SQLite

目录

使用 SQLite 编程的快速介绍

这告诉你怎么开始实验 SQLite，没有冗长的说明和配置：

下载代码

- 取得一份二进制拷贝，或者是源代码并自己编译它。 关于下载的更多信息。

创建一个新数据库

- 在 shell 或 DOS 命令行下，输入："sqlite3 test.db". 将创建一个新的数据库文件名叫"test.db"。 （你可以使用不同的名字）
- 输入 SQL 命令在提示符下创建和写入新的数据。
- 这里有更多相关文档。

使用 SQLite 编写程序

- 下面是一个简单的 TCL 程序，让我们看看怎么使用 SQLite 的 TCL 接口。此程序在由第一个参数定义的数据库上执行第二个参数给出的 SQL 语句。这个命令是第 7 行的 sqlite3 命令，用于打开一个 SQLite 数据库并且创建一个新的 TCL 命令"db"访问数据库，这个 db 命令在第 8 行对数据库执行 SQL 命令，并且在最后一行关闭与数据库的连接。
  - #!/usr/bin/tclsh
  - if {$argc!=2} {
  - puts stderr "Usage: %s DATABASE SQL-STATEMENT"
  - exit 1
  - }
  - load /usr/lib/tclsqlite3.so Sqlite3
  - sqlite3 db [lindex $argv 0]
  - db eval [lindex $argv 1] x {
  - foreach v $x(*) {
  - puts "$v = $x($v)"
  - }
  - puts ""
  - }
  - db close
- 下面是一个 C 程序的例子，显示怎么使用 sqlite 的 C/C++ 接口。数据库的名字由第一个参数取得，第二个参数是一条或更多的 SQL 执行语句。这个函数在 22 行调用 sqlite3_open() 打开数据库，在第 27 行 sqlite3_exec()对数据库执行 SQL 语句，在第 31 行由 sqlite3_close()关闭数据库连接。
  - #include
  - #include
  - static int callback(void *NotUsed, int argc, char **argv, char **azColName){

```
  int i;
  for(i=0; isqlite3_open(argv[1], &db);
  if( rc ){
    fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
    sqlite3_close(db);
    exit(1);
  }
  rc = sqlite3_exec(db, argv[2], callback, 0, &zErrMsg);
  if( rc!=SQLITE_OK ){
    fprintf(stderr, "SQL error: %s\n", zErrMsg);
  }
  sqlite3_close(db);
  return 0;
}
```

SQLite 适用的范围

SQLite 不同于其他大部分的 SQL 数据库引擎,因为它的首要设计目标就是简单化:

- 易于管理
- 易于使用
- 易于嵌入其他大型程序
- 易于维护和配置

许多人喜欢 SQLite 因为它的小巧和快速. 但是这些特性只是它的部分优点, 使用者还会发现 SQLite 是非常稳定的. 出色的稳定性源于它的简单, 越简单就越不容易出错. 除了上述的简单、小巧和稳定性外, 最重要的在于 SQLite 力争做到简单化.

简单化在一个数据库引擎中可以说是一个优点, 但也可能是个缺点, 主要决定于你想要做什么. 为了达到简单化, SQLite 省略了一些人们认为比较有用的特性, 例如高并发性、 严格的存取控制、 丰富的内置功能、 存储过程、复杂的 SQL 语言特性、 XML 以及 Java 的扩展, 超大的万亿级别的数据测量等等. 如果你需要使用上述的这些特性并且不介意它们的复杂性, 那么 SQLite 也许就不适合你了. SQLite 没有打算作为一个企业级的数据库引擎, 也并不打算和 Oracle 或者 PostgreSQL 竞争.

仅凭经验来说 SQLite 适用于以下场合: 当你更看中简单的管理、使用和维护数据库, 而不是那些企业级数据库提供的不计其数的复杂功能的时候, 使用 SQLite 是一个比较明智的选择. 事实也证明, 人们在许多情况下已经清楚的认识到简单就是最好的选择.

SQLite 最佳试用场合

- **网站**

  作为数据库引擎 SQLite 适用于中小规模流量的网站(也就是说, 99.9%的网站). SQLite 可以处理多少网站流量在于网站的数据库有多大的压力. 通常来说, 如果一个网站的点击率少于 100000 次/天的话, SQLite 是可以正常运行的. 100000 次/天是一个保守的估计, 不是一个准确的上限. 事实证明, 即使是 10 倍的上述流量的情况下 SQLite 依然可以正常运行.

- **嵌入式设备和应用软件**

  因为 SQLite 数据库几乎不需要管理, 因此对于那些无人值守运行或无人工技术支持的设备或服务, SQLite 是一个很好的选择. SQLite 能很好的适用于手机, PDA, 机顶盒, 以及其他仪器. 作为一个嵌入式数据库它也能够很好的应用于客户端程序.

- **应用程序文件格式**

  SQLite 作为桌面应用程序的本地磁盘文件格式取得了巨大成功. 例如金融分析工具、CAD 包、档案管理程序等等. 一般的数据库打开操作需要调用 sqlite3_open()函数, 并且标记一个显式本地事务的起始点 (BEGIN TRANSACTION) 来保证以独占的方式得到文件的内容. 文件保存将执行一个提交 (COMMIT) 同时标记另一个显式本地事务起始点. 这种事务处理的作用就是保证对于应用程序数据文件的更新是原子的、持久的、独立的和一致的.

数据库里可以加入一些临时的触发器,用来把所有的改变记录在一张临时的取消/重做日志表中. 当用户按下取消/重做按钮的时候这些改变将可以被回滚. 应用这项技术实现一个无限级的取消/重做功能只需要编写很少的代码.

- **替代某些特别的文件格式**

  许多程序使用 fopen(), fread(), 或 fwrite()函数创建和管理一些自定义的文件用来保存数据. 使用 SQLite 替代这些自定义的文件格式将是一种很好的选择.

- **内部的或临时的数据库**

  对于那些有大量的数据需要用不同的方式筛选分类的程序, 相对于编写同样功能的代码, 如果你把数据读入一个内存中的 SQLite 数据库, 然后使用连接查询和 ORDER BY 子句按一定的顺序和排列提取需要的数据, 通常会更简单和快速. 按照上述的方法使用内嵌的 SQLite 数据库将会使程序更富有灵活性, 因为添加新的列或索引不用重写任何查询语句.

- **命令行数据集分析工具**

  有经验的 SQL 用户可以使用 SQLite 命令行程序去分析各种混杂的数据集. 原是数据可以从 CSV（逗号分隔值文件）文件中导入, 然后被切分产生无数的综合数据报告. 可能得用法包括网站日志分析, 运动统计分析, 编辑规划标准, 分析试验结果.

  当然你也可以用企业级的客户端/服务器数据库来做同样的事情. 在这种情况下使用 SQLite 的好处是: SQLite 的部署更为简单并且结果数据库是一个单独的文件, 你可以把它存储在软盘或者优盘或者直接通过 email 发给同事.

- **在 Demo 或测试版的时候作为企业级数据库的替代品**

  如果你正在编写一个使用企业级数据库引擎的客户端程序, 使用一个允许你连接不同 SQL 数据库引擎的通用型数据库后台将是很有意义的. 其更大的意义在于将 SQLite 数据库引擎静态的连接到客户端程序当中，从而内嵌 SQLite 作为混合的数据库支持. 这样客户端程序就可以使用 SQLite 数据库文件做独立的测试或者验证.

- **数据库教学**

  因为 SQLite 的安装和使用非常的简单(安装过程几乎忽略不计, 只需要拷贝 SQLite 源代码或 sqlite.exe 可执行文件到目标主机, 然后直接运行就可以) 所以它非常适合用来讲解 SQL 语句. 同学们可以非常简单的创建他们喜欢的数据库, 然后通过电子邮件发给老师批注或打分. 对于那些感兴趣怎样实现一个关系型数据库管理系统（RDBMS）的高层次的学生, 按照模块化设计且拥有很好的注释和文档的 SQLite 源代码, 将为他们打下良好的基础. 这并不是说 SQLite 就是如何实现其他数据库引擎的精确模型, 但是很适合学生们了解 SQLite 是如何快速工作的, 从而掌握其他数据库系统的设计实现原则.

- **试验 SQL 语言的扩展**

  SQLite 简单且模块化的设计使得它可以成为一个用来测试数据库语言特性或新想法的优秀的原型平台.

哪些场合适合使用其他的关系型数据库管理系统（RDBMS）

- **客户端/服务器程序**

  如果你有许多的客户端程序要通过网络访问一个共享的数据库，你应当考虑用一个客户端/服务器数据库来替代 SQLite. SQLite 可以通过网络文件系统工作，但是因为和大多数网络文件系统都存在延时，因此执行效率不会很高. 此外大多数网络文件系统在实现文件逻辑锁的方面都存在着 bug(包括 Unix 和 windows). 如果文件锁没有正常的工作，就可能出现在同一时间两个或更多的客户端程序更改同一个数据库的同一部分，从而导致数据库出错. 因为这些问题是文件系统执行的时候本质上存在的 bug，因此 SQLite 没有办法避免它们.

  好的经验告诉我们，应该避免在许多计算机需要通过一个网络文件系统同时访问同一个数据库的情况下使用 SQLite.

- **高流量网站**

  SQLite 通常情况下用作一个网站的后台数据库可以很好的工作. 但是如果你的网站的访问量大到你开始考虑采取分布式的数据库部署，那么你应当毫不犹豫的考虑用一个企业级的客户端/服务器数据库来替代 SQLite.

- **超大的数据集**

  当你在 SQLite 中开始一个事务处理的时候(事务处理会在任何写操作发生之前产生，而不是必须要显示的调用 BEGIN...COMMIT)，数据库引擎将不得不分配一小块脏页(文件缓冲页面)来帮助它自己管理回滚操作. 每 1MB 的数据库文件 SQLite 需要 256 字节. 对于小型的数据库这些空间不算什么，但是当数据库增长到数十亿字节的时候，缓冲页面的尺寸就会相当的大了. 如果你需要存储或修改几十 GB 的数据，你应该考虑用其他的数据库引擎.

- **高并发访问**

  SQLite 对于整个数据库文件进行读取/写入锁定. 这意味着如果任何进程读取了数据库中的某一部分，其他所有进程都不能再对该数据库的任何部分进行写入操作. 同样的，如果任何一个进程在对数据库进行写入操作，其他所有进程都不能再读取该数据库的任何部分. 对于大多数情况这不算是什么问题. 在这些情况下每个程序使用数据库的时间都很短暂，并且不会独占，这样锁定至多会存在十几毫秒. 但是如果有些程序需要高并发，那么这些程序就需要寻找其他的解决方案了.

SQLite 第三版总览(简介)

SQLite 第三版主要主要介绍关于类库的一些变化, 包括:

- 介绍了一个关于数据库文件的更紧凑的格式.
- 若类型和 BLOB 支持.
- 支持 UTF-8 and UTF-16 文本.
- 用户定义的文件排列顺序.
- 64 字节的行编号.
- 针对并发性的一些改良.

这篇文档简易的介绍了 SQlite3.0 版针对于 2.8 版的一些改进,适用于对 SQlite2.8 版比较了解的用户。

命名上的变化

在可预见的未来，错误修正这项功能将继续支持 SQLite 2.8 版。为了保证这两个版本可以共存，在 3.0 版本中，一些主要文件的名称和 API 的名称中都加了个”3“。例如, c 程序内含文件的名称已经从"sqlite.h"改为"sqlite3.h".还有，用来和数据库一起操作的数据命令解释程序也 从 "sqlite.exe" 改为 "sqlite3.exe"。有了这些命名上的区别，SQLite 2.8 版和 SQLite 3.0 版就可以同时安装在同一个系统下了。另外，名称上的区别也使同一个 C 程序可以同时和 2.8，3.0 两个版本同时相连并使用同一个类库。

新的文件格式

SQlite 的数据库文件格式已被完全更新， 2.1 版的格式和 3.0 版的格式是互不兼容的。比如，2.8 版的 SQLite 是无法读取 3.0 版的 SQlite 数据库文件的，同样，3.0 版的 SQLite 也是无法读取 2.8 版的 SQlite 数据库文件的。

如想把 SQLite 2.8 版的数据库转换成 3.0 版的数据库的话，你可以用一些现成的命令行操作，比如输入下面的命令：

```
sqlite OLD.DB .dump | sqlite3 NEW.DB
```

新的数据库文件格式使用 B+树型数据表格。在 B+树中，所有的数据都被存储在(数据结构中)树结构端结点，而不是既在(数据结构中)树结构端结点又 在树的分支节点。B+树型数据表格具有很好的测量性，并且可以存储比较大的数据组。此外，传统的 B-树也仍被应用在 SQlite3.0 的许多目录中。

新的文件格式可以存储的可变页的长度在 512 和 32768 字节之间。每页的文件长度都可以在页眉显示出来,所以从理论上来说,同一个类库可以读取不同长度的数据库,但实际上这一点还没有完全实现。

新的文件格式在磁盘映像中省略了没有被应用的区域。例如，目录仅仅显示 B-树所存储的主要部分而不是显示所有的数据。也就是说，记录数据长度的区域 被省略了。整数值，比如说关键字的长度和关键数据可以用变长量来编译，这样一来，最常见的数据就可以只用一两个字节来显示了，还有，

如果需要的话，最高 64 字节的数据信息也是可以编译的。 在 3.0 版中，整数和浮动的点数据是用二进制来记录的，但在 2.0 版中，它们则是被转换成 ASCII 码的。所以，同一个数据库文件，如果是记录在 SQlite3.0 中，就可以比记录在 SQlite2.8 中少占用 25%至 30%的磁盘空间。

