

Неполное Руководство по SQLite для
пользователей Windows

Перевод: А.Г. Пискунов

8 августа 2018 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 второй, третьей, четвертой и пятой глав монографии Grant Allen и Mike Owens 'The Definitive Guide to SQLite'. Существенную помощь в работе оказал George Brink, а так же многие участники форума <http://www.sql.ru/forum>. Поэтому, пользуясь случаем хочется, выразить благодарность всем им и отдельно Alex-у Sibilev-у за создание форума и многолетние усилия по его поддержанию.

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, компактная, 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. Приложение просто делает его работу, движок БД просто содержится внутри. Это избавляет от необходимости сетевых настроек и администрирования. Подумайте как это облегчает жизнь: никаких файрволов, 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 прикладным интерфейсом C. Таким образом, SQLite встроены 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 (-ax) базы данных. SQLite позволяет им управляться с синхронизацией и блокировками.

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

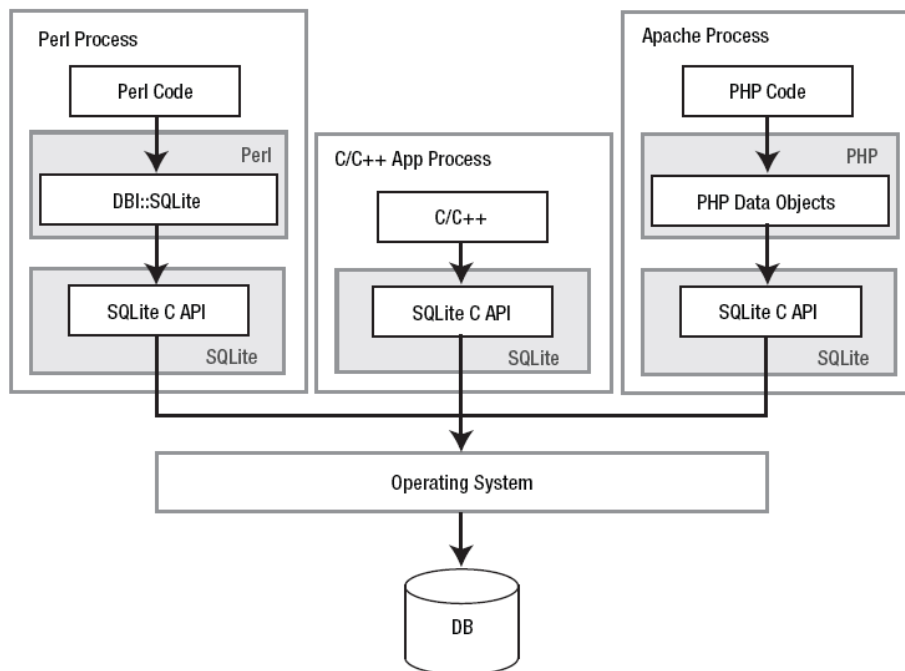


Рис. 2.1: SQLite встраивается в приложения

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 конвейеру. Верхние модули компилируют запросы, средние выполняют их, а нижние управляют с диском и взаимодействуют с операционной системой.

2.2.1 Интерфейс

Интерфейс является верхним модулем и состоит из программного интерфейса языка C (API). Это означает, что через него с SQLite взаимодействуют приложения, скрипты и библиотеки. Образно говоря, через этот интерфейс разработчики, администраторы, студенты и сумасшедшие ученые разговаривают с SQLite .

2.2.2 Компилятор

Процесс компиляции начинается с лексического анализатора и парсера.

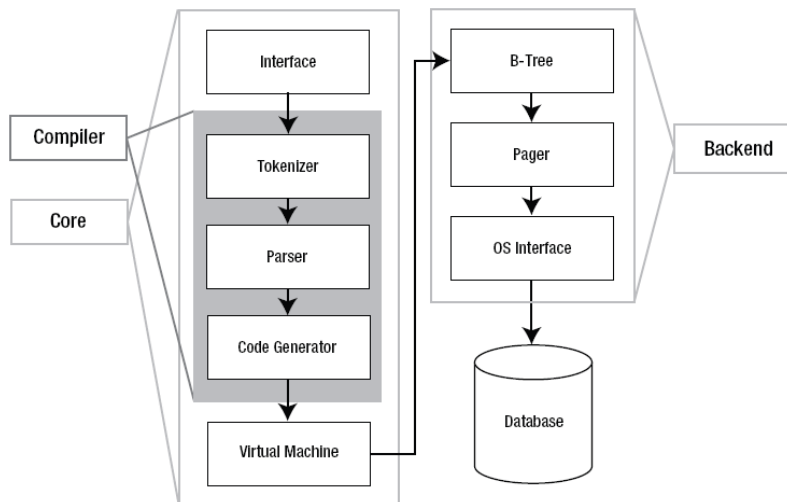


Рис. 2.2: архитектура SQLite

2.2.3 Виртуальная Машина

Виртуальная машина (движок БД) называется virtual database engine (VDBE)

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.

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 комментарии, 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 утилита 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 подгружаемая 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.

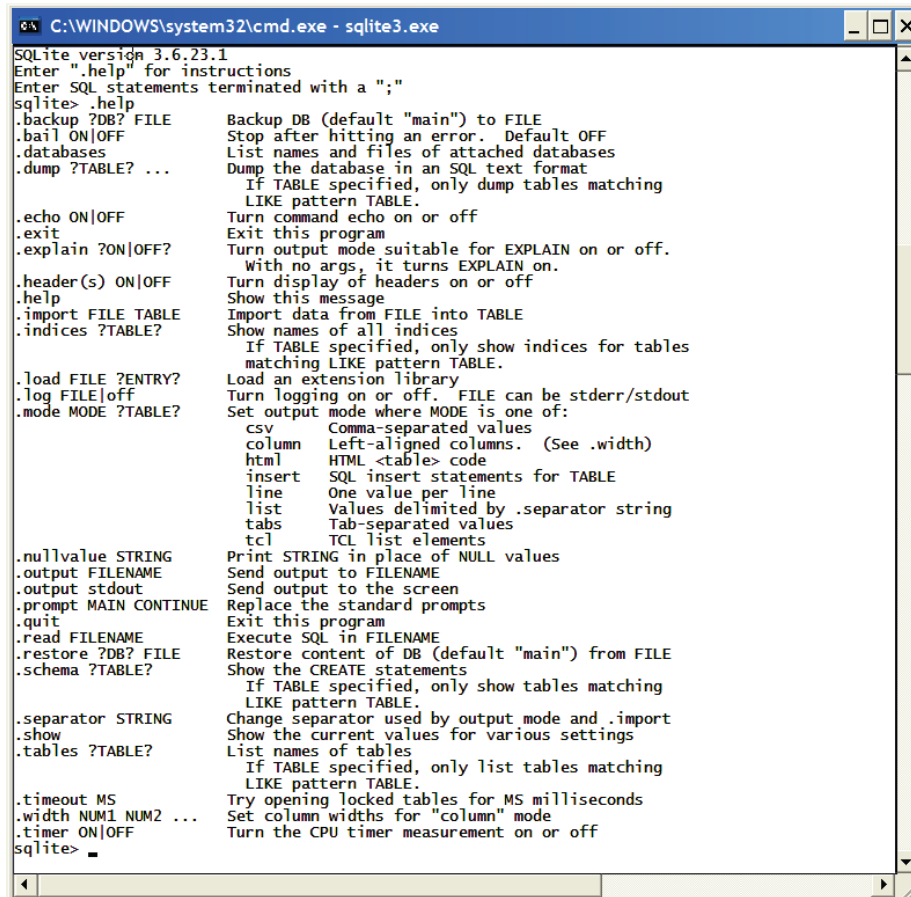
3.2 SQLite под Windows

Собираетесь ли Вы использовать 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' - 'A bundle of command-line tools for managing SQLite database files'. В загруженном архиве (с именем приблизительно sqlite-tools-win32-x86-3240000.zip) 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 невероятные 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.



```

C:\WINDOWS\system32\cmd.exe - sqlite3.exe
SQLite version 3.6.23.1
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .help
.backup ?DB? FILE      Backup DB (default "main") to FILE
.bail ON|OFF           Stop after hitting an error. Default OFF
.databases              List names and files of attached databases
.dump ?TABLE? ...     Dump the database in an SQL text format
                       If TABLE specified, only dump tables matching
                       LIKE pattern TABLE.
.echo ON|OFF           Turn command echo on or off
.exit                  Exit this program
.explain ?ON|OFF?     Turn output mode suitable for EXPLAIN on or off.
                       With no args, it turns EXPLAIN on.
.header(s) ON|OFF     Turn display of headers on or off
.help                  Show this message
.import FILE TABLE    Import data from FILE into TABLE
.indices ?TABLE?      Show names of all indices
                       If TABLE specified, only show indices for tables
                       matching LIKE pattern TABLE.
.load FILE ?ENTRY?    Load an extension library
.log FILE|off         Turn logging on or off. FILE can be stderr/stdout
.mode MODE ?TABLE?   Set output mode where MODE is one of:
                       csv      Comma-separated values
                       column   Left-aligned columns. (See .width)
                       html     HTML <table> code
                       insert   SQL insert statements for TABLE
                       line     One value per line
                       list     Values delimited by .separator string
                       tabs     Tab-separated values
                       tcl      TCL list elements
.nullvalue STRING     Print STRING in place of NULL values
.output FILENAME      Send output to FILENAME
.output stdout        Send output to the screen
.prompt MAIN CONTINUE Replace the standard prompts
.quit                 Exit this program
.read FILENAME        Execute SQL in FILENAME
.restore ?DB? FILE    Restore content of DB (default "main") from FILE
.schema ?TABLE?      Show the CREATE statements
                       If TABLE specified, only show tables matching
                       LIKE pattern TABLE.
.separator STRING     Change separator used by output mode and .import
.show                 Show the current values for various settings
.tables ?TABLE?      List names of tables
                       If TABLE specified, only list tables matching
                       LIKE pattern TABLE.
.timeout MS           Try opening locked tables for MS milliseconds
.width NUM1 NUM2 ... Set column widths for "column" mode
.timer ON|OFF        Turn the CPU timer measurement on or off
sqlite>

```

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

Чтобы 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 можно 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 точки (`.`) 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 создаст БД, только после того, как пользователь создаст в БД нечто вроде таблицы или представления. 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, предшествующие 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 представление 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 содержимом БД. Можно получить список таблиц (равно как представлений) используя команду *.tables [pattern]*, где необязательный параметр *[pattern]* может быть регулярным выражением в смысле операции *like* языка SQL. В ответе окажутся таблицы и представления подходящие под регулярное выражение. Без регулярного выражения ответ будет содержать список всех таблиц и представлений:

```
sqlite> .tables
```

```
schema test
```

В нашем ответе видно название таблицы *test* и представления *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 представления можно получить используя команду `.schema [table name]`. Если имени таблицы не будет задано SQLite вернет определения всех объектов БД - таблиц, индексов, представлений 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 системного представления SQLite , который называется `sqlite_master`. Это представление является системным каталогом. Его поля описываются ниже.

- `type` - тип объекта (таблица, индекс, представление, 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 представление `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, представление 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 главе [5](#) .

Замечание

Не забывайте, что большинство утилит вроде SQLite CLP, хранит историю команд, введенных оператором. Чтобы 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 представлений. Утилиты будет экспортировать таблицы или представления, 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`. Чтобы 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 экспортом данных. Можно экспортировать различные подмножества таблиц и представлений 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 нижеследующим примере удаляются два объекта БД (таблица `test` и представление `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 представление 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 'sqlite> '
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 с разделителем по-умолчанию. То есть, для экспортирования таблицы в CSV 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 Экспортирование таблицы (Exporting Delimited Data)

Рассмотренные выше возможности для экспортирования, импортирования и форматирования данных позволяют экспортировать таблицы (как записи). Например, чтобы экспортировать записи таблицы `test`, в которых поле `value` начинается на букву `m` в файл `test.csv`, сделайте 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`), сделайте 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 планировщике 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, чтобы 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 содержать вполне читабельные команды подязыков DDL или DML для создания и наполнения базы данных test.db. Чтобы извлечь все записи из таблицы test, выполните следующее:

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

Так же выполнить CLR 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
```

CLR выполнит все команды из файла test.sql, чтобы 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. Чтобы обойти такую особенность нужно дописать любую команду в строку параметров. Например:

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

Команда .exit переведет утилиту в пакетный режим.

Замечание

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

3.4.8 Бэкап базы данных

Есть несколько возможностей для выполнения бэкапа. Бэкап 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 имеет значения сколько бекапов своей базы данных сделано, если восстановление из них не удастся выполнить. Имеют 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 средством для получения информации о логической структуре базы данных (внутренняя схема БД), является использование представления `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 Эксперт Персонал - <http://www.sqliteexpert.com/download.html> - минимум инсталляции, хорошая функциональность;
- 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;
- 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

Расположен на странице <http://system.data.sqlite.org/index.html/doc/trunk/www/downloads.wiki>; Рекомендуется использовать провайдер, построенный в native mode assembly (родном режиме). Это архивы вроде <http://system.data.sqlite.org/downloads/1.0.97.0/sqlite-netFx35-binary-Win32-2008-1.0.97.0.zip>. Кроме того, подразумевается, что на компьютере уже установлена студия для Visual C# и, значит, The Visual C# 20XX YYYYYY runtime тоже установлен.

3.6.1 Построение консольного приложения на C-Sharp

3.6.1.1 Три версии для выполнения оператора *select*

В приложении строится три версии одной и той же программы. Первая использует конкретный адаптер SQLite :

```
---- File:./db/0sqlite.cs
```

```
using System;
using System.IO;
using System.Data;
using System.Data.Common;
using System.Threading;
using System.Configuration;
using System.Data.SQLite;

partial class test
{
    public static void sqlite(string scnn){
        Console.WriteLine ("\n *** using SQLite provider only");
        using (SQLiteConnection cnn = new SQLiteConnection(scnn))
        {
            cnn.Open();
        }
    }
}
```



```

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

partial class test
{
    public static void Db(string scmn){
        Console.WriteLine ("\n *** using abstract provider only");
        try
        { // чтобы избежать установки SQLite или
          // необходимости иметь файл конфигурации
          // я в существующую конфигурацию провайдеров
          // искусственно добавляю System.Data.SQLite.dll
          // по-другому получить нужную фабрику не удавалось
          var dataSet = ConfigurationManager.GetSection("system.data")
            as System.Data.DataSet;
          dataSet.Tables[0].Rows.Add("SQLite Data Provider"
            , ".Net Framework Data Provider for SQLite"
            , "System.Data.SQLite"
            , "System.Data.SQLite.SQLiteFactory, System.Data.SQLite");
        }
        catch (System.Data.ConstraintException) { }

        DbProviderFactory dbF = DbProviderFactories.GetFactory("System.Data.SQLite");
        using (DbConnection cnn = dbF.CreateConnection()) {
            cnn.ConnectionString = scmn;
            cnn.Open();
            if (Settings.tblFill){ // использование DataTable
                using (DataTable ds = new DataTable()) {
                    DbDataAdapter da = dbF.CreateDataAdapter();
                    da.SelectCommand = cnn.CreateCommand();
                    da.SelectCommand.CommandText = "select * from food_types";
                    da.Fill(ds);
                    PrintTable(ds, true);
                }
            }
            else { // использование датаридера
                DbCommand cmd = cnn.CreateCommand();
                cmd.CommandText = "select * from food_types";
                DbDataReader 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();
        }
    }
}

```

```

    }
  }
}
}

```

---- End Of File:./db/2db.cs

3.6.1.2 Метод Main и построение приложения

Остальной код приложения:

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

```

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

```

```

struct Settings
{
    static public bool    tblFill
                        , sqlite = false
                        , db     = false
                        , ins    = false
                        , del    = false
    ;
}

```

```

partial class test
{

```

```

    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_ = "SQLite";

    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], "l", "list")) {
            DataTable fs = DbProviderFactories.GetFactoryClasses();
            PrintTable(fs, false);
        }
        else if(IsOneOf(args[i], "i", "insert")) {
            Settings.ins = true;
        }
        else if(IsOneOf(args[i], "d", "delete")) {
            Settings.del = true;
        }
        else if(IsOneOf(args[i], "b", "batch")) {
            Settings.tblFill = true;
        }
        else if(IsOneOf(args[i], "s", "sqlite")) {
            Settings.sqlite = true;
        }
        else if(IsOneOf(args[i], "db")) {
            Settings.db = true;
        }
        else if (IsOneOf (args[i], "ver", "version")) {
            int ma, mi, b;
            version (out ma, out mi, out b);
            Console.WriteLine("#{0} ({1}.{2}.{3} for {4})", Path.GetFileNameWithoutExtension(
                System.Windows.Forms.Application.ExecutablePath), ma, mi, b, me_);
            return 0;
        }
    }
}

if (Settings.ins == true)          ins("Data Source=foods.db");
else if (Settings.del == true)    del("Data Source=foods.db");
else if (Settings.sqlite == true && Settings.db == true) sqliteDb("Data Source=foods.db");
else if(Settings.sqlite == true && Settings.db == false)  sqlite("Data Source=foods.db");
else if(Settings.sqlite == false && Settings.db == true)  Db("Data Source=foods.db");
else {
    Console.WriteLine("nothing to do!");
    goto error;
}

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

error:
string msg = "usage "+": app [-b] [-l] [-ver] { -db | -sqlite}\n"
+ "\t -b      : to read all the table (DataAdapter) , opposite - DataReader\n"
+ "\t -l      : to show list of the ADO provider fabrics\n"
+ "\t -i      : to split lines of stdin based on ',' and make insert to foods_type table\n"
+ "\t -d      : to delete extra records from foods_type table\n"
+ "\t -db     : to use abstract (DbXxxxx) way\n"
+ "\t -sqlite : to use sqlite (SQLiteXxxxx) way\n"

```



```

+ "\t -ver    : to print version\n";
  Console.WriteLine("sqlite example: \n {0}", msg);
return 0;
}

```

```

public static void PrintTable(DataTable iTbl, bool verbose)
{
  string sep = "\t";
  if (iTbl != null) {
    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();
    }
  }
}

```

