SQLite 高级教程

1. PRAGMA（编译指示、注解）

PRAGMA 命令是一个特殊的命令，可以用在 SQLite 环境内控制各种环境变量和状态标志。一个 PRAGMA 值可以被读取，也可以根据需求进行设置。

1.1 语法

要查询当前的 PRAGMA 值，只需要提供该 pragma 的名字：

```
PRAGMA pragma_name;
```

要为 PRAGMA 设置一个新的值，语法如下：

```
PRAGMA pragma_name = value;
```

设置模式，可以是名称或等值的整数，但返回的值将始终是一个整数。

1.2 各种 Pragma

1.2.1 auto_vacuum（清理）

auto_vacuum Pragma 获取或设置 auto-vacuum 模式。语法如下：

```
PRAGMA [database.]auto_vacuum;
PRAGMA [database.]auto_vacuum = mode;
```

其中，**mode** 可以是以下任何一种：

Pragma 值	描述
0 或 NONE	禁用 Auto-vacuum。这是默认模式，意味着数据库文件尺寸大小不会缩小，除非手动使用 VACUUM 命令。
1 或 FULL	启用 Auto-vacuum，是全自动的。在该模式下，允许数据库文件随着数据从数据库移除而缩小。
2 或 INCREMENTAL	启用 Auto-vacuum，但是必须手动激活。在该模式下，引用数据被维持，自由页面只放在自由列表中。这些页面可在任何时候使用 **incremental_vacuum pragma** 进行覆盖。

1.2.2 cache_size

cache_size Pragma 可获取或暂时设置在内存中页面缓存的最大尺寸。语法如下：

```
1 PRAGMA [database.]cache_size;
2 PRAGMA [database.]cache_size = pages;
```

pages 值表示在缓存中的页面数。内置页面缓存的默认大小为 2,000 页，最小尺寸为 10 页

1.2.3 case_sensitive_like Pragma

case_sensitive_like Pragma 控制内置的 LIKE 表达式的大小写敏感度。默认情况下，该 Pragma 为 false，这意味着，内置的 LIKE 操作符忽略字母的大小写。语法如下：

```
1 PRAGMA case_sensitive_like = [true|false];
```

目前没有办法查询该 Pragma 的当前状态。

1.2.4 count_changes Pragma

count_changes Pragma 获取或设置数据操作语句的返回值，如 INSERT、UPDATE 和 DELETE。语法如下

```
1 PRAGMA count_changes;
2 PRAGMA count_changes = [true|false];
```

默认情况下，该 Pragma 为 false，这些语句不返回任何东西。如果设置为 true，每个所提到的语句将返回一个单行单列的表，由一个单一的整数值组成，该整数表示操作影响的行。

1.2.5 database_list Pragma

database_list Pragma 将用于列出了所有的数据库连接。语法如下：

```
1 PRAGMA database_list;
```

该 Pragma 将返回一个单行三列的表格，每当打开或附加数据库时，会给出数据库中的序列号，它的名称和相关的文件。

1.2.6 encoding Pragma

encoding Pragma 控制字符串如何编码及存储在数据库文件中。语法如下：

```
1 PRAGMA encoding;
2 PRAGMA encoding = format;
```

格式值可以是 UTF-8、UTF-16le 或 UTF-16be 之一。

1.2.7 freelist_count Pragma

freelist_count Pragma 返回一个整数，表示当前被标记为免费和可用的数据库页数。语法如下：

```
1 PRAGMA [database.]freelist_count;
```

1.2.8 index_info Pragma

index_info Pragma 返回关于数据库索引的信息。语法如下：

```
1 PRAGMA [database.]index_info( index_name );
```

结果集将为每个包含在给出列序列的索引、表格内的列索引、列名称的列显示一行。

1.2.9 index_list Pragma

index_list Pragma 列出所有与表相关联的索引。语法如下：

```
1 PRAGMA [database.]index_list( table_name );
```

结果集将为每个给出列序列的索引、索引名称、表示索引是否唯一的标识显示一行。

1.2.10 journal_mode Pragma

journal_mode Pragma 获取或设置控制日志文件如何存储和处理的日志模式。语法如下：

```
PRAGMA journal_mode;
PRAGMA journal_mode = mode;
PRAGMA database.journal_mode;
PRAGMA database.journal_mode = mode;
```

这里支持五种日志模式：

Pragma 值	描述
DELETE	默认模式。在该模式下，在事务结束时，日志文件将被删除。
TRUNCATE	日志文件被截断为零字节长度。
PERSIST	日志文件被留在原地，但头部被重写，表明日志不再有效。
MEMORY	日志记录保留在内存中，而不是磁盘上。
OFF	不保留任何日志记录。

1.2.11 max_page_count Pragma

max_page_count Pragma 为数据库获取或设置允许的最大页数。语法如下：

```
PRAGMA [database.]max_page_count;
PRAGMA [database.]max_page_count = max_page;
```

默认值是 1,073,741,823，这是一个千兆的页面，即如果默认 1 KB 的页面大小，那么数据库中增长起来的一个兆字节。

1.2.12 page_count Pragma

page_count Pragma 返回当前数据库中的网页数量。语法如下：

```
PRAGMA [database.]page_count;
```

数据库文件的大小应该是 page_count * page_size。

1.2.13 page_size Pragma

page_size Pragma 获取或设置数据库页面的大小。

```
PRAGMA [database.]page_size;
PRAGMA [database.]page_size = bytes;
```

默认情况下，允许的尺寸是 512、1024、2048、4096、8192、16384、32768 字节。改变现有数据库页面大小的唯一方法就是设置页面大小，然后立即 VACUUM 该数据库。

1.2.14 parser_trace Pragma

parser_trace Pragma 随着它解析 SQL 命令来控制打印的调试状态

```
PRAGMA parser_trace = [true|false];
```

默认情况下，它被设置为 false，但设置为 true 时则启用，此时 SQL 解析器会随着它解析 SQL 命令来打印出它的状态。

1.2.15 recursive_triggers

recursive_triggers Pragma 获取或设置递归触发器功能。如果未启用递归触发器，一个触发动作将不会触发另一个触发。

```
PRAGMA recursive_triggers;
PRAGMA recursive_triggers = [true|false]
```

1.2.16 schema_version

schema_version Pragma 获取或设置存储在数据库头中的的架构版本值。

```
PRAGMA [database.]schema_version;
PRAGMA [database.]schema_version = number;
```

这是一个 32 位有符号整数值，用来跟踪架构的变化。每当一个架构改变命令执行（比如 CREATE... 或 DROP...）时，这个值会递增。

1.2.17 secure_delete

secure_delete Pragma 用来控制内容是如何从数据库中删除。

```
PRAGMA secure_delete;
```

```
PRAGMA secure_delete = [true|false];
PRAGMA database.secure_delete;
PRAGMA database.secure_delete = [true|false];
```

安全删除标志的默认值通常是关闭的，但是这是可以通过 SQLITE_SECURE_DELETE 构建选项来改变的。

1.2.18 sql_trace

sql_trace Pragma 用于把 SQL 跟踪结果转储到屏幕上。

```
PRAGMA sql_trace;
PRAGMA sql_trace = [true|false];
```

SQLite 必须通过 SQLITE_DEBUG 指令来编译要引用的该 Pragma。

1.2.19 synchronous

synchronous Pragma 获取或设置当前磁盘的同步模式，该模式控制积极的 SQLite 如何将数据写入物理存储。语法如下

```
PRAGMA [database.]synchronous;
PRAGMA [database.]synchronous = mode;
```

SQLite 支持下列同步模式：

Pragma 值	描述
0 或 OFF	不进行同步。
1 或 NORMAL	在关键的磁盘操作的每个序列后同步。
2 或 FULL	在每个关键的磁盘操作后同步。

1.2.20 temp_store

temp_store Pragma 获取或设置临时数据库文件所使用的存储模式。

```
PRAGMA temp_store;
PRAGMA temp_store = mode;

```

SQLite 支持下列存储模式：

Pragma 值	描述
0 或 DEFAULT	默认使用编译时的模式。通常是 FILE。
1 或 FILE	使用基于文件的存储。
2 或 MEMORY	使用基于内存的存储。

1.2.21 temp_store_directory

temp_store_directory Pragma 获取或设置用于临时数据库文件的位置

```
PRAGMA temp_store_directory;
PRAGMA temp_store_directory = 'directory_path';
```