关于 SQLite3.0 版所采用的 B-树的文件格式具体细节，你可以点击 btree.c 看标题注释。

弱类型和 BLOB 技术支持

SQLite2.8 在数据库内部可以用不同的文件格式处理文件，但是，如果想把信息写到硬盘或是通过 API 和其它数据相连，所有文件格式都必须被转 换成 ASCII 文本格式。SQLite 3.0 则不同，它可以把数据可内部的主要数据，也就是上文所提到的用变长量所表示的数据显示给用户，并在适当的时候用二进制的形式在磁盘中显示。为了支持 BLOB，非 ASCII 格式的数据也可以在磁盘中显示。

SQLite 2.8 版有一个特点，就是任何类型的数据都可以存储在任意的数据列中，不受数据列所要求存储的文件类型的限制。这个特性被保留在 3.0 版中并有所改进。虽 然数据文件的格式决定了文件的类型，每列所存储的文件都要有规定的属性，但在 3.0 中每个列都是可以存储不同类型的数据的。当数据被存入一个数据列的时 候，这个列将尽全力把存入的数据的文件格式转换成该列所要求的文件格式。所有的 SQL 数据库引擎都是这样的。所不同的是，SQLite 3.0 将存储数据即使转换该数据的文件类型是不可能的。

例如，一个数据列要求所存储的文件的类型是 "INTEGER"，你输入一个字符串，这个列将自动检查所输入的字符串是否是数字，如果它确实看起来像是数字，字符串将被转换成数字，然后，如果这个数字没有分数部分的话，它将被转换成整数存储起来。但是如果这个字符串不是一个规则的数字的话，它将仍被保存为一个字符串。如果一个列要求所存储的文件的类 型是"TEXT"的话，在存储数据之前，列将尝试把数字转换成 ASCII-Text 来表示数据。但是，BLOBS 在文本列仍然被保存成 BLOBS，因为在通常情况下你是不可能把 BLOB 转换成文本的。

在大部分的其他的 SQL 数据库引擎中，数据类型是和他该数据所处列的类型紧密相连的，但在 SQlite3.0 中，一个数据的类型只和自身有关，和所属列所要求的数据类型没有任何关联。 Paul Graham 在他的 *ANSI Common Lisp* 一书中称这个特性"manifest typing"为"弱类型". 其他的作者对"manifest typing"有不同的定义，我们不要混淆，不管它的名称是什么，我们知道它是 3.0 的一个特性就好。

关于 SQLite3.0 版的数据类型的更多内容，点击 separately.

支持 UTF-8 和 UTF-16

在 SQLite 3.0 中，API 中的程序可以识别 UTF-8 和 UTF-16 文本，并不改变主机中原来字节的顺序。每个数据库文件都可以用 UTF-8, UTF-16BE (big-endian)和 UTF-16LE (little-endian)两种方式处理文本。在磁盘文件的内部，到处可见到同样的文本显示。如果数据库文件中所记载的文本显示（在文件标题中）和接 口程序所要求的文本显示不相符合的话，文本将被转换成当前所要求的文本格式。经常转换文本格式对于程序来说是非常麻烦的，所以建议程序员在一个应用程序中 自始至终使用一种文本格式。

当前，在执行 SQLite 的时候，SQL 的语法只能识别 UTF-8 文本，所以如果是 UTF-16 文本的话将被转换成 UTF-8 文本。当然这只是程序执行的问题，在不远的将来，更新版本的 SQLite 将完全有实力识别用 SQL 所编译的 UTF-16 文本。

当开发用户所定义的 SQL 函数和分类排序功能的时候，不管是 UTF-8，UTF-16be 或是 UTF-16le 文本格式，每个函数和分类排序都能清晰的识别文件。程序所执行的每一步都被完整无误的记录下来以便编译。当 SQL 函数或分类排序要求相应的格式，但当前版 本的文件编译又无能为力的时候，文本将自动被转换。像以前一样，这种转换需要一定的时间，所以建议程序员挑选一种编码并自始至终使用以避免不必要的文本格 式转换。

不只是对于一个完整的文本，即使是某个文本中的格式不规则的字符串和格式不规则的 UTF-8 和 UTF-16 文本行，SQLite 也要竭尽全力把他们 编译成规则的文件格式。所以如果你想存储 ISO8859 数据的话，你可以使用 UTF-8 接口程序。只要不使用 UTF-16 分类排序或 SQL 函数，文本的字 节顺序将不会被变更。

用户定义的分类排序

所谓的分类排序是指文本的排列顺序。当 SQLite 3.0 分类的时候(比如使用比较操作符"<" or ">=")，它是按照数据类型分类的。

- 首先把空行分类
- 按数字的顺序把数值分类
- 再把文本分类
- 最后把 BLOBs 分类

当比较两个文本行的时候使用分类排序。分类排序不改变空行，数字和 BLOBS 的顺序，它只改变文本的顺序。

分类排序功能是作为一个函数来执行的，它使用两个字符串来表示， 如果第一个字符串小于第二个字符串，则显示为负数， 如等于则显示为零，如大于则显示为正数。 SQLite 3.0 版设置了一个名叫 BINARY 的独立的内置分类排序功能， 它是依靠标准 C 类库中的 memcmp()程序来实现的。 BINARY 分类排序功能很好的支持英文文档。对于其他语言或者在其他场合，程序会根据需要来选择合适的分类排序功能。

使用哪种分类排序是由 SQL 中的 COLLATE 语句来决定的。一个 COLLATE 语句可分辨一种表格，它可以定义并默认表格中的每列该使用哪种分类 排序，目录的每个区段该使用哪种分类排序，或 SELECTED 语句中 ORDER BY 字句该使用哪种分类排序。按照计划，SQLite 还将使用标准的 CAST()句法来具体定义每个分类排序。

64 字节的行编号

表格中的每一行都有行标识符。如果表格中的某一列被定义为整数初级键控，那么那一列就成为其所属行的别名。但不管有没有定义这样一个列， 每个行都是有行编号的。

在 SQLite 3.0 版中，行编号是 64 字节的带符号整数。而在 SQLite2.8 中，行编号是 32 字节的。

为了占有最小的存储空间，64 字节的行编号被存储为变长量。行编号在 0 和 127 之间的都只占有一个字节的空间。行编号在 128 和 16383 之间的只 占用 2 个字节的空间。 行编号在 16383 和 2097152 之间的占有三个字节的空间。依此类推。也可以使用负行编号， 担负行编号通常要占用 9 个字节的空间， 所以建议不要使用。SQLite 所自动生成的行编号通常不是负行编号。

改良的并发性

SQLite 2.8 允许多个进程同时读取或一个进程读取，但不允许读取和写入同时进行。SQLite 3.0 则允许一个进程写入和其它多个进程同时读取。为了确认所更新的数据，输入者必须为数据库中的简短区间设置一个额外的锁，但这个锁不再像以前那样必须 贯穿在整个输入过程中。可以点击找到关于 SQLite3.0 版中的锁的具体信息。

SQLite 中也有一个在格式上有局限性的表格极的锁. 如果每个表格都存储了不同类型的文件，这些文件可以被连接到主数据库（使用 ATTACH 命令），这个被整合在一起的数据库依然可以作为一个整体来运行。但 各个文件将按照需要生成各自的锁。所以如果你把“数据库”重新定义成两个或更多数据库文件，那么在同一个数据库中同时对两个进程写入则是完全可能的。为了 更好的支持这项功能，同时处理两个或更多的 ATTACHed 型数据库是最基本的了。

致谢

SQLite3.0 版的成功发行得利于 AOL 开发者的大力支持和开源软件的帮助。

SQLite 第三版中的数据类型

1.存储类别

第二版把所有列的值都存储成 ASCII 文本格式。第三版则可以把数据存储成整数和实数,还可以存储 BLOB 数据.

Each value stored in an SQLite 数据库中存储的每个值都有一个属性,都属于下面所列类中的一种,(被数据库引擎所控制)

- **空**.这个值为空值
- **整数**.值被标识为整数,依据值的大小可以依次被存储为 1,2,3,4,5,6,7,8.
- **实数**. 所有值都是浮动的数值,被存储为 8 字节的 IEEE 浮动标记序号.
- **文本**. 值为文本字符串,使用数据库编码存储(TUTF-8, UTF-16BE or UTF-16-LE).
- **BLOB**. 值是 BLOB 数据,如何输入就如何存储,不改变格式.

像 SQLite2.0 版一样,在 3.0 版中,除了 INTEGER PRIMARY KEY,数据库中的任何列都可以存储任何类型的数据.这一规则也有例外,在下面的"严格相似模式"中将描述.

输入 SQLite 的所有值,不管它是嵌入 SQL 语句中的文字还是提前编译好的绑定在 SQL 语句中的值,在 SQL 语句执行前都被存储为一个类.在下面所描述的情况下,数据库引擎将在执行时检查并把值在数字存储类(整数和实数)和文本类间转换.

存储的类别最初被分类为如下:

- 具体的值比如 SQL 语句部分的带双引号或单引号的文字被定义为文本,如果文字没带引号并没有小数点或指数则被定义为整数,如果文字没带引号但有小数点或指数则被定义为实数,如果值是空则被定义为空值.BLOB 数据使用符号 X'ABCD'来标识.
- Values supplied using the 被输入的值使用 sqlite3_bind_* APIs 的被分类一个存储等级,这等级是和原来的类基本相一致的. (比如 sqlite3_bind_blob()绑定一个 BLOB 的值).

值的分类是 SQL 分等级操作的结果,决定于最远的操作表达式.用户定义的功能也许会把值返回任意的类.在编译的时候来确定表达式的存储类基本是不可能的.

2. 列之间的亲和性

在 SQLite3.0 版中,值被定义为什么类型只和值自身有关,和列没有关系,和变量也没有关系.(这有时被称作 弱类型.)所有其它的我们所使用的数据库引擎都受静态类型系统的限制,其中的所有值的类是由其所属列的属性决定的,而和值无关.

为了最大限度的增加 SQLite 数据库和其他数据库的兼容性,SQLite 支持列的"类型亲和性". 列的亲和性是为该列所存储的数据建议一个类型.我们要注意是建议而不是强迫.在理论上来讲,任何列依然是可以存储任何类型的数据的. 只是针对某些列,如果给建议类型的话,数据库将按所建议的类型存储.这个被优先使用的数据类型则被称为"亲和类型".

在 SQLite3.0 版中,数据库中的每一列都被定义为以下亲和类型中的一种:

- 文本
- 数字的
- 整数
- 无

一个具有类型亲和性的列按照无类型,文本,或 BLOB 存储所有的数据.如果数字数据被插入一个具有文本类型亲和性的列,在存储之前数字将被转换成文本.

一个具有数字类型亲和性的列也许使用所有的五个存储类型存储值.当文本数据被插入一个数字列时,在存储之前,数据库将尝试着把文本转换成整数或实 数.如果能成功转换的话,值将按证书活实数的类型被存储. 如果不能 成功转换的话,值则只能按文本类型存储了,而不会被转换成无类型或 BLOB 类型来存储.

一个具有整数亲和力的列在转换方面和具有数字亲和力的列是一样的,但也有些区别 ,比如没有浮动量的实值(文本值转换的值)被插入具有整数亲和力的列时,它将被转换成整数并按整数类型存储.

一个具有无类型亲和力的列不会优先选择使用哪个类型.在数据被输入前它不会强迫数据转换类型.

2.1 列的亲和性的决定

一个列的亲和类型是由该列所宣称的类型决定的.遵守以下规则:

1. 如果数据类型包括字符串"INT"那么它被定义成具有整数亲和性.
2. 如果列中的数据类型包括以下任何的字符串 "CHAR", "CLOB", or "TEXT" 那么这个列则具有文本亲和性.要注意 VARCHAR 类型包括字符串"CHAR"因此也具有文本类型亲和性.
3. 如果一个列的数据类型包括字符串"BLOB"或者如果数据类型被具体化了,那么这个列具有无类型亲和性.
4. 否则就具有数字类型亲和性.

如果表格使用 If "CREATE TABLE AS SELECT..."语句生成的,那么所有的列则都没有具体的数据类型,则没有类型亲和性.

2.2 列的亲和性的例子

```
CREATE TABLE t1(
    t  TEXT,
    nu NUMERIC,
    i  INTEGER,
    no BLOB
);

-- Storage classes for the following row:
-- TEXT, REAL, INTEGER, TEXT
INSERT INTO t1 VALUES('500.0', '500.0', '500.0', '500.0');
```

```
-- Storage classes for the following row:
-- TEXT, REAL, INTEGER, REAL
INSERT INTO t1 VALUES(500.0, 500.0, 500.0, 500.0);
```

3.比较表达式

像 SQLite2.0 版一样,3.0 版的一个特性是二进制比较符'=', '<', '<=', '>=' and '!=',一个操作'IN'可以测试固定的成员资格, 三重的比较操作符'BETWEEN'.

比较的结果决定于被比较的两个值的存储类型。遵循以下规则：

- 一个具有空存储类型的值被认为小于任何值（包括另外一个具有空存储类型的值）。
- 一个整数值或实数值小于任何文本值和 BLOB 值。当一个整数或实数和另一个整数或实数相比较的时候，则按照实际数值来比较。
- 一个文本值小于 BLOB 值。当两个文本值相比较的时候，则用 C 语言类库中的 memcmp()函数来比较。然而，有时候也不是这样的，比如在下面所描述的“用户定义的整理顺序”情况下。
- 当两个 BLOB 文本被比较的时候，结果决定于 memcmp()函数。

在开始比较前，SQLite 尝试着把值在数字存储级（整数和实数）和文本之间相互转换。下面列举了关于如何比较二进制值的例子。在着重号 below 中使用的表达式可以表示 SQL 标量表达式或是文本但不是一个列值。

- 当一个列值被比拟为表达式结果的时候，在比较开始前，列的亲和性将被应用在表达结果中。
- 当两个列值比较的时候，如果一个列有整数或数字亲和性的时候，而另外一列却没有，那么数字亲和性适用于从非数字列提取的任何具有文本存储类型的值. P>
- 当比较两个表达式的结果时,不发生任何转换,直接比较结果.如果一个字符串和一个数字比较,数字总是小于字符串.

在 SQLite 中, 表达式"a BETWEEN b AND c"等于表达式 "a >= b AND a <= c",在比较表达式时,a 可以是具有任何亲和性.