```

static public void version (out int major, out int minor, out int build){
  System.Reflection.Assembly asm = System.Reflection.Assembly.GetExecutingAssembly();
  System.Version ver = asm.GetName().Version;
  major = ver.Major;
  minor = ver.Minor;
  build = ver.Build;
}

```

```

}

```

```

//  Encoding e = Encoding.GetEncoding(1251);
//      dr.GetBytes(1, (Int64)0, buf, 0, 100);
//      foo = e.GetString(buf);

```

```

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

```

Пакетный файл для построения приложения:

```

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

```

```

echo off
call params.%COMPUTERNAME%.cmd

```

```

if %1 == -? %X%\csc.exe -?
if %1 == -? exit

```

```

SET S=/r:%X%\System.Data.dll /r:%X%\System.dll /r:%X%\System.Drawing.dll /r:%X%\System.EnterpriseService
SET S=/r:%X%\System.Data.dll /r:%X%\System.dll /r:%X%\System.Core.dll /r:%X%\System.Drawing.dll /r:%X%

```

```
SET FLGS= /nologo /noconfig /unsafe
```

```
if exist local.cmd call local.cmd
```

```
rm %1.exe .*
rm ".eRr."
echo on
%X%\csc.exe %S% %R% %FLGS% /t:exe /out:%1.exe *.cs %ADD% %2 >>".eRr."
```

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

Динамически подгружаемые библиотеки и версия .NET:

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

```
rem //http://www.sql.ru/forum/actualthread.aspx?tid=584888&pg=1&hl=%ea%ee%ec%e0%ed%e4%ed%e0%ff%20%
rem agp
```

```
SET X=d:\WINDOWS\Microsoft.NET\Framework\v4.0.30319
SET X=C:\WINDOWS\MICROS~1.NET\FRAMEW~1\V40~1.303\
```

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

```
---- File:./db/local.cmd
```

```
SET R=/r:System.Data.SQLite.dll /r:System.Configuration.dll
```

```
SET NAMEZIP=start
```

```
echo %NAMEZIP%
```

```
rm -rf _bld
rm *.exe *.log
```

```
rem
SET EXZIP=-x *.exe -x *.eRr -x *.log
```

```
---- End Of File:./db/local.cmd
```

Файл выполнения тестов приложения:

```
---- File:./db/test.cmd
```

```
rem версия приложения
```

```
app -ver >.app.txt
rem подсказка по юниттесту
app -? >>.app.txt
```

rem прямое использование SQLite провайдера

```
rem DataAdapter
app -sqlite -b >>.app.txt
rem DataReader
app -sqlite >>.app.txt
```

rem используем фабрику SQLite провайдера и абстрактный провайдер

```
rem DataAdapter
app -sqlite -db -b >>.app.txt
rem DataReader
app -sqlite -db >>.app.txt
```

rem используем абстрактный провайдер

```
rem DataAdapter
app -db -b >>.app.txt
rem DataReader
app -db >>.app.txt
```

rem добавить записи в таблицу

```
app -i <food_types_add.csv >>.app.txt
```

rem удалить добавленные записи

```
app -d >>.app.txt
```

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

Естественно в всех трех случаях работы результаты должны быть одинаковые:

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

```
*** using SQLite provider only
```

```
#0:id;
#1: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;
total time 00:00:00.2500000
```

```

*** using SQLite provider only
fields number: '2'
coll/name: '1'/'id', col2: 'Bakery', val column: '1'
coll/name: '2'/'id', col2: 'Cereal', val column: '2'
coll/name: '3'/'id', col2: 'Chicken/Fowl', val column: '3'
coll/name: '4'/'id', col2: 'Condiments', val column: '4'
coll/name: '5'/'id', col2: 'Dairy', val column: '5'
coll/name: '6'/'id', col2: 'Dip', val column: '6'
coll/name: '7'/'id', col2: 'Drinks', val column: '7'
coll/name: '8'/'id', col2: 'Fruit', val column: '8'
coll/name: '9'/'id', col2: 'Junkfood', val column: '9'
coll/name: '10'/'id', col2: 'Meat', val column: '10'
coll/name: '11'/'id', col2: 'Rice/Pasta', val column: '11'
coll/name: '12'/'id', col2: 'Sandwiches', val column: '12'
coll/name: '13'/'id', col2: 'Seafood', val column: '13'
coll/name: '14'/'id', col2: 'Soup', val column: '14'
coll/name: '15'/'id', col2: 'Vegetables', val column: '15'
total time 00:00:00.2656250

```

```

*** using SQLite fabric && abstract providers
#0:id;
#1: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;
total time 00:00:00.2500000

```

```

*** using SQLite fabric && abstract providers
fields number: '2'
coll/name: '1'/'id', col2: 'Bakery', val column: '1'
coll/name: '2'/'id', col2: 'Cereal', val column: '2'
coll/name: '3'/'id', col2: 'Chicken/Fowl', val column: '3'
coll/name: '4'/'id', col2: 'Condiments', val column: '4'
coll/name: '5'/'id', col2: 'Dairy', val column: '5'
coll/name: '6'/'id', col2: 'Dip', val column: '6'
coll/name: '7'/'id', col2: 'Drinks', val column: '7'
coll/name: '8'/'id', col2: 'Fruit', val column: '8'
coll/name: '9'/'id', col2: 'Junkfood', val column: '9'
coll/name: '10'/'id', col2: 'Meat', val column: '10'
coll/name: '11'/'id', col2: 'Rice/Pasta', val column: '11'
coll/name: '12'/'id', col2: 'Sandwiches', val column: '12'
coll/name: '13'/'id', col2: 'Seafood', val column: '13'

```

```

coll/name: '14'/id', col2: 'Soup', val column: '14'
coll/name: '15'/id', col2: 'Vegetables', val column: '15'
total time 00:00:00.2500000

```

```

*** using abstract provider only
#0:id;
#1: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;
total time 00:00:00.3125000

```

```

*** using abstract provider only
fields number: '2'
coll/name: '1'/id', col2: 'Bakery', val column: '1'
coll/name: '2'/id', col2: 'Cereal', val column: '2'
coll/name: '3'/id', col2: 'Chicken/Fowl', val column: '3'
coll/name: '4'/id', col2: 'Condiments', val column: '4'
coll/name: '5'/id', col2: 'Dairy', val column: '5'
coll/name: '6'/id', col2: 'Dip', val column: '6'
coll/name: '7'/id', col2: 'Drinks', val column: '7'
coll/name: '8'/id', col2: 'Fruit', val column: '8'
coll/name: '9'/id', col2: 'Junkfood', val column: '9'
coll/name: '10'/id', col2: 'Meat', val column: '10'
coll/name: '11'/id', col2: 'Rice/Pasta', val column: '11'
coll/name: '12'/id', col2: 'Sandwiches', val column: '12'
coll/name: '13'/id', col2: 'Seafood', val column: '13'
coll/name: '14'/id', col2: 'Soup', val column: '14'
coll/name: '15'/id', col2: 'Vegetables', val column: '15'
total time 00:00:00.2656250

```

```

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

```

3.6.1.3 Выполнение параметризованного оператора редактирования (*ExecuteScalar*)

Для редактирования таблицы при помощи нескольких операторов сделаем параметризованный оператор *insert* при помощи которого вставим в таблицу несколько записей, прочитанных со стандартного ввода:

```

---- File:./db/3ins.cs

```

```

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

partial class test
{
    public static void ins(string scnn){
        Console.WriteLine ("\n *** using SQLite fabric && abstract providers");
        using (SQLiteFactory dbF = new SQLiteFactory()){
            using (DbConnection cnn = dbF.CreateConnection()) {
                cnn.ConnectionString = scnn;
                cnn.Open();
                string s = "";
                int n = 0;
                string[] separator = {" , "};
                for (int i = 1;(s = Console.ReadLine()) != null;i++)
                {
                    string[] sint = s.Split(separator,
                        StringSplitOptions.RemoveEmptyEntries);
                    if (sint.Length >= 2) {
                        if (int.TryParse (sint[0], out n)){
                            // есть строка и целое число
                            // можно выполнять инсерт
                            DbCommand cmd = cnn.CreateCommand();
                            cmd.CommandText =
                                // выполнить вставку и вернуть id
                                "INSERT INTO food_types Values(@id, @nm); SELECT last_insert_rowid(";
                            DbParameter id = cmd.CreateParameter();
                            //http://stackoverflow.com/questions/7113667/creating-dynamic-sql-dbparameter-values
                            id.ParameterName = "@id";
                            id.Value = n;
                            id.DbType = DbType.Int32;
                            cmd.Parameters.Add(id);
                            // первый параметр добавлен
                            DbParameter nm = cmd.CreateParameter();
                            nm.ParameterName = "@nm";
                            nm.Value = sint[1];
                            nm.DbType = DbType.String;
                            cmd.Parameters.Add(nm);
                            // второй параметр добавлен
                            int rc = Convert.ToInt32(cmd.ExecuteScalar());
                            Console.WriteLine (
                                "\n *** line #{0}/' {1}' inserted: id:{2}", i, s, rc);
                        }
                        else{
                            Console.WriteLine (
                                "\n *** line #{0}/' {1}' skipped: wrong first field", i, s);
                        }
                    }
                }
            }
        }
    }
}

```



```
        "\n *** {0} records has been deleted", rc);  
    }  
  }  
}
```

---- End Of File:./db/4del.cs

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 должен показаться слишком трудным, даже если Вы никогда ранее не использовали SQL. В текущей главе для обсуждения языка используется такой минимум понятий, чтобы внести 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.

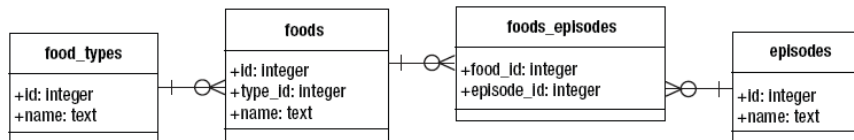


Рис. 4.1: Схема БД продуктов сериала

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 изменения 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, чтобы 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.

```
select id from foods where name='JuJyFruit';
```

select	id	from	foods	where	name='JuJyFruit';
verb	subject				predicate

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 чизбургера. 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 гамбургер с перцем-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 чизбургер, в то время как императивному 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='jalapenos'
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, состояются 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 х. Примеры:

```
x'01'
X'offf'
x'oFoEFF'
X'ofoeffab'
```

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 F00;
```

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, чтобы 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). Таким 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. Этот термин используется, чтобы отличать таблицы созданные оператором *create table* от системных таблиц или других объектов, подобных таблицам (представлений).

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 специальный 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, чтобы 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

Структуру 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) );
```

Чтобы 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 подязык **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 среде 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 операций (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 группирования к операции основной проекции. Внешнее соединение расширяет возможности операции соединения и позволяет извлекать дополнительную информацию 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 является 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.

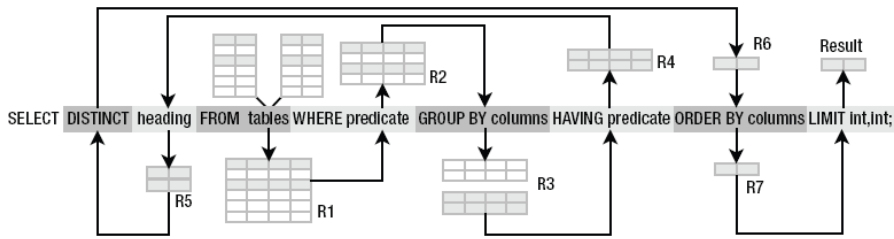


Рис. 4.3: Фразы оператора *select* в SQLite

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, представлений и подзапросов (представленных переменной *tables* на рис. 4.3), разделенных запятыми. Более чем одна таблица (представление 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.

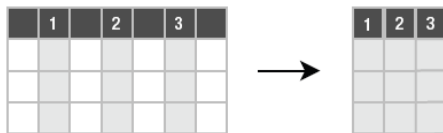


Рис. 4.4: Проекция

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*.

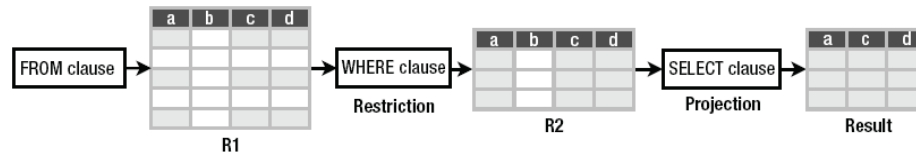


Рис. 4.5: Ограничение и проекция во фразе *select*

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. Рассмотрим 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 (отношение - результат). 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).

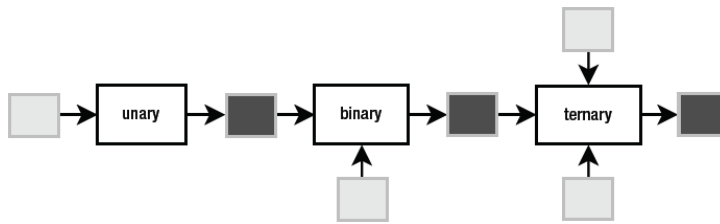


Рис. 4.6: Унарные, бинарные и тернарные операции могут образовывать конвейер

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
<i>IN</i>	принадлежит
<i>AND</i>	логическое и
<i>OR</i>	логическое или
<i>IS</i>	логическая эквивалентность
<i>LIKE</i>	подобие строк
<i>GLOB</i>	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 загогоулина, 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 выполняться для любой записи, выбранной в ответ.

Замечание переводчика

В документации по SQLite операции `&` и `|` относятся к математическим, хотя в книге названы логическими. Они, в отличие от *AND* и *OR*, выполняются побитово. Например,

```

sqlite3> select 7 | 0;

```

возвращает 7, то есть, три установленных бита. А оператор

```

sqlite3> select 7 OR 0;

```

возвращает 1.

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, чтобы 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 (%) обозначает образец (в русской литературе - регулярное выражение), которому подходит последовательность любых символов в строке, включая пустую. Символ подчеркивания (__) означает регулярное выражение, которому подходит любой одиночный символ из строки. Алгоритм, подбирающий подстрочку для символа процента, является 'жадным', то есть, из нескольких подходящих вариантов выбирается максимально 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 операцию поиска файлов в операционных системах Unix или Linux. В операции в качестве спецсимволов используются символы звездочка (*) и подчеркивание (_), и подбор строчек под регулярное выражение выполняется с учетом регистра. 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 упорядочение 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 чтобы 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 ту же функцию. В качестве входных параметров 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 выражениях 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, чтобы посчитать количество записей таблицы *foods*, относящихся к типу номер 1 ('baked goods', с *type_id = 1*), используйте групповую функцию *count()* так, как показано ниже:

```
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, чтобы 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 функций, но хорошо бы помнить, что программный интерфейс C SQLite 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 *group by* слегка отличается от остальных фраз оператора *select*, поэтому требуется напрячь воображение глядя на рисунок 4.7 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*. 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.

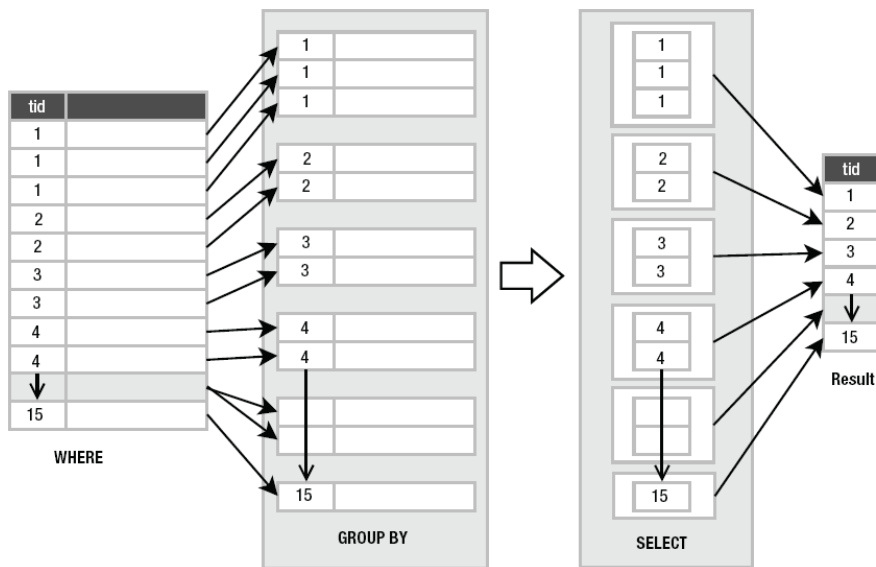


Рис. 4.7: Работа фразы group by

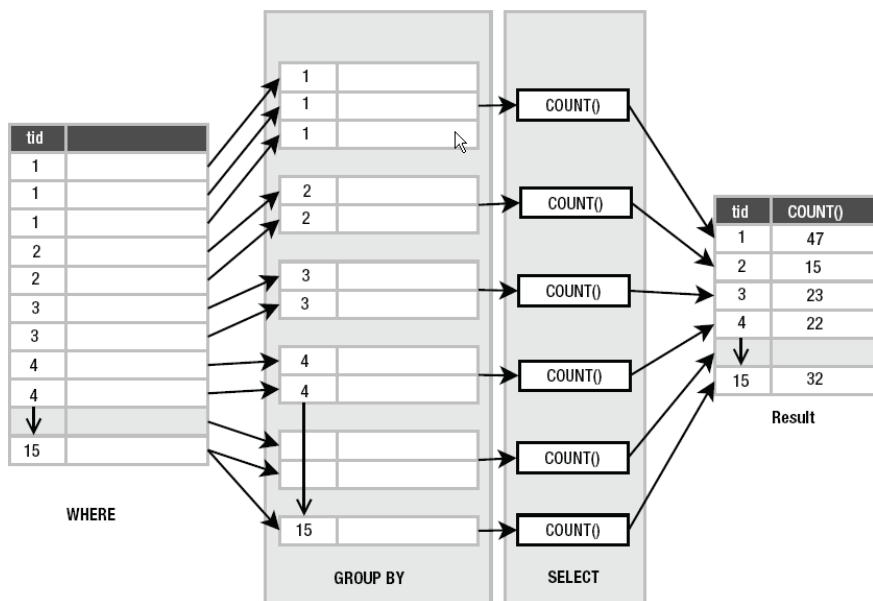
В случае, когда используется *group by* операции агрегирования применяются к каждой группе отдельно (собственно поэтому и называются групповыми функциями), при этом каждая группа, как бы, собираются в отдельную запись отношения-результата. Применим функцию *count()* для подсчета количества записей в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

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 (Chicken/Fowl, птицы) – 23, и

Рис. 4.8: Операции агрегирования и *group by*

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, используя *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.