1.2.22 user_version

user_version Pragma 获取或设置存储在数据库头的用户自定义的版本值。

```
PRAGMA [database.]user_version;
PRAGMA [database.]user_version = number;
```

这是一个 32 位的有符号整数值，可以由开发人员设置，用于版本跟踪的目的。

1.2.23 writable_schema

writable_schema Pragma 获取或设置是否能够修改系统表。语法如下：

```
PRAGMA writable_schema;
PRAGMA writable_schema = [true|false];
```

如果设置了该 Pragma，则表以 sqlite_ 开始，可以创建和修改，包括 sqlite_master 表。使用该 Pragma 时要注意，因为它可能导致整个数据库损坏。

2. 约束

约束是在表的数据列上强制执行的规则，这些是用来限制可以插入到表中的数据类型，这确保了数据库中数据的准确性和可靠性。

约束可以是列级或表级。列级约束仅适用于列，表级约束被应用到整个表。

以下是在 SQLite 中常用的约束。

- ***NOT NULL 约束**：确保某列不能有 NULL 值。*
- ***DEFAULT 约束**：当某列没有指定值时，为该列提供默认值。*
- ***UNIQUE 约束**：确保某列中的所有值是不同的。*
- ***PRIMARY Key 约束**：唯一标识数据库表中的各行/记录。*
- ***CHECK 约束**：CHECK 约束确保某列中的所有值满足一定条件。*

2.1 NOT NULL 约束

默认情况下，列可以保存 NULL 值。如果您不想某列有 NULL 值，那么需要在该列上定义此约束，指定在该列上不允许 NULL 值。

NULL 与没有数据是不一样的，它代表着未知的数据。

实例

创建一个新的表 COMPANY，并增加了五列，其中 ID、NAME 和 AGE 三列指定不接受 NULL 值：

```
CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
```

2.2 DEFAULT 约束

DEFAULT 约束在 INSERT INTO 语句没有提供一个特定的值时，为列提供一个默认值。

实例

创建一个新的表 COMPANY，并增加了五列。在这里，SALARY 列默认设置为 5000.00。所以当 INSERT INTO 语句没有为该列提供值时，该列将被设置为 5000.00。

```
CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL    DEFAULT 50000.00
);
```

2.3 UNIQUE 约束

UNIQUE 约束防止在一个特定的列存在两个记录具有相同的值。在 COMPANY 表中，例如，您可能要防止两个或两个以上的人具有相同的年龄。

实例

创建一个新的表 COMPANY，并增加了五列。在这里，AGE 列设置为 UNIQUE，所以不能有两个相同年龄的记录：

```
CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL UNIQUE,
   ADDRESS        CHAR(50),
   SALARY         REAL    DEFAULT 50000.00
);
```

2.4 PRIMARY KEY 约束

PRIMARY KEY 约束唯一标识数据库表中的每个记录。在一个表中可以有多个 UNIQUE 列，但只能有一个主键。在设计数据库表时，主键是很重要的。主键是唯一的 ID。

我们使用主键来引用表中的行。可通过把主键设置为其他表的外键，来创建表之间的关系。由于"长期存在编码监督"，在 SQLite 中，主键可以是 NULL，这是与其他数据库不同的地方。

主键是表中的一个字段，唯一标识数据库表中的各行/记录。主键必须包含唯一值。主键列不能有 NULL 值。

一个表只能有一个主键，它可以由一个或多个字段组成。当多个字段作为主键，它们被称为**复合键**。

如果一个表在任何字段上定义了一个主键，那么在这些字段上不能有两个记录具有相同的值。

实例

创建以 ID 作为主键的 COMAPNY 表的各种实例：

```
CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
```

2.5 CHECK 约束

CHECK 约束启用输入一条记录要检查值的条件。如果条件值为 false，则记录违反了约束，且不能输入到表。

实例

例如，下面的 SQLite 创建一个新的表 COMPANY，并增加了五列。在这里，我们为 SALARY 列添加 CHECK，所以工资不能为零：

```
CREATE TABLE COMPANY3(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL    CHECK(SALARY > 0)
);
```

2.6 删除约束

在 SQLite 中，要删除表的约束，通常需要使用 **ALTER TABLE** 语句，并指定要删除的约束类型。

删除主键约束:

```
ALTER TABLE table_name
DROP CONSTRAINT primary_key_name;
```

在这里，**table_name** 是你要操作的表名，**primary_key_name** 是要删除的主键约束的名称。

删除唯一约束:

```
ALTER TABLE table_name
DROP CONSTRAINT unique_constraint_name;
```

同样，**table_name** 是表名，**unique_constraint_name** 是要删除的唯一约束的名称。

删除外键约束:

```
ALTER TABLE table_name
DROP CONSTRAINT foreign_key_constraint_name;
```

在这里，**table_name** 是表名，**foreign_key_constraint_name** 是要删除的外键约束的名称。

3. Join

***Join** 子句用于结合两个或多个数据库中表的记录。JOIN 是一种通过共同值来结合两个表中字段的手段。*

SQL 定义了三种主要类型的连接：

- *交叉连接 - CROSS JOIN*
- *内连接 - INNER JOIN*
- *外连接 - OUTER JOIN*

假设有两个表 COMPANY 和 DEPARTMENT。我们已经看到了用来填充 COMPANY 表的 INSERT 语句。现在让我们假设 COMPANY 表的记录列表如下：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

另一个表是 DEPARTMENT，定义如下：

```
CREATE TABLE DEPARTMENT(
   ID INT PRIMARY KEY      NOT NULL,
   DEPT           CHAR(50) NOT NULL,
   EMP_ID         INT      NOT NULL
);
```

填充 DEPARTMENT 表的 INSERT 语句：

```
INSERT INTO DEPARTMENT (ID, DEPT, EMP_ID)
VALUES (1, 'IT Billing', 1 );

INSERT INTO DEPARTMENT (ID, DEPT, EMP_ID)
```

```
VALUES (2, 'Engineering', 2 );

INSERT INTO DEPARTMENT (ID, DEPT, EMP_ID)
VALUES (3, 'Finance', 7 );
```

DEPARTMENT 表中有下列的记录列表：

```
ID          DEPT        EMP_ID
----------  ----------  ----------
1           IT Billing  1
2           Engineerin  2
3           Finance     7
```

3.1 交叉连接 - CROSS JOIN

交叉连接（CROSS JOIN）把第一个表的每一行与第二个表的每一行进行匹配。如果两个输入表分别有 x 和 y 行，则结果表有 x*y 行。由于交叉连接（CROSS JOIN）有可能产生非常大的表，使用时必须谨慎，只在适当的时候使用它们。

交叉连接的操作，它们都返回被连接的两个表所有数据行的笛卡尔积，返回到的数据行数等于第一个表中符合查询条件的数据行数乘以第二个表中符合查询条件的数据行数。

交叉连接（CROSS JOIN）的语法：

```
SELECT ... FROM table1 CROSS JOIN table2 ...
```

基于上面的表，我们可以写一个交叉连接（CROSS JOIN），如下所示：

```
sqlite> SELECT EMP_ID, NAME, DEPT FROM COMPANY CROSS JOIN DEPARTMENT;
```

3.2 内连接 - INNER JOIN

内连接（INNER JOIN）根据连接谓词结合两个表（table1 和 table2）的列值来创建一个新的结果表。查询会把 table1 中的每一行与 table2 中的每一行进行比较，找到所有满足连接谓词的行的匹配对。当满足连接谓词时，A 和 B 行的每个匹配对的列值会合并成一个结果行。

内连接（INNER JOIN）是最常见的连接类型，是默认的连接类型。INNER 关键字是可选的。

下面是内连接（INNER JOIN）的语法：

```
SELECT ... FROM table1 [INNER] JOIN table2 ON conditional_expression ...
```

为了避免冗余，并保持较短的措辞，可以使用 **USING** 表达式声明内连接（INNER JOIN）条件。这个表达式指定一个或多个列的列表：

```
SELECT ... FROM table1 JOIN table2 USING ( column1 ,... ) ...
```

自然连接（NATURAL JOIN）类似于 **JOIN...USING**，只是它会自动测试存在两个表中的每一列的值之间相等值：

```
SELECT ... FROM table1 NATURAL JOIN table2...
```