表达式 "a IN (SELECT b ....)" 在比较时遵循上面所提到的三条规则,是二进制比较.(例如, 在一个相似的样式 "a = b"). 例如,如果'b'是一个列值, 'a' 是一个表达式,那么,在开始比较前,'b'的亲和性就被转换为'a'的亲和性了.

SQLite 把表达式 "a IN (x, y, z)" 和 "a = z OR a = y OR a = z"视为相等.

3.1 比较例子

```
CREATE TABLE t1(
    a TEXT,
    b NUMERIC,
    c BLOB
);
```

```
-- Storage classes for the following row:
-- TEXT, REAL, TEXT
INSERT INTO t1 VALUES('500', '500', '500');

-- 60 and 40 are converted to '60' and '40' and values are compared as TEXT.
SELECT a < 60, a < 40 FROM t1;
1|0

-- Comparisons are numeric. No conversions are required.
SELECT b < 60, b < 600 FROM t1;
0|1

-- Both 60 and 600 (storage class NUMERIC) are less than '500'
-- (storage class TEXT).
SELECT c < 60, c < 600 FROM t1;
0|0
```

4. 运算符

所有的数学运算符(所有的运算符而不是连锁作用标记符"||")运算对象首先具有数字亲和性, 如果一个或是两个都不能被转换为数字那么操作的结果将是空值。

对于连接作用操作符，所有操作符将首先具有文本亲和性。如果其中任何一个操作符不能被转换为文本（因为它是空值或是 BLOB）连接作用操作符将是空值。

5. 分类，排序，混合挑选

当用子句 ORDER 挑选值时,空值首先被挑选出来, 然后是整数和实数按顺序被挑选出来, 然后是文本值按 memcmp()顺序被挑选出来, 最后是 BLOB 值按 memcmp()顺序被挑选出来.在挑选之前, 没有存储类型的值都被转换了.

When grouping values with the 当用 GROUP BY 子句给值分组时,具有不同存储类型的值被认为是不同的, 但也有例外, 比如,一个整数值和一个实数值从数字角度来说是相等的,那么它们则是相等的.用 GROUP by 子句比较完后,值不具有任何亲和性.

混合挑选操作符 UNION, INTERSECT and EXCEPT 在值之间实行绝对的比较,同样的亲和性将被应用于所有的值,这些值将被存储在一个单独的具有混合 SELECT 的结果组的列中. 被赋予的亲和性是该列的亲和性,这个亲和性是由剩下的大部分的混合 SELECTS 返回的,这些混合 SELECTS 在那个位置上有列值(而不是其它类型的表 达式). 如果一个给定的混合 SELECT 列没有 SELECTS 的量, 那么在比较前,该列的值将不具有任何亲和性.

6. 其它亲和性模式

以上的部分所描述的都是数据库引擎在正常亲和性模式下所进行的操作, SQLite 将描述其它两种亲和性模式,如下:

- **严格亲和性**模式. 在这种模式下, 如果需要值之间相互转换数据存储类型的话, 数据库引擎将发送错误报告, 当前语句也将会重新运行.
- **无亲和性**模式. 在这种模式下, 值的数据存储类型不发生转换. 具有不同存储类型的值之间不能比较, 但整数和实数之间可以比较.

7.用户定义的校对顺序

By default, when 当 SQLite 比较两个文本值的时候, 通过系统设定, 不管字符串的编码是什么, 用 memcmp()来比较. SQLite 第三版允许用户提供任意的函数来代替 memcmp(), 也就是用户定义的比较顺序.

除了系统预设的 BINARY 比较顺序, 它是用 memcmp()函数比较, SQLite 还包含了两个额外的内置比较顺序函数, NOCASE 和 REVERSE:

- **BINARY** -使用 memcmp()比较字符串数据, 不考虑文本编码.
- **REVERSE** -用倒序比较二进制文本.
- **NOCASE** - 和二进制一样,但在比较之前,26 位的大写字母盘要被折合成相应的小写字母盘.

7.1 分配比较顺序

每个表格中的每个列都有一个预设的比较类型. 如果一个比较类型不是二进制所要求的, 比较的子句将被具体化为 列的定义 来定义该列.

当用 SQLite 比较两个文本值时, 比较顺序将按照以下的规则来决定比较的结果. 文档的第三部分和第五部分描述在何种场合下发生这种比较.

对于二进制比较符(=, <, >, <= and >=), 如果每个操作数是一列的话, 那么该列的默认比较类型决定于所使用的比较顺序. 如果两个操作数都是列的话, 那么左边的操作数的比较类型决定了所要使用的比较顺序. 如果两个操作数都不是一列, 将使用二进制来比较.

表达式"x BETWEEN y and z"和 "x >= y AND x <= z"是相同的. 表达式"x IN (SELECT y ...)" 和表达式 "x = y" 使用同样的方法来操作, 这是为了决定所要使用的比较顺序. 如果 X 是一列或者二进制的, 则"x IN (y, z ...)" 形式的表达式所使用的比较顺序是 X 的默认的比较类型.

ORDER BY clause that is part of a SELECT statement may be assigned a collation sequence to be used for the sort operation explicitly. In this case the explicit collation sequence is always used. Otherwise, if the expression sorted by an ORDER BY clause is a column, then the default collation type of the column is used to determine sort order. If the expression is not a column, then the BINARY collation sequence is used.

7.2 比较顺序的例子

下面的例子介绍了 The examples below identify the collation sequences that would be used to determine the results of text comparisons that may be performed by various SQL statements.

Note that a text comparison may not be required, and no collation sequence used, in the case of numeric, blob or NULL values.

```
CREATE TABLE t1(
    a,                 -- default collation type BINARY
    b COLLATE BINARY,  -- default collation type BINARY
    c COLLATE REVERSE, -- default collation type REVERSE
    d COLLATE NOCASE   -- default collation type NOCASE
);

-- Text comparison is performed using the BINARY collation sequence.
SELECT (a = b) FROM t1;

-- Text comparison is performed using the NOCASE collation sequence.
SELECT (d = a) FROM t1;

-- Text comparison is performed using the BINARY collation sequence.
SELECT (a = d) FROM t1;

-- Text comparison is performed using the REVERSE collation sequence.
SELECT ('abc' = c) FROM t1;

-- Text comparison is performed using the REVERSE collation sequence.
SELECT (c = 'abc') FROM t1;

-- Grouping is performed using the NOCASE collation sequence (i.e. values
-- 'abc' and 'ABC' are placed in the same group).
SELECT count(*) GROUP BY d FROM t1;

-- Grouping is performed using the BINARY collation sequence.
SELECT count(*) GROUP BY (d || '') FROM t1;

-- Sorting is performed using the REVERSE collation sequence.
SELECT * FROM t1 ORDER BY c;

-- Sorting is performed using the BINARY collation sequence.
SELECT * FROM t1 ORDER BY (c || '');

-- Sorting is performed using the NOCASE collation sequence.
SELECT * FROM t1 ORDER BY c COLLATE NOCASE;
```

SQLite 不支持的 SQL 特性

相对于试图列出 SQLite 支持的所有 SQL92 特性，只列出不支持的部分要简单得多。下面显示的就是 SQLite 所不支持的 SQL92 特性。

这个列表的顺序关系到何时一个特性可能被加入到 SQLite。接近列表顶部的特性更可能在不远的将来加入。接近列表底部的特性尚且没有直接的计划。

完整的触发器支持（Complete trigger support）	现在有一些触发器的支持，但是还不完整。 缺少的特性包括 FOR EACH STATEMENT 触发器（现在所有的触发器都必须是 FOR EACH ROW ）， 在表上的 INSTEAD OF 触发器（现在 INSTEAD OF 触发器只允许在视图上），以及递归触发器——触发自身的触发器。
完整的 ALTER TABLE 支持（Complete ALTER TABLE support）	只支持 ALTER TABLE 命令的 RENAME TABLE 和 ADD COLUMN。 其他类型的 ALTER TABLE 操作如 DROP COLUMN，ALTER COLUMN，ADD CONSTRAINT 等等均被忽略。
RIGHT 和 FULL OUTER JOIN（RIGHT and FULL OUTER JOIN）	LEFT OUTER JOIN 已经实现，但还没有 RIGHT OUTER JOIN 和 FULL OUTER JOIN。
可写视图（Writing to VIEWs）	SQLite 中的视图是只读的。无法在一个视图上执行 DELETE，INSERT，UPDATE。 不过你可以创建一个试图在视图上 DELETE，INSERT，UPDATE 时触发的触发器，然后在触发器中完成你所需要的工作。
GRANT 和 REVOKE（GRANT and REVOKE）	由于 SQLite 读和写的是一个普通的磁盘文件， 因此唯一可以获取的权限就是操作系统的标准的文件访问权限。 一般在客户机/服务器架构的关系型数据库系统上能找到的 GRANT 和 REVOKE 命令对于一个嵌入式的数据库引擎来说是没有意义的， 因此也就没有去实现。

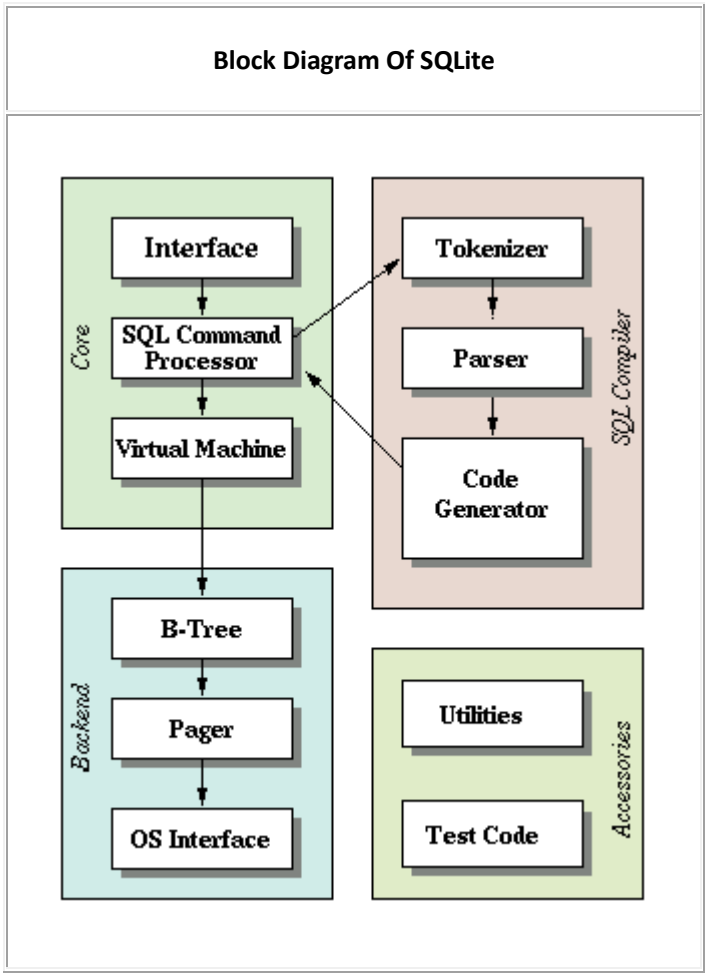
SQLite 的体系结构简介

简介

这篇文档主要描述了 SQLite 类库的结构。这篇文档的内容对于那些想了解和修改 SQLite 内部结构的人将会非常有用。

右侧是一个结构图，它显示了 SQLite 的主要成分及各成分之间是如何相互关联的。接下来的文本将简要的介绍每个单一的成分。

这篇文档描述 SQLite 第三版，它和 2.8 版以及早期的版本基本相似，但在一些细节上是有区别的。



接口程序

SQLite 类库大部分的公共接口程序是由 main.c, legacy.c, 和 vdbeapi.c 源文件中的功能执行的。但有些程序是分散在其他文件夹的，因为在其他文件夹里他们可以访问有文件作用域的数据结构。sqlite3_get_table() 这个程序是在 table.c 中执行的。 sqlite3_mprintf()在 printf.c 中执行。sqlite3_complete()在 tokenize.c 中执行。Tcl 接口程序用 tclsqlite.c 来执行。

为了避免和其它软件在名字上有冲突，SQLite 类库中所有的外部符号都是以 sqlite3 为前缀来命名的。这些被用来做外部使用的符号（换句话说，这些符号用来形成 SQLite 的 API）是以 sqlite3_.

来命名的。

Tokenizer

当执行一个包含 SQL 语句的字符串时，接口程序要把这个字符串传递给 tokenizer。Tokenizer 的任务是把原有字符串分成一个个标示符，并把这些标示符传递给剖析器。Tokenizer 是在 C 文件夹 tokenize.c 中用手编译的。

在这个设计中需要注意的一点是，tokenizer 调用 parser。熟悉 YACC 和 BISON 的人们也许会习惯于用 parser 调用 tokenizer。 The author of SQLite 的作者已经尝试了这两种方法，并发现用 tokenizer 调用 parser 会使程序运行的更顺利。YACC 使程序更滞后一些。

Parser

The parser 是一个部分，它基于文件场景赋予 tokens 意思。SQLite 的 parser 是由 Lemon LALR(1) parser generator 产生的。Lemon 和 YACC/BISON 一样做同样的工作，但是它使用不同的输入语句，这个输入语句是不易出错的。Lemon 也产生一个 parser，这个 parser 是可重入的并且是线程安全的。Lemon 定义了无终端解除程序的概念，所以当遇到语法错误的时候，它不会泄露内存。驱动 Lemon 的原文件在 parse.y.