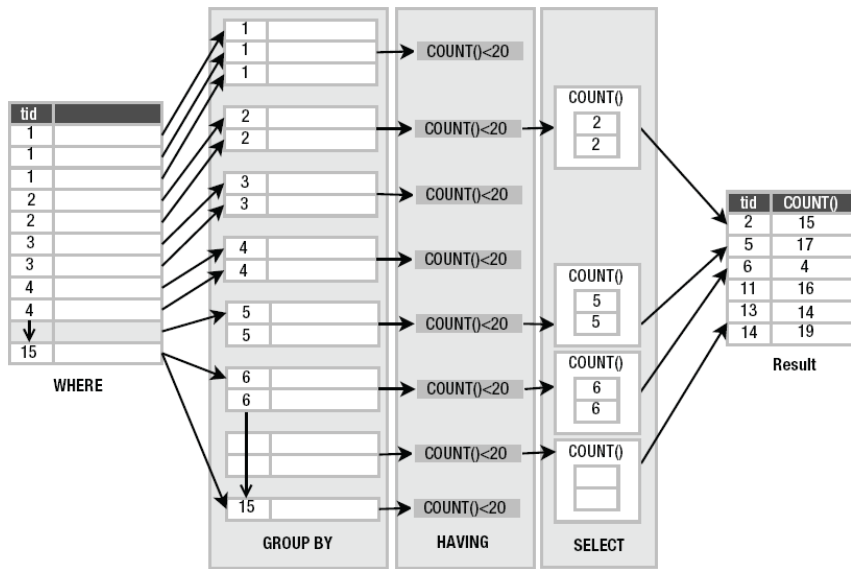


Рис. 4.9: Выборка над группами при помощи *having*

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, чтобы дать возможность эту группу увидеть.

Значит, фразы *group by* и *having* предоставляют дополнительные возможности для выборки. *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:

```
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 ни печально, но он не имеет смысла. Все поля фразы *select*, не используемые в операции агрегирования, необходимо перечислять во фразе *group by*. И только такие SQL - операторы нужно использовать.

4.4.7 Удаление повторяющихся записей

Фраза *distinct* удаляет все повторяющиеся записи из отношения - результата фразы *select*. Её можно использовать, чтобы 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 фразы *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 операции соединения невозможно представить работу с несколькими таблицами (или отношениям), с неё начинает свою работу оператор *select*. Результат соединения является входом для последующих операций фильтрации из оператора *select*.

Рассмотрим пример, чтобы лучше 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*, и поле *id* называется первичным ключом (*primary key*) таблицы *foods_types*. В свою очередь, поле *foods.type_id*, вследствие связи между таблицами, называется внешним ключом (*foreign key*), оно содержит (или ссылается на) значения из первичного ключа 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, которое содержит более детальную информацию. Например, можно в отношении 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* результата следует поле *food_types.name*. Каждая запись из таблицы *foods* связана с записью с таблицей *food_types* при помощи связи *foods.type_id* → *foods_types.id* (см. 4.10)

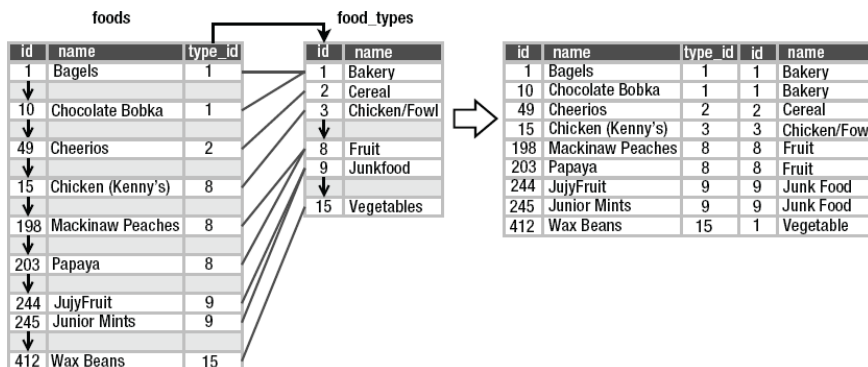


Рис. 4.10: Соединение таблиц *foods* и *food_types*

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 среди имен всех таблиц оператора *select*, то СУБД разберется из какой таблицы поле. В противном случае СУБД не сможет понять какое именно поле подразумевается 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.

Таким образом, чтобы соединить таблицы, СУБД подбирает соответствующие записи. Для каждой записи из первой таблицы СУБД найдет все записи из второй таблицы, которые имеют такие же значения в сравниваемых полях и включает полученные строки в результат. Так в текущем примере фраза *from* строит составное отношение, соединяя строки 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, чтобы выбрать элементы, принадлежащими обоим 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.

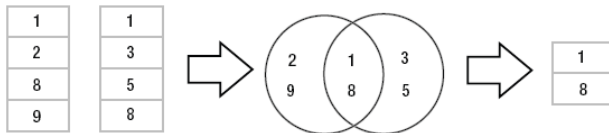


Рис. 4.11: Пересечение множеств

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

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?

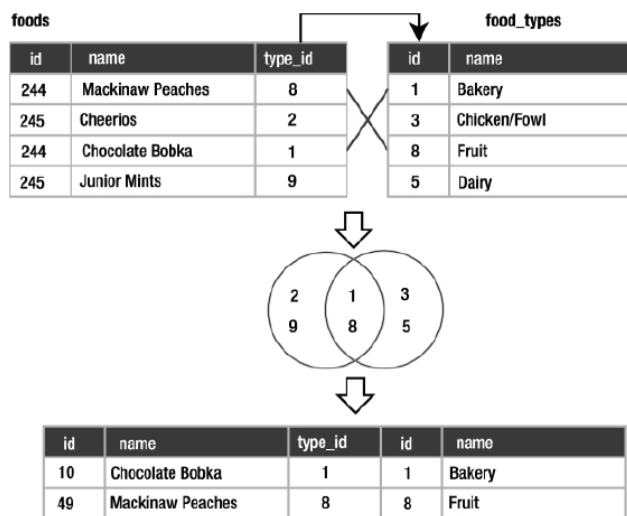


Рис. 4.12: Пересечение и соединение таблиц

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 соединения, 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 естественное или 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 соединение 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 использовать 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 соединения – 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 *select id*, 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 продуктах. 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 собой, 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 уменьшает вероятность перепутывания синонимов с именами таблиц и выражений.

4.4.10 Подзапросы

Подзапросом называется оператор *select* внутри другого оператора *select*. 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 бинарной операции *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

В этом операторе, при помощи фраз *order by* и *limit*, создается список из 10 наиболее часто встречающихся блюд, с хотдогами во главе. 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 примере называется 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 представлением 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. Очень часто в языке SQL 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* берет два различных отношения *A* и *B* и соединяет их в одно, содержащее все неповторяющиеся различные записи из обеих. 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, появившихся 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*, чтобы 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 удастся выразить 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, возвращаются 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 выполнит 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 ветки *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 согласиться с этим утверждением. Приведем несколько основных правил и идей, которые необходимо узнать, чтобы научиться пользоваться этим термином.

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

<i>x</i>	<i>y</i>	<i>x AND y</i>	<i>x OR y</i>
<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>
<i>true</i>	<i>false</i>	<i>false</i>	<i>true</i>
<i>true</i>	<i>NULL</i>	<i>NULL</i>	<i>true</i>
<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>
<i>false</i>	<i>NULL</i>	<i>false</i>	<i>NULL</i>
<i>NULL</i>	<i>NULL</i>	<i>NULL</i>	<i>NULL</i>

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

Второе, для определения наличия или отсутствия значения *NULL* используются операции *is NULL* и *is NOT NULL*. Попытка использовать любую другую операцию, такую как равно, не равно, больше чем и так далее, может вызвать удивление, 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* в других РСУБД и в языках программирования вероятно создаст проблемы.

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 пустого 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 особая 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 И так

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 создание 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

Чтобы 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 сгенерировало 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 чтобы увидеть, какое именно значение было использовано для последней вставки.

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. Чтобы 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, чтобы 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. Если 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 поля 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 BOVKA';
```

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 и 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 вводятся 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 действия 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 некотором 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 убедиться, 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.

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 ли в таблице 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 сослаться при помощи двух синонимов: *_rowid_* и *oid*. SQLite всегда автоматически генерирует умолчательные 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 для этого поля. На деле, такое поле будет еще одним синонимом для *rowid*. Все такие поля будут ссылаться на одно и то же значение. Так как SQLite использует шестидесятичетырехбитные целые со знаком, то максимально значение для такого поля равно 9,223,372,036,854,775,807.

Даже если удастся достичь этой границы, SQLite просто начнет подбирать уникальные значения, которые еще отсутствуют в таблицы. При удалении записей из таблицы, значения в первичном ключе освобождаются и могут быть использованы в последующих вставках. Как следствие, получаем, что такие значения не обязаны быть упорядочены в возрастающем порядке.

Внимание

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

Если хочется, чтобы SQLite использовал уникальные, автоматически генерируемые значения для первичного ключа на время жизни таблицы и не брал значения меньше максимального в таблице (not fill in the gaps - не заполнял разрывы), то нужно добавить ключевое слово *autoincrement* после фразы *integer primary key*.

В рассмотренных выше примерах было вставлено две записи в таблицу *contacts*. Нужно заметить, что 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 дополнение 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 абсолютного 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 ключом 10. Первая попытка вставить запись с первичным ключом равным 9 выполняется успешно. 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 *autoincrement* 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 принадлежать 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 требования, 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].

Прямо 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 формате 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, чтобы 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 *NOT NULL*. 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*. Заметим, что для проверки целостности данных можно использовать триггер, причем последний имеют больше возможностей. Если окажется невозможным записать нужную функциональность ограничением *check*, попробуйте использовать триггер. Позднее в этой главе, в разделе [5.2.6](#) они будут обсуждаться.

5.2.2.4 Внешние ключи

SQLite поддерживает понятие такое понятие реляционной теории БД как ссылочная целостность

примечание

При определении четырех видов целостности выше, использовался термин 'referential 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. Классические 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 *create table* (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, чтобы любое значение поля *type_id* всегда определяло существующий тип блюда, то оператор *create table* 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 дочерней таблицы устанавливается в *NULL*, если значение в родительской таблице удаляется.

set default

любое значение из дочерней 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 дочерней таблице. Если удаляются записи в родительской таблице, то должны удаляться записи дочерней 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 дочерняя.

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 транзакции.

5.2.2.5 Сортирующие последовательности (*collation*)

Сортирующие последовательности задают способ сравнения текстов. Различные сортирующие последовательности приведут к различным результатам 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 различные кодировки, 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 блока 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 цифр 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;

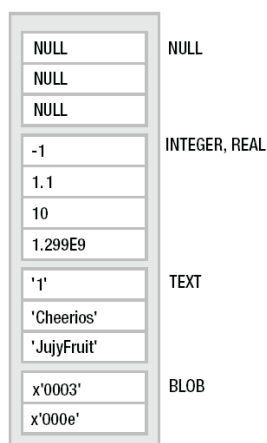
rowid      x          typeof(x)
-----
1          3.142     real
2          3.142     text
3          3142      integer
4          1B        blob
5          NULL      null
```

Пример влечет несколько вопросов. Каким образом величины в полях сортируются и сравниваются? Как сортируются поля с целыми, вещественными, текстовыми, блобами и отсутствующими значениями? 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 имеют одинаковое значение класса. Величины из *integer* и *real* сравниваются численно.
- Значения класса памяти *text* считаются 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, чтобы 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 Представления

Представления (view, иногда называются обзорами) 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 представления похожи на настоящие таблицы, они ими не являются. Содержимое настоящих таблиц представляет собой настоящие данные, в то время, как содержимое представления динамически генерируется во время обращения к нему. Синтаксис для создания представления следующий:

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

Название представления задается текстом *name*, а определяется оператором *select – stmt*. В результате оно будет выглядеть, как таблица с названием *name*. Представьте, что существует запрос, который требуется часто выполнять. Представление - это средство, которое позволяет избежать 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 представления. 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 представление *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 представлений генерируется динамически. Таким образом, каждый раз, при обращении к *details*, выполняется SQL оператор ассоциированный с представлением, который создает результат, основываясь на данных в БД на момент времени выполнения. Некоторые особенности представлений связанные с безопасностью, доступные в других СУБД, вообще говоря, отсутствуют в SQLite . А некоторые доступны (при помощи свойств управления операторами) будут обсуждаться в следующих главах.

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

```
drop view name;
```

Название удаляемого представления задается текстом *name*;

Обновляемые представления

Реляционная модель требует обновляемые представления. Это такие представления, данные в которых могут быть изменены. Например, выполнение операторов *insert* или *update* на представлении, должно приводить к соответствующим изменениям в содержании таблиц. Такие возможности не поддерживаются в SQLite . Однако, если использовать триггеры, то можно создать нечто, похожее на обновляемые представления. В секции [5.2.6](#) будут обсуждаться технические подробности их использования.

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, основанные 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 строиться 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 неиспользования 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. Должен 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 (на 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 некоторым 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 функцию 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, которые 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 представления

Как упоминалось выше, при помощи триггеров можно создавать нечто, вроде обновляемых представлений. Идея состоит в том, что для представления создается триггер на обновление. В SQLite триггеры для представлений могут использовать ключевое слово *instead* в своем определении. Создадим представление, соединяющее 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 ключом. Нужно заметить, что для всех полей представления использовались синонимы. Это

требуется, чтобы в теле будущего триггера различать поля *id* и *name* из обеих таблиц. Далее, запишем триггер для представления:

```
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 возьмет 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 представление.

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 безвозвратно 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 оператора *commit*. Оператор *commit* завершает выполнение всех операторов с начала транзакции, фиксирует транзакцию. Подобно ему *rollback* отменяет действие всех операторов с начала транзакции. Область видимости транзакции это несколько операторов SQL, которые вместе либо завершаются и фиксируются в БД (коммитятся) либо откатываются (Начинаются с *begin*, 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 транзакцией. 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 завершённый 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 позволяют расширить гибкость транзакций. Точку сохранения можно указать среди несколько SQL операторов таким образом, что откат будет выполняться не до начала транзакции, а до указанной точки. Как показано в следующем примере, чтобы создать точку сохранения требуется просто написать оператор *savepoint* с произвольным 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 Политика 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 возможность 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* принимает 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 задают весь спектр чувствительности к ошибкам: от наиболее терпимого способа - *replace*, до наиболее жесткого - *rollback*. Способы разрешения конфликтов определяются в следующем 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, чтобы 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 так и остаются. 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 годиться те же 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: регистрация записи с упреждением

Начиная с версии 3.7.0 в SQLite будет использоваться регистрация записи с упреждением (write-ahead logging (WAL)). Придется изменить поведение транзакций и блокировок чтобы 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 выделяется пять различных состояний блокировок: *unlocked* - незаблокированное, *shared* - разделяемое, *reserved* - резервирования, *pending* - отложенное и *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 незаблокированном состоянии блокировки не требуется.

Незаблокированное 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 находиться 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. Чтобы 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.

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.

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

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 сессия 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 транзакцию 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. Транзакции `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 транзакции и блокировки будут обсуждаться в главе 6