基于上面的表，我们可以写一个内连接（INNER JOIN），如下所示：

```
sqlite> SELECT EMP_ID, NAME, DEPT FROM COMPANY INNER JOIN DEPARTMENT
        ON COMPANY.ID = DEPARTMENT.EMP_ID;
```

上面的查询会产生以下结果：

```
EMP_ID      NAME        DEPT
----------  ----------  ----------
1           Paul        IT Billing
2           Allen       Engineerin
7           James       Finance
```

3.3 外连接 - OUTER JOIN

外连接（OUTER JOIN）是内连接（INNER JOIN）的扩展。虽然 SQL 标准定义了三种类型的外连接：LEFT、RIGHT、FULL，但 SQLite 只支持 **左外连接（LEFT OUTER JOIN）**。

外连接（OUTER JOIN）声明条件的方法与内连接（INNER JOIN）是相同的，使用 ON、USING 或 NATURAL 关键字来表达。最初的结果表以相同的方式进行计算。一旦主连接计算完成，外连接（OUTER JOIN）将从一个或两个表中任何未连接的行合并进来，外连接的列使用 NULL 值，将它们附加到结果表中。

下面是左外连接（LEFT OUTER JOIN）的语法：

```
SELECT ... FROM table1 LEFT OUTER JOIN table2 ON conditional_expression ...
```

为了避免冗余，并保持较短的措辞，可以使用 **USING** 表达式声明外连接（OUTER JOIN）条件。这个表达式指定一个或多个列的列表：

```
SELECT ... FROM table1 LEFT OUTER JOIN table2 USING ( column1 ,... ) ...
```

基于上面的表，我们可以写一个外连接（OUTER JOIN），如下所示：

```
sqlite> SELECT EMP_ID, NAME, DEPT FROM COMPANY LEFT OUTER JOIN DEPARTMENT
        ON COMPANY.ID = DEPARTMENT.EMP_ID;
```

上面的查询会产生以下结果：

```
EMP_ID      NAME        DEPT
----------  ----------  ----------
1           Paul        IT Billing
2           Allen       Engineerin
            Teddy
            Mark
            David
            Kim
7           James       Finance
```

4. Unions 子句

***UNION** 子句/运算符用于合并两个或多个 SELECT 语句的结果，不返回任何重复的行。*

为了使用 UNION，每个 SELECT 被选择的列数必须是相同的，相同数目的列表达式，相同的数据类型，并确保它们有相同的顺序，但它们不必具有相同的长度。

4.1 语法

UNION 的基本语法如下：

```
SELECT column1 [, column2 ]
```

```
FROM table1 [, table2 ]
[WHERE condition]

UNION

SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
```

这里给定的条件根据需要可以是任何表达式。

4.2 实例

假设有下面两个表，（1）COMPANY 表如下所示：

```
sqlite> select * from COMPANY;
ID          NAME                AGE         ADDRESS     SALARY
----------  ------------------  ----------  ----------  ----------
1           Paul                32          California  20000.0
2           Allen               25          Texas       15000.0
3           Teddy               23          Norway      20000.0
4           Mark                25          Rich-Mond   65000.0
5           David               27          Texas       85000.0
6           Kim                 22          South-Hall  45000.0
7           James               24          Houston     10000.0
```

（2）另一个表是 DEPARTMENT，如下所示：

```
ID          DEPT                EMP_ID
----------  ------------------  ----------
1           IT Billing          1
2           Engineering         2
3           Finance             7
4           Engineering         3
5           Finance             4
6           Engineering         5
7           Finance             6
```

现在，让我们使用 SELECT 语句及 UNION 子句来连接两个表，如下所示：

```
sqlite> SELECT EMP_ID, NAME, DEPT FROM COMPANY INNER JOIN DEPARTMENT
```

```
      ON COMPANY.ID = DEPARTMENT.EMP_ID
UNION
  SELECT EMP_ID, NAME, DEPT FROM COMPANY LEFT OUTER JOIN DEPARTMENT
     ON COMPANY.ID = DEPARTMENT.EMP_ID;
```

这将产生以下结果：

```
EMP_ID      NAME                 DEPT
----------  -------------------  ----------
1           Paul                 IT Billing
2           Allen                Engineerin
3           Teddy                Engineerin
4           Mark                 Finance
5           David                Engineerin
6           Kim                  Finance
7           James                Finance
```

4.3 UNION ALL 子句

UNION ALL 运算符用于结合两个 SELECT 语句的结果，包括重复行。

适用于 UNION 的规则同样适用于 UNION ALL 运算符。

语法

UNION ALL 的基本语法如下：

```
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]

UNION ALL

SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
```

这里给定的条件根据需要可以是任何表达式。

实例

使用 SELECT 语句及 UNION ALL 子句来连接两个表，如下所示：

```
sqlite> SELECT EMP_ID, NAME, DEPT FROM COMPANY INNER JOIN DEPARTMENT
        ON COMPANY.ID = DEPARTMENT.EMP_ID
   UNION ALL
     SELECT EMP_ID, NAME, DEPT FROM COMPANY LEFT OUTER JOIN DEPARTMENT
        ON COMPANY.ID = DEPARTMENT.EMP_ID;
```

这将产生以下结果：

```
EMP_ID      NAME                DEPT
----------  ------------------  ----------
1           Paul                IT Billing
2           Allen               Engineerin
3           Teddy               Engineerin
4           Mark                Finance
5           David               Engineerin
6           Kim                 Finance
7           James               Finance
1           Paul                IT Billing
2           Allen               Engineerin
3           Teddy               Engineerin
4           Mark                Finance
5           David               Engineerin
6           Kim                 Finance
7           James               Finance
```

5. NULL 值

NULL 是用来表示一个缺失值的项。表中的一个 NULL 值是在字段中显示为空白的一个值。

带有 NULL 值的字段是一个不带有值的字段。NULL 值与零值或包含空格的字段是不同的。

5.1 语法

创建表时使用 **NULL** 的基本语法如下：

```
SQLite> CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
```

```
);
```

在这里，**NOT NULL** 表示列总是接受给定数据类型的显式值。这里有两个列我们没有使用 NOT NULL，这意味着这两个列可以为 NULL。

带有 NULL 值的字段在记录创建的时候可以保留为空。

5.2 实例

NULL 值在选择数据时会引起问题，因为当把一个未知的值与另一个值进行比较时，结果总是未知的，且不会包含在最后的结果中。假设有下面的表，COMPANY 的记录如下所示：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

使用 UPDATE 语句来设置一些允许空值的值为 NULL，如下所示：

```
sqlite> UPDATE COMPANY SET ADDRESS = NULL, SALARY = NULL where ID IN(6,7);
```

现在，COMPANY 表的记录如下所示：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22
7           James       24
```

接下来，看看 **IS NOT NULL** 运算符的用法，它用来列出所有 SALARY 不为 NULL 的记录：

```
sqlite> SELECT  ID, NAME, AGE, ADDRESS, SALARY
        FROM COMPANY
        WHERE SALARY IS NOT NULL;
```

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
6           Kim         22
7           James       24
```

6. 别名

*可以暂时把表或列重命名为另一个名字，这被称为**别名**。使用表别名是指在一个特定的 SQLite 语句中重命名表。重命名是临时的改变，在数据库中实际的表的名称不会改变。*

列别名用来为某个特定的 SQLite 语句重命名表中的列。

6.1 语法

表 别名的基本语法如下：

```
SELECT column1, column2....
FROM table_name AS alias_name
WHERE [condition];
```

列 别名的基本语法如下：

```
SELECT column_name AS alias_name
FROM table_name
WHERE [condition];
```

6.2 实例

假设有下面两个表，（1）COMPANY 表如下所示：

```
sqlite> select * from COMPANY;
ID          NAME                AGE         ADDRESS     SALARY
----------  ------------------  ----------  ----------  ----------
```

```
4  1          Paul                32          California  20000.0
5  2          Allen               25          Texas       15000.0
6  3          Teddy               23          Norway      20000.0
7  4          Mark                25          Rich-Mond   65000.0
8  5          David               27          Texas       85000.0
9  6          Kim                 22          South-Hall  45000.0
10 7          James               24          Houston     10000.0
```