因为 lemon 是一个在发展机械上不常见的程序，所以 lemon 的源代码（只是一个 C 文件）是在 SQLite 分布区的"tool"子目录下的。 lemon 的文档是在分布区的 "doc"子目录下的。

代码发生器

在剖析器收集完符号并把之转换成完全的 SQL 语句时，它调用代码产生器来产生虚拟的机器代码，这些机器代码将按照 SQL 语句的要求来工作。在代码产生器中有许多文件；attach.c, auth.c, build.c, delete.c, expr.c, insert.c, pragma.c, select.c, trigger.c, update.c, vacuum.c and where.c. 正是在这些文件中，最具有重要意义的事情发生了。 expr.c 处理表达式代码的生成。 where.c 处理 SELECT, UPDATE and DELETE 语句中 WHERE 子句的代码的生成。文件 attach.c, delete.c, insert.c, select.c, trigger.c update.c, 和 vacuum.c 处理 SQL 语句中具有同样名字的语句的代码的生成。(每个文件调用 expr.c and where.c 中的程序） All other 所有 SQL 的其它语句的代码是由 build.c 生成的。文件 auth.c 执行 sqlite3_set_authorizer()的功能.

虚拟机器

由代码生成器产生的程序由虚拟机器来运行。总而言之，虚拟机器主要用来执行一个为操作数据库而设计的抽象的计算引擎。机器有一个用来存储中间数据的存储栈。每个指令包含一个操作代码和三个额外的操作数。

虚拟机器本身是被包含在一个单独的文件 vdbe.c 中的。虚拟机器也有它自己的标题文件：vdbe.h 它在虚拟机器和剩下的 SQLite 类库之间定义了一个接口程序，vdbeInt.h 它定义了虚拟机器的结构。文件 vdbeaux.c 包含了虚拟机器所使用的实用程序和一些被其它类库用来建立 VM 程序的接口程序模块。文件 vdbeapi.c 包含虚拟机器的外部接口，比如 sqlite3_bind_... 类的函数。单独的值（字符串，整数，浮动点数值，BLOBS）被存储在一个叫 "Mem"的内部目标程序里，"Mem"是由 vdbemem.c 执行的。

SQLite 使用 C 语言程序来来执行 SQL 函数。即使内置的 SQL 函数也是用这种方法来执行的。大部分的 SQL 内置函数(ex: coalesce(), count(),substr(), and so forth)可以在 func.c 里发现。日期和时间转换函数在 date.c.

B-树

SQLite 数据库在磁盘里维护，使用源文件 btree.c 中的 B-树执行。数据库中的每个表格和目录使用一个单独的 B-tree。所有的 B-trees 被存储在同样的磁盘文件里。文件格式的细节被记录在 btree.c.

开头的备注里。

B-tree 子系统的接口程序被标题文件 btree.h 所定义。.

页面高速缓存

B-tree 模块要求信息来源于磁盘上固定规模的程序块。默认程序块的大小是 1024 个字节，但是可以在 512 和 65536 个字节间变化。 页面高速缓存负责读，写和高速缓存这些程序块。页面高速缓存还提供重新运算和提交抽象命令，它还管理关闭数据库文件夹。 B-tree 驱动器要求页面高速缓存器中的特别的页，当它想修改页或重新运行改变的时候，它会通报页面高速缓存。为了保证所有的需求被快速，安全和有效的 处理，页面高速缓存处理所有的微小的细节。

运行页面高速缓存的代码在专门的 C 源文件 pager.c 中。页面高速缓存的子系统的接口程序被目标文件 pager.h 所定义。

OS 接口程序

为了在 POSIX 和 Win32 之间提供一些可移植性，SQLite 操作系统的接口程序使用一个提取层。 OS 提取层的接口程序被定义在 os.h. 每个支持的操作系统有它自己的执行文件：Unix 使用 os_unix.c，windows 使用 os_win.c。每个具体的操作器具有它自己的标题文件： os_unix.h, os_win.h, etc.

Utilities

内存分配和字符串比较程序位于 util.c。剖析器使用的表格符号被 hash.c 中的无用信息表格维护。源文件 utf.c 包含 UNICODE 转换子程序。SQLite 有它自己的执行文件 printf() （有一些扩展）在 printf.c 中，还有它自己随机数量产生器在 random.c.

测试代码

如果你计算回归测试脚本，多于一半的 SQLite 代码数据库的代码将被测试。 在主要代码文件中有许多 assert()语句。另外,源文件 test1.c 通过 test5.c 和 md5.c 执行只为测试用的扩展名。os_test.c 向后的接口程序用来模拟断电，来验证页面调度程序中的系统性事故恢复机制。

SQLite 与其他数据库的速度比较

附注：这篇文档是一篇旧文档。它把老版的 SQLite 的速度和老版的 MySQL 和 PostgreSQL 的速度进行了对比。读者被热诚地邀请贡献更先进的速度对比在 SQLite Wiki.

这里的数字非常老，几乎没什么意义。在更新之前仍使用这篇文档只是为了证明 SQLite 不是停滞不前的。

执行程序总结

一系列的测试程序已经被运行去测量 SQLite 2.7.6，PostgreSQL 7.1.3，和 MySQL 3.23.41 的相对性能。以下是根据实验得出的一些总结：

- SQLite 2.7.6 是非常快的，有时可以比安装在 RedHat 7.2 上的预置的 PostgreSQL 7.1.3 快 10 倍甚至 20 倍。
- SQLite 2.7.6 总是很快， 有时比 MySQL 3.23.41 快 2 倍多。
- SQLite 执行 CREATE INDEX or DROP TABLE 时不像其它数据库那样快。但这不是什么问题，因为这些是很少运用的操作。
- 如果你把 multiple 操作聚集到一个单独的事物项，SQLite 可以更快些。

但以下几点是值得注意的：

- 这些测试程序不是在测量多用户使用时数据库的性能，也不是在测量包含 multiple 的最优化复杂查询。
- 这些测试是在一个相对小的数据库里完成的（大约 14 兆字节）。 它们没有测试数据库引擎的大小对程序运行速度的影响有多大。

测试环境

测试所用的平台是一个具有 1GB 内存的 1.6GHz Athlon 和一个 IDE 驱动硬盘。操作系统是具有一个 stock 内核的 RedHat Linux 7.2 。

使用的 PostgreSQL 和 MySQL 服务器被 RedHat 7.2 中的默认程序所传送。（PostgreSQL 7.1.3 版和 MySQL3.23.41 版）所使用的引擎自始至终没有被调整过。需要特别注意的是，RedHat 7.2 中默认的 MYSQL 配置不支持处理事物项，这一点使 MYSQL 的速度大大增加。但 SQLite 还是有能力完成大部分的测试的。

我被告知，RedHat 7.3 中预设的 PostgreSQL 配置是非常落后的（它必须在具有 8MB RAM 的机器上工作）。 and that PostgreSQL 则可以通过调整一些配置来运行的快些。 Matt Sergeant 报道说他已经调整了他的 PostgreSQL 装置，并且像下面所显示的一样重新进行测试。他的结果显示 PostgreSQL 和 MySQL 运行速度一样。如想查看结果，访问：

http://www.sergeant.org/sqlite_vs_pgsync.html

SQLite 实在同样的配置下被测试的，它是用 -O6 optimization 和 -DNDEBUG=1 交叉编写，这样使许多 SQLite 代码中的"assert()"语句 无法运行。

所有的测试都是在一个静止的机器上进行的。所有的测试时由一个简单的 TCL 文稿编排程序产生和运行的。 A copy of this Tcl script can be found in the 你可以在源文件目录文件 SQLite source tree in the file tools/speedtest.tcl 中发现 TCL 文稿编排程序的副本.

所有测试中的时间都是以精确到秒的背景时钟来计算的.SQLite 有两个单独的时间值. 第一个时间值在一个完整磁盘同步化打开的默认装置里.同步话打开后,为了确保重要数据已被真正的写入磁盘驱动表面,SQLite 在关键的时候执行 fsync()系 统调用. 在数据库更新过程中,当操作系统崩溃时或者计算机突然断电时,为了保证数据库的完整性,同步化是非常有必要的.第二个时间值是当同步化关闭的时候.关闭同 步化, SQLite 有时会运行的快些,但如果系统崩溃或者突然断电数据库将会受到损失. 通常来说,同步化的 SQLite 的时间是为了和 PostgreSQL(因为他也是同步化的)比较,异步化的 SQLite 是为了和也是异步化的 MySQL 引 擎比较.

测试 1: 1000 INSERTs

```
CREATE TABLE t1(a INTEGER, b INTEGER, c VARCHAR(100));
INSERT INTO t1 VALUES(1,13153,'thirteen thousand one hundred fifty three');
INSERT INTO t1 VALUES(2,75560,'seventy five thousand five hundred sixty');
... 995 lines omitted
INSERT INTO t1 VALUES(998,66289,'sixty six thousand two hundred eighty nine');
INSERT INTO t1 VALUES(999,24322,'twenty four thousand three hundred twenty two');
INSERT INTO t1 VALUES(1000,94142,'ninety four thousand one hundred forty two');
PostgreSQL:                4.373
MySQL:                     0.114
SQLite 2.7.6:             13.061
SQLite 2.7.6 (nosync):     0.223
```

因为没有一个中央服务器来控制访问，SQLite 必须先关闭再打开数据库文件，这样高速缓存器就失去了作用。在这个测试中，每个 SQL 语句都是一个独立的事务元，所以数据库文件必须被打开和关闭，高速缓存必须刷新 1000 次。 尽管这样，异步版本的 SQLite 还是和 MYSQL 一样快。但同步版本的却是非常慢。SQLite 在每个同步事务元后调用 fsync()，因而确保了磁盘表面所有的数据都是安全的。13 秒的测试时间大部分都被用于磁盘 I/O。

测试 2:在事务处理程序中的 25000 INSERTs

```
BEGIN;
CREATE TABLE t2(a INTEGER, b INTEGER, c VARCHAR(100));
INSERT INTO t2 VALUES(1,59672,'fifty nine thousand six hundred seventy two');
... 24997 lines omitted
INSERT INTO t2 VALUES(24999,89569,'eighty nine thousand five hundred sixty nine');
INSERT INTO t2 VALUES(25000,94666,'ninety four thousand six hundred sixty six');
COMMIT;
PostgreSQL:                4.900
```

MySQL: 2.184
SQLite 2.7.6: 0.914
SQLite 2.7.6 (nosync): 0.757

当所有的 INSERTs 被放入一个事务处理程序时，SQLite 不在需要关闭再打开数据库，或者使各个语句间的高速缓存器无效。在结束之前它也不用再执行 fsync()s。当解决了这个问题后 SQLite 比 PostgreSQL 和 MySQL 快很多。.

测试 3: 在编入索引表格中的 25000 INSERTs

```
BEGIN;
CREATE TABLE t3(a INTEGER, b INTEGER, c VARCHAR(100));
CREATE INDEX i3 ON t3(c);
... 24998 lines omitted
INSERT INTO t3 VALUES(24999,88509,'eighty eight thousand five hundred nine');
INSERT INTO t3 VALUES(25000,84791,'eighty four thousand seven hundred ninety one');
COMMIT;
```

PostgreSQL: 8.175
MySQL: 3.197
SQLite 2.7.6: 1.555
SQLite 2.7.6 (nosync): 1.402

有报告说 SQLite 在编入索引的表格中表现的不是很好。这个测试证明事实不是这样的，虽然 SQLite 在创建索引登录时不是很快，但它整体的速度仍是比 PostgreSQL 和 MySQL 快。

测试 4:没有索引的 100 SELECTs

```
BEGIN;
SELECT count(*), avg(b) FROM t2 WHERE b>=0 AND b<1000;
SELECT count(*), avg(b) FROM t2 WHERE b>=100 AND b<1100;
... 96 lines omitted
SELECT count(*), avg(b) FROM t2 WHERE b>=9800 AND b<10800;
SELECT count(*), avg(b) FROM t2 WHERE b>=9900 AND b<10900;
COMMIT;
```

PostgreSQL: 3.629
MySQL: 2.760
SQLite 2.7.6: 2.494
SQLite 2.7.6 (nosync): 2.526

这个测试是在一个没有索引的 25000 登录表格中进行 100 查询，所以需要浏览一个完整的表格。SQLite 之前的版本在这个测试上比 PostgreSQL 和 MySQL 慢，但最新版本的 SQLite 是比前两者都快的。

测试 5: 在一个字符串比较上的 100 SELECTs

```
BEGIN;
SELECT count(*), avg(b) FROM t2 WHERE c LIKE '%one%';
SELECT count(*), avg(b) FROM t2 WHERE c LIKE '%two%';
... 96 lines omitted
SELECT count(*), avg(b) FROM t2 WHERE c LIKE '%ninety nine%';
SELECT count(*), avg(b) FROM t2 WHERE c LIKE '%one hundred%';
COMMIT;
```

PostgreSQL:	13.409
MySQL:	4.640
SQLite 2.7.6:	3.362
SQLite 2.7.6 (nosync):	3.372

这个测试浏览 100 完整表格，但它使用字符串比较，而不是数字比较。This test still does 100 full table scans but it uses uses string comparisons instead of numerical comparisons. SQLite 比 PostgreSQL 快三倍，比 MYSQL 快 30%。

测试 6: 创建索引

```
CREATE INDEX i2a ON t2(a);
CREATE INDEX i2b ON t2(b);
```

PostgreSQL:	0.381
MySQL:	0.318
SQLite 2.7.6:	0.777
SQLite 2.7.6 (nosync):	0.659

SQLite 在创建新索引时是比较慢的，但这并不是什么大问题，因为创建新目录并不常用，希望 SQLite 之后的版本在这个问题上可以变得快些。

测试 7: 没有索引的 5000 SELECTs

```
SELECT count(*), avg(b) FROM t2 WHERE b>=0 AND b<100;
SELECT count(*), avg(b) FROM t2 WHERE b>=100 AND b<200;
SELECT count(*), avg(b) FROM t2 WHERE b>=200 AND b<300;
... 4994 lines omitted
SELECT count(*), avg(b) FROM t2 WHERE b>=499700 AND b<499800;
SELECT count(*), avg(b) FROM t2 WHERE b>=499800 AND b<499900;
SELECT count(*), avg(b) FROM t2 WHERE b>=499900 AND b<500000;
```