5.4 Администрирование БД

К администрированию 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 текущей 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 показано 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 индексов. 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 использованию 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 несколько групп. Некоторыми можно пользоваться 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. Чтобы 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, чтобы хранить изменения передаваемые в БД. Как описывалось в разделе 5.3.3 сессия в этот момент находится в состоянии резервирования (и имеет 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     0     1
1    type_id   integer   0     0     0
2    name      text      0     0     0
```

5.4.3.3 Перенос изменений в файл БД

Обычно SQLite фиксирует (commit) все изменения на жестком диске 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 используется прагма *synchronous*. 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 *synchronous* 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*, чтобы 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 (*little-endian*) и UTF-16be (*big-endian*).

Чтобы 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 трассировки 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, представления, 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 В-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 приводятся 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 ключом к пониманию того, как SQLite получает доступ к данным и отвечает на запрос. Он проливает свет на то, когда и как используются индексы, на порядок соединения таблиц и оказывает неоспоримую пользу для разрешения трудностей с медленно работающими запросами.

5.5 Итак

Может SQL и простой язык для использования, но уже рассмотренный диалект от SQLite потребовал две главы для введения основных понятий. Этому 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 познакомиться с подробностями выполнения SQLite -ом этих операторов SQL. Для этого полезно было бы почитать главу 6, которая является введением в программный интерфейс SQLite и объяснением как выполняется SQL с точки зрения программного интерфейса.

Глава 6

ДИЗАЙН И КОНЦЕПЦИИ SQLite

Эта глава описывает базис для последующих глав, каждая из которых будет фокусироваться на различных аспектах программирования для SQLite . Она апеллирует к вещам, которые должен знать программист, использующий SQLite в своих приложениях. Эти знания помогут понять не только программный интерфейс SQLite , но так же его архитектуру и реализацию независимо от языка программирования: все равно будет ли это родной для SQLite C, толи язык разработки скриптов. Вооруженный этими знаниями разработчик будет лучше подготовлен для создания более быстрых и надежных приложений, избегающий потенциальных трудностей, связанных с блокировками и неожиданными ошибками. Зная как SQLite работает относительно кода разработчика, можно быть более уверенным что задача решается с правильной начальной позиции.

Нет нужды просматривать исходный код SQLite , чтобы понять тонкости его использования равно как и нет нужды уметь программировать на C. Концепции и дизайн SQLite очевидны и легки для использования. Есть лишь несколько фактов, которые требуется знать. Эта глава излагает главные концепции и компоненты, при помощи которых можно выстроить необходимый уровень понимания SQLite . Знание о работе программного интерфейса является ключем к пониманию SQLite . Таким образом, эта глава начинается введением в программный интерфейс, иллюстрирует его основные структуры, его общий дизайн и его главные функции. Также уделяется внимание основным подсистемам SQLite , играющим важную роль в обработке запросов.

Кроме знания какая функция выполняет какую работу, уделяется внимание тому, как программный интерфейс функционирует в терминах транзакций. Все, включая базу данных SQLite , рассматривается в контексте транзакций. Затем приходится заглянуть ниже программного интерфейса, чтобы понять как транзакции выполняются в терминах блокировок. Блокировки могут создавать трудности, если не понимать как они выполняются. Понимание блокировок позволяет не только избегать потенциальных проблем конкуренции, но также оптимизировать выполняемые запросы, кон-

тролируя как приложение использует блокировки.

Наконец, требуется понимание того, как это все вместе используется для разработки приложений. Последняя часть главы обсуждает все три темы - программный интерфейс, транзакции и блокировки - вместе и рассматривает различные примеры хорошего и плохого кода. Там уточняются сценарии, которые могут приводить к трудностям и предоставляется некоторый взгляд, позволяющих избегать их.

Применяя все изложенное в правильном порядке, можно уверенно добиваться хороших результатов вне зависимости от языка (все равно С или любой другой), на котором используется программный интерфейс.

6.1 Программный интерфейс

Функционально, программный интерфейс (иногда API: application program interface) SQLite можно разделить на две части: базовый программный интерфейс и расширенный. Базовый интерфейс состоит из функций, которые используются для выполнения основных операций БД: соединение с БД, выполнение операторов SQL и получения результатов. Кроме того, он содержит дополнительные функции, которые полезны при выполнении таких задач, как форматирование строк, управления операторами, отладка и обработка ошибок. Расширенный программный интерфейс предлагает различные возможности для создания дополнений, определяемых пользователем, которые интегрируются в язык SQL диалекта SQLite .

6.1.1 Основные структуры данных

Как было показано в главе 2 , SQLite состоит из нескольких компонент - парсера, лексического анализатора, виртуальной машины и так далее. С точки зрения программиста, главными компонентами, о которых надо помнить, являются соединения, операторы, В-дерево и менеджер страниц (pager). На рис. 6.1 изображены взаимоотношения между перечисленными компонентами. Эти объекты описывают три принципиальные вещи, которые необходимо знать про SQLite , чтобы разрабатывать хорошие приложения: программный интерфейс, транзакции и блокировки. Технически, В-дерево и менеджер страниц не является частью программного интерфейса, они находятся за его границами. Но они играют критическую роль в транзакциях и блокировках. Более подробно участие В-дерева и менеджера страниц в транзакциях и блокировках будет исследовано в этой секции ниже и в разделе 6.3 'Транзакции'.

6.1.1.1 Соединения и Операторы

Двумя фундаментальными структурами данных в программном интерфейсе, которые ассоциированы с обработкой запросов, являются соединения и операторы. В добавлениях к большинству языков (in most language extensions -

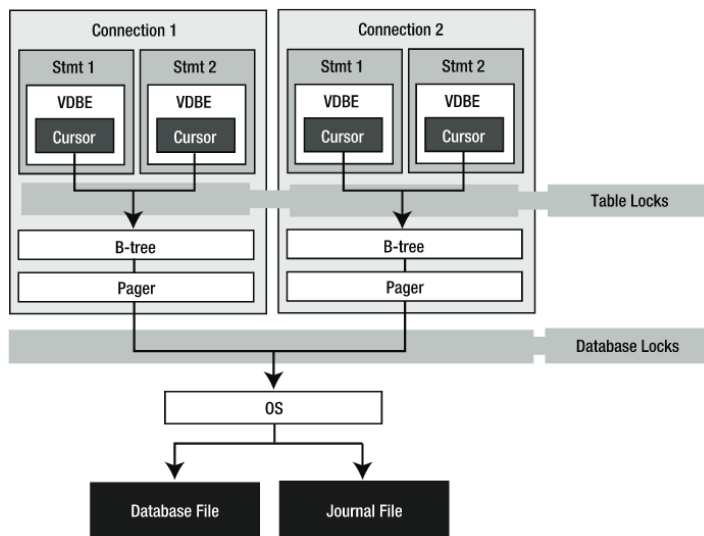


Рис. 6.1: Модель объектов API языка C для SQLite

видимо, библиотека для доступа к SQLite) для SQLite можно увидеть объекты для соединений и для операторов, которые используются для выполнения запросов. В программном интерфейсе для C эти структуры напрямую ссылаются на хендлеры *sqlite3* и *sqlite3_stmt* соответственно. Практически любое действие при помощи программного интерфейса выполняется с использованием этих двух структур.

Структура соединения предоставляет одиночное соединение с БД, равно как и контекст для одиночной транзакции. Операторы создаются на основе соединений. Таким образом, для каждого оператора существует соединение, ассоциированное с ним. Структура оператор представляет собой одиночный, откомпилированный оператор SQL. С точки зрения разработчиков SQLite , оператор представляется в форме байт кода для VDBE - программы, которая должна выполнять его работу. Такие программы содержат в себе все необходимое. Она содержит ресурсы для пошагового запоминания состояния программы VDBE по мере выполнения, курсор В-дерева, указывающего на запись на диске, равно как и другие вещи, такие как связанные параметры. Связанные параметры будут обсуждаться позже в секции 6.1.2.3 'Использование Параметризованного SQL'. Хотя команды содержат множество различных вещей, о них можно думать как о курсоре, используемом в циклах для прохода через набор записей результата работы оператора SQL или как о хендлере, ссылающемся на оператор SQL.

6.1.1.2 В-дерево и Менеджер Страниц

Каждое соединение может содержать несколько объектов БД. Каждый объект БД имеет один объект называемый В-дерево, который в свою очередь, имеет один объект, называемый менеджер страниц.

Операторы используют В-дерево и менеджер страниц чтобы читать дан-

ные из БД и записать данные в БД. Операторы, которые читают БД, делают это последовательно через В-дерево, используя курсоры. Курсоры последовательно просматривают записи, которые хранятся группами в виде страниц. Так как курсор перебирает записи, то ему также приходится перебирать страницы. Чтобы курсор заглянул в очередную страницу, её надо предварительно загрузить с диска в оперативную память. Эту работу выполняет менеджер страниц. Как только В-дерево нуждается в конкретной странице из БД, оно обращается к менеджеру страниц, чтобы он доставил её с диска. Менеджер страниц загружает страницу в кэш страниц, играющий роль буфера памяти. Как только страница оказывается в кэше страниц, В-дерево и связанный с ним курсор могут добраться к записям внутри страницы.

Если курсор меняет данные страницы, то менеджер страниц должен предпринять меры чтобы сохранить исходную страницу для выполнения отката транзакции. То есть, менеджер страниц ответственен за чтение из БД; запись в БД; поддержание кэша памяти или страниц; управление транзакциями. В дополнение к этому, он управляет блокировками и восстановлением после краха. Все это позднее раскрывается в разделе 6.3 'Транзакции'.

Вообще говоря, нужно знать два факта про соединения и транзакции. Первый - любую операцию над БД соединение всегда выполняет в транзакции. Второй - соединение никогда не имеет более одной транзакции в один момент времени. Это означает, что все операторы, построенные для данного соединения, выполняются в одном и том же контексте транзакции. Требование выполнять более одного оператора в отдельных транзакциях влечет необходимость использовать несколько соединений, одно соединение для каждого контекста транзакции.

6.1.2 Базовый Программный Интерфейс

Как ранее упоминалось, базовый программный интерфейс занимается выполнением команд SQL. Для этого сделано несколько функций, выполняющих запросы, равно как и несколько вспомогательных функций для управления другими аспектами БД. Существует два важных метода для выполнения команд SQL: подготовленные запросы и немедленные запросы (There are two essential methods for executing SQL commands: prepared queries and wrapped queries).

Замечание переводчика

Подготовленные запросы (как и немедленные) - это не запросы, а группы функций. Во избежание путаницы, далее переводиться как группы функций.

Все операторы SQLite выполняет, в конце концов, именно группой функций подготовленные запросы. Это представляет собой трехфазный процесс включающий подготовку, выполнение и финализацию. В интерфейсе есть отдельные функции, связанные с каждой фазой. Запись и информацию о

поле из отношения - результата предоставляют функции, из фазы выполнения.

В дополнение к трехфазному методу выполнения запроса, существует две функции немедленного выполнения, которые все три фазы выполняют за единственный вызов. Они предоставляют удобный путь для выполнения оператора SQL за раз. Это только некоторые из многих различных вспомогательных функций в программном интерфейсе. В этой секции будут рассмотрены оба метода выполнения запросов вместе со связанными с ними вспомогательными функциями. Но перед этим необходимо взглянуть на процесс соединения с БД.

6.1.2.1 Соединение с БД

Соединение с БД влечет несколько больше, чем открытие файла. Любая БД SQLite -а хранится в одиночном файле файловой системы - одна БД в одном файле. Функция, которая используется для соединения или открытия БД в программном интерфейсе C называется *sqlite3_open()* и, в основном, является просто системным вызовом для открытия файла. SQLite , кроме того, может создавать БД в оперативной памяти компьютера (RAM).

Для большинства языков, для этого нужно использовать либо строчку *:memory:*, либо пустую строку в качестве названия для БД. Такая БД будет доступна только для соединения, создавшего её (БД в RAM нельзя совместно использовать с другим соединением). Более того, такая БД будет существовать только во время жизни соединения. Она удаляется из памяти после закрытия последнего.

При выполнении соединения с БД на жестком диске, SQLite открывает файл, если он существует. При попытке открытия несуществующего файла SQLite предполагает, что требуется создание новой БД. Но SQLite не создает немедленно новый файл в файловой системе. Он будет создаваться только в случае, если в новую БД будет добавлено нечто вроде таблицы, или представления или еще какогонибудь объекта БД. Если просто открыть и закрыть новую БД, SQLite не будет заморачиваться с созданием файла для неё, все равно он останется пустым.

Кроме того, существует важная причина для того, чтобы немедленно не создавать новый файл. Некоторые опции БД, такие как кодирование, размер страницы и автоуплотнение можно менять только перед созданием БД. По умолчанию, SQLite использует страницы размером в 1024 байта. При этом размер страницы можно задавать степенью двойки в диапазоне от 512 до 32768 байт. Их можно выбирать другими по причине производительности. Например, совпадение размер страницы с размером страницы операционной системы может сделать операции ввода - вывода более быстрыми. Большой размер страницы ускоряет работу приложений, которые обрабатывают большие объемы двоичных данных. Для выбора желаемого размера страницы необходимо использовать прагму *page_size*.

Кодировка - это еще один неизменяемый параметр БД. Можно выбрать кодировку для БД используя специальную прагму *encoding*, выбирая из значений UTF-8, UTF-16, UTF-16le (little endian) и UTF-16be (big endian).

Наконец, прагма *auto_vacuum* позволяет использовать автоупаковку БД. По умолчанию, SQLite не возвращает удаленные из БД страницы операционной системе. После удаления записей из БД, размер файла БД не меняется. Чтобы освободить ненужные для БД страницы, необходимо явно выполнить команду упаковки БД - *vacuum*. Возможность автоупаковки вынуждает SQLite автоматически ужимать файл БД, каждый раз когда из неё удаляются данные. Эта возможность часто очень полезна для встроенных приложений, когда ресурс памяти является критичным.

После открытия любая БД, все равно на диске или в памяти, будет доступна при помощи хендлера соединения *sqlite3*. Такой хендлер представляет собой одиночное соединение с БД. Объекты соединения из [добавлений к языкам программирования](#) абстрагируются от этого хендлера и иногда реализуют методы, которые соответствуют функциям программного интерфейса, получающие этот хендлер в качестве аргумента.

6.1.2.2 Применение Функций Подготовленных Запросов

Как утверждалось выше, метод подготовленных запросов является именно тем процессом, при помощи которого SQLite выполняет все операторы SQL. При этом, выполнение оператора SQL является трех шаговым процессом:

- Подготовка: парсер, лексический анализатор и генератор кода готовят оператор SQL, компилируя его в [байт код VDBE](#). В программном интерфейсе языка C это выполняется при помощи функции *sqlite3_prepare_v2()*, которая вызывается непосредственно из компилятора. Компилятор создает хендлер *sqlite3_stmt* который содержит байт код и другие ресурсы, необходимые для выполнения оператора и, если команда будет генерировать отношение - результат, последующего прохода по результату выполнения команды;
- Выполнение: VDBE выполняет байт код. Это выполнение является пошаговым процессом. В программном интерфейсе C каждый шаг начинается при помощи функции *sqlite3_step()*, принуждающее VDBE к следующему шагу по байт коду. Обычно первый вызов *sqlite3_step()* запрашивает блокировку некоторого типа. Тип блокировки выбирается соответственно действиям, выполняемым оператором (чтение или запись). Для оператора *select* каждое обращение к функции *sqlite3_step()* передвигает курсор хендлера к следующей записи отношения - результата. Для каждой записи в отношении функция будет возвращать SQLITE_ROW до тех пор, пока не достигнет конца отношения, после чего вернет SQLITE_DONE. Для других операторов SQL (*insert*, *update*, *delete* и так далее), первый вызов функции *sqlite3_step()* принуждает VDBE целиком выполнить оператор;
- Финализация: VDBE закрывает оператор и освобождает ресурсы. В программном интерфейсе C, это происходит при помощи функции *sqlite_finalize()*, которая приводит к прекращению программы на байт коде, освобождению ресурсов и закрытию хендлера оператора.

Каждый шаг - подготовка, выполнение, финализация - соответствует подходящему состоянию хендлера - подготовленному, активному или фи-

нализированному. Подготовленное состояние означает, что все необходимые ресурсы уже выделены и оператор готов к выполнению, однако выполнение не начато. Не была затребована ни одна из блокировок, и не будет затребовано никакой блокировки до первого вызова функции *sqlite3_step()*. Состояние оператор становится активным после первого вызова *sqlite3_step()*. В этот момент он начинает выполняться и в игру вступает блокировка одного из типов. Финализированное состояние означает, что оператор закрыт, все связанные с ним ресурсы были освобождены. На рис. 6.2 изображены эти шаги и состояния.

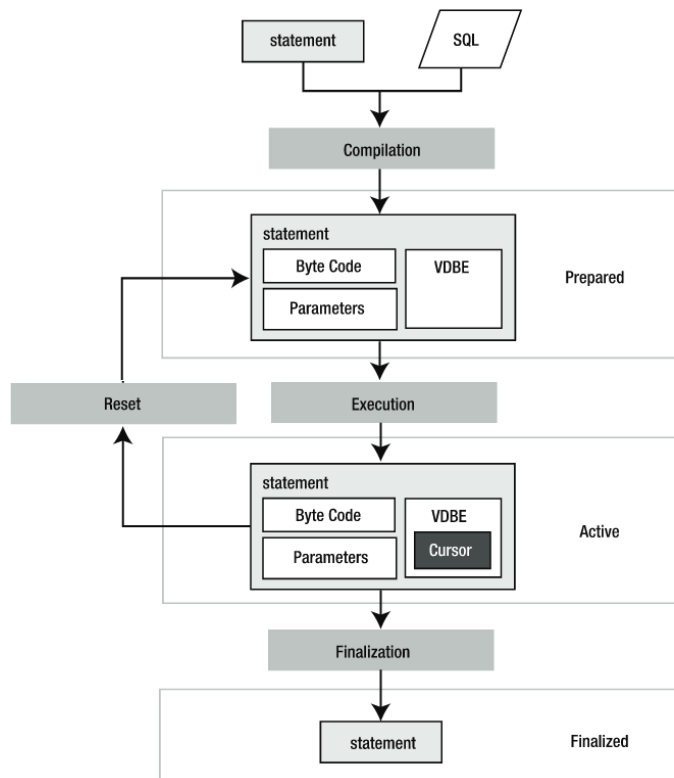


Рис. 6.2: Обработка оператора SQL

Следующий псевдокод иллюстрирует общий процесс выполнения запроса в SQLite :

```

# 1. Open the database, create a connection object (db)
db = open('foods.db')

# 2.A. Prepare a statement
stmt = db.prepare('select * from episodes')

# 2.B. Execute. Call step() is until cursor reaches end of result set.
while stmt.step() == SQLITE_ROW
    print stmt.column('name')
end
  
```

```

# 2.C. Finalize. Release read lock.
stmt.finalize()

# 3. Insert a record
stmt = db.prepare('INSERT INTO foods VALUES (...)')
stmt.step()
stmt.finalize()

# 4. Close database connection.
db.close()

```

Этот псевдокод является объектно - ориентированным аналогом программного интерфейса, подобного тому, который можно найти в языках для написания скриптов. Все методы соответствуют функциям программного интерфейса SQLite . Например, *prepare()* подразумевает вызов функции *sqlite3_prepare_v2()* и так далее. Этот пример выполняет оператор *select*, проходя через все возвращаемые строки, за которым выполняется оператор *insert*, обработанный одиночным вызовом функции *step()*.

Временное хранилище

Временное хранилище является важной частью обработки запроса. SQLite периодически нуждается в хранении промежуточных результатов, производимых в процессе выполнения оператора - например, при необходимости сортировки вследствие использования фразы *orderby* или когда записи из одной таблицы соединяются с записями другой. Тогда информация часто сохраняется во временном хранилище. Временное хранилище находится либо в оперативной памяти компьютера, либо в файле. Хотя SQLite имеет умолчания подходящие для всех платформ, сохраняется возможность контролировать как и где оно располагается. Прагма *temp_store* позволяет указывать место (оперативная память или файл) для размещения временного хранилища. При размещении временного хранилища в файле, можно использовать прагму *temp_store_directory* чтобы указать каталог для хранения файла временного хранилища.

6.1.2.3 Использование Параметризованного SQL

Операторы SQL могут содержать параметры. Параметрами называются специальные обозначения (placeholder - спецификатор места вывода, дословно - держатель места), задающие место в операторе SQL, в которое после компиляции будут подставляться значения (bound - связываемые). Следующий операторы представляют примеры параметризованных запросов:

```

insert into foods (id, name) values (?,?);
insert into episodes (id, name) (:id, :name);

```

Здесь можно увидеть две формы связываемых параметров: связываемые по порядку (positional) и именованные (named). Первая строка содержит

параметры, связываемые по порядку, вторая использует именованные параметры.

Параметры, связываемые по порядку, задаются позицией символа вопроса в тексте оператора. Первый вопрос отмечает позицию для первой величины, второй - для второй, и так далее. Именованные параметры используют имена переменных, которые начинаются с символа двоеточие. Когда `sqlite3_prepare_v2()` компилирует оператор с параметрами, она располагает спецификаторы места вывода в хендлере подготовленного оператора. Затем, перед выполнением, оператор ожидает получить значения для подстановки в эти места. Если значения для связывания не предоставляются, SQLite будет использовать значение `NULL` во время выполнения оператора.

Преимущество использования связывания параметров заключается в возможности выполнять откомпилированный оператор несколько раз. Необходимо просто связать новый набор значений с уже откомпилированным оператором и выполнить его. Это то место, где использование параметров полезнее чем финализация байт кода: оно позволяет избавиться от ненужной компиляции операторов SQL. Становится ненужной вся работа по лексическому и синтаксическому разбору оператора SQL, равно как дополнительной генерации байт кода. Повторное связывание параметров выполняется при помощи функции `sqlite3_reset()`.

Еще одно преимущество параметров заключается в ненужности экранирования символа кавычка в связываемых строчках. К примеру, процесс связывания параметров самостоятельно сконвертирует строку вроде `'Kenny's Chicken'` в строку `'Kenny"s Chicken'` - экранируя одиночную кавычку. Это помогает избегать как синтаксических ошибок, так и возможных атак на БД при помощи SQL инъекции. SQL инъекции будут рассмотрены ниже в секции `'Форматирование оператора SQL'`. Следующий псевдокод иллюстрирует основы использования связанных параметров:

```
db = open('foods.db')
stmt = db.prepare('insert into episodes (id, name) values (:id, :name)')

stmt.bind('id', '1')
stmt.bind('name', 'Soup Nazi')
stmt.step()

# Reset and use again
stmt.reset()
stmt.bind('id', '2')
stmt.bind('name', 'The Junior Mint')

# Done
stmt.finalize()

db.close()
```

Здесь, `reset()` просто удаляет связанные ресурсы, но оставляет нетронутым байт код VDBE, равно как и параметры.

Замечание переводчика

В примере, по видимому, отсутствует вызов метода `stmt.step()`

Ранее подготовленный оператор не нуждается в дополнительном вызове функции *prepare()*. Это позволяет улучшить производительность повторяемых запросов, так как полностью исключается использование компилятора.

6.1.2.4 Executing Wrapped Queries

wrapped

Как уже упоминалось ранее, есть две очень полезные вспомогательные функции, которые позволяют выполнять оператор SQL за одиночный вызов функции. Первая - *sqlite3_exec()* - спроектирована для запросов, не возвращающих данные. Вторая - *sqlite3_get_table()* - спроектирована для запросов, которые возвращают их. Во многих [добавлениях к языкам](#) можно обнаружить аналоги для обеих. Часто на первую ссылаются просто как на *exec()*, а на вторую - *get_table()*.

Функция *exec()* является быстрым и удобным способом выполнять операторы *insert*, *update*, *delete* или операторы языка определения данных (DDL) для создания и удаления объектов БД. Она работает непосредственно на соединении, получая хендлер *sqlite3* к открытой БД и строку, содержащую один или более операторов SQL. Это правда: функция *exec()* способна обработать строку с несколькими операторами SQL, которые разделяются символом точка с запятой, и выполнить их все вместе.

Для этого *exec()* выполняет синтаксический анализ строки, выделяет отдельные операторы и обрабатывает их один за другим. Она захватывает память для хендлеров операторов, подготавливает, выполняет и финализирует каждый из них. Если она получила несколько SQL операторов и один из них сбойнул, функция *exec()* прекращает работу и возвращает соответствующий код ошибки. Иначе она возвращает код успешного завершения. Следующий псевдокод концептуально иллюстрирует применение функции *exec()* в добавлении к некоторому языку.

```
db = open('foods.db')
db.exec("insert into episodes (id, name) values (1, 'Soup Nazi')")
db.exec("insert into episodes (id, name) values (2, 'The Fusilli Jerry')")
db.exec("begin; delete from episodes; rollback")
db.close()
```

Хотя можно использовать *exec()* для обработки записей, возвращаемых из оператора *select*, но это подразумевает изоциренные методы, полностью поддерживаемые только в программном интерфейсе C.

Название второй функции, *sqlite3_get_table()*, может вводить в заблуждение, ибо она не ограничивается запросами к одиночной таблице. Скорее, её имя подразумевает отношение - результат выполнения любого оператора *select*. Естественно, ей можно с таким же успехом обрабатывать соединения таблиц. Во многом, *get_table()* выполняется так же как и *exec()*, но возвращает результат - отношение. Это отношение может быть представлено различными способами, в зависимости от языка. Следующий псевдокод иллюстрирует типичное использование функции:

```

db = open('foods.db')
table = db.get_table("select * from episodes limit 10")

for i=0; i < table.rows; i++
  for j=0; j < table.cols; j++
    print table[i][j]
  end
end

db.close()

```

Важной чертой `get_table()` является то, что она предоставляет способ для выполнения запроса и получения отношения - результата за одно действие. Это означает, что чем больше отношение - результат, тем больше памяти использует функция. Не должно быть сюрпризом, что не следует её использовать для получения больших наборов данных. С другой стороны, помните, что метод подготавливаемых запросов использует память только для одной записи (в реальности для одной страницы, на которой расположена запись), что лучше подходит для прохода через большие отношения - результаты.

Отметим, что хотя эти функции предоставляют некоторое удобство, они прячут доступ к хендлеру оператора, что приводит к ослаблению контроля. Например, с ними нельзя использовать параметризованные операторы SQL. Таким образом, они не такие эффективные для повторяемых задач, которые выигрывают при использовании параметров. Кроме того, программный интерфейс предоставляет функции, которые пользуются хендлерами оператора. Эти функции предоставляют информацию про поля в отношении - результате (как про данные, так и про метаданные). Но эта информация не доступна при использовании функций немедленных запросов.

6.1.2.5 Обработка Ошибок

Приведенные примеры были слишком упрощены для иллюстрации основ обработки запросов. В реальности, необходимо уделять внимание обработке ошибок. Почти каждая из уже обсужденных функций может вызвать какую нибудь ошибку. Коды ошибок, к обработке которых нужно быть готовым, включают `SQLITE_ERROR` и `SQLITE_BUSY`. Последний из которых, ссылается на условие занятости, возникающее в случае невозможности поставить блокировку. Условие занятости обсуждается в разделе 6.3 'Гран-закции', в тоже время, детальное описание ошибок можно найти в главе 7.

Применительно к большинству ошибок, программный интерфейс при помощи `sqlite3_errcode()` предоставляет код возврата функции, вызванной перед `sqlite3_errcode()`. При помощи `sqlite3_errmsg()` можно получить подробное описание этой ошибки. Многие добавления к языкам поддерживают эти функции тем или иным способом.

Если озаботится ошибками, то каждый вызов функции в предыдущих примерах должен проверять код возврата приблизительно в таком виде:

```

# Check and report errors
if db.errcode() != SQLITE_OK
  print db.errmsg(stmt)
end

```

Вообще говоря, обработка ошибок не слишком сложна. Способ обработки зависит от того, что точно требуется сделать. Наиболее простой из них, такой же как и в любом другом программном интерфейсе - аккуратно читать документацию про используемую функцию и возвращаемые ей коды ошибок.

6.1.2.6 Форматирование SQL Операторов

Еще одна удобная и приятная функция, которую могут предлагать некоторые [добавления к языкам](#), называется `sqlite3_mprintf()`. Она является аналогом стандартной функции C - `sprintf()`. Её спецификаторы места вывода, специфические для SQL, могут быть весьма полезны. Они обозначаются как `%q` и `%Q` и используются для значений, специфических для SQL. `%q` работает похоже на `%s`, и используется для вывода строк, заканчивающихся нулем. Но кроме того, спецификатор удваивает каждую одиночную кавычку, облегчая жизнь программиста и предоставляя защиту против SQL инъекции. Подробности см. в секции 'Атака при помощи SQL инъекции'. Далее, смотри пример:

```
char* before = "Hey, at least %q no pig-man.";
char* after = sqlite3_mprintf(before, "he's");
```

Величина переменной `after` равняется 'Hey, at least he's no pig-man'. Одиночная кавычка в `he's` удваивается, делая строчную константу пригодной к использованию в операторе SQL. Спецификатор `%Q` все делает точно так как `%q`, но дополнительно заключает результирующую строку в одинарные кавычки. Но, если через `%Q` выводится нулевой указатель (в смысле C), то результирующая строка будет `NULL` без одинарных кавычек. Для дальнейшей информации смотри документацию по `sqlite3_mprintf()` в справочнике по программному интерфейсу C.

Атака при помощи SQL инъекции

Приложение становится уязвимым для атаки при помощи SQL инъекции, если оно будет полагаться на вводимые данные для конструирования операторов SQL. Если невнимательно относится к фильтрованию пользовательского ввода, то для недобросовестного пользователя появляется возможность ввести дополнительный оператор SQL. К примеру, приложение использует пользовательский ввод, чтобы подставить его значение в следующий оператор SQL:

```
select * from foods where name='%s';
```

Тут спецификатор места вывода `%s` заменяется на все, что введет пользователь приложения. Если пользователь имеет информацию о БД приложения, он может ввести строку, которая драматически изменит рассматриваемый оператор SQL. Таким примером является следующая строка:

```
nothing' limit 0; select name from sqlite_master where name='%'
```

После выполнения подстановки в рассматриваемый оператор SQL, он превратится в два:

```
select * from foods where name='nothing' limit 0; select name from
sqlite_master where name='%';
```

Первый оператор не вернет ничего, а второй - вернет в приложение имена объектов из БД. Естественно, такие попытки требуют некоторых знаний о атакуемом приложении, но тем не менее они не являются невозможными. Даже коммерческие веб приложения хранят операторы SQL в своих скриптах языка JavaScript, а они имеют достаточно много подсказок о используемой БД. В рассмотренном примере, злоумышленник может вставить оператор *drop table* для каждой таблицы, найденной в *sqlite3_master*, что приведет к необходимости продираться через бекапы БД.

6.1.3 Управление Операторами

Программный интерфейс предоставляет набор возможностей для наблюдения, управления и ограничения событий, происходящего в БД (). SQLite реализует эти возможности либо в форме фильтров, либо в форме функций обратного вызова (callback functions), которые регистрируются для вызова при наступлении заданных событий. Для такой регистрации есть три hook-функции: *sqlite3_hook()* - для наблюдения за выполнением операторов *commit* на соединении; *sqlite3_rollback_hook()* - для наблюдения за откатами; *sqlite3_update_hook()* - для наблюдения за изменениями в записях, которые вызываются операторами *insert*, *update* и *delete* (operational control).

Функции обратного вызова вызываются во время выполнения приложения - во время выполнения оператора SQL. Каждая hook - функция позволяет регистрировать функцию обратного вызова для каждого соединения и так же позволяет передавать им данные, специфические для приложения. Ниже приведен пример использования функций управления операторами:

```
def commit_hook(cnx)
  log('Attempted commit on connection %x', cnx)
  return -1
end
db = open('foods.db')
db.set_commit_hook(rollback_hook, cnx)
db.exec("begin; delete from episodes; rollback")
db.close()
```

Замечание переводчика

По видимому, в примере есть описка с названием функции: *commit_hook(cnx)* и *rollback_hook(cnx)* - скорее всего, должно быть одно и тоже имя.

Значение, которое возвращает hook-функция, имеет возможность некоторым образом, зависящем от функции, менять событие в SQLite. В этом

примере выполнится откат, так как функция обратного вызова возвращает не нулевое значение.

Дополнительно, программный интерфейс предлагает очень мощную hook - функцию времени компиляции с названием *sqlite3_set_authorized()*. Она предоставляет очень тонкий контроль над практически всем, что может произойти в БД, так же как и предоставляет возможность ограничивать и доступ, и редактирование БД, таблиц и полей. Детально она будет описываться в главе 7 .

6.1.4 Использование Нитей (Потоков)

SQLite предоставляет несколько функций для использования в многопоточной среде. Начиная с версии 3.3.1, SQLite ввел уникальный режим управления с названием режим разделяемого кэша (shared cache mode), который спроектирован для встроенных многопоточных серверов. Он предоставляет возможность для одиночного потока (нити) открывать несколько соединений так, что бы они разделяли общий кэш страниц, снижая таким образом общее использование памяти сервером. Кроме того, разработана еще одна модель конкуренции, с ней добавлены функции, которые позволяют управлять памятью и тонкой настройкой сервера. Еще раз эта модель конкуренции будет обсуждаться позже, в секции 'Режим разделяемого кэша' и, более детально, в главе 7 .

6.2 Расширенный Программный Интерфейс

Расширенный программный интерфейс в программном интерфейсе SQLite C предлагает поддержку определяемых пользователем функций, групповых функций и сортирующих последовательностей. Функция, определяемая пользователем, разрабатывается на языке C или любом другом языке и может вызываться из запросов языка SQL в SQLite . Если приложение использует программный интерфейс C, то функция надо разрабатывать на C или C++. В добавлениях к другим языкам, функция, определяемая пользователем, должна использовать язык, для которого разработано добавление.

Функции, определяемые пользователем, надо регистрировать посессионно, так как она хранится в памяти приложения. То есть, они не сохраняются в БД, подобно хранимым процедурам в других РСУБД. Они хранятся в приложении. Когда приложение (или скрипт) начинает работать, оно ответственно за регистрацию пользовательского расширения для каждого соединения, которое должно использовать расширение.

6.2.1 Создание Функций, Определяемых Пользователем

Разработка определяемой пользователем функции является двухшаговым процессом. Сначала пишется обработчик, который делает работу, которую предполагается использовать из языка SQL. Затем обработчик регистрируется с некоторым именем для языка SQL, списком формальных параметров и указателем на обработчик.

Предположим, требуется создать специальную функцию для SQL с названием `hello_newman()`, которая возвращает строку 'Hello Jerry'. Сначала требуется написать функцию на языке C, вроде следующей:

```
void hello_newman(sqlite3_context* ctx, int nargs, sqlite3_value** values)
{
    /* Create Newman's reply */
    const char *msg = "Hello Jerry";

    /* Set the return value.*/
    sqlite3_result_text(ctx, msg, strlen(msg), SQLITE_STATIC);
}
```

Для читателей не знакомых с C заметим, что этот обработчик просто возвращает заданную строку. Далее, требуется её зарегистрировать при помощи `sqlite3_create_function()` (или, в случае [добавления к другому языку](#), при помощи её эквивалента):

```
sqlite3_create_function(db, "hello_newman", 0, hello_newman);
```

Первый фактический параметр `db` - это соединение с БД. Второй - имя функции, как предполагается её называть в языке SQL, третий означает, что у функции нет формальных параметров (Значение -1 на этом месте означает, что она может обрабатывать любое число формальных параметров). Последний параметр - это указатель на выше приведенную функцию C `hello_newman()`. Она и будет вызываться, в случае использования `hello_newman()` на языке SQL.

После такой регистрации, SQLite знает, что встретив в SQL запросе функцию `hello_newman()`, он должен вызвать сишную функцию `hello_newman()` для получения результата. То есть, следующий запрос

```
select hello_newman()
```

в приложении, выполнившем описанную регистрацию, вернет одну запись с одним полем, которое содержит текст 'Hello Jerry'. Как упоминалось в главе 5 определяемые пользователем функции являются удобным способом для создания специальных ограничений целостности, встраиваемых во фразу *check*.

6.2.2 Создание Групповых Функций, Определяемых Пользователем

Групповые функции (agregates) это функции, которые применяются ко всем записям в отношении - результате и по ним всем вычисляют некоторое интегральное значение. Примерами стандартных групповых функций из языка SQL являются функции `sum()` (сумма), `count()` (количество) и `avr()` (среднее).

В SQLite , при помощи процесса из трех шагов можно создать дополнительную групповую функцию. Таким образом необходимо:

- зарегистрировать групповую функцию;
- разработать функцию, которая вызывается для каждой записи в отношении результате;

- разработать финальную функцию, которая вызывается после обработки всех записей. Она позволяет вычислять окончательное значение и выполняет очистку памяти.

Проиллюстрируем сказанное примером создания групповой функции *pysum* для вычисления суммы для [добавления к языку Python](#):