（2）另一个表是 DEPARTMENT，如下所示：

```
1 ID          DEPT                EMP_ID
2 ----------  ------------------  ----------
3 1           IT Billing          1
4 2           Engineering         2
5 3           Finance             7
6 4           Engineering         3
7 5           Finance             4
8 6           Engineering         5
9 7           Finance             6
```

现在，下面是 **表别名** 的用法，在这里我们使用 C 和 D 分别作为 COMPANY 和 DEPARTMENT 表的别名：

```
1 sqlite> SELECT C.ID, C.NAME, C.AGE, D.DEPT
2         FROM COMPANY AS C, DEPARTMENT AS D
3         WHERE  C.ID = D.EMP_ID;
```

上面的 SQLite 语句将产生下面的结果：

```
1 ID          NAME        AGE         DEPT
2 ----------  ----------  ----------  ----------
3 1           Paul        32          IT Billing
4 2           Allen       25          Engineerin
5 3           Teddy       23          Engineerin
6 4           Mark        25          Finance
7 5           David       27          Engineerin
8 6           Kim         22          Finance
9 7           James       24          Finance
```

让我们看一个 **列别名** 的实例，在这里 COMPANY_ID 是 ID 列的别名，COMPANY_NAME 是 name 列的别名：

```
sqlite> SELECT C.ID AS COMPANY_ID, C.NAME AS COMPANY_NAME, C.AGE, D.DEPT
        FROM COMPANY AS C, DEPARTMENT AS D
        WHERE  C.ID = D.EMP_ID;
```

上面的 SQLite 语句将产生下面的结果：

```
COMPANY_ID  COMPANY_NAME  AGE         DEPT
----------  ------------  ----------  ----------
1           Paul          32          IT Billing
2           Allen         25          Engineerin
3           Teddy         23          Engineerin
4           Mark          25          Finance
5           David         27          Engineerin
6           Kim           22          Finance
7           James         24          Finance
```

7. 触发器

***触发器（Trigger）**是数据库的回调函数，它会在指定的数据库事件发生时自动执行/调用。以下是关于 SQLite 的触发器（Trigger）的要点：*

- *SQLite 的触发器（Trigger）可以指定在特定的数据库表发生 DELETE、INSERT 或 UPDATE 时触发，或在一个或多个指定表的列发生更新时触发。*
- *SQLite 只支持 FOR EACH ROW 触发器（Trigger），没有 FOR EACH STATEMENT 触发器（Trigger）。因此，明确指定 FOR EACH ROW 是可选的。*
- *WHEN 子句和触发器（Trigger）动作可能访问使用表单 **NEW.column-name** 和 **OLD.column-name** 的引用插入、删除或更新的行元素，其中 column-name 是从与触发器关联的表的列的名称。*
- *如果提供 WHEN 子句，则只针对 WHEN 子句为真的指定行执行 SQL 语句。如果没有提供 WHEN 子句，则针对所有行执行 SQL 语句。*
- *BEFORE 或 AFTER 关键字决定何时执行触发器动作，决定是在关联行的插入、修改或删除之前或者之后执行触发器动作。*
- *当触发器相关联的表删除时，自动删除触发器（Trigger）。*

- *要修改的表必须存在于同一数据库中，作为触发器被附加的表或视图，且必须只使用 **tablename**，而不是 **database.tablename**。*
- *一个特殊的 SQL 函数 RAISE() 可用于触发器程序内抛出异常。*

7.1 语法

创建 **触发器（Trigger）** 的基本语法如下：

```
CREATE  TRIGGER trigger_name [BEFORE|AFTER] event_name
ON table_name
BEGIN
  -- 触发器逻辑....
END;
```

在这里，**event_name** 可以是在所提到的表 **table_name** 上的 *INSERT*、*DELETE* 和 *UPDATE* 数据库操作。您可以在表名后选择指定 FOR EACH ROW。

以下是在 UPDATE 操作上在表的一个或多个指定列上创建触发器（Trigger）的语法：

```
CREATE  TRIGGER trigger_name [BEFORE|AFTER] UPDATE OF column_name
ON table_name
BEGIN
  -- 触发器逻辑....
END;
```

7.2 实例

假设一个情况，我们要为被插入到新创建的 COMPANY 表（如果已经存在，则删除重新创建）中的每一个记录保持审计试验：

```
sqlite> CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
```

为了保持审计试验，我们将创建一个名为 AUDIT 的新表。每当 COMPANY 表中有一个新的记录项时，日志消息将被插入其中：

```
sqlite> CREATE TABLE AUDIT(
    EMP_ID INT NOT NULL,
    ENTRY_DATE TEXT NOT NULL
);
```

在这里，ID 是 AUDIT 记录的 ID，EMP_ID 是来自 COMPANY 表的 ID，DATE 将保持 COMPANY 中记录被创建时的时间戳。所以，现在让我们在 COMPANY 表上创建一个触发器，如下所示：

```
sqlite> CREATE TRIGGER audit_log AFTER INSERT
ON COMPANY
BEGIN
   INSERT INTO AUDIT(EMP_ID, ENTRY_DATE) VALUES (new.ID, datetime('now'));
END;
```

现在，我们将开始在 COMPANY 表中插入记录，这将导致在 AUDIT 表中创建一个审计日志记录。因此，让我们在 COMPANY 表中创建一个记录，如下所示：

```
sqlite> INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Paul', 32, 'California', 20000.00 );
```

这将在 COMPANY 表中创建如下一个记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
```

同时，将在 AUDIT 表中创建一个记录。这个纪录是触发器的结果，这是我们在 COMPANY 表上的 INSERT 操作上创建的触发器（Trigger）。类似的，可以根据需要在 UPDATE 和 DELETE 操作上创建触发器（Trigger）。

```
EMP_ID      ENTRY_DATE
----------  -------------------
1           2013-04-05 06:26:00
```

7.3 列出触发器（TRIGGERS）

可以从 **sqlite_master** 表中列出所有触发器，如下所示：

```
sqlite> SELECT name FROM sqlite_master
WHERE type = 'trigger';
```

```
name
----------
audit_log
```

想要列出特定表上的触发器，则使用 AND 子句连接表名，如下所示：

```
sqlite> SELECT name FROM sqlite_master
WHERE type = 'trigger' AND tbl_name = 'COMPANY';
```

7.4 删除触发器（TRIGGERS）

下面是 DROP 命令，可用于删除已有的触发器：

```
sqlite> DROP TRIGGER trigger_name;
```

7.5 for each row 和 when

关于 for each row 和 when 的实例，这里补充一下。

for each row 是操作语句每影响到一行的时候就触发一次，也就是删了 10 行就触发 10 次，而 for each state 一条操作语句就触发一次，有时没有被影响的行也执行。sqlite 只实现了 for each row 的触发。when 和 for each row 用法是这样的：

```
CREATE TRIGGER trigger_name
AFTER UPDATE OF id ON table_1
FOR EACH ROW
WHEN new.id>30
BEGIN
UPDATE table_2 SET id=new.id WHERE table_2.id=old.id;
END;
```

上面的触发器在 table_1 改 id 的时候如果新的 id>30 就把 表table_2 中和表table_1 id 相等的行一起改为新的 id

8. 索引

索引（Index）是一种特殊的查找表，数据库搜索引擎用来加快数据检索。简单地说，索引是一个指向表中数据的指针。一个数据库中的索引与一本书的索引目录是非常相似的。

拿汉语字典的目录页（索引）打比方，我们可以按拼音、笔画、偏旁部首等排序的目录（索引）快速查找到需要的字。

索引有助于加快 SELECT 查询和 WHERE 子句，但它会减慢使用 UPDATE 和 INSERT 语句时的数据输入。索引可以创建或删除，但不会影响数据。

使用 CREATE INDEX 语句创建索引，它允许命名索引，指定表及要索引的一列或多列，并指示索引是升序排列还是降序排列。

索引也可以是唯一的，与 UNIQUE 约束类似，在列上或列组合上防止重复条目。

8.1 CREATE INDEX 命令

CREATE INDEX 的基本语法如下：

```
1 CREATE INDEX index_name ON table_name;
```

8.2 单列索引

单列索引是一个只基于表的一个列上创建的索引。基本语法如下：

```
1 CREATE INDEX index_name
2 ON table_name (column_name);
```

8.3 唯一索引

使用唯一索引不仅是为了性能，同时也为了数据的完整性。唯一索引不允许任何重复的值插入到表中。基本语法如下：