PostgreSQL:	4.614
MySQL:	1.270
SQLite 2.7.6:	1.121
SQLite 2.7.6 (nosync):	1.162

当有索引的时候，所有这三个数据库都运行的非常快，但 SQLite 仍是最快的。

测试 8: 没有索引的 1000 UPDATEs

```
BEGIN;
UPDATE t1 SET b=b*2 WHERE a>=0 AND a<10;
UPDATE t1 SET b=b*2 WHERE a>=10 AND a<20;
... 996 lines omitted
UPDATE t1 SET b=b*2 WHERE a>=9980 AND a<9990;
UPDATE t1 SET b=b*2 WHERE a>=9990 AND a<10000;
COMMIT;
PostgreSQL:           1.739
MySQL:                8.410
SQLite 2.7.6:         0.637
SQLite 2.7.6 (nosync): 0.638
```

对于这个特别的 UPDATE 测试，MySQL 比 PostgreSQL 和 SQLite 慢五或十倍。我不知为什么。 MySQL 通常情况下是个非常快的引擎。也许这个问题在之后的版本中被详细说明了。

测试 9: 有索引的 25000 UPDATEs

```
BEGIN;
UPDATE t2 SET b=468026 WHERE a=1;
UPDATE t2 SET b=121928 WHERE a=2;
... 24996 lines omitted
UPDATE t2 SET b=35065 WHERE a=24999;
UPDATE t2 SET b=347393 WHERE a=25000;
COMMIT;
PostgreSQL:           18.797
MySQL:                 8.134
SQLite 2.7.6:          3.520
SQLite 2.7.6 (nosync):  3.104
```

在这个测试中，最近的 2.7.0 版 SQLite 和 MYSQL 运行速度一样，但是最近对 SQLite 的优化使它速度比 UPDATEs 快一倍。

测试 10: 有索引的 25000 text UPDATEs

```
BEGIN;
UPDATE t2 SET c='one hundred forty eight thousand three hundred eighty two' WHERE a=1;
UPDATE t2 SET c='three hundred sixty six thousand five hundred two' WHERE a=2;
... 24996 lines omitted
UPDATE t2 SET c='three hundred eighty three thousand ninety nine' WHERE a=24999;
UPDATE t2 SET c='two hundred fifty six thousand eight hundred thirty' WHERE a=25000;
COMMIT;
PostgreSQL:           48.133
```

MySQL: 6.982
SQLite 2.7.6: 2.408
SQLite 2.7.6 (nosync): 1.725

2.7.0 版本的 SQLite 过去和 MYSQL 运行速度一样，但现在 2.7.6 版的 SQLite 的速度是 MYSQL 的两倍，是 PostgreSQL 的 20 倍。

在这个测试中，PostgreSQL 也很慢，一个有经验的管理者可以通过调试服务器使之运行的快些。

测试 11: 来源于 SELECT 的 INSERTs

```
BEGIN;
INSERT INTO t1 SELECT b,a,c FROM t2;
INSERT INTO t2 SELECT b,a,c FROM t1;
COMMIT;
```

PostgreSQL: 61.364
MySQL: 1.537
SQLite 2.7.6: 2.787
SQLite 2.7.6 (nosync): 1.599

在这个测试中,异步的 SQLite 比 MYSQL 慢(MYSQL 似乎特别擅长 INSERT...SELECT 语句)。PostgreSQL 引擎仍然是非常慢的， 61 秒中的大部分时间被用来等待磁盘 I/O。

测试 12: 没有索引的 DELETE

```
DELETE FROM t2 WHERE c LIKE '%fifty%';
```

PostgreSQL: 1.509
MySQL: 0.975
SQLite 2.7.6: 4.004
SQLite 2.7.6 (nosync): 0.560

在这个测试中，The synchronous version of 同步版本的 SQLite 是这组中最慢的，但异步版本的 SQLite 是最快的。不同的是，它需要额外的时间去执行 fsync()。

测试 13: 有索引的 DELETE

```
DELETE FROM t2 WHERE a>10 AND a<20000;
```

PostgreSQL: 1.316
MySQL: 2.262
SQLite 2.7.6: 2.068
SQLite 2.7.6 (nosync): 0.752

这个测试非常重要，因为在这里 PostgreSQL 比 MySQL 要快。SQLite 比前两者都要快。

测试 14: 一个大 DELETE 之后的一个大 INSERT

```
INSERT INTO t2 SELECT * FROM t1;
PostgreSQL:           13.168
MySQL:                 1.815
SQLite 2.7.6:          3.210
SQLite 2.7.6 (nosync): 1.485
```

一些老版的 SQLite（2.4.0 以前的版本）在执行完 DELETEs 及新 INSERTs 后明显慢下来，但在这个测试中我们可以看到，这个问题已经被解决了。

测试 15: 一个大的 DELETE 及许多小 INSERTs

```
BEGIN;
DELETE FROM t1;
INSERT INTO t1 VALUES(1,10719,'ten thousand seven hundred nineteen');
... 11997 lines omitted
INSERT INTO t1 VALUES(11999,72836,'seventy two thousand eight hundred thirty six');
INSERT INTO t1 VALUES(12000,64231,'sixty four thousand two hundred thirty one');
COMMIT;
PostgreSQL:           4.556
MySQL:                1.704
SQLite 2.7.6:         0.618
SQLite 2.7.6 (nosync): 0.406
```

SQLite 通常总是会在一个事务处理程序中执行 INSERTs，这也许就是为什么在这个测试中 SQLite 通常比其它数据库快很多的原因。

测试 16: DROP TABLE

```
DROP TABLE t1;
DROP TABLE t2;
DROP TABLE t3;
PostgreSQL:           0.135
MySQL:                0.015
SQLite 2.7.6:         0.939
SQLite 2.7.6 (nosync): 0.254
```

SQLite 在执行撤销表格这一操作时比其它数据库要慢一些。 这也许是因为当 SQLite 撤销一个表格的时候，它必须全面检查并清除数据库文件中的记录。与之不同的是，MySQL 和 PostgreSQL 分别的文件夹来代表每个表格，所以如果它们想撤销一个表格，它们只需删除一个文件，这当然要快一些了。

但是，撤销表格并不是一个常用的操作，所以 SQLite 慢一些也不会有什么问题。

SQLite 中的空处理与其它数据库引擎的比较

我的目标是使 SQLite 用一种标准和顺从的方法来处理空值。但是在 SQL 标准中关于如何处理空值的描述似乎不太明确。从标准文档中，我们不太容易弄清楚空值在所有场合下是如何被处理的。

所以标准文档被取代，各种流行的 SQL 引擎被用来测试，看它们是如何处理空值的。我的目的是想 SQLite 像其他引擎一样工作。志愿者们开发了 SQL 的测试脚本并使之在 SQL RDBMSes 上运行，运用测试的结果来推论空值在各种引擎上是如何被处理的。最初的测试是在 2002 年 5 月运行的。测试脚本的副本在这篇文档的最后。

SQLite 最初是这样编译的，对于下面表格中的所有问题，它的答案都是"Yes"。 但是在其它 SQL 引擎上的测试表明没有一个引擎是这样工作的。所以 SQLite 被改进了，改进后它像 Oracle, PostgreSQL, and DB2 一样工作。改进后，对于 SELECT DISTINCT 语句和 SELECT 中的 UNIQUE 操作符，空值是模糊的。在 UNIQUE 列中空值仍然是清晰的。这看起来有些独裁的意思，但是使 SQLite 和其它数据 库引擎兼容似乎比这个缺陷更重要。

为了 SELECT DISTINCT 和 UNION，使 SQLite 认为空值是清晰的是有可能的。但是你需要在 sqliteInt.h 原文件中改变 NULL_ALWAYS_DISTINCT #define 的值，并重新编译。

更新于 2003-07-13: 这篇文档写的很早，一些被测试的数据库引擎已经被更新，忠实地使用者也发送了一些关于下面表格的修正意见。原始数据显示了各种不同的状态，但是随着时间的 变化，数据的状态已经逐渐向 PostgreSQL/Oracle 模式汇合。唯一的突出的不同是 Informix and MS-SQL 在 UNIQUE 列中都认为空值是模糊的。

令人迷惑的一点是，NULLs 对于 UNIQUE 列是清晰的，但对于 SELECT DISTINCT 和 UNION 是模糊的。空值应该是清晰或模糊都可以。但 SQL 标准文档建议空值在所有地方都是清晰的。但在这篇作品中，被测试的 SQL 引擎认为在 SELECT DISTINCT 或在 UNION 中，空值是清晰的。

下面的表格显示了空处理实验的结果。

	SQLite	PostgreSQL	Oracle	Informix	DB2	MS-SQL	OCELOT
Adding anything to null gives null	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Multiplying null by zero gives null	Yes	Yes	Yes	Yes	Yes	Yes	Yes
nulls are distinct in a UNIQUE column	Yes	Yes	Yes	No	(Note 4)	No	Yes
nulls are distinct in SELECT DISTINCT	No	No	No	No	No	No	No
nulls are distinct in a UNION	No	No	No	No	No	No	No
"CASE WHEN null THEN 1 ELSE 0 END" is 0?	Yes	Yes	Yes	Yes	Yes	Yes	Yes

"null OR true" is true	Yes	Yes	Yes	Yes	Yes	Yes	Yes
"not (null AND false)" is true	Yes	Yes	Yes	Yes	Yes	Yes	Yes

	MySQL 3.23.41	MySQL 4.0.16	Firebird	SQL Anywhere	Borland Interbase
Adding anything to null gives null	Yes	Yes	Yes	Yes	Yes
Multiplying null by zero gives null	Yes	Yes	Yes	Yes	Yes
nulls are distinct in a UNIQUE column	Yes	Yes	Yes	(Note 4)	(Note 4)
nulls are distinct in SELECT DISTINCT	No	No	No (Note 1)	No	No
nulls are distinct in a UNION	(Note 3)	No	No (Note 1)	No	No
"CASE WHEN null THEN 1 ELSE 0 END" is 0?	Yes	Yes	Yes	Yes	(Note 5)
"null OR true" is true	Yes	Yes	Yes	Yes	Yes

Notes:
1. Older versions of firebird omits all NULLs from SELECT DISTINCT and from UNION.
2. Test data unavailable.
3. MySQL version 3.23.41 does not support UNION.
4. DB2, SQL Anywhere, and Borland Interbase do not allow NULLs in a UNIQUE column.
5. Borland Interbase does not support CASE expressions.

"not (null AND false)" is true	No	Yes	Yes	Yes	Yes

下面的脚本被用来收集关于上面表格的信息。

-- 我认为 SQL 关于空值的处理是不定的，所以不能靠逻辑来推断，必须同过实验来发现结果。为了实现这个目标，我已经准备了下列的脚本来测试不同的 SQL 数据库如何处理空值。
我的目标是使用这个脚本收集的信息来使 SQLite 尽量像其它数据库一样工作。如果你可以在你的数据库引擎里运行这个脚本并把结果发送到 drh@hwaci.com，那将是对我的莫大帮助。
请标识你为这个测试所使用的引擎。谢谢。如果你为了在你的数据库引擎上运行测试而修改了原来的脚本，请把你所修改的脚本和结果一起发送过来。

--

```
-- 创建一个具有数据的测试表格
create table t1(a int, b int, c int);
insert into t1 values(1,0,0);
insert into t1 values(2,0,1);
insert into t1 values(3,1,0);
insert into t1 values(4,1,1);
insert into t1 values(5,null,0);
insert into t1 values(6,null,1);
```

```
insert into t1 values(7,null,null);

-- 看 CASE 在测试表达式中如何处理空值
select a, case when b<>0 then 1 else 0 end from t1;
select a+10, case when not b<>0 then 1 else 0 end from t1;
select a+20, case when b<>0 and c<>0 then 1 else 0 end from t1;
select a+30, case when not (b<>0 and c<>0) then 1 else 0 end from t1;
select a+40, case when b<>0 or c<>0 then 1 else 0 end from t1;
select a+50, case when not (b<>0 or c<>0) then 1 else 0 end from t1;
select a+60, case b when c then 1 else 0 end from t1;
select a+70, case c when b then 1 else 0 end from t1;

-- 如果你用空值乘以零会发生什么结果?
select a+80, b*0 from t1;
select a+90, b*c from t1;

-- 其它操作符和空值会发生什么?
select a+100, b+c from t1;

-- 关于集合操作符的测试
select count(*), count(b), sum(b), avg(b), min(b), max(b) from t1;

-- 在 WHERE 子句中检查空值的状态
select a+110 from t1 where b<10;
select a+120 from t1 where not b>10;
select a+130 from t1 where b<10 OR c=1;
select a+140 from t1 where b<10 AND c=1;
select a+150 from t1 where not (b<10 AND c=1);
select a+160 from t1 where not (c=1 AND b<10);

-- 在 DISTINCT 查询中检查空值的状态
select distinct b from t1;

-- 在 UNION 查询中检查空值的状态
select b from t1 union select b from t1;

-- 用 unique 列创建一个新表格，检查空值是否被认为是清晰的。
create table t2(a int, b int unique);
insert into t2 values(1,1);
insert into t2 values(2,null);
insert into t2 values(3,null);
select * from t2;

drop table t1;
drop table t2;
```

SQLite 数据库的速度比较(wiki)

来运行这些测试的脚本文件是here 中的一个修改的版本。我所改良的版本是连接在这页上的。但是我本人不是一个 TCL 程序员，我所修改的版本只是由于数据库的需要。如果你看到一些错误影响了测试的结果，请你发送你的结果到这里。

所有的数据库都有自己默认的设置。如果是 SQLite，我就使用二进制。

测试所运行的环境是 1.6GHz Sempron，1GB 随机存取内存，7200rpm SATA 硬盘， Windows 2000 + SP4 和所有的更新材料。