```
connection=apsw.Connection("foods.db")

def step(context, *args):
    context['value'] += args[0]

def finalize(context):
    return context['value']

def pysum():
    return ({'value' : 0}, step, finalize)

connection.createaggregatefunction("pysum", pysum)

c = connection.cursor()
print c.execute("select pysum(id) from foods").next()[0]
```

Вызов функции SQLite *createaggregatefunction()* регистрирует групповую функцию, получая функции Python-а *step()* и *finalize()*. SQLite использует переменную *context*, для хранения промежуточных значений между вызовами *step()*. После обработки последней записи SQLite вызывает *finalize()*, в данном примере она просто выводит посчитанную сумму. Об очистке *context* SQLite позаботится самостоятельно.

6.2.3 Создание Определяемых Пользователем Сортирующих Последовательностей

Сортирующие последовательности определяют как сравниваются строчные величины. Таким образом, сортирующие последовательности, определяемые пользователем, являются способом для дополнительных способов сортировки и сравнения тестов. Это делается при помощи функции программного интерфейса *sqlite3_create_collation()*. По умолчанию SQLite предоставляет три сортирующих последовательности: BINARY, NOCASE и RTRIM. BINARY сравнивает строки при помощи функции *memcmp()* из библиотеки C (она чувствительна к регистру во всех случаях). Сортировка NOCASE наоборот, нечувствительна к регистру. RTRIM сравнивает строки так же, как и сортировка BINARY, за исключением игнорирования завершающих пробелов.

Сортирующие последовательности, определяемые пользователем, особенно полезны для настроек локальных языков (в смысле русский или арабский, а не Pascal), которые не слишком хорошо сравниваются при помощи последовательности BINARY, или тех, которые нуждаются в поддержке для кодировки UTF-16. Кроме того, их полезно применять в специальных случаях, таких как сортировки дат, формат которых, не склонен сам по себе ни к хронологическому, ни к лексикографическому порядку. В главе 8 приводится иллюстрация применения в SQLite сортирующей последовательности дат, используемой в СУБД Oracle.

6.3 Транзакции

К этому моменту общее представление о программном интерфейсе уже должно быть сформировано. Были рассмотрены способы выполнения операторов SQL при помощи полезных вспомогательных функций. Однако выполнение операторов SQL подразумевает некоторые знания за пределами собственно функций из программного интерфейса. С обработкой запросов тесно связаны транзакции и блокировки. Запросы всегда выполняются внутри транзакций, транзакции влекут блокировки, а блокировки, без должного исполнения, вызывают проблемы. Управлять типом и длительностью блокировок можно и при помощи SQL, и способами разработки кода для приложения.

В разделе 5.3.4 обсуждалось возникновение мертвых блокировок (deadlock, дедлок, клинч) в ситуации, когда два соединения управляют своими транзакциями исключительно при помощи операторов SQL. Программист имеет еще одну возможность жонглировать -, собственно, код приложения. Код приложения тоже может содержать несколько соединений в различных состояниях внутри различных обработчиков операторов на различных таблицах в любые моменты времени. Все, что требуется, чтобы приложение застряло на блокировке EXCLUSIVE (даже не заметив этого), не давая другим соединениям ничего доделать, это обработчик одиночного оператора.

Поэтому критически важно хорошо представлять как работают и транзакции, и блокировки, также связь между ними и используемыми функциями программного интерфейса при выполнении запросов. Было бы идеально, при взгляде на несколько написанных строчек кода, объяснить в каком состоянии находится транзакция или хотя бы предположить место потенциальных трудностей. В этой секции исследуется механизм выполнения транзакций и блокировок, а в следующей секции посмотрим как они работают в реальных программах.

6.3.1 Жизненный Цикл Транзакции

Касательно программирования и транзакций необходимо знать пару фактов. Сначала требуется знать какие объекты выполняются в какой транзакции. Затем возникает вопрос длительности транзакций - когда транзакция начинается, как долго она тянется и когда начинает влиять на другие соединения? Первый вопрос касается исключительно программного интерфейса. Второй - касается SQL вообще и его конкретной реализации в SQLite в частности.

Как известно, при помощи единственного соединения можно создать несколько обработчиков команд. И, как показано на рис. 6.2, каждое соединение с БД имеет в точности одно В-дерево и один, ассоциированный с ним менеджер страниц. Менеджер страниц играет большую роль, чем соединение в этой теме, потому что он управляет транзакциями, блокировками, кэшем памяти и восстановлением после краха - это все будет объясняться в нескольких следующих секциях. Для упрощения можно говорить, что

объект соединения занимается этим всем, но на деле, этим занимается менеджер страниц внутри него. Важная деталь, которую надо помнить про запись в БД при одном соединении это то, что в каждый момент времени есть только одна транзакция. То есть, в каждой транзакции соединения выполняются все объекты операторов, которые были построены для этого соединения. Это ответ на первый вопрос.

Что до второго вопроса, то длительность транзакции (или жизненный цикл транзакции) может быть либо такой короткой, как один оператор или такой длинной как надо, до тех пор, пока не будет сказано: 'Стоп'. По умолчанию, соединение работает в режиме автокоммита, это означает, что каждый оператор выполнялся в отдельной транзакции. И напротив, после встреченного ключевого слова *begin*, транзакция тянется до тех пор, пока не встретит слова *commit* или *rollback*, или до тех пор пока один из операторов SQL не вызовет нарушение целостности БД и завершит работу откатом. Далее надо рассмотреть вопрос как транзакции связаны с блокировками.

6.3.2 Состояния Блокировок

В большинстве случаев продолжительность блокировки приближается к продолжительности транзакции. Хотя они не всегда начинаются вместе, они всегда вместе заканчиваются. При завершении транзакции, освобождается связанная с ней блокировка. Другими словами, блокировка не заканчивается до тех пор, пока не завершается её транзакция или не сбойнет приложение. После сбоя приложения или краха системы транзакция не завершается, в этом случае в БД остается неявная блокировка с которой начнет работу следующее соединение. Эта подробнее уточняется позднее в секции 'Блокировки и Восстановление После Краха'.

В SQLite существует пять различных состояний блокировок и соединения, не зависимо от того, чем оно занимается, всегда находится в одном из них. Состояния блокировок SQLite и переходы между ними изображены на рис. 6.3 . Рисунок раскрывает каждое возможное состояние блокировки, в которое может находиться соединение равно как и путь, через который транзакция может пройти за время своей жизни. Рисунок изображен в терминах состояний блокировок, переходов между блокировками и жизненного цикла транзакций. На рисунке действительно можно проследить жизнь транзакции в терминах блокировок.

За исключением состояния UNLOCKED, каждое из них имеет соответствующую блокировку. Таким образом, можно говорить, что соединение имеет 'имеет блокировку RESERVED', или соединение 'находится в состоянии RESERVED' или просто 'в RESERVED' - это все значит одно и то же. За исключением UNLOCKED, чтобы находиться в таком-то состоянии, соединение сначала должно получить блокировку, соответствующую этому состоянию.

Транзакция может начинаться или с UNLOCKED, или RESERVED или EXCLUSIVE. По умолчанию, как видно на 6.3 , транзакция начинается с UNLOCKED. Состояния блокировок из белых прямоугольников - UNLOCKED,

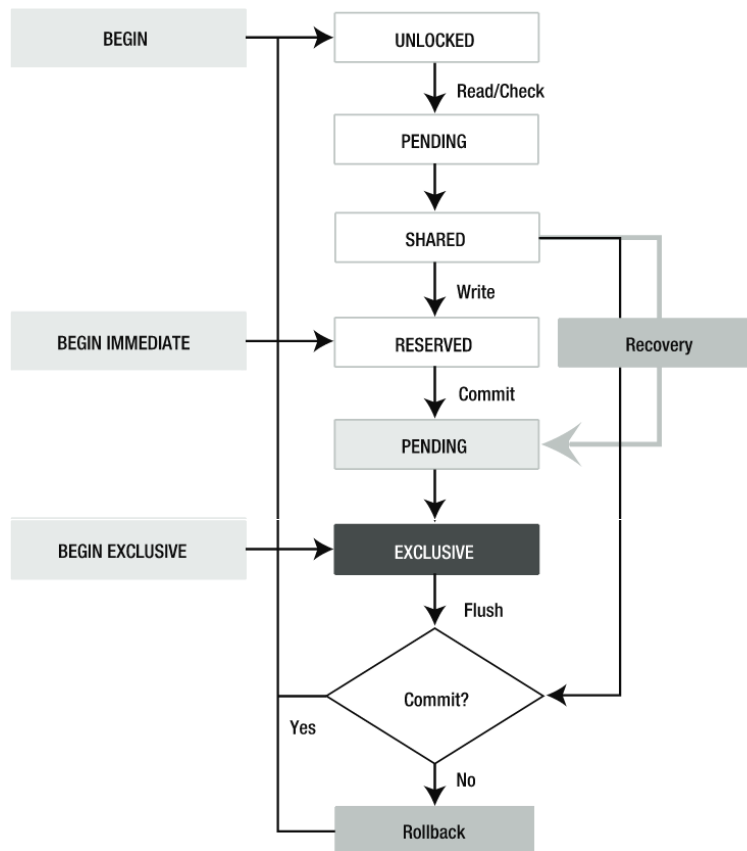


Рис. 6.3: Переходы между блокировками SQLite

PENDING, SHARED или RESERVED - могут существовать в БД при отсутствии соединений с последней. Ограничения появляются Начиная с PENDING в сером прямоугольнике. Это состояние представляет собой блокировку, которую удерживает одиночное соединение, назовем его писателем, желающее получить состояние EXCLUSIVE. И напротив, состояние PENDING в белом прямоугольнике представляет путь, на котором соединение запрашивает и освобождает блокировку по дороге к состоянию SHARED. Невзирая на все представленные состояния блокировок, каждая транзакция в SQLite сводится к одному из двух типов: к транзакциям, которые читают; или к транзакциям, которые пишут. В конце концов этот факт отражен на рисунке: читающие и пишущие транзакции и как они ладят друг с другом.

6.3.3 Читающие Транзакции

Для начала пройдем по процессу блокировки оператора *select*. Это очень простой путь. Соединение, которое выполняет оператор *select*, начинает транзакцию, которая меняется от UNLOCKED к SHARED и, по достижении оператора *commit*, возвращается к UNLOCKED. Конец истории.

Теперь, поинтересуемся, что случится при выполнении двух операторов?

Каким путем в этом случае станет меняться блокировка в этом случае? Ответ зависит от использования или не использования режима автокомита. Рассмотрим следующий пример:

```
db = open('foods.db')
db.exec('begin')
db.exec('select * from episodes')
db.exec('select * from episodes')
db.exec('commit')
db.close()
```

В этом случае, с явно написанным *begin*, оба оператора *select* выполняются в одной транзакции и, поэтому, - в одном состоянии SHARED. Первый вызов функции *exec()* выполняется, оставляя соединение в SHARED, затем вызывается следующий *exec()*. Наконец, явно выполненный оператор *commit* переводит соединение из SHARED обратно в UNLOCKED. Все изменения блокировки выглядит следующим образом:

UNLOCKED→PENDING →SHARED →UNLOCKED

Теперь рассмотрим случай, как будто бы в примере нет ни оператора *begin*, ни оператора *commit*. Это означает, что два оператора *select* выполняются в режиме автофиксации (autocommit). Тогда они проходят полный путь изменений состояния блокировок независимо друг от друга. В этом случае состояния блокировок меняется следующим образом:

UNLOCKED →PENDING →SHARED →UNLOCKED →PENDING →SHARED →UNLOCKED

Так как в примере просто читаются данные, это не выглядит совсем уж по другому, но в режиме автофиксации через блокировку файла приходится пройти дважды. И, как вскоре будет понятно, другое, пишущее соединение может успеть сработать между двумя операторами *select* и изменить БД. Таким образом, нельзя быть уверенным, что эти два оператора вернут один и тот же результат. С другой стороны, при использовании *begin..commit*, гарантируется, что у тех же запросов отношения - результаты будут идентичными.

6.3.4 Пишущие Транзакции

Теперь рассмотрим оператор, который пишет в БД, такой как *update*. В-первых, соединение должно менять свое состояние так же как и в случае *select* и закончится на состоянии SHARED. Каждый оператор - все равно, читающий или пишущий - должен начинаться с последовательного изменения UNLOCKED → PENDING → SHARED, причем PENDING, как будет вскоре видно, является основной блокировкой.

6.3.4.1 Состояние RESERVED

В момент, когда соединение пытается записать что нибудь в БД, оно должно свое состояние SHARED изменить на RESERVED. И, после получения блокировка RESERVED, соединение готово менять содержимое БД. Даже если и не удастся на деле изменить содержимое БД в этот момент, эти изменения сохранит [менеджер страниц](#) в упомянутом ранее [кэше страниц](#). Этот кэш, именно тот, которому меняли размер при помощи прагмы *cache_size* в разделе [5.4.3](#) .

После перехода соединения в состояния `RESERVED`, менеджер страниц инициализирует журнал откатов (`rollback journal`). Это файл (отображен на рис. 6.1), который используется для откатов и восстановлений после краха. В частности, он хранит страницы, необходимые для восстановления БД, в состоянии, которое они имели перед началом транзакции. Эти страницы в журнал откатов складывает менеджер страниц, после изменения оных В-деревом. В данном случае, для каждой записи, измененной оператором `update`, менеджер страниц выбирает из БД страницу, содержащую исходную запись и сохраняет её в журнале. То есть, журнал хранит частичное содержание БД до транзакции. Следовательно, все что менеджер страниц должен сделать для отката любой транзакции, это просто скопировать содержимое журнала обратно в файл БД. Тогда БД восстановит свое состояние перед транзакцией.

Для `RESERVED` существует три набора страниц, которыми управляет менеджер страниц: измененные страницы, не измененные страницы и сохраненные в журнале. Измененные страницы - это страницы, которые содержат записи, которые отредактировало В-дерево. Они хранятся в кэше страниц. Не измененные страницы - это страницы, которые В-дерево прочитало, но не меняло. Это результат работы операторов `select`. Наконец, страницы, сохраненные в журнале - это исходные версии отредактированных страниц. Они не сохраняются в кэше страниц, а записываются в журнал перед, собственно, их изменением.

Кэш страниц позволяет пишущему соединению закончить свою работу, находясь в состоянии `RESERVED` без взаимодействия с другими (читающими) соединениями. То есть, SQLite эффективно справляется с несколькими читающими соединениями и одним пишущим, которые работают одновременно в одной и той же БД. Хитростью является то, что пишущее соединение должно сохранять свои изменения в кэше страниц, а не в БД. Напомним, что в один момент времени для данной БД только одно соединение может быть в состоянии `RESERVED` или `EXCLUSIVE` - то есть, несколько читающих соединений и только одно пишущее.

6.3.4.2 Состояние `PENDING`

Когда соединение завершает все изменения оператора `update` и приближается время фиксации транзакции, менеджер страниц пытается перейти к состоянию блокировки `EXCLUSIVE`. Ради полноты картины повторим изложенное в разделе 5.3.3. Менеджер страниц пытается получить блокировку состояния `PENDING` из состояния `RESERVED`. Если ему это удастся, он находится в нем, предотвращая все другие соединения от перехода в это состояние. Взглянув на 6.3 можно увидеть произведенный эффект. Напомним, что важность блокировки `PENDING` уже отмечалась. Теперь понятно почему. Так как пишущее соединение находится в состоянии `PENDING`, ни одно из других соединений не может больше перейти в состояние `SHARED` из состояния `UNLOCKED`. Это означает, что теперь не появится ни одного нового соединения с БД; ни читающего, ни пишущего. ?? Это состояние является отсеивающим действием. Оно гарантирует, что пишущее соединение может ждать своей очереди к БД и - если все соединения ведут себя правильно - дожидаться своего. Продолжать свою работу как обычно смогут

только те соединения, которые находятся в состоянии SHARED. А пишущее соединение, находясь в PENDING, ждет их завершения и освобождения их блокировок. Что еще влечет это ожидание является дополнительным вопросом, который будет вскоре рассмотрен в секции 6.5 .

После освобождения остальными соединениями их блокировок, БД переходит в монопольное владение к пишущему соединению. Значит менеджер страниц может поменять состояние блокировки своего соединения из PENDING в EXCLUSIVE.

6.3.4.3 Состояние EXCLUSIVE

В состоянии EXCLUSIVE производится сохранение изменённых страниц из кэша страниц в файл БД. Так как менеджер страниц собирает изменять БД, то лежащая на нем ответственность возрастает. Эту работу надо сделать с максимальной тщательностью. Перед началом вывода этих страниц в БД, он обращается к журналу. От него требуется убедиться, что все содержимое журнала было записано на поверхность диска (в его файле). Ведь весьма вероятно, что операционная система продолжает хранить многое, если не все в оперативной памяти, несмотря на попытки менеджера сбросить страницы на диск. Он в буквальном смысле сообщает операционной системе о сохранении всех этих страниц на диске. И теперь в игру вступает описанная в разделе 5.4.3 прагма синхронизации - *synhronous*. ?? Метод, заданный прагмой, определяет насколько тщательно менеджер страниц обеспечивает вывод операционной системой страниц журнала на диск. Нормальная установка заключается в выполнении полной синхронизации перед продолжением работы, операционная система должна подтвердить реальную запись всех буферизованных страниц журнала на поверхность диска. Если прагма *synhronous* установлена в FULL, менеджер страниц выполняет две полных синхронизации перед продолжением. Если прагма *synhronous* установлена в NONE, менеджер страниц не заботится о журнале вообще (при этом он может быть в 50 раз быстрее, но можно забыть о надежности транзакций).

Фиксация журнала на диске так важна, так как это единственная возможность восстановить файл БД после сбоя приложения или краха операционной системы во время записи в БД менеджером страниц. Если все страницы журнала не перенести из оперативной памяти на диск перед крахом системы, то БД невозможно вернуть в начальное состояние, так как данные, оставшиеся в оперативной памяти, оказываются потерянными навсегда. В лучшем случае, БД окажется в несогласованном состоянии, в худшем - файл окажется испорченным.

Внимание

Даже если было использовано наиболее консервативное значение для прагмы *synhronous* журнал может все таки не полностью сбросится на диск. Это не вина SQLite , это следствие определенных типов аппаратных средств или операционных систем. SQLite использует системный вызов *fsync()* под Юниксами и *FlushFileBuffers()* под Windows чтобы заставить записать страницы журнала на диск. Однако есть сообщения о том,

что эти функции не всегда работают, особенно на дешевых IDE дисках. По всей видимости, некоторые производители IDE дисков используют микросхемы, которые привирают о реальном состоянии дел при записи. В некоторых случаях микросхемы кешируют данные и сообщают что они уже записаны на поверхность диска. Кроме этого, есть сообщения (непроверенные) о том, что Windows иногда игнорирует вызовы *FlushFileBuffers()*. Таким образом, в случае сомнительных программных или аппаратных средств надежность транзакций может пострадать.

Как только менеджер страниц позаботился о журнале, он копирует все измененные страницы в файл БД. Что произойдет потом, зависит от режима транзакции. В случае автофиксации транзакций менеджер страниц очищает журнал и кэш страниц, переходит из состояния EXCLUSIVE в UNLOCKED. Если транзакция не добралась до фиксации, то менеджер страниц продолжает держать состояние блокировки EXCLUSIVE и журнал используется до достижения либо оператора *commit*, либо оператора *rollback*.

6.3.4.4 Автофиксация и Производительность

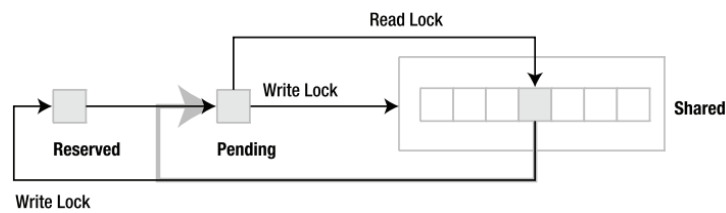
С полученным багажем знаний, рассмотрим что происходит с оператором *update*, выполняемым в явной транзакции, и оператором *update*, выполняемым в режиме автофиксации. При автофиксации каждая команда, изменяющая БД в отдельной транзакции последовательно меняет свои состояния следующим образом:

UNLOCKED →PENDING →SHARED →RESERVED →PENDING →EXCLUSIVE →UNLOCKED

При этом, каждый такой цикл включает в себя создание, фиксацию и очистку журнала откатов. Несмотря на то, что выполнение нескольких выборок в режиме автофиксации не является проблемой с точки зрения эффективности, необходимо тщательно обдумать использование автофиксации для частых запросов на запись. Кроме того, как упоминалось в случае *select*, при выполнении нескольких операторов с автофиксацией, ничто не сможет удержать другое работающее соединение от изменения БД между операторами. По этой причине, если два обновления БД зависят от заданных значений в данных, требуется всегда выполнять их в одной и той же транзакции.

Блокировки и восстановление

В SQLite реализация блокировок использует обычные файловые блокировки. SQLite есть три различных файловых блокировок для файла БД: резервированный байт (*reserved byte*), байт ожидания (*pending byte*) и разделяемая область (*shared region*).



Все начинается с байта ожидания. Чтобы перейти от состояния UNLOCKED к SHARED соединение пробует установить блокировку для чтения на байте ожидания. В случае успеха, соединение берет блокировку для чтения на случайном байте в разделяемой области и освобождает блокировку для чтения на байте ожидания. Чтобы изменить состояние от SHARED к RESERVED, соединение вытаскивается получить блокировку записи для зарезервированного байта. Чтобы перейти от RESERVED к EXCLUSIVE приложение пытается получить блокировку для записи для зарезервированного байта. В случае успеха, это приводит к процессу истощения, так как не позволяет другим соединениям брать блокировку для чтения на байте ожидания, чтобы перейти к состоянию SHARED. Наконец, чтобы изменить состояние своей блокировки на EXCLUSIVE, соединение пробует получить блокировку для записи на всей разделяемой области. Так как разделяемая область содержит блокировки для чтения всех остальных активных соединений, этот шаг гарантирует, что блокировка соединения перейдет в состояние EXCLUSIVE только после того, как сначала все другие блокировки соединений освободят состояние SHARED.

Механизм восстановления после краха в SQLite использует резервированный байт чтобы определить когда БД нуждается в восстановлении. Так как файл журнала и блокировка RESERVED передается из рук в руки, менеджер страниц видит журнал без блокировки, если случилось что то плохое. Такая проверка целостности выполняется каждый раз, когда менеджер страниц открывает БД или пытается получить страницу из неё. Наличие файла журнала и отсутствие блокировки RESERVED на БД означает, что процесс, создавший журнал потерпел крах или упала операционная система. В этом случае журнал называется горячим журналом и считается, что БД может находится в противоречивом состоянии. Такой журнал должен быть 'отыгран назад', чтобы вернуть БД к её состоянию, которое было перед прерыванием транзакции.

Для этого менеджер страниц переводит БД в режим восстановления. Для этого, как показано выше (серой линией на рис. 6.3) менеджер изменяет свое состояние от SHARED прямо к PENDING в сером прямоугольнике. Такой переход возможен только в этой ситуации. Есть две причины, чтобы пропустить состояние RESERVED. Во-первых, блокирование байта ожидания не допускает никаких новых соединений с БД. Во-вторых, уже существующие соединения с БД (в состоянии SHARED) при сле-

дующей попытке доступиться к данным увидят, что журнал горячий. Такие соединения тоже попробуют перейти в режим восстановления и отыграть журнал. Однако не смогут сделать это, так как первое соединение уже захватило блокировку PENDING. То есть, немедленно меняя блокировку SHARED на PENDING, первое соединение может быть уверенно, что новых соединений с БД не появится, соединения с БД уже находящиеся в состоянии SHARED не смогут начать восстановление БД. Работа всех соединений, кроме начавшего восстановление, приостановлена.

По существу, горячий журнал является неявной блокировкой EXCLUSIVE. После краха пишущего соединения, никакая активность не может продолжаться до тех пор, пока одно из соединений не восстановит её. При попытке доступа к любой странице следующий менеджер страниц увидит горячий журнал, блокирующий все и начнет восстановление. При отсутствии активных соединений, горячий журнал обнаружит первое приложение, попытавшееся соединиться с БД. Оно же и начнет восстановление.

6.4 Настройка Кэша Страниц

Предположим, предыдущий пример был изменен, теперь он начинается с оператора *begin*, за которым следуют *update*. Причем в процессе всех изменений переполняется кэш страниц. Как должен на это реагировать SQLite? То есть, что произойдет, если *update* потребует хранить больше страниц, чем может поместиться в кэше страниц?

6.4.1 Переход в Состояние EXCLUSIVE

Когда конкретно менеджер страниц меняет состояние RESERVED на EXCLUSIVE и почему? Существует два сценария и будут рассмотрены оба из них. Либо соединение достигает момента фиксации изменений и явно начинает блокировку EXCLUSIVE, либо кэш страниц переполняется без возможности какого либо выбора. Рассмотрим первый сценарий. Что случается при заполнении кэша страниц? Выражаясь простым языком, менеджер больше не может запомнить ни одной измененной страницы и, естественно, не может выполнять свою работу. Его переводят в состояние EXCLUSIVE, чтобы продолжать. На деле, это не совсем правда, потому что существуют мягкий и жесткий пределы.

Первому заполнению кэша страниц соответствует мягкий предел. В этот момент кэш хранит смесь из неизменных и измененных страниц. Менеджер страниц пытается очистить кэш. Он проходит по кэшу, избавляясь от неизменных страниц. После очистки менеджер продолжает работу с освобожденной памятью до тех пор, пока кэш не заполнится снова. Далее, если есть хоть одна неизменная страница, можно повторить процесс очистки от них. Это означает появление жесткого предела. В этот момент менеджер страниц не имеет другой возможности, кроме как переходить к состоянию EXCLUSIVE.

С другой стороны, прагма `cach_size`, проявляется в состоянии `RESERVED`. Как объяснялось в секции 5.4.3, прагма управляет размером кэша страниц. Чем больше кэш, тем больше измененных страниц в нем может запомнить менеджер страниц и тем больше работы соединение может выполнить, не меняя свое состояние на `EXCLUSIVE`. И, как отмечалось выше, выполнение работы в состоянии `RESERVED` уменьшает время, когда соединение находится в состоянии `EXCLUSIVE`. Если все закончено в `RESERVED`, то `EXCLUSIVE` нужен исключительно на время сбрасывания измененных страниц на диск, не будет компиляции следующих запросов, не будет обработки следующих результатов и, затем, вывода их на диск. Обработка, которая выполняется в состоянии `RESERVED`, может значительно улучшить общую производительность. При больших транзакциях или перегруженной БД и достаточного объема доступной памяти было бы идеально, чтобы размер кэша позволял находиться соединению в `RESERVED` так долго как возможно.

6.4.2 Выбор Размера Кэша

Итак, как подобрать размер кэша? Это зависит от выполняемой работы. Предположим, требуется поменять каждую запись в таблице `episodes`. В этом случае каждая страница таблицы будет изменена. Тогда надо узнать размер таблицы `episodes` в страницах и соответственно ему изменить размер кэша страниц. Для получения такой информации о таблице `episodes` требуется использовать `sqlite3_analyzer.exe`. Для каждой таблицы он может вывалить детальную статистику, включая число страниц. К примеру, для таблицы `episodes` из БД `foods` можно получить следующую информацию:

```
*** Table EPISODES *****
Percentage of total database..... 20.0%
Number of entries..... 181
Bytes of storage consumed..... 5120
Bytes of payload..... 3229          63.1%
Average payload per entry..... 17.84
Average unused bytes per entry..... 5.79
Average fanout..... 4.00
Maximum payload per entry..... 38
Entries that use overflow..... 0          0.0%

Index pages used..... 1
Primary pages used..... 4
Overflow pages used..... 0
Total pages used..... 5
Unused bytes on index pages..... 990          96.7%
Unused bytes on primary pages..... 58          1.4%
Unused bytes on overflow pages..... 0
Unused bytes on all pages..... 1048          20.5%
```

Общее число страниц таблицы - 5. Из них четыре страницы действительно отведено под таблицу и одна - под индекс. Поскольку умолчательный размер кэша - 2000 страниц, то волноваться не о чем. В таблице `episodes` 400 записей, следовательно, каждая страница хранит 100 записей. Не зачем беспокоится об изменении кэша, до тех пор, пока не придется менять каждую из, по меньшей мере, 196 000 записей в таблице `episodes`. И, вообще, этим нужно заниматься при нескольких соединениях с БД и решении вопроса коллективной работы. Для одиночного пользователя БД это все не имеет значения.

6.5 Ожидание Блокировок

Ранее упоминался менеджер страниц, ожидающий смену состояния с PENDING на EXCLUSIVE. Но что конкретно подразумевает это ожидание блокировки? Во-первых, ожидание блокировки может повлечь любой вызов *exec()* или *step()*. Всякий раз, когда SQLite натывается на ситуацию, когда он не может захватить блокировку, умолчательным поведением является возврат SQLITE_BUSY в функцию, которая вынудила его к попытке её получить. Признак SQLITE_BUSY можно получить не зависимо от исполняемого оператора. Как уже известно, даже оператор *select* может сбойнуть на попытке получить блокировку SHARED, если пишущее соединение пишет или находится в состоянии PENDING. Самое простое, что можно сделать после получения SQLITE_BUSY - это повторить предыдущий вызов. Однако, вскоре будет ясно, что это не всегда лучший способ поведения.

6.5.1 Использование Обработчика Занятости

Вместо того, чтобы снова и снова вызывать функцию программного интерфейса, можно использовать обработчик занятости (busy handler). Если соединение не может захватить блокировку, то не обязательно упорно получать SQLITE_BUSY, вместо этого лучше использовать обработчик занятости.

Обработчик занятости - это функция, которую пишут, чтобы убить время или делать еще что нибудь по желанию, вроде заботливой рассылки спама тещам. Обработчики занятости предполагается вызывать, когда SQLite не может получить блокировку. Единственное, что обязан сделать обработчик это предоставить код возврата, уведомляющий SQLite о требуемых следующих действиях. По соглашению, если обработчик возвращает *true*, то SQLite будет продолжать попытки получить блокировку. Если обработчик возвращает *false*, то SQLite вернет SQLITE_BUSY в функцию. запрашивающую блокировку. Рассмотрим следующий пример:

```
counter = 1

def busy()
    counter = counter + 1
    if counter == 2
        return 0
    end

    spam_mother_in_law(100)
    return 1
end

db.busy_handler(busy)
stmt = db.prepare('select * from episodes;')
stmt.step()
stmt.finalize()
```

Собственно, разработка функции *spam_mother_in_law()* остается как упражнение для читателя.

Функция *step()* должна получить блокировку SHARED на БД, чтобы выполнить оператор *select*. Однако предположим что у этой БД есть еще активное пишущее соединение. Обычно функция *step()* вернула бы

SQLite_BUSY. Однако в нашем случае это не так. Здесь менеджер страниц (тот из них, который обращается за блокировкой) вызывает функцию *busy()*, так как она была зарегистрирована в качестве обработчика занятости. *busy()* увеличивает счетчик *counter*, посылает тебе сотню случайных писем из своей папки со спамом и возвращает 1. Менеджер страниц интерпретирует её как *true* (сигнал продолжать захватывать блокировку) и повторяет попытку получить блокировку SHARED. Пусть БД осталось заблокированной. Менеджер страниц опять вызывает обработчик. На этот раз, *busy()* возвращает 0, который трактуется как *false*. Тогда менеджер страниц, вместо повторения захвата блокировки, возвращает код SQLite_BUSY, которым закончивается работа функции *step()*.

Разрабатывать обработчик занятости просто для ожидания (чтобы убить время) нет нужды. Он уже есть в программный интерфейс SQLite. Это функция, которая просто засыпает на заданный период времени для ожидания блокировки. Её название *sqlite3_busy_timeout()* и поддерживается библиотеками некоторых [добавлений к языку](#). По существу можно просто сказать 'засни на 10 секунд, если не сможешь получить блокировку' и менеджер страниц сделает это. Он подождет 10 секунд, и затем, если не сможет получить блокировку, вернет SQLite_BUSY.

6.5.2 Правильное Использование Транзакций

Ещё раз пересмотрим предыдущий пример, но теперь в нем заменим оператор *select*, на оператор *update*. Что на деле будет означать SQLite_BUSY? После применения *select* это просто значит: 'Я не могу получить блокировку SHARED'. А что это значит в случае *update*? На деле, это неизвестно. SQLite_BUSY может означать, что соединение не смогло получить блокировку SHARED, в связи с наличием другого пишущего соединения в состоянии PENDING. Это может означать, что соединение, получив блокировку SHARED, не может перейти к RESERVED. Важно то, что таким способом нельзя узнать состояние БД или состояние рассматриваемого соединения. В режиме автофиксации SQLite_BUSY для запросов на запись полностью неопределен. Итак, что же теперь делать дальше? Надо ли вызывать *step()* снова и снова, пока оператор не выполнится?

Над этим надо подумать. Предположим, попытка получить блокировку SHARED (а не RESERVED), закончилась SQLite_BUSY и теперь соединение удерживает состояние RESERVED. Напомним, состояние БД - не известно. И нахрапистая попытка любой ценой выполнять свою транзакцию не обязательно сработает, и в рассматриваемом соединении, и в любом другом. Попытки вызова *step()* заведут в тупик – в задницу соединения в состоянии RESERVED, и, если никто из двоих не отступит, к клинчу.

Замечание

SQLite пытается избежать мертвой блокировки в конкретно этом сценарии, игнорируя вызов обработчика занятости (*busy handler*). Обработчик занятости соединения в состоянии SHARED не вызывается, если предполагается, что он будет мешать завершиться соединению

в состоянии RESERVED. Однако, понята ли подсказка, зависит полностью от приложения. В случае, если оно будет просто продолжать вызовы *step()* ничто не спасет SQLite .

как уже должно быть понятно, запрос на запись надо начинать с *beginimmediate*. По крайней мере, при получении ответа SQLITE_BUSY, будет известно состояние соединения. Пишущее соединение сможет повторять свои попытки без блокирования другого соединения. И, как только достигнут успех, так же известно, что пишущее соединение перешло в состояние RESERVED. Теперь можно делать что угодно. С другой стороны, *beginexclusive* является гарантией что не придется иметь дела с условием занятости вообще. Но при этом надо помнить, что пишущее соединение находится в состоянии EXCLUSIVE, которое снижает эффективность многократных соединений, по сравнению с работой в состоянии RESERVED.

Блокировки и сетевая файловая система

Уже можно хорошо представлять о трудностях использования файла ДБ через сетевую файловую систему (в расшаренном каталоге). SQLite поддерживает многократные соединения размещая файл блокировки в файловой системе. Очень важно, чтобы этот файл и размещался и удалялся в правильные моменты времени. А корректность управления блокировками многократных соединения у SQLite полностью зависит от файловой системы. SQLite использует один и тот же механизм блокировок не различая обычную файловую систему от сетевой. Под Unix SQLite использует POSIX advisory locks, а под Windows - системные вызовы *LockFile()*, *LockFileEx()* и *UnlockFile()*. Эти стандартные системные вызовы корректно работают в несетевой файловой системе. Это сетевая файловая система обязана эмулировать работу обычной. И, к сожалению, в некоторых реализациях эта эмуляция не всегда корректна. Даже в случае корректной работы сетевой файловой системы остаются поводы для размышлений.

Возьмем к примеру NFS. Это отличная сетевая файловая система. Однако, многие реализации исходной NFS имеют известные ошибки и, в некоторых случаях, не имеет реализованных блокировок. Это означает серьезную проблему для SQLite . Без блокировок два соединения могут перейти в состояние EXCLUSIVE на одной ДБ и в одно время, что почти гарантированно приведет к повреждению БД. Это не проблема протокола NFS вообще, это проблема некоторых её реализаций. Благодаря последним реализациям NFS эти трудности преодолены и некоторые из них (такие, которые используются в Sun) вполне работоспособны.

6.6 Прикладные Программы

Наконец, получена хорошая картина для программного интерфейса, транзакций и блокировок. Чтобы покончить с этим, рассмотрим все это с точки зрения прикладной программы и рассмотрим пару сценариев, которые было бы желательно обсудить.

6.6.1 Использование Нескольких Соединений

Опытные программисты вполне могли создавать приложения, использующее несколько соединений внутри простой программы. Классический пример: одно соединение проходит по таблице в то время, как другое изменяет её. Такие приложения с несколькими соединениями могут приводить к трудностям в работе SQLite, поэтому их надо аккуратно обдумывать. Рассмотрим пример:

```
c1 = open('foods.db')
c2 = open('foods.db')

stmt = c1.prepare('select * from episodes')

while stmt.step()
  print stmt.column('name')
  c2.exec('update episodes set <some columns, criteria, etc.>')
end

stmt.finalize()

c1.close()
c2.close()
```

Можно спорить, что здесь легко указать трудность. Находясь в цикле *while*, соединение *c2* выполняет обновления таблицы, в то время как соединение *c1* имеет блокировку в состоянии SHARED. Эта блокировка не будет освобождена до финализации оператора *stmt* после цикла. Таким образом, нет никакой возможности выполнить запрос обновления БД внутри цикла. Либо *c2.exec()* будет тихо сбоить, либо, в случае обработчиков занятости, они будут просто тормозить приложение. Более правильная версия этого примера должна использовать одно соединение и выполнять запросы в рамках транзакции *begin immediate*. Новую версию приводим ниже:

```
c1 = open('foods.db')

# Keep trying until we get it
while c1.exec('begin immediate') != SQLITE_SUCCESS
end

stmt = c1.prepare('select * from episodes')

while stmt.step()
  print stmt.column('name')
  c1.exec('update episodes set <some columns, criteria, etc.>')
end

stmt.finalize()
c1.exec('commit')
c1.close()
```

В подобных случаях надо использовать операторы из одного соединения для запросов чтения и записи. Тогда не за чем беспокоиться о трудностях,

вызываемых блокировками БД. Но, как оказывается, этот конкретный пример остается нерабочим. Процесс блокирование при чтении таблицы одним оператором и редактирование её другим имеет дополнительную тонкость, которую тоже надо знать. Сейчас она будет раскрыта.

6.6.2 Важность Финализации

Обычной промашкой в обработке оператора *select* является непонимание того, что блокировка SHARED не обязательно освобождается до вызова функции *finalize()* (или *reset()*), более того, обычно не освобождается. Рассмотрим пример:

```
stmt = c1.prepare('select * from episodes')

while stmt.step()
  print stmt.column('name')
end

c2.exec('begin immediate; update episodes set ...; commit;')

stmt.finalize()
```

Хотя практически так программы не пишут, это можно написать случайно, просто потому, что так можно сделать. Эквивалент этого примера, написанный с помощью программного интерфейса C, будет на деле работать. Даже если не вызвать *finalize()*, второе соединение изменит БД безо всяких проблем. Прежде чем сказать почему, взглянем на следующий пример:

```
c1 = open('foods.db')
c2 = open('foods.db')

stmt = c1.prepare('select * from episodes')

stmt.step()
stmt.step()
stmt.step()

c2.exec('begin immediate; update episodes set ...; commit;')

stmt.finalize()
```

Предположим в таблице *episodes* есть 100 записей. Приложение прошло только три из них. Что случится теперь? Второе приложение получит SQLITE_BUSY.

В предыдущем примере, SQLite освобождает блокировку SHARED, когда оператор достигает окончания отношения - результата. То есть, в момент последнего вызова *step()*, на котором программный интерфейс C возвращает SQLITE_DONE, VDBE встретит инструкцию *Close*, и SQLite закроет курсор и сбросит блокировку SHARED. То есть, *c2* будет иметь возможность выполнить свой *insert* хотя *c1* еще не вызвал *finalize()*.

Замечание переводчика

По видимому, не *insert*, а *update*

Во втором случае, оператор *select* не достиг последней записи из отношения результата. Поэтому следующий выход *step()* должен вернуть SQLITE_RESULT, что означает наличие непрочитанных записей в отношении - результате, это же означает, что блокировка SHARED остается. Значит, *c2* не может из - за

неё выполнить *insert* (на деле *update*), так как *c1* продолжает удерживать состояние SHARED.

Мораль этой истории в том, что не надо делать все, что можно. Вызвать *finalize()* или *reset()* перед запросами на редактирование в другом соединении нужно всегда. Также хорошо бы помнить, что *step()* и *finalize()* в режиме автофиксации более или менее ограничивают транзакции и блокировки. Они начинают и заканчивают транзакции. Они начинают и освобождают блокировки. Нужно очень аккуратно работать в другом соединении между этими двумя функциями.

6.6.3 Режим Разделяемого Кэша

Теперь, когда все понятно про правила работы конкурирующих соединений, настало время для некоторого усложнения. SQLite имеет еще одну модель для конкурирующих соединений, которая называется режим разделяемого кэша. Она регулирует работу соединений внутри отдельных нитей.

В этом режиме нити могут создавать несколько соединений, которые сообща используют (разделяют) один и тот же самый кэш страниц. Более того, такая группа соединений может иметь несколько читающих соединений и одно пишущее (в состоянии EXCLUSIVE), работающих в одно время с одной и той же БД. Уловкой есть то, что эти соединения из разных нитей не могут разделять кэш, они строго ограничены одной нитью (именно работающей в режиме разделяемого кэша), создавшей их. Более того, пишущие и читающие соединения должны быть готовы обрабатывать специальные условия, включающие блокировку таблиц.

Когда читающие соединения читают таблицы, SQLite автоматически ставит на них блокировку для чтения. Это предохраняет таблицы от изменения пишущим соединением. Если пишущее соединение пытается изменить таблицу, заблокированную для чтения, то получает SQLITE_LOCKED. Та же логика применяется к читающим соединениям, пытающимся обратиться к таблицам, заблокированным для записи. Но если пишущее соединение работает в специальном режиме чтения не зафиксированных данных (read-uncommitted mode), то читающие соединения могут читать эти таблицы. Этот режим можно задавать при помощи прагмы *read_uncommitted*. В этом случае SQLite не ставит на них блокировку для записи. Как результат, читающие соединения вообще не замечают пишущее. Однако, тогда они могут получить несогласованные данные, так как пишущее соединение может изменить таблицы во время чтения. Этот режим похож на встречающийся в других РСУБД уровень изолирования транзакций с названием грязное чтение.

Этот режим разработан для встраиваемых серверов, которые нуждаются в экономии оперативной памяти и должны обслуживать несколько большее число соединений. Дополнительную информацию о том, как использовать этот режим с программным интерфейсом для C можно найти в гл. 7 .

6.7 И так

Программный интерфейс SQLite гибкий, интуитивно понятный и легкий в использовании. Он состоит из двух основных частей: базовый интерфейс и дополнительный. Базовый интерфейс вращается возле двух основных структур данных, которые используются для выполнения команд SQL: соединений (connection) и операторов (statement). Операторы выполняются за три шага: компиляция, выполнение и финализация. Функции `SQL exec()` и `get_table()` выполняют эти три шага за один вызов функции, автоматически обрабатывая объекты, ассоциированные с операторами. Дополнительный интерфейс предоставляет возможности для дополнительной настройки SQLite тремя способами: при помощи функций, определяемых пользователем; при помощи групповых функций, определяемых пользователем; при помощи сортирующих последовательностей, определяемых пользователем.

Так как модель конкуренции в SQLite несколько отличается от других СУБД, то важно иметь некоторое понимание как SQLite управляется с транзакциями и блокировками, как они на деле выполняются и как они работают внутри кода пользователя. Вообще говоря, эти концепции не трудны для понимания и состоят из нескольких простых правил, которые необходимо помнить, чтобы разрабатывать программы с использованием SQLite.

Все, уже рассказанное выше, будет уточняться в последующих главах, так как эти концепции применимы не только к программному интерфейсу для языка C, но к другим языкам, поскольку они построены на использовании интерфейса для C.

Глава 7

The Core C API

Глава 8

The Extension C API

Глава 9

SQLite Internals and New Features

Предметный указатель

- NULL*, 13
- encoding*, 116
- like*, 49
- page_size*, 116
- temp_store* , 119
- temp_store_directory*, 119
- ./db/0sqlite.cs* , 18
- ./db/1sqlitedb.cs* , 19
- ./db/2db.cs* , 20
- ./db/3ins.cs* , 28
- ./db/4del.cs* , 30
- ./db/app.cs*, 22
- ./db/app.txt* , 26
- ./db/cs.cmd*, 24
- ./db/food_types_add.csv* , 30
- ./db/local.cmd* , 25
- ./db/params.agp-x.cmd* , 25
- ./db/test.cmd* , 25
- .dump*, 12
- .echo*, 13
- .exit*, 7
- .headers*, 13
- .help*, 8
- .import*, 12
- .indices*, 10
- .mode*, 13
- .nullvalue*, 13
- .output*, 12
- .read*, 12
- .schema*, 11
- .schema* , 40
- .separator*, 12
- .show*, 13
- .tables*, 10
- Система управления базой данных, 45
- Утилита CLP, 13
- автофиксация, 131
- автоприращение (autoincrement), 39
- байт код VDBE, 114
- байт код VDBE., 117
- байт ожидания, 134
- целостность данных, 77
- дедлок, 128
- добавления к языку, 127
- естественное соединение (*natural join*), 61
- фиксация, 98
- фиксирует, 109
- фраза, предложение, clause , 42
- функция обратного вызова, 124
- групповая функция, 126
- групповые функции, операции агрегирования, 52
- грязное чтение , 143
- хендлер, 114
- именованные параметры, 119
- классы памяти (storage class), 88
- клинча, 128
- конвейер операций, 57
- конвейер операций , 42
- конвейере операций , 53
- лексема, 37
- лексический анализатор, 4, 117
- менеджер страниц, 113, 131
- мертвые блокировки, 104
- немедленный запрос, 115
- обработчик занятости, 138, 139
- обзор (view), 91
- ограничения целостности, 77
- ограничения целостности (column constraint), 39
- операции агрегирования , 54
- отношение - результат, 45
- отношение внешнего ключа, 59
- отношение внешнего ключа (*foreign key relationship*), 57
- парсер, 4, 117
- пересечение (*intersection*), 59
- первичный ключ, 78

- первичный ключ (*primary key*), 57
- подготовленный запрос, 115
- подзапросы (*subquery*), 64
- пользовательское расширение, 125
- прагма, 105
- прагма (*pragma*) , 107
- прагма *read_uncommitted*, 143
- представление (*view*), 91
- прямое произведение (*cross join*), 60
- разделяемая область, 134
- разделяют, 143
- разрешение конфликтов (*conflict resolution*), 99
- резервированный байт, 134
- режим разделяемого кэша, 125
- режиме чтения не зафиксированных данных, 143
- синонимы (*alias*), 62
- сообща используют, 143
- соотнесенный подзапрос, 64
- сортирующая последовательность, 127
- спецификатор места вывода, 119
- связываемые по порядку параметры, 119
- педуплер операционной системы, 14
- транзакции, 97
- управление операторами, 92, 113, 124
- утилите CLP, 93
- виртуальная машина, 5
- внешним ключом (*foreign key*), 57
- внутреннее произведение (*inner join*), 59
- временное хранилище, 119
- язык манипулирования данными , 38
- язык определения данных , 38
- языка DDL, 12
- журнал откатов, 132
- добавление к языку, 114
- добавлений к языкам программирования, 117
- добавлений к языку., 139
- добавления к другому языку., 126
- добавления к языкам., 123
- добавлениях к языкам, 121
- кэш страниц, 115
- кэше страниц., 131
- менеджер страниц, 115
- разделитель операторов, 36
- шелл (*shell*), 6
- внешнее соединение (*outer join*), 61
- встроенное представление (*inline view*), 65
- command terminator, 36
- DML, 12
- VDBe, 5
- VDBe , 114
- agregates, 126
- API, 4
- autocommit, 131
- big endian, 116
- busy handler, 138, 139
- callback function, 124
- CLP, 6–10, 14, 34
- CLP , 8
- CLP , 8
- CLP., 6–8
- collation, 127
- commit, 98, 109
- CSV формат, 12
- data definition language - DDL, 38
- data manipulation language - DML, 38
- DDL -, 11
- DDL., 11, 71
- deadlock , 128
- DML (Языка Манипулирования Данными), 72
- DML., 71
- DML., 40
- hook-функция, 124
- language extensions, 114
- like, 10
- little endian, 116
- named parameter, 119
- operational control, 124
- pager, 113
- pending byte, 134
- placeholder, 119
- positional parameter, 119
- prepared query, 115
- R1, 43
- R1., 44

R2, [43](#), [45](#)

RAM, [116](#)

read-uncommitted mode, [143](#)

read_uncommitted, [143](#)

reserved byte, [134](#)

rollback journal, [132](#)

shared cache mode, [125](#)

shared region, [134](#)

SQLITE_DONE, [117](#)

SQLITE_ROW, [117](#)

Temporary storage, [119](#)

vacuum, [16](#)

VDBE, [117](#)

wrapped query, [115](#)

Литература

- [1] Mike Owens Grant Allen. The definitive guide to sqlite, 2010. <http://www.apress.com/9781430232254>. 82

Оглавление

1	НАЧАЛО	2
1.1	Получение исходных кодов примеров книги	2
2	ВВЕДЕНИЕ В SQLite	3
2.1	Встраиваемая база данных	3
2.2	Архитектура	4
2.2.1	Интерфейс	4
2.2.2	Компилятор	4
2.2.3	Виртуальная Машина	5
2.3	Особенности и философия	5
2.4	Примеры	5
3	НАЧИНАЕМ	6
3.1	Где брать SQLite ?	6
3.2	SQLite под Windows	6
3.2.1	Загрузка CLP	6
3.2.2	Загрузка DLL	7
3.3	Утилита CLP	7
3.3.1	Интерактивное использование CLP	8
3.3.2	CLP в пакетном режиме	8
3.4	Администрирование	9
3.4.1	Создаем файл базы данных	9
3.4.2	Получение информации о внутренней схеме базы дан- ных	10
3.4.3	Экспортирование данных	12
3.4.4	Импортирование данных	12
3.4.5	Форматирование	13
3.4.6	Экспортирование таблицы (Exporting Delimited Data)	14
3.4.7	Автоматизация обслуживания БД	14
3.4.8	Бекап базы данных	15
3.4.9	Получение информации о файле базы данных	16
3.5	Дополнительные утилиты	17
3.6	Ado.Net провайдер	18
3.6.1	Построение консольного приложения на C-Sharp	18
3.6.1.1	Три версии для выполнения оператора <i>select</i>	18
3.6.1.2	Метод Main и построение приложения	22
3.6.1.3	Выполнение параметризованного оператора редактирования (<i>ExecuteScalar</i>)	28

3.6.1.4	Выполнение одного оператора редактирования (<i>ExecuteNonQuery</i>)	30
4	ЯЗЫК SQL В SQLite	32
4.1	Пример базы данных	32
4.1.1	Подготовка БД	33
4.1.2	Выполнение примеров	33
4.2	Синтаксис	35
4.2.1	Операторы	36
4.2.2	Константы	37
4.2.3	Ключевые слова и идентификаторы	37
4.2.4	Комментарии	38
4.3	Создание базы данных	38
4.3.1	Создание таблиц	38
4.3.2	Обновление таблиц	39
4.4	Запросы к базе данных	40
4.4.1	Операции реляционной алгебры	40
4.4.2	<i>select</i> и конвейер операций	42
4.4.3	Выборка	44
4.4.3.1	Значения	45
4.4.3.2	Операции	45
4.4.3.3	Бинарные операции	46
4.4.3.4	Логические операции	48
4.4.3.5	Операция <i>LIKE</i> и <i>GLOB</i>	49
4.4.4	Ограничение и упорядочение	50
4.4.5	Функции и операции агрегирования	51
4.4.6	Группировки	53
4.4.7	Удаление повторяющихся записей	57
4.4.8	Соединение таблиц	57
4.4.8.1	Внутреннее соединение	59
4.4.8.2	Прямое произведение	60
4.4.8.3	Внешнее соединение	61
4.4.8.4	Естественное соединение	61
4.4.8.5	Предпочтительный синтаксис	62
4.4.9	Имена и алиасы	62
4.4.10	Подзапросы	64
4.4.11	Составные запросы	65
4.4.12	Условные выражения	67
4.4.13	Обработка значений <i>NULL</i> в SQLite	69
4.5	Итак	70
5	ПРОДОЛЖАЕМ ИЗУЧАТЬ SQL В SQLite	72
5.1	Изменение данных	72
5.1.1	Вставка записей	72
5.1.1.1	Вставка одной записи	73
5.1.1.2	Вставка множества записей	74
5.1.1.3	Опять о вставке множества записей	75
5.1.2	Обновление записей	76
5.1.3	Удаление записей	76
5.2	Целостность данных	77

5.2.1	Целостность сущности	77
5.2.1.1	Ограничение уникальности (<i>unique</i>)	78
5.2.1.2	Ограничение первичного ключа	79
5.2.2	Доменная целостность	82
5.2.2.1	Значения по умолчанию	83
5.2.2.2	Ограничение <i>NOT NULL</i>	83
5.2.2.3	Ограничение <i>check</i>	84
5.2.2.4	Внешние ключи	85
5.2.2.5	Сортирующие последовательности (<i>collation</i>)	87
5.2.3	Классы памяти	88
5.2.4	Представления	91
5.2.5	Индексы	92
5.2.5.1	Сортирующие последовательности	93
5.2.5.2	Применение индексов	93
5.2.6	Триггеры	95
5.2.6.1	Триггер при обновлении	95
5.2.6.2	Обработка ошибок	96
5.2.6.3	Обновляемые представления	96
5.3	Транзакции	97
5.3.1	Область видимости транзакций	98
5.3.2	Политика разрешения конфликтов	99
5.3.3	Блокировки в базе данных	102
5.3.4	Мертвые блокировки	103
5.3.5	Типы транзакций	104
5.4	Администрирование БД	105
5.4.1	Присоединение БД	105
5.4.2	Уплотнение БД	106
5.4.3	Конфигурирование БД	107
5.4.3.1	Размер кеша для соединения	107
5.4.3.2	Получение информации о базе данных	108
5.4.3.3	Перенос изменений в файл БД	109
5.4.3.4	Хранилище временных объектов	109
5.4.3.5	Размер страницы БД, кодировка и уплотнение	110
5.4.3.6	Отладка	110
5.4.4	Системный каталог	110
5.4.5	План запроса	111
5.5	Итак	111
6	ДИЗАЙН И КОНЦЕПЦИИ SQLite	112
6.1	Программный интерфейс	113
6.1.1	Основные структуры данных	113
6.1.1.1	Соединения и Операторы	113
6.1.1.2	В-дерево и Менеджер Страниц	114
6.1.2	Базовый Программный Интерфейс	115
6.1.2.1	Соединение с БД	116
6.1.2.2	Применение Функций Подготовленных Запросов	117
6.1.2.3	Использование Параметризованного SQL	119
6.1.2.4	Executing Wrapped Queries	121
6.1.2.5	Обработка Ошибок	122

6.1.2.6	Форматирование SQL Операторов	123
6.1.3	Управление Операторами	124
6.1.4	Использование Нитей (Потоков)	125
6.2	Расширенный Программный Интерфейс	125
6.2.1	Создание Функций, Определяемых Пользователем	125
6.2.2	Создание Групповых Функций, Определяемых Пользователем	126
6.2.3	Создание Определяемых Пользователем Сортирующихся Последовательностей	127
6.3	Транзакции	128
6.3.1	Жизненный Цикл Транзакции	128
6.3.2	Состояния Блокировок	129
6.3.3	Читающие Транзакции	130
6.3.4	Пишущие Транзакции	131
6.3.4.1	Состояние RESERVED	131
6.3.4.2	Состояние PENDING	132
6.3.4.3	Состояние EXCLUSIVE	133
6.3.4.4	Автофиксация и Производительность	134
6.4	Настройка Кэша Страниц	136
6.4.1	Переход в Состояние EXCLUSIVE	136
6.4.2	Выбор Размера Кэша	137
6.5	Ожидание Блокировок	138
6.5.1	Использование Обработчика Занятости	138
6.5.2	Правильное Использование Транзакций	139
6.6	Прикладные Программы	141
6.6.1	Использование Нескольких Соединений	141
6.6.2	Важность Финализации	142
6.6.3	Режим Разделяемого Кэша	143
6.7	Итак	144
7	The Core C API	145
8	The Extension C API	146
9	SQLite Internals and New Features	147