```
1 CREATE UNIQUE INDEX index_name
2 on table_name (column_name);
```

8.4 组合索引

组合索引是基于一个表的两个或多个列上创建的索引。

```
CREATE INDEX index_name
on table_name (column1, column2);
```

是否要创建一个单列索引还是组合索引，要考虑到您在作为查询过滤条件的 WHERE 子句中使用非常频繁的列。

如果只使用到一个列，则选择使用单列索引。如果在作为过滤的 WHERE 子句中有两个或多个列经常使用，则选择使用组合索引。

8.5 隐式索引

隐式索引是在创建对象时，由数据库服务器自动创建的索引。索引自动创建为主键约束和唯一约束。

8.6 实例

在 COMPANY 表的 salary 列上创建一个索引：

```
sqlite> CREATE INDEX salary_index ON COMPANY (salary);
```

现在，让我们使用 **.indices** 或 **.indexes** 命令列出 COMPANY 表上所有可用的索引，如下所示：

```
sqlite> .indices COMPANY
```

这将产生如下结果，其中 *sqlite_autoindex_COMPANY_1* 是创建表时创建的隐式索引。

```
salary_index
sqlite_autoindex_COMPANY_1
```

可以列出数据库范围的所有索引，如下所示：

```
sqlite> SELECT * FROM sqlite_master WHERE type = 'index';
```

8.7 DROP INDEX 命令

一个索引可以使用 SQLite 的 **DROP** 命令删除。当删除索引时应特别注意，因为性能可能会下降或提高。

```
DROP INDEX index_name;
```

8.8 什么情况下要避免使用索引?

虽然索引的目的在于提高数据库的性能，但这里有几个情况需要避免使用索引。使用索引时，应重新考虑下列准则：

- 索引不应该使用在较小的表上。
- 索引不应该使用在有频繁的大批量的更新或插入操作的表上。
- 索引不应该使用在含有大量的 NULL 值的列上。
- 索引不应该使用在频繁操作的列上

9. Indexed By

"INDEXED BY index-name" 子句规定必须需要命名的索引来查找前面表中值。

如果索引名 index-name 不存在或不能用于查询，然后 SQLite 语句的准备失败。

"NOT INDEXED" 子句规定当访问前面的表（包括由 UNIQUE 和 PRIMARY KEY 约束创建的隐式索引）时，没有使用索引。

然而，即使指定了 "NOT INDEXED"，INTEGER PRIMARY KEY 仍然可以被用于查找条目。

9.1 语法

INDEXED BY 子句的语法，它可以与 DELETE、UPDATE 或 SELECT 语句一起使用：

```
SELECT|DELETE|UPDATE column1, column2...
INDEXED BY (index_name)
table_name
WHERE (CONDITION);
```

9.2 实例

假设有表 COMPANY，我们将创建一个索引，并用它进行 INDEXED BY 操作。

```
sqlite> CREATE INDEX salary_index ON COMPANY(salary);
```

现在使用 INDEXED BY 子句从表 COMPANY 中选择数据，如下所示：

```
sqlite> SELECT * FROM COMPANY INDEXED BY salary_index WHERE salary > 5000;
```

10. Alter 命令

***ALTER TABLE** 命令不通过执行一个完整的转储和数据的重载来修改已有的表。您可以使用 ALTER TABLE 语句重命名表，使用 ALTER TABLE 语句还可以在已有的表中添加额外的列。*

在 SQLite 中，除了重命名表和在已有的表中添加列，ALTER TABLE 命令不支持其他操作。

10.1 语法

用来重命名已有的表的 **ALTER TABLE** 的基本语法如下：

```
ALTER TABLE database_name.table_name RENAME TO new_table_name;
```

用来在已有的表中添加一个新的列的 **ALTER TABLE** 的基本语法如下：

```
ALTER TABLE database_name.table_name ADD COLUMN column_def...;
```

10.2 实例

假设我们的 COMPANY 表有如下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
```

```
9  7           James       24          Houston     10000.0
```

现在，让我们尝试使用 ALTER TABLE 语句重命名该表，如下所示：

```
1  sqlite> ALTER TABLE COMPANY RENAME TO OLD_COMPANY;
```

上面的 SQLite 语句将重命名 COMPANY 表为 OLD_COMPANY。现在，尝试在 OLD_COMPANY 表中添加一个新的列，如下所示：

```
1  sqlite> ALTER TABLE OLD_COMPANY ADD COLUMN SEX char(1);
```

现在，COMPANY 表已经改变，使用 SELECT 语句输出如下：

```
1  ID          NAME        AGE         ADDRESS     SALARY      SEX
2  ----------  ----------  ----------  ----------  ----------  ---
3  1           Paul        32          California  20000.0
4  2           Allen       25          Texas       15000.0
5  3           Teddy       23          Norway      20000.0
6  4           Mark        25          Rich-Mond   65000.0
7  5           David       27          Texas       85000.0
8  6           Kim         22          South-Hall  45000.0
9  7           James       24          Houston     10000.0
```

请注意，新添加的列是以 NULL 值来填充的。

测试发现 sqlite3 并不支持直接添加带有 unique 约束的列：

```
1  sqlite alter table company add department text unique;
2  Error: Cannot add a UNIQUE column
```

可以先直接添加一列数据，然后再添加 unique 索引。其实在建表的时候如果列有 unique 约束，通过查询系统表 SQLITE_MASTER 可以看到，会自动创建相应的索引。

SQLITE_TEMP_MASTER 跟 SQLITE_MASTER 是 sqlite 的系统表，SQLITE_TEMP_MASTER 表存储临时表有关的所有内容。

11. Truncate Table

在 SQLite 中，并没有 TRUNCATE TABLE 命令，但可以使用 SQLite 的 **DELETE** 命令从已有的表中删除全部的数据。

11.1 语法

DELETE 命令的基本语法如下

```
sqlite> DELETE FROM table_name;
```

但这种方法无法将递增数归零。

如果要将递增数归零，可以使用以下方法：

```
sqlite> DELETE FROM sqlite_sequence WHERE name = 'table_name';
```

当 SQLite 数据库中包含自增列时，会自动建立一个名为 sqlite_sequence 的表。这个表包含两个列：name 和 seq。name 记录自增列所在的表，seq 记录当前序号（下一条记录的编号就是当前序号加 1）。如果想把某个自增列的序号归零，只需要修改 sqlite_sequence 表就可以了。

```
UPDATE sqlite_sequence SET seq = 0 WHERE name = 'table_name';
```

11.2 实例

假设 COMPANY 表有如下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面为删除上表记录的实例：

```
SQLite> DELETE FROM sqlite_sequence WHERE name = 'COMPANY';
SQLite> VACUUM;
```

现在，COMPANY 表中的记录完全被删除，使用 SELECT 语句将没有任何输出。

12. 视图

视图（View）只不过是通过相关的名称存储在数据库中的一个 SQLite 语句。视图（View）实际上是一个以预定义的 SQLite 查询形式存在的表的组合。

视图（View）可以包含一个表的所有行或从一个或多个表选定行。视图（View）可以从一个或多个表创建，这取决于要创建视图的 SQLite 查询。、

视图（View）是一种虚表，允许用户实现以下几点：

- *用户或用户组查找结构数据的方式更自然或直观。*
- *限制数据访问，用户只能看到有限的数据，而不是完整的表。*
- *汇总各种表中的数据，用于生成报告。*

SQLite 视图是只读的，因此可能无法在视图上执行 DELETE、INSERT 或 UPDATE 语句。但是可以在视图上创建一个触发器，当尝试 DELETE、INSERT 或 UPDATE 视图时触发，需要做的动作在触发器内容中定义。

12.1 创建视图

SQLite 的视图是使用 **CREATE VIEW** 语句创建的。SQLite 视图可以从一个单一的表、多个表或其他视图创建。

CREATE VIEW 的基本语法如下：

```
CREATE [TEMP | TEMPORARY] VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE [condition];
```

可以在 SELECT 语句中包含多个表，这与在正常的 SQL SELECT 查询中的方式非常相似。如果使用了可选的 TEMP 或 TEMPORARY 关键字，则将在临时数据库中创建视图。

12.2 实例

现在，下面是一个从 COMPANY 表创建视图的实例。视图只从 COMPANY 表中选取几列：

```
sqlite> CREATE VIEW COMPANY_VIEW AS
SELECT ID, NAME, AGE
FROM  COMPANY;
```