考虑到 SQL 的一些细小的变化，原来的测试脚本被改变了，因而可以使之在所有的数据库引擎下都可以运行。如果是 MySQL，我就必须改变 CREATE TABLE 语句，使之可以使用 InnoDB and MyISAM 存储引擎，如果是 InnoDB 引擎则关闭自动更新数据库模式（即使文档要求 BEGIN 自动执行，但实际上却不是如此）如果是 Firebird 我则必须把 BEGIN 改成 SET TRANSACTION.
除了上述情况外，所有的数据库使用同样的 SQL 执行。

所有的 SELECTs 在一个行里被执行三次，平均的时间在结果中显示。

我不知道如何在 windows 下如何刷新磁盘缓冲器。这个脚本使用 Linux 中的 sync 命令来执行刷新过程。如果你知道如何在 windows 下执行刷新，请告知我。你也许在{link: http://www.checksum.org/cso/downloads/ from here}调试 NTSync。

如果你想要个除了数字的解释，我建议你读original tests. 那里所讲的大部分在这里都是使用的。

我觉得在测试 6 中服务器也许做了下列操作：

```
CREATE INDEX t1_b ON t1(b);
CREATE INDEX t2_b ON t2(b);
SELECT t1.a FROM t1 INNER JOIN t2 ON t1.b=t2.b;
DROP INDEX t1_b;
DROP INDEX t2_b;
```

上面的查询，当在同样的数据库里运行时，就像测试 6 中的查询，可以在 0.5 秒内完成，而不是 14+ 秒。 事实上它比测试 7 的时间还要快一点，但这是由于可变的运行时间状态。

我不太清楚测试 8 中的 Postgres 有什么问题， 如果你知道的话，并且如果你认为那个特殊的测试使 Postgres 有问题是因为连接了 SQL，请你告知我。

A Speak UP: Postgres 是用来默认安装的，在 shoebox 上运行。如果被使用在默认设置中用来做一系列的测试，它运行的并不太理想。Postgres 只有一个 sync 模式来保证 ACID 的一致性。Postgres 依据它静态数据库中的数据来决定如何执行查询。数据只有通过"Analyze" 命令来更新。其他数据库也许用不同的命令来跟踪那些数据。在其他无事务处理的插入测试中，如果想看到一个 7200 RPM 的驱动在 0.7 秒内做 1000 次的 I/O 操作是非常困难的。sync 会强制每个事务处理程序在硬盘里执行，它就是每个 INSERT 操作。

Reply: 从测试开始运行后，Postgres 的 conf 文件就被调试了一些。我正在把 conf 文件连接到所有数据库，这样任何人都可以评论并提出修改意见，因而在 sync 测试中的 ACID 一致性方面使所有数据库都使用大致相同的硬件资源和函数。一旦所有数据库的输入都被收集到的时候，我将重新执行测试。
在实际操作中，Postgres 在每个查询之前就运行 ANALYZE 到底是不是很经常呢？如果是的话，我将更新测试脚本使之在每个测试前运行 ANALYZE。
测试 1 是非常奇怪的，这一点我也同意。如果每个处理程序只使用一个 fsync，理想中完美的数据库也要用 8 秒以上的时间来完成这个测试。

Another speak UP :-) : 在大量插入或大部分行被更新后分析表格。产生的统计量将允许 planner 为 SELECTs -选择一个最有效的查询计划。一个典型的例子就是一个充满数据的表格具有数百万个记录，如果之前从没被分析过， planner 也许认为表格只有 100 行，然后从目录扫描中按顺序来扫描表格，但这也许会和实际上的表格大小相差甚远。还要注意的 是，PostgreSQL 在高并发加载数据时表现的非常出色。我也考虑到扩大测试，这样你就可以在同一时刻在多个事务处理程序中有数十个连接既读又写数 据。可以在性能更优越的硬件上把测试扩展到数百个甚至数千个连接。

Reply: 对于 ANALYZE，在表格发生重大变化后，我将尽全力做我所能做的，并把它加入到测试脚本中。我认为这点我还是能做到的。我计划把 ANALYZE 花费的 时间包含到对 Postgress 测试中。我认为这是非常合理的，因为没有任何的其他数据库有这样特殊的措施，并且 AFAICT ANALYZE 是非常快的。
不需怀疑的是，Postgres 在高并发性的测试中会表现得很出色。但如果你需要数百个同步的复写器或读卡器，你则不太可能对 SQLite 感兴趣。所以，到现在为止，我还没有运行这些测试的意向。

关于结果的一些注释：

- 时间是以秒为单位的，它代表了背景时钟的时间。
- nosync 如果是 SQLite 就意味着 SQLite 用 PRAGMA synchronous=OFF 运行; ，如果是 MySQL 则意味着表格的类型是 MyISAM。
- sync 如果是 SQLite 意味着 SQLite 用 PRAGMA synchronous=FULL; ，如果是 MySQL 就意味着表格的类型是 InnoDB。

Comment: 你可以通过安装 postgresql.conf 中的’fsync = false’ 来关闭 PostgreSQL 中的 syncing (ie: nosync)。 这个安装不是很轻松，但是这会便于和其他数据库中你正在关闭或打开的 sync 做对比。另外，在变化很大的表格中运行 analyze 是很普遍的。通常情况 下，PostgreSQL 认为使用中的表格的行自始至终是被插入的，被更新的和被删除的，而通常的运行的规模和统计量是不变的。因为整体的统计量不会变化 很多,所以你可能不会在 every 查询前运行 analyze。但你可能运行 ’vacuum analyze’ 来标记一些可以替换的行和更新统计量。在表格有些改动后，这样的测试对 analyze 非常有意义。如果表格是可以再生的（行可以更新和被删除），应该运行 vacuum and/or vacuum full 。
测试 8 如果不运行 analyze 可能会出现一些问题， Postgres 不使用索引也许是因为它运行默认统计量，这意味着只有 100 行或像 100 行这样的情况下有顺序的浏览会更快一些。 另外一个优点是每个查询都用’explain analyze’来重新运行 Postgres 并同时提供输出量，以此来验证正在使用的查询计划。

Reply: nosync 测试运行起来很简单，这样可以使 SQLite 的开发者和使用者对之有更好的理解。
MySQL 被加入到 mix，因为 AFAIK 对于 MYSQL 的运行来说是一个特别常见的设置。此外，我对 sync 的表现非常感兴趣，因为它是最常见的操作模 式。对于 MySQL 中的 Except ，我不知道三年中它会不会改变。

Comment:我下拉网页 postgresql.conf 注意到你打开了行级别的数据统计。这对于有许多查询的测试来说是有些负面影响的。为什么你这样做？
另外我建议一些测试的准备好的语句版本在支持它的数据库里运行。

Reply: 我认为我没有改变它，也许他是默认装成那样的。我所做的改动只是在邮寄目录中，这里是相关参考：

```
shared_buffers=10000
effective_cache_size=100000
work_mem=10000
vacuum_cost_delay=50
autovacuum=on
autovacuum_vacuum_scale_factor=0.2
```

关于应答的说明: Postgres 8.x 的默认装置 definitely 有

```
#stats_row_level = false
```

在他们默认的设置中，你应该关闭 row stats collector，从新运行测试。

关于测试

Test 1: 1000 INSERTs

```
CREATE TABLE t1(a INTEGER, b INTEGER, c VARCHAR(100));
INSERT INTO t1 VALUES(1,13153,'thirteen thousand one hundred fifty three');
INSERT INTO t1 VALUES(2,75560,'seventy five thousand five hundred sixty');
... 995 lines omitted
INSERT INTO t1 VALUES(998,66289,'sixty six thousand two hundred eighty nine');
INSERT INTO t1 VALUES(999,24322,'twenty four thousand three hundred twenty two');
INSERT INTO t1 VALUES(1000,94142,'ninety four thousand one hundred forty two');
SQLite 3.3.3 (sync):          3.823
SQLite 3.3.3 (nosync):        1.668
SQLite 2.8.17 (sync):         4.245
SQLite 2.8.17 (nosync):       1.743
PostgreSQL 8.1.2:             4.922
MySQL 5.0.18 (sync):          2.647
MySQL 5.0.18 (nosync):        0.329
FirebirdSQL 1.5.2:            0.320
```

Test 2: 25000 INSERTs in a transaction

```
BEGIN;
CREATE TABLE t2(a INTEGER, b INTEGER, c VARCHAR(100));
INSERT INTO t2 VALUES(1,298361,'two hundred ninety eight thousand three hundred sixty one');
... 24997 lines omitted
```

INSERT INTO t2 VALUES(24999,447847,'four hundred forty seven thousand eight hundred forty seven');
INSERT INTO t2 VALUES(25000,473330,'four hundred seventy three thousand three hundred thirty');
COMMIT;
SQLite 3.3.3 (sync): 0.764
SQLite 3.3.3 (nosync): 0.748
SQLite 2.8.17 (sync): 0.698
SQLite 2.8.17 (nosync): 0.663
PostgreSQL 8.1.2: 16.454
MySQL 5.0.18 (sync): 7.833
MySQL 5.0.18 (nosync): 7.038
FirebirdSQL 1.5.2: 4.280

Test 3: 25000 INSERTs into an indexed table

BEGIN;
CREATE TABLE t3(a INTEGER, b INTEGER, c VARCHAR(100));
CREATE INDEX i3 ON t3(c);
... 24998 lines omitted
INSERT INTO t3 VALUES(24999,442549,'four hundred forty two thousand five hundred forty nine');
INSERT INTO t3 VALUES(25000,423958,'four hundred twenty three thousand nine hundred fifty eight');
COMMIT;
SQLite 3.3.3 (sync): 1.778
SQLite 3.3.3 (nosync): 1.832
SQLite 2.8.17 (sync): 1.526
SQLite 2.8.17 (nosync): 1.364
PostgreSQL 8.1.2: 19.236
MySQL 5.0.18 (sync): 11.524
MySQL 5.0.18 (nosync): 12.427
FirebirdSQL 1.5.2: 6.351

Test 4: 100 SELECTs without an index

SELECT count(*), avg(b) FROM t2 WHERE b>=0 AND b<1000;
SELECT count(*), avg(b) FROM t2 WHERE b>=100 AND b<1100;
SELECT count(*), avg(b) FROM t2 WHERE b>=200 AND b<1200;
... 94 lines omitted
SELECT count(*), avg(b) FROM t2 WHERE b>=9700 AND b<10700;
SELECT count(*), avg(b) FROM t2 WHERE b>=9800 AND b<10800;
SELECT count(*), avg(b) FROM t2 WHERE b>=9900 AND b<10900;
SQLite 3.3.3 (sync): 3.153
SQLite 3.3.3 (nosync): 3.088
SQLite 2.8.17 (sync): 3.993
SQLite 2.8.17 (nosync): 3.983
PostgreSQL 8.1.2: 5.740
MySQL 5.0.18 (sync): 2.718

MySQL 5.0.18 (nosync): 1.641
FirebirdSQL 1.5.2: 2.976

Test 5: 100 SELECTs on a string comparison

SELECT count(*), avg(b) FROM t2 WHERE c LIKE '%one%';
SELECT count(*), avg(b) FROM t2 WHERE c LIKE '%two%';
SELECT count(*), avg(b) FROM t2 WHERE c LIKE '%three%';
... 94 lines omitted
SELECT count(*), avg(b) FROM t2 WHERE c LIKE '%ninety eight%';
SELECT count(*), avg(b) FROM t2 WHERE c LIKE '%ninety nine%';
SELECT count(*), avg(b) FROM t2 WHERE c LIKE '%one hundred%';
SQLite 3.3.3 (sync): 4.853
SQLite 3.3.3 (nosync): 4.868
SQLite 2.8.17 (sync): 4.511
SQLite 2.8.17 (nosync): 4.500
PostgreSQL 8.1.2: 6.565
MySQL 5.0.18 (sync): 3.424
MySQL 5.0.18 (nosync): 2.090
FirebirdSQL 1.5.2: 5.803

Test 6: INNER JOIN without an index

SELECT t1.a FROM t1 INNER JOIN t2 ON t1.b=t2.b;
SQLite 3.3.3 (sync): 14.473
SQLite 3.3.3 (nosync): 14.445
SQLite 2.8.17 (sync): 47.776
SQLite 2.8.17 (nosync): 47.750
PostgreSQL 8.1.2: 0.176
MySQL 5.0.18 (sync): 3.421
MySQL 5.0.18 (nosync): 3.443
FirebirdSQL 1.5.2: 0.141

Test 7: Creating an index

CREATE INDEX i2a ON t2(a);
CREATE INDEX i2b ON t2(b);
SQLite 3.3.3 (sync): 0.552
SQLite 3.3.3 (nosync): 0.526
SQLite 2.8.17 (sync): 0.650
SQLite 2.8.17 (nosync): 0.605
PostgreSQL 8.1.2: 0.276
MySQL 5.0.18 (sync): 1.159
MySQL 5.0.18 (nosync): 0.275

FirebirdSQL 1.5.2: 0.264

Test 8: 5000 SELECTs with an index

SELECT count(*), avg(b) FROM t2 WHERE b>=0 AND b<100;
SELECT count(*), avg(b) FROM t2 WHERE b>=100 AND b<200;
SELECT count(*), avg(b) FROM t2 WHERE b>=200 AND b<300;
... 4994 lines omitted
SELECT count(*), avg(b) FROM t2 WHERE b>=499700 AND b<499800;
SELECT count(*), avg(b) FROM t2 WHERE b>=499800 AND b<499900;
SELECT count(*), avg(b) FROM t2 WHERE b>=499900 AND b<500000;
SQLite 3.3.3 (sync): 1.872
SQLite 3.3.3 (nosync): 1.853
SQLite 2.8.17 (sync): 2.444
SQLite 2.8.17 (nosync): 2.478
PostgreSQL 8.1.2: 199.823
MySQL 5.0.18 (sync): 3.763
MySQL 5.0.18 (nosync): 3.725
FirebirdSQL 1.5.2: 5.187
* Performance of PostgreSQL in this test is most probably heavily impacted by *psql* command line utility. Same test when run form pgAdmin III GUI completed in 5 seconds.

