием. Иначе SQLite возвращает текст ok. Остальные прагмы используются для трассировок работы парсера и движка БД, в случае если SQLite скомпилирован с отладочной информацией. Глава 9 содержит подробную информацию об этих прагмах.

5.4.4 Системный каталог

Роль системного каталога содержащим информацию про таблицы, обзоры, индексы и триггера в Бд играет таблица $sqlite\_master$. Для примера, можно посмотреть содержание этой таблицы для базы данных foods:

```
sqlite> select type, name, rootpage from sqlite_master;

type        name                        rootpage
----------  --------------------------  ----------
table       episodes                    2
table       foods                       3
table       foods_episodes              4
table       food_types                  5
index       foods_name_idx              30
table       sqlite_sequence             50
trigger     foods_update_trg            0
trigger     foods_insert_trg            0
trigger     foods_delete_trg            0
```

Поле *type* указывает тип объекта, поле *name* - название, поле *rootpage* содержит первую страницу B-дерева для данный объект в БД. Это поле имеет смысл только для таблиц и индексов.

Кроме этого, таблица *sqlite_master* содержит поле *sql*, в котором хранится оператор языка DML, которым был создан объект БД:

```
sqlite> select sql from sqlite_master where name='foods_update_trg';

create trigger foods_update_trg
before update of type_id on foods
begin
  select case
     when (select id from food_types where id=new.type_id) is null
     then raise( abort,
                 'Foreign Key Violation: foods.type_id is not in food_types.id')
  end;
end
```

5.4.5 План запроса

Оператором *explain query plan* можно просмотреть подробности выполнения запроса. В ответ на этот оператор SQLite вернет список выполненных шагов, которые СУБД сделала для выполнения запроса.

Для получения плана запроса требуется написать текст *explain query plan* за которым немедленно следует обычный текст запроса. Ниже приводится объяснения SQLite по поводу выполнения запроса к таблице *foods*:

```
sqlite> explain query plan select * from foods where id = 145;
order       from        detail
----------  ----------  ----------------------------
0           0           TABLE foods USING PRIMARY KEY
```

Это значит, что для доступа к нужной записи SQLite использовал первичный ключ, а не перебирал записи таблицы. Изучение плана запроса является ключем к пониманию как SQLite получает доступ к данным и отвечает на запрос. Он проливает свет на то, когда и как используются индексы и на порядок соединнения таблиц и оказывает неоспоримую пользу для разрешения трудностей с медленно работающими запросами.

5.5 Итак

Может SQL и простой язык для использования, но уже рассмотренная небольшая диалекта от SQLite потребовала две главы для введения основ-

ных понятий. Но не надо слишком удивляться, так как SQL является универсальным средством работы с реляционными базами данных. Любой, кто собирается пользоваться реляционной базой данных, должен знать SQL, не зависимо от того, является ли он случайным пользователем, системным администратором или разработчиком. Програмисту, использующему SQLite , тоже рекомендуется начинать работу с операторов SQL.

Теперь хорошо бы немного познакомится с подробностями выполнения SQLite -ом этих операторв SQL. Для этого полезно было бы почитать главу 6 , которая является введением в програмный интрефейс SQLite и объяснением как выполняется SQL с точки зрения програмного интерфейса.

Глава 6

SQLite Design and Conceptions

Глава 7

The Core C API

Глава 8

The Extension C API

Глава 9

SQLite Internals and New Features

Предметный указатель

Литература

[1] Mike Owens Grant Allen. The definitive guide to sqlite, 2010. http://www.apress.com/9781430232254. 72

Оглавление