现在，可以查询 COMPANY_VIEW，与查询实际表的方式类似。下面是实例：

```
sqlite> SELECT * FROM COMPANY_VIEW;
```

这将产生以下结果:

```
ID          NAME        AGE
----------  ----------  ----------
1           Paul        32
2           Allen       25
3           Teddy       23
4           Mark        25
5           David       27
6           Kim         22
7           James       24
```

12.3 删除视图

要删除视图，只需使用带有 **view_name** 的 DROP VIEW 语句。DROP VIEW 的基本语法如下

```
sqlite> DROP VIEW view_name;
```

13. 事务

事务（Transaction）是一个对数据库执行工作单元。事务（Transaction）是以逻辑顺序完成的工作单位或序列，可以是由用户手动操作完成，也可以是由某种数据库程序自动完成。

事务（Transaction）是指一个或多个更改数据库的扩展。例如，如果您正在创建一个记录或者更新一个记录或者从表中删除一个记录，那么您正在该表上执行事务。重要的是要控制事务以确保数据的完整性和处理数据库错误。

实际上，您可以把许多的 SQLite 查询联合成一组，把所有这些放在一起作为事务的一部分进行执行。

13.1 事务的属性

事务（Transaction）具有以下四个标准属性，简称ACID：

- **原子性（Atomicity）：**确保工作单位内的所有操作都成功完成，否则，事务会在出现故障时终止，之前的操作也会回滚到以前的状态。
- **一致性（Consistency)：**确保数据库在成功提交的事务上正确地改变状态。
- **隔离性（Isolation）：**使事务操作相互独立和透明。
- **持久性（Durability）：**确保已提交事务的结果或效果在系统发生故障的情况下仍然存在。

13.2 事务控制

使用下面的命令来控制事务：

- **BEGIN TRANSACTION**：开始事务处理。
- **COMMIT**：保存更改，或者可以使用 **END TRANSACTION** 命令。
- **ROLLBACK**：回滚所做的更改。

事务控制命令只与 DML 命令 INSERT、UPDATE 和 DELETE 一起使用。他们不能在创建表或删除表时使用，因为这些操作在数据库中是自动提交的。

13.3 BEGIN TRANSACTION 命令

事务（Transaction）可以使用 BEGIN TRANSACTION 命令或简单的 BEGIN 命令来启动。此类事务通常会持续执行下去，直到遇到下一个 COMMIT 或 ROLLBACK 命令。不过在数据库关闭或发生错误时，事务处理也会回滚。以下是启动一个事务的简单语法：

```
BEGIN;

or

BEGIN TRANSACTION;
```

13.4 COMMIT 命令

COMMIT 命令是用于把事务调用的更改保存到数据库中的事务命令。

COMMIT 命令把自上次 COMMIT 或 ROLLBACK 命令以来的所有事务保存到数据库。

COMMIT 命令的语法如下：

```
COMMIT;

```

```
or

END TRANSACTION;
```

13.5 ROLLBACK 命令

ROLLBACK 命令是用于撤消尚未保存到数据库的事务的事务命令。

ROLLBACK 命令只能用于撤销自上次发出 COMMIT 或 ROLLBACK 命令以来的事务。

ROLLBACK 命令的语法如下：

```
ROLLBACK;
```

13.6 实例

COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

现在，让我们开始一个事务，并从表中删除 age = 25 的记录，最后，我们使用 ROLLBACK 命令撤消所有的更改。

```
sqlite> BEGIN;
sqlite> DELETE FROM COMPANY WHERE AGE = 25;
sqlite> ROLLBACK;
```

检查 COMPANY 表，仍然有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
```

```
4  2           Allen       25          Texas       15000.0
5  3           Teddy       23          Norway      20000.0
6  4           Mark        25          Rich-Mond   65000.0
7  5           David       27          Texas       85000.0
8  6           Kim         22          South-Hall  45000.0
9  7           James       24          Houston     10000.0
```

现在，让我们开始另一个事务，从表中删除 age = 25 的记录，最后我们使用 COMMIT 命令提交所有的更改。

```
1  sqlite> BEGIN;
2  sqlite> DELETE FROM COMPANY WHERE AGE = 25;
3  sqlite> COMMIT;
```

检查 COMPANY 表，有以下记录：

```
1  ID          NAME        AGE         ADDRESS     SALARY
2  ----------  ----------  ----------  ----------  ----------
3  1           Paul        32          California  20000.0
4  3           Teddy       23          Norway      20000.0
5  5           David       27          Texas       85000.0
6  6           Kim         22          South-Hall  45000.0
7  7           James       24          Houston     10000.0
```

14. 子查询

子查询或称为内部查询、嵌套查询，指的是在 SQLite 查询中的 WHERE 子句中嵌入查询语句。

一个 SELECT 语句的查询结果能够作为另一个语句的输入值。

子查询可以与 SELECT、INSERT、UPDATE 和 DELETE 语句一起使用，可伴随着使用运算符如 =、<、>、>=、<=、IN、BETWEEN 等。

以下是子查询必须遵循的几个规则：

- *子查询必须用括号括起来。*
- *子查询在 SELECT 子句中只能有一个列，除非在主查询中有多列，与子查询的所选列进行比较。*
- *ORDER BY 不能用在子查询中，虽然主查询可以使用 ORDER BY。可以在子查询中使用 GROUP BY，功能与 ORDER BY 相同。*

- *子查询返回多于一行，只能与多值运算符一起使用，如 IN 运算符。*
- *BETWEEN 运算符不能与子查询一起使用，但是，BETWEEN 可在子查询内使用。*

14.1 SELECT 语句中的子查询使用

子查询通常与 SELECT 语句一起使用。基本语法如下：

```
SELECT column_name [, column_name ]
FROM   table1 [, table2 ]
WHERE  column_name OPERATOR
       (SELECT column_name [, column_name ]
       FROM table1 [, table2 ]
       [WHERE])
```

实例

检查 SELECT 语句中的子查询使用：

```
sqlite> SELECT *
     FROM COMPANY
     WHERE ID IN (SELECT ID
                  FROM COMPANY
                  WHERE SALARY > 45000)
```

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

14.2 INSERT 语句中的子查询使用

子查询也可以与 INSERT 语句一起使用。INSERT 语句使用子查询返回的数据插入到另一个表中。在子查询中所选择的数据可以用任何字符、日期或数字函数修改。

基本语法如下：

```
INSERT INTO table_name [ (column1 [, column2 ]) ]
           SELECT [ *|column1 [, column2 ]
           FROM table1 [, table2 ]
           [ WHERE VALUE OPERATOR ]
```

实例

假设 COMPANY_BKP 的结构与 COMPANY 表相似，且可使用相同的 CREATE TABLE 进行创建，只是表名改为 COMPANY_BKP。现在把整个 COMPANY 表复制到 COMPANY_BKP，语法如下：

```
sqlite> INSERT INTO COMPANY_BKP
     SELECT * FROM COMPANY
     WHERE ID IN (SELECT ID
                  FROM COMPANY) ;
```

14.3 UPDATE 语句中的子查询使用

子查询可以与 UPDATE 语句结合使用。当通过 UPDATE 语句使用子查询时，表中单个或多个列被更新。

基本语法如下：

```
UPDATE table
SET column_name = new_value
[ WHERE OPERATOR [ VALUE ]
   (SELECT COLUMN_NAME
   FROM TABLE_NAME)
   [ WHERE) ]
```

实例

假设，我们有 COMPANY_BKP 表，是 COMPANY 表的备份。

下面的实例把 COMPANY 表中所有 AGE 大于或等于 27 的客户的 SALARY 更新为原来的 0.50 倍：

```
sqlite> UPDATE COMPANY
     SET SALARY = SALARY * 0.50
     WHERE AGE IN (SELECT AGE FROM COMPANY_BKP
                   WHERE AGE >= 27 );
```

14.4 DELETE 语句中的子查询使用

子查询可以与 DELETE 语句结合使用，就像上面提到的其他语句一样。

基本语法如下：

```
DELETE FROM TABLE_NAME
[ WHERE OPERATOR [ VALUE ]
   (SELECT COLUMN_NAME
   FROM TABLE_NAME)
   [ WHERE) ]
```

实例

假设，我们有 COMPANY_BKP 表，是 COMPANY 表的备份。

下面的实例删除 COMPANY 表中所有 AGE 大于或等于 27 的客户记录：