Test 9: 1000 UPDATEs without an index

BEGIN;
UPDATE t1 SET b=b*2 WHERE a>=0 AND a<10;
UPDATE t1 SET b=b*2 WHERE a>=10 AND a<20;
... 996 lines omitted
UPDATE t1 SET b=b*2 WHERE a>=9980 AND a<9990;
UPDATE t1 SET b=b*2 WHERE a>=9990 AND a<10000;
COMMIT;
SQLite 3.3.3 (sync): 0.562
SQLite 3.3.3 (nosync): 0.573
SQLite 2.8.17 (sync): 0.543
SQLite 2.8.17 (nosync): 0.532
PostgreSQL 8.1.2: 1.663
MySQL 5.0.18 (sync): 1.930
MySQL 5.0.18 (nosync): 4.656
FirebirdSQL 1.5.2: 1.804

Test 10: 25000 UPDATEs with an index

BEGIN;
UPDATE t2 SET b=271822 WHERE a=1;

```
UPDATE t2 SET b=28304 WHERE a=2;
... 24996 lines omitted
UPDATE t2 SET b=442549 WHERE a=24999;
UPDATE t2 SET b=423958 WHERE a=25000;
COMMIT;
```

SQLite 3.3.3 (sync):	1.883
SQLite 3.3.3 (nosync):	1.894
SQLite 2.8.17 (sync):	1.994
SQLite 2.8.17 (nosync):	1.973
PostgreSQL 8.1.2:	23.933
MySQL 5.0.18 (sync):	16.348
MySQL 5.0.18 (nosync):	17.383
FirebirdSQL 1.5.2:	15.542

Test 11: 25000 text UPDATEs with an index

```
BEGIN;
UPDATE t2 SET c='four hundred sixty eight thousand twenty six' WHERE a=1;
UPDATE t2 SET c='one hundred twenty one thousand nine hundred twenty eight' WHERE a=2;
... 24996 lines omitted
UPDATE t2 SET c='thirty five thousand sixty five' WHERE a=24999;
UPDATE t2 SET c='three hundred forty seven thousand three hundred ninety three' WHERE a=25000;
COMMIT;
```

SQLite 3.3.3 (sync):	1.386
SQLite 3.3.3 (nosync):	1.365
SQLite 2.8.17 (sync):	1.168
SQLite 2.8.17 (nosync):	1.121
PostgreSQL 8.1.2:	24.672
MySQL 5.0.18 (sync):	16.469
MySQL 5.0.18 (nosync):	15.491
FirebirdSQL 1.5.2:	21.583

Test 12: INSERTs from a SELECT

```
BEGIN;
INSERT INTO t1 SELECT * FROM t2;
INSERT INTO t2 SELECT * FROM t1;
COMMIT;
```

SQLite 3.3.3 (sync):	1.179
SQLite 3.3.3 (nosync):	1.116
SQLite 2.8.17 (sync):	1.864
SQLite 2.8.17 (nosync):	1.526
PostgreSQL 8.1.2:	1.091
MySQL 5.0.18 (sync):	0.986
MySQL 5.0.18 (nosync):	0.933

FirebirdSQL 1.5.2: 4.782

Test 13: INNER JOIN with index on one side

SELECT t1.a FROM t1 INNER JOIN t2 ON t1.b=t2.b;
SQLite 3.3.3 (sync): 0.371
SQLite 3.3.3 (nosync): 0.369
SQLite 2.8.17 (sync): 0.273
SQLite 2.8.17 (nosync): 0.275
PostgreSQL 8.1.2: 5.981
MySQL 5.0.18 (sync): 0.408
MySQL 5.0.18 (nosync): 0.603
FirebirdSQL 1.5.2: 1.099

Test 14: INNER JOIN on text field with index on one side

SELECT t1.a FROM t1 INNER JOIN t3 ON t1.c=t3.c;
SQLite 3.3.3 (sync): 0.383
SQLite 3.3.3 (nosync): 0.376
SQLite 2.8.17 (sync): 0.309
SQLite 2.8.17 (nosync): 0.291
PostgreSQL 8.1.2: 1.324
MySQL 5.0.18 (sync): 0.404
MySQL 5.0.18 (nosync): 0.558
FirebirdSQL 1.5.2: 0.454

Test 15: 100 SELECTs with subqueries. Subquery is using an index

SELECT t1.a FROM t1 WHERE t1.b IN (SELECT t2.b FROM t2 WHERE t2.b>=0 AND t2.b<1000);
SELECT t1.a FROM t1 WHERE t1.b IN (SELECT t2.b FROM t2 WHERE t2.b>=100 AND t2.b<1100);
SELECT t1.a FROM t1 WHERE t1.b IN (SELECT t2.b FROM t2 WHERE t2.b>=200 AND t2.b<1200);
... 94 lines omitted
SELECT t1.a FROM t1 WHERE t1.b IN (SELECT t2.b FROM t2 WHERE t2.b>=9700 AND t2.b<10700);
SELECT t1.a FROM t1 WHERE t1.b IN (SELECT t2.b FROM t2 WHERE t2.b>=9800 AND t2.b<10800);
SELECT t1.a FROM t1 WHERE t1.b IN (SELECT t2.b FROM t2 WHERE t2.b>=9900 AND t2.b<10900);
SQLite 3.3.3 (sync): 7.877
SQLite 3.3.3 (nosync): 8.040
SQLite 2.8.17 (sync): 4.387
SQLite 2.8.17 (nosync): 4.381
PostgreSQL 8.1.2: 6.245
MySQL 5.0.18 (sync): 16.891
MySQL 5.0.18 (nosync): 38.447

FirebirdSQL 1.5.2: 37.439

Test 16: DELETE without an index

DELETE FROM t2 WHERE c LIKE '%fifty%';
SQLite 3.3.3 (sync): 0.528
SQLite 3.3.3 (nosync): 0.429
SQLite 2.8.17 (sync): 1.228
SQLite 2.8.17 (nosync): 0.984
PostgreSQL 8.1.2: 0.336
MySQL 5.0.18 (sync): 0.394
MySQL 5.0.18 (nosync): 0.532
FirebirdSQL 1.5.2: 0.404

Test 17: DELETE with an index

DELETE FROM t2 WHERE a>10 AND a<20000;
SQLite 3.3.3 (sync): 0.866
SQLite 3.3.3 (nosync): 0.627
SQLite 2.8.17 (sync): 1.275
SQLite 2.8.17 (nosync): 0.817
PostgreSQL 8.1.2: 0.283
MySQL 5.0.18 (sync): 0.541
MySQL 5.0.18 (nosync): 1.336
FirebirdSQL 1.5.2: 5.033

Test 18: A big INSERT after a big DELETE

INSERT INTO t2 SELECT * FROM t1;
SQLite 3.3.3 (sync): 0.973
SQLite 3.3.3 (nosync): 0.865
SQLite 2.8.17 (sync): 1.680
SQLite 2.8.17 (nosync): 1.336
PostgreSQL 8.1.2: 0.727
MySQL 5.0.18 (sync): 0.762
MySQL 5.0.18 (nosync): 1.088
FirebirdSQL 1.5.2: 4.171

Test 19: A big DELETE followed by many small INSERTs

BEGIN;
DELETE FROM t1;
INSERT INTO t1 VALUES(1,29676,'twenty nine thousand six hundred seventy six');

... 2997 lines omitted

```
INSERT INTO t1 VALUES(2999,37835,'thirty seven thousand eight hundred thirty five');
INSERT INTO t1 VALUES(3000,97817,'ninety seven thousand eight hundred seventeen');
COMMIT;
```

SQLite 3.3.3 (sync):	0.155
SQLite 3.3.3 (nosync):	0.133
SQLite 2.8.17 (sync):	0.160
SQLite 2.8.17 (nosync):	0.255
PostgreSQL 8.1.2:	2.635
MySQL 5.0.18 (sync):	1.402
MySQL 5.0.18 (nosync):	1.133
FirebirdSQL 1.5.2:	0.667

Test 20: DROP TABLE

```
DROP TABLE t1;
DROP TABLE t2;
```

SQLite 3.3.3 (sync):	0.138
SQLite 3.3.3 (nosync):	0.392
SQLite 2.8.17 (sync):	0.188
SQLite 2.8.17 (nosync):	0.257
PostgreSQL 8.1.2:	0.229
MySQL 5.0.18 (sync):	0.125
MySQL 5.0.18 (nosync):	0.058
FirebirdSQL 1.5.2:	0.133

附加文件

- speedtest.tcl 9602 bytes added by anonymous on 2006-Feb-07 04:52:02 UTC.
  TCL script used to run the tests
- my.ini 9249 bytes added by anonymous on 2006-Feb-12 02:54:22 UTC.
- postgresql.conf 14045 bytes added by anonymous on 2006-Feb-12 02:54:35 UTC.
- firebird.conf 19729 bytes added by anonymous on 2006-Feb-12 02:54:51 UTC.

SQLite 在 Windows 中的性能调试

为了使 SQLite 在大型数据库和小型用户系统里飞快地运行，我花费了大量时间，我遇到了许多可以加快它们程序运行的方法。

作为侧面注释，这些大型数据库有 300 万行之多，但 SQLite 仍可以很好的处理这么大量的数据。

我也把"包含在内，被动去做"，如果你读的足够多（或者欺骗，只是跳到最后一个对象）。它是不能处理数据的，这是我的错误，所以我有义务来向大家澄清这一点。

这主要发生在 Windows 和 Delphi 环境下，但在其它情景下也许是个优点，请查阅 SQLite 3.1.0:

直接使用 SQLite

我建议你直接尝试使用 SQLite，直接在数据库里执行程序要比使用一些假地址或查询函数好一些。（因为 SQLite 没有这些函数，有能力的程序员通常需要生成复杂的和代价昂贵的 CPU/HDD 编码来模仿 BDE 的功能性）。

我建议你一定到改变你编码的方法，因为从我的经历来看，从 DLL 导出的函数使用起来很简单，可以满足你的需要，虽然它也许和某些人使用 Delphi DBE 的方法有些许的不同。

我自己发现 SqliteWrappers 对我们有很大帮助，它使我们可以很快地把 SQL 送到数据库，并马上得到我们所需要的结果。当然，这决定于你要做什么，和你对你所导出的 DLL 函数有多大信心。

如果你想往数据库里附加，插入，或者更新数据，下面的 SQL 语句：

```
DB.ExecSQL('INSERT OR REPLACE INTO tableName (Field1, Field2, Field 3) ' +
           'VALUES (Value1, Value2, Value3)');
```

可以像下面一样代替全部的旧数据：

```
IF Locate(Field1, Value1) THEN
    UpdateTableFunction() // ...Lots of code here
 ELSE
    InsertIntoTableFunction();  // ...even more code HERE
```

保持代码是精确的并加快速度。记住在更新 SQL 时换 Begin Transactions 和 End Transactions。为了达到很高的效能，在一个事物处理程序中，我们运行上千个 SQL INSERTS。

这不是说使用 componants 不好，使用它们可以大大加快你开发程序的时间。即使你正在使用 componants，你也一定要看看通过 SQL 测试 SQlite(-> ExecSQL)。一个好的和谐的 SQL 语句能详细而精确的满足你的需要，可以比一般的解决办法更快。

2：Indexes 和数据库结构是非常重要的。

当然，一个普通的数据库遵循法则一，但这是一个 SQL 数据库，添加你所需要的索引是非常有必要的，即使是添加一个 DON'T 你不需要的索引也是很重要的。首先用你的判断力来计划一下数据库，使每个事都有一个功能，如果你不需要，则不必要它。

一个操作键有两个索引（例如复合和编入索引）可以减慢数据插入的速度， 当检索时它不会给你带来什么好处，因为 SQLite 将忽略这些。它将以最小的程度增加你的数据库的规模，即使速度对你来说不是什么大问题。

有一些其它的 Wiki 页面，比如 PerformanceTuning 它具体的讲述了这个问题。

3：页面规模也很重要

Windows NTFS 系统默认的群的规格似乎是 4096 字节。把 SQLite 数据库的页面规格也设置成 4096 字节将加快数据库在系统中的速度，当然簇也要相同。

（注意，我认为 Linux 的群的规格是 1024，这是新的 SQLite 数据库所默认的。）

判断你的群的规格的最简单的办法是用碎片整理程序整理驱动器，然后分析。这里有相关的讲解。

为了设置 SQLite 的页面规格，你需要创造一个新的 EMPTY 数据库，然后做

```
PRAGMA page_size=4096;
```

现在创建表格 immediately（如果你想关闭 SQLite 指令行程序，重新打开数据库， 页面规格就被重新设置成 1024）。在创建第一个表格前一定要设置页面的规格。

一旦表格建成，就不能改变页面规格了。

键入：

```
PRAGMA page_size;
```

将告诉你当时正在设置什么。

4：成群的索引

SQLite 不支持成群的索引（简单来说，就是索引使数据库中的数据存入时索引的顺序是什么样，数据就怎样放置）

这意味着，如果你的索引是整数顺序，记录就会把数据库中的数据按整数顺序安排，1 然后 2 然后 3。

你不能生成一个成群的索引，但是你可以按顺序选择你的数据，这样任何历史数据都是整齐的安排好的。当然，当数据库成熟的时候，你丢失的数据，它都可以帮助你。

有人邮寄了下面的例子，这是个好例子，如果你有个 WIBBLE 表格，你想分析它的主要区段，如果有顺序那就好办啦。使用命令行工具，你可以通过下面指导创建一个家的群：

```
create table wibble2 as select * from wibble;
delete from wibble;
insert into wibble select * from wibble2 order by key;
drop table wibble2;
```

5：作为读这篇文章的收获，这里有个不智能的事情需要提醒你。

要**非常非常**谨慎你的数据库的名字，特别是 *extension*

例如，如果你把你所有的数据库命名为 extension.sdb（ SQLite Database，好名字吧？我认为是，无论我怎么选都行）但你会发现 SDB extension 已然和 APPFIX PACKAGES 相关联了。