```
sqlite> DELETE FROM COMPANY
     WHERE AGE IN (SELECT AGE FROM COMPANY_BKP
                   WHERE AGE > 27 );
```

15. Autoincrement

*SQLite 的 **AUTOINCREMENT** 是一个关键字，用于表中的字段值自动递增。我们可以在创建表时在特定的列名称上使用 **AUTOINCREMENT** 关键字实现该字段值的自动增加。*

*关键字 **AUTOINCREMENT** 只能用于整型（INTEGER）字段。*

15.1 语法

AUTOINCREMENT 关键字的基本用法如下：

```
CREATE TABLE table_name(
   column1 INTEGER AUTOINCREMENT,
   column2 datatype,
   column3 datatype,
   .....
   columnN datatype,
);

```

15.2 实例

假设要创建的 COMPANY 表如下所示：

```
sqlite> CREATE TABLE COMPANY(
   ID INTEGER PRIMARY KEY   AUTOINCREMENT,
   NAME           TEXT      NOT NULL,
   AGE            INT       NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
```

16. SQLite 注入

如果您的站点允许用户通过网页输入，并将输入内容插入到 SQLite 数据库中，这个时候您就面临着一个被称为 SQL 注入的安全问题。本章节将向您讲解如何防止这种情况的发生，确保脚本和 SQLite 语句的安全。

注入通常在请求用户输入时发生，比如需要用户输入姓名，但用户却输入了一个 SQLite 语句，而这语句就会在不知不觉中在数据库上运行。

永远不要相信用户提供的数据，所以只处理通过验证的数据，这项规则是通过模式匹配来完成的。在下面的实例中，用户名 username 被限制为字母数字字符或者下划线，长度必须在 8 到 20 个字符之间 - 请根据需要修改这些规则。

```
if (preg_match("/^\w{8,20}$/", $_GET['username'], $matches)){
   $db = new SQLiteDatabase('filename');
   $result = @$db->query("SELECT * FROM users WHERE username=$matches[0]");
}else{
   echo "username not accepted";
}
```

为了演示这个问题，假设考虑此摘录：

```
$name = "Qadir'; DELETE FROM users;";
@$db->query("SELECT * FROM users WHERE username='{$name}'");
```

函数调用是为了从用户表中检索 name 列与用户指定的名称相匹配的记录。正常情况下，**$name** 只包含字母数字字符或者空格，比如字符串 ilia。但在这里，向 $name 追加了一个全新的查询，这个对数据库的调用将会造成灾难性的问题：注入的 DELETE 查询会删除 users 的所有记录。

虽然已经存在有不允许查询堆叠或在单个函数调用中执行多个查询的数据库接口，如果尝试堆叠查询，则会调用失败，但 SQLite 和 PostgreSQL 里仍进行堆叠查询，即执行在一个字符串中提供的所有

查询，这会导致严重的安全问题。

16.1 防止 SQL 注入

在脚本语言中，比如 PERL 和 PHP，您可以巧妙地处理所有的转义字符。编程语言 PHP 提供了字符串函数 **SQLite3::escapeString($string)** 和 **sqlite_escape_string()** 来转义对于 SQLite 来说比较特殊的输入字符。

虽然编码使得插入数据变得安全，但是它会呈现简单的文本比较，在查询中，对于包含二进制数据的列，**LIKE** 子句是不可用的。

请注意，addslashes() 不应该被用在 SQLite 查询中引用字符串，它会在检索数据时导致奇怪的结果。

17. Explain

在 SQLite 语句之前，可以使用 "EXPLAIN" 关键字或 "EXPLAIN QUERY PLAN" 短语，用于描述表的细节。

如果省略了 EXPLAIN 关键字或短语，任何的修改都会引起 SQLite 语句的查询行为，并返回有关 SQLite 语句如何操作的信息。

- *来自 EXPLAIN 和 EXPLAIN QUERY PLAN 的输出只用于交互式分析和排除故障。*
- *输出格式的细节可能会随着 SQLite 版本的不同而有所变化。*
- *应用程序不应该使用 EXPLAIN 或 EXPLAIN QUERY PLAN，因为其确切的行为是可变的且只有部分会被记录。*

17.1 语法

EXPLAIN 的语法如下：

```
1 EXPLAIN [SQLite Query]
```

EXPLAIN QUERY PLAN 的语法如下：

```
1 EXPLAIN  QUERY PLAN [SQLite Query]
```

17.2 实例

检查 SELECT 语句中的 **Explain** 使用：

```
sqlite> EXPLAIN SELECT *  FROM COMPANY  WHERE Salary >= 20000;
```

这将产生以下结果：

```
addr        opcode      p1          p2          p3
----------  ----------  ----------  ----------  ----------
0           Goto        0           19
1           Integer     0           0
2           OpenRead    0           8
3           SetNumColu  0           5
4           Rewind      0           17
5           Column      0           4
6           RealAffini  0           0
7           Integer     20000       0
8           Lt          357         16          collseq(BI
9           Rowid       0           0
10          Column      0           1
11          Column      0           2
12          Column      0           3
13          Column      0           4
14          RealAffini  0           0
15          Callback    5           0
16          Next        0           5
17          Close       0           0
18          Halt        0           0
19          Transactio  0           0
20          VerifyCook  0           38
21          Goto        0           1
22          Noop        0           0
```

现在，让我们检查 SELECT 语句中的 **Explain Query Plan** 使用：

```
SQLite> EXPLAIN QUERY PLAN SELECT * FROM COMPANY WHERE Salary &gt= 20000;
```

```
order       from        detail
----------  ----------  -------------
0           0           TABLE COMPANY
```

18. Vacuum

VACUUM 命令通过复制主数据库中的内容到一个临时数据库文件，然后清空主数据库，并从副本中重新载入原始的数据库文件。这消除了空闲页，把表中的数据排列为连续的，另外会清理数据库文件结构。

如果表中没有明确的整型主键（INTEGER PRIMARY KEY），VACUUM 命令可能会改变表中条目的行 ID（ROWID）。VACUUM 命令只适用于主数据库，附加的数据库文件是不可能使用 VACUUM 命令。

如果有一个活动的事务，VACUUM 命令就会失败。VACUUM 命令是一个用于内存数据库的任何操作。由于 VACUUM 命令从头开始重新创建数据库文件，所以 VACUUM 也可以用于修改许多数据库特定的配置参数。

18.1 手动 VACUUM

下面是在命令提示符中对整个数据库发出 VACUUM 命令的语法：

```
$sqlite3 database_name "VACUUM;"
```

也可以在 SQLite 提示符中运行 VACUUM，如下所示：

```
sqlite> VACUUM;
```

也可以在特定的表上运行 VACUUM，如下所示：

```
sqlite> VACUUM table_name;
```

18.2 自动 VACUUM

SQLite 的 Auto-VACUUM 与 VACUUM 不大一样，它只是把空闲页移到数据库末尾，从而减小数据库大小。通过这样做，它可以明显地把数据库碎片化，而 VACUUM 则是反碎片化。所以 Auto-VACUUM 只会让数据库更小。

在 SQLite 提示符中，您可以通过下面的编译运行，启用/禁用 SQLite 的 Auto-VACUUM：

```
sqlite> PRAGMA auto_vacuum = NONE;  -- 0 means disable auto vacuum
sqlite> PRAGMA auto_vacuum = INCREMENTAL;  -- 1 means enable incremental vacuum
sqlite> PRAGMA auto_vacuum = FULL;  -- 2 means enable full auto vacuum
```

可以从命令提示符中运行下面的命令来检查 auto-vacuum 设置：

```
$sqlite3 database_name "PRAGMA auto_vacuum;"
```

19. 日期 & 时间

SQLite 支持以下五个日期和时间函数：

序号	函数	实例
1	date(timestring, modifier, modifier, ...)	以 YYYY-MM-DD 格式返回日期。
2	time(timestring, modifier, modifier, ...)	以 HH:MM:SS 格式返回时间。
3	datetime(timestring, modifier, modifier, ...)	以 YYYY-MM-DD HH:MM:SS 格式返回。
4	julianday(timestring, modifier, modifier, ...)	这将返回从格林尼治时间的公元前 4714 年 11 月 24 日正午算起的天数。
5	strftime(format, timestring, modifier, modifier, ...)	这将根据第一个参数指定的格式字符串返回格式化的日期。具体格式见下边讲解。

上述五个日期和时间函数把时间字符串作为参数。时间字符串后跟零个或多个 modifier 修饰符。strftime() 函数也可以把格式字符串 format 作为其第一个参数。下面将为您详细讲解不同类型的时间字符串和修饰符。

19.1 时间字符串

一个时间字符串可以采用下面任何一种格式：

序号	时间字符串	实例
1	YYYY-MM-DD	2010/12/30
2	YYYY-MM-DD HH:MM	2010/12/30 12:10
3	YYYY-MM-DD HH:MM:SS.SSS	10:04.1
4	MM-DD-YYYY HH:MM	12-30-2010 12:10
5	HH:MM	12:10
6	YYYY-MM-DD**T**HH:MM	2010/12/30 12:10
7	HH:MM:SS	12:10:01
8	YYYYMMDD HHMMSS	20101230 121001
9	now	2013/5/7

您可以使用 "T" 作为分隔日期和时间的文字字符。

19.2 修饰符（Modifier）

时间字符串后边可跟着零个或多个的修饰符，这将改变有上述五个函数返回的日期和/或时间。任何上述五大功能返回时间。修饰符应从左到右使用，下面列出了可在 SQLite 中使用的修饰符：

- NNN days
- NNN hours
- NNN minutes
- NNN.NNNN seconds
- NNN months
- NNN years
- start of month
- start of year
- start of day
- weekday N
- unixepoch
- localtime
- utc

19.3 格式化

SQLite 提供了非常方便的函数 **strftime()** 来格式化任何日期和时间。您可以使用以下的替换来格式化日期和时间：

替换	描述
%d	一月中的第几天，01-31
%f	带小数部分的秒，SS.SSS
%H	小时，00-23
%j	一年中的第几天，001-366
%J	儒略日数，DDDD.DDDD
%m	月，00-12
%M	分，00-59
%s	从 1970-01-01 算起的秒数
%S	秒，00-59
%w	一周中的第几天，0-6 (0 is Sunday)
%W	一年中的第几周，01-53
%Y	年，YYYY
%%	% symbol

19.4 实例

现在让我们使用 SQLite 提示符尝试不同的实例。下面是计算当前日期：

```
sqlite> SELECT date('now');
2013-05-07
```

计算当前月份的最后一天：

```
sqlite> SELECT date('now','start of month','+1 month','-1 day');
2013-05-31
```

计算给定 UNIX 时间戳 1092941466 的日期和时间：

```
sqlite> SELECT datetime(1092941466, 'unixepoch');
2004-08-19 18:51:06
```

计算给定 UNIX 时间戳 1092941466 相对本地时区的日期和时间：

```
sqlite> SELECT datetime(1092941466, 'unixepoch', 'localtime');
```

```
2 2004-08-19 11:51:06
```

计算当前的 UNIX 时间戳：

```
1 sqlite> SELECT strftime('%s','now');
2 1367926057
```

计算美国"独立宣言"签署以来的天数：

```
1 sqlite> SELECT julianday('now') - julianday('1776-07-04');
2 86504.4775830326
```

计算从 2004 年某一特定时刻以来的秒数：

```
1 sqlite> SELECT strftime('%s','now') - strftime('%s','2004-01-01 02:34:56');
2 295001572
```

计算当年 10 月的第一个星期二的日期：

```
1 sqlite> SELECT date('now','start of year','+9 months','weekday 2');
2 2013-10-01
```

计算从 UNIX 纪元算起的以秒为单位的时间（类似 strftime('%s','now')，不同的是这里有包括小数部分）：

```
1 sqlite> SELECT (julianday('now') - 2440587.5)*86400.0;
2 1367926077.12598
```

在 UTC 与本地时间值之间进行转换，当格式化日期时，使用 utc 或 localtime 修饰符，如下所示：

```
1 sqlite> SELECT time('12:00', 'localtime');
2 05:00:00
3
4 sqlite>  SELECT time('12:00', 'utc');
5 19:00:00
```

20. 常用函数

SQLite 有许多内置函数用于处理字符串或数字数据。下面列出了一些有用的 SQLite 内置函数，且所有函数都是大小写不敏感，这意味着您可以使用这些函数的小写形式或大写形式或混合形式。欲了解更多详情，请查看 SQLite 的官方文档：

序号	函数 & 描述
1	**SQLite COUNT 函数** SQLite COUNT 聚集函数是用来计算一个数据库表中的行数。
2	**SQLite MAX 函数** SQLite MAX 聚合函数允许我们选择某列的最大值。
3	**SQLite MIN 函数** SQLite MIN 聚合函数允许我们选择某列的最小值。
4	**SQLite AVG 函数** SQLite AVG 聚合函数计算某列的平均值。
5	**SQLite SUM 函数** SQLite SUM 聚合函数允许为一个数值列计算总和。
6	**SQLite RANDOM 函数** SQLite RANDOM 函数返回一个介于 -9223372036854775808 和 +9223372036854775807 之间的伪随机整数。
7	**SQLite ABS 函数** SQLite ABS 函数返回数值参数的绝对值。
8	**SQLite UPPER 函数** SQLite UPPER 函数把字符串转换为大写字母。
9	**SQLite LOWER 函数** SQLite LOWER 函数把字符串转换为小写字母。
10	**SQLite LENGTH 函数** SQLite LENGTH 函数返回字符串的长度。
11	**SQLite sqlite_version 函数** SQLite sqlite_version 函数返回 SQLite 库的版本。

20.1 COUNT 函数

COUNT 聚集函数是用来计算一个数据库表中的行数。下面是实例：

```
sqlite> SELECT count(*) FROM COMPANY;

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

```
5 7
```

20.2 MAX 函数

MAX 聚合函数允许我们选择某列的最大值。下面是实例：

```
1 sqlite> SELECT max(salary) FROM COMPANY;
2
3 max(salary)
4 ------------
5 85000.0
```

20.3 MIN 函数

MIN 聚合函数允许我们选择某列的最小值。下面是实例：

```
1 sqlite> SELECT min(salary) FROM COMPANY;
```

20.4 AVG 函数

AVG 聚合函数计算某列的平均值。下面是实例：

```
1 sqlite> SELECT avg(salary) FROM COMPANY;
```

20.5 SUM 函数

SQLite SUM 聚合函数允许为一个数值列计算总和。下面是实例：

```
1 sqlite> SELECT sum(salary) FROM COMPANY;
```

20.6 RANDOM 函数

SQLite RANDOM 函数返回一个介于 -9223372036854775808 和 +9223372036854775807 之间的伪随机整数。下面是实例：

```
1 sqlite> SELECT random() AS Random;
```

```
2
3 Random
4 -------------------
5 5876796417670984050
```

20.7 ABS 函数

ABS 函数返回数值参数的绝对值。下面是实例：

```
1 sqlite> SELECT abs(5), abs(-15), abs(NULL), abs(0), abs("ABC");
2
3 abs(5)      abs(-15)    abs(NULL)   abs(0)      abs("ABC")
4 ----------  ----------  ----------  ----------  ----------
5 5           15                      0           0.0
```

20.8 UPPER 函数

UPPER 函数把字符串转换为大写字母。下面是实例：

```
1 sqlite> SELECT upper(name) FROM COMPANY;
2
3 upper(name)
4 -----------
5 PAUL
6 ALLEN
7 TEDDY
8 MARK
9 DAVID
10 KIM
11 JAMES
```

20.9 LOWER 函数

LOWER 函数把字符串转换为小写字母。下面是实例：

```
1 sqlite> SELECT lower(name) FROM COMPANY;
```

20.10 LENGTH 函数

SQLite LENGTH 函数返回字符串的长度。下面是实例：

```
sqlite> SELECT name, length(name) FROM COMPANY;

NAME        length(name)
----------  ------------
Paul        4
Allen       5
Teddy       5
Mark        4
David       5
Kim         3
James       5
```

20.11 sqlite_version 函数

SQLite sqlite_version 函数返回 SQLite 库的版本。下面是实例：

```
sqlite> SELECT sqlite_version() AS 'SQLite Version';

SQLite Version
--------------
3.6.20

```