现在是个非常精彩的部分，APPFIX 是 Windows XP 认可的可执行的程序，它将（强调我所说的 **ADD THE DATABASE TO THE SYSTEM RESTORE FUNCTIONALITY**

这意味着，每次你往数据库里输入任何数据，Windows XP 系统都认为一个可执行程序已经被改变，它将把你整个 800 meg 的数据库恢复到目录。

我认为 DB 或 DAT 是可取的。

SQLite 中如何用触发器执行取消和重做逻辑

这页主要描述一个使用 SQLite 作为主要数据结构的应用程序如何使用触发器去执行取消和重做逻辑。

我的想法是创建一个特殊的表格(例如名为撤销记录),表格保存数据库撤销和重做变化所需的信息。因为数据库中的每个表格都需要参与撤销和重 做，每个 DELETE，INSERT，和 UPDATE 都生成了触发器，DELETE，INSERT，和 UPDATE 可以在撤销日志表格中生成登记项，这个 登记项将撤销操作。撤销表格中的登记项由一般的 SQL 语句组成，为了完成撤销，SQL 语句可以被重新运行。

例如，如果你想在类似下面表格中执行撤销或重：

```
  CREATE TABLE ex1(a,b,c);
```

Then triggers would be created as follows:

```
  CREATE TEMP TRIGGER _ex1_it AFTER INSERT ON ex1 BEGIN
    INSERT INTO undolog VALUES(NULL,'DELETE FROM ex1 WHERE rowid='||new.rowid);
  END;
  CREATE TEMP TRIGGER _ex1_ut AFTER UPDATE ON ex1 BEGIN
    INSERT INTO undolog VALUES(NULL,'UPDATE ex1
       SET a='||quote(old.a)||',b='||quote(old.b)||',c='||quote(old.c)||'
     WHERE rowid='||old.rowid);
  END;
  CREATE TEMP TRIGGER _ex1_dt BEFORE DELETE ON ex1 BEGIN
    INSERT INTO undolog VALUES(NULL,'INSERT INTO ex1(rowid,a,b,c)
      VALUES('||old.rowid||','||quote(old.a)||','||quote(old.b)||
             ','||quote(old.c)||')');
  END;
```

在 ex1 表格中执行每个 INSERT 后， the _ex1_it 触发器生成 DELETE 语句的文本，它将撤销 INSERT 操作。 The _ex1_ut 触发器生成 UPDATE 语句，这语句将取消一个 UPDATE 所产生的作用。_ex1_dt 触发器生成一个语句，这语句将取消一个 DELETE 所具有的作用。

要注意 quote()函数在这些触发器中的使用。quote()函数在 SQLite 中是标准的。它把它的参数转换成一种适合被包含在 SQL 语句中的形式。数字值不改变。单个的 quotes 被加在字符串之前或之后，任何内在的单个 quotes 都被逃逸。quote()函数被加入 SQLite 是 为了执行撤销和重做操作。

当然,你也可以像上面一样用手生成触发器。但这个技术最突出的特点就是这些触发器可以自动生成。

例子中编码的语言是 TCL。使用其它语言也是可以的，但是有可能要做更多工作。记住，这里的编码是 demonstration 技术,不是一个方便的模式,不能自动的为你做每件事。下面所示的 demonstration 编码是源于程序执行过程中所使用的真实的代码。但你在使用它的时候，为了满足你的需要，你需要做些改变。

为了激活撤销和重做的逻辑，激活要参加撤销和重做的所有种类的（表格）undo::activate 指令作为参数。用 undo::deactivate, undo::freeze, and undo::unfreeze 来控制 undo/redo 机制的状态。

The undo::activate 指令在数据库中生成临时的触发器，它记录表格中所有被命名为参数的变化。

在一系列的改变定义了一个单独的undo/redo步骤后,激活undo::barrier指令来定义那步的局限性。在一个交互式的程序中，在做任何改动后，你可以调用 undo::event，undo::barrier 将被自动调用作为一个等待的回调。

当使用者按下 Undo 按钮，激活 undo::undo。当使用者按下 Redo 按钮，激活 undo::redo。

在调用 undo::undo or undo::redo，undo/redo 模块将自动在所有顶级有名字的空间激活程序 status_refresh 和 reload_all。 这些程序应该被定义用来重建画面，或者更新基于 undone/redon 在数据库中变化的程序的状态。

下面的 demonstration 代码包含一个 status_refresh 程序，它激活 Undo and Redo 按钮， 根据没做的和要重做的事来选菜单。你需要重新定义这个程序，用来涉及特定的 Undo 和 Redo 按钮，为你的应用线则进入菜单。这里所提供的执行只是一个例 子。

demonstration 代码假定 SQLite 数据库是用一个名为"db"的句柄打开的。对于一个内置内存的数据库来说，合适的指令应如下：

```
sqlite3 db :memory:
```

这里是 demonstration 代码：

```
# 每件事都在一个私有的有名称的空间里进行
namespace eval ::undo {

# proc:  ::undo::activate TABLE ...
# title: Start up the undo/redo system
#
# 参数应是在一个或多个数据库表格（在数据库中和句柄"db"相关联）
# 它们的变化被记录下来是为了撤销和重做的目的。
proc activate {args} {
  variable _undo
  if {$_undo(active)} return
  eval _create_triggers db $args
  set _undo(undostack) {}
  set _undo(redostack) {}
  set _undo(active) 1
  set _undo(freeze) -1
  _start_interval
}

# proc:  ::undo::deactivate
# title: Halt the undo/redo system and delete the undo/redo stacks
#
proc deactivate {} {
```

```
  variable _undo
  if {!$_undo(active)} return
  _drop_triggers db
  set _undo(undostack) {}
  set _undo(redostack) {}
  set _undo(active) 0
  set _undo(freeze) -1
}

# proc:  ::undo::freeze
# title: Stop accepting database changes into the undo stack
#
# 当这个例行程序被启用直到下一步被解冻前，新的数据库的变化将不被记录在撤销存储栈。
#
proc freeze {} {
  variable _undo
  if {![info exists _undo(freeze)]} return
  if {$_undo(freeze)>=0} {error "recursive call to ::undo::freeze"}
  set _undo(freeze) [db one {SELECT coalesce(max(seq),0) FROM undolog}]
}

# proc:  ::undo::unfreeze
# title: Begin accepting undo actions again.
#
proc unfreeze {} {
  variable _undo
  if {![info exists _undo(freeze)]} return
  if {$_undo(freeze)<0} {error "called ::undo::unfreeze while not frozen"}
  db eval "DELETE FROM undolog WHERE seq>$_undo(freeze)"
  set _undo(freeze) -1
}

# proc:  ::undo::event
# title: Something undoable has happened
#
# 当一个不可撤销的操作发生的时候调用这个程序。在下一个空闲时刻之前激活::undo::barrier。
#
proc event {} {
  variable _undo
  if {$_undo(pending)==""} {
    set _undo(pending) [after idle ::undo::barrier]
  }
}

# proc:  ::undo::barrier
# title: Create an undo barrier right now.
#
```

```
proc barrier {} {
  variable _undo
  catch {after cancel $_undo(pending)}
  set _undo(pending) {}
  if {!$_undo(active)} {
    refresh
    return
  }
  set end [db one {SELECT coalesce(max(seq),0) FROM undolog}]
  if {$_undo(freeze)>=0 && $end>$_undo(freeze)} {set end $_undo(freeze)}
  set begin $_undo(firstlog)
  _start_interval
  if {$begin==$_undo(firstlog)} {
    refresh
    return
  }
  lappend _undo(undostack) [list $begin $end]
  set _undo(redostack) {}
  refresh
}

# proc:  ::undo::undo
# title: Do a single step of undo
#
proc undo {} {
  _step undostack redostack
}

# proc:  ::undo::redo
# title: Redo a single step
#
proc redo {} {
  _step redostack undostack
}

# proc:   ::undo::refresh
# title:  Update the status of controls after a database change
#
# 基于现行数据库的状态，为了合理的进行控制，撤销模块在任何撤销和重做后调用这个例行程序。这个模块通过激活所有顶级有名称的空间中的 status_refresh 模块来工作。
#
proc refresh {} {
  set body {}
  foreach ns [namespace children ::] {
    if {[info proc ${ns}::status_refresh]==""} continue
    append body ${ns}::status_refresh\n
  }
```

```
  proc ::undo::refresh {} $body
  refresh
}

# proc:   ::undo::reload_all
# title:  Redraw everything based on the current database
#
# 为了使屏幕在数据库内容的基础上被完全重新绘图，撤销模块在任何的撤销和重做后都调用这个
# 例行程序。通过在每个顶级的有名称的空间调用"reload" 模块而不是::undo 来完成这个程序。
proc reload_all {} {
  set body {}
  foreach ns [namespace children ::] {
    if {[info proc ${ns}::reload]==""} continue
    append body ${ns}::reload\n
  }
  proc ::undo::reload_all {} $body
  reload_all
}
################################################################################
# 这个模块的公共接口程序在上面。例行程序和变量静态追踪（名字以"_"开头的）是这个模块私有的。
################################################################################

# state information
#
set _undo(active) 0
set _undo(undostack) {}
set _undo(redostack) {}
set _undo(pending) {}
set _undo(firstlog) 1
set _undo(startstate) {}


# proc:  ::undo::status_refresh
# title: Enable and/or disable menu options a buttons
#
proc status_refresh {} {
  variable _undo
  if {!$_undo(active) || [llength $_undo(undostack)]==0} {
    .mb.edit entryconfig Undo -state disabled
    .bb.undo config -state disabled
  } else {
    .mb.edit entryconfig Undo -state normal
    .bb.undo config -state normal
  }
  if {!$_undo(active) || [llength $_undo(redostack)]==0} {
    .mb.edit entryconfig Redo -state disabled
    .bb.redo config -state disabled
```

```
  } else {
    .mb.edit entryconfig Redo -state normal
    .bb.redo config -state normal
  }
}

# xproc:   ::undo::_create_triggers DB TABLE1 TABLE2 ...
# title:   Create change recording triggers for all tables listed
#
# 在数据库中创建一个名为"undolog"的临时表格。创建可以激发任何 insert, delete, or update of TABLE1, TABLE2, .... 的触发器。
# 当这些触发器激发的时候，insert records in 在未做日志中插入记录，这些未做日志中包含 SQL 语句的文本，这些语句将撤销 insert， delete，或 update。
#
proc _create_triggers {db args} {
  catch {$db eval {DROP TABLE undolog}}
  $db eval {CREATE TEMP TABLE undolog(seq integer primary key, sql text)}
  foreach tbl $args {
    set collist [$db eval "pragma table_info($tbl)"]
    set sql "CREATE TEMP TRIGGER _${tbl}_it AFTER INSERT ON $tbl BEGIN\n"
    append sql "   INSERT INTO undolog VALUES(NULL,"
    append sql "'DELETE FROM $tbl WHERE rowid='||new.rowid);\nEND;\n"

    append sql "CREATE TEMP TRIGGER _${tbl}_ut AFTER UPDATE ON $tbl BEGIN\n"
    append sql "   INSERT INTO undolog VALUES(NULL,"
    append sql "'UPDATE $tbl "
    set sep "SET "
    foreach {x1 name x2 x3 x4 x5} $collist {
      append sql "$sep$name='||quote(old.$name)||'"
      set sep ","
    }
    append sql " WHERE rowid='||old.rowid);\nEND;\n"

    append sql "CREATE TEMP TRIGGER _${tbl}_dt BEFORE DELETE ON $tbl BEGIN\n"
    append sql "   INSERT INTO undolog VALUES(NULL,"
    append sql "'INSERT INTO ${tbl}(rowid"
    foreach {x1 name x2 x3 x4 x5} $collist {append sql ,$name}
    append sql ") VALUES('||old.rowid||'"
    foreach {x1 name x2 x3 x4 x5} $collist {append sql ,'||quote(old.$name)||'}
    append sql ")');\nEND;\n"

    $db eval $sql
  }
}

# xproc:   ::undo::_drop_triggers DB
# title:   Drop all of the triggers that _create_triggers created
```

```
#
proc _drop_triggers {db} {
  set tlist [$db eval {SELECT name FROM sqlite_temp_master
                WHERE type='trigger'}]
  foreach trigger $tlist {
    if {![regexp {^_.*_(i|u|d)t$} $trigger]} continue
    $db eval "DROP TRIGGER $trigger;"
  }
  catch {$db eval {DROP TABLE undolog}}
}

# xproc: ::undo::_start_interval
# title: Record the starting conditions of an undo interval
#
proc _start_interval {} {
  variable _undo
  set _undo(firstlog) [db one {SELECT coalesce(max(seq),0)+1 FROM undolog}]
}

# xproc: ::undo::_step V1 V2
# title: Do a single step of undo or redo
#
# For an undo V1=="undostack" and V2=="redostack".   For a redo,
# V1=="redostack" and V2=="undostack".
#
proc _step {v1 v2} {
  variable _undo
  set op [lindex $_undo($v1) end]
  set _undo($v1) [lrange $_undo($v1) 0 end-1]
  foreach {begin end} $op break
  db eval BEGIN
  set q1 "SELECT sql FROM undolog WHERE seq>=$begin AND seq<=$end
       ORDER BY seq DESC"
  set sqllist [db eval $q1]
  db eval "DELETE FROM undolog WHERE seq>=$begin AND seq<=$end"
  set _undo(firstlog) [db one {SELECT coalesce(max(seq),0)+1 FROM undolog}]
  foreach sql $sqllist {
    db eval $sql
  }
  db eval COMMIT
  reload_all

  set end [db one {SELECT coalesce(max(seq),0) FROM undolog}]
  set begin $_undo(firstlog)
  lappend _undo($v2) [list $begin $end]
  _start_interval
  refresh
```

```
}


# End of the ::undo namespace
}
```

SQLite3 C/C++ 开发接口简介（API 函数）

1.0 总览

SQLite3 是 SQLite 一个全新的版本,它虽然是在 SQLite 2.8.13 的代码基础之上开发的,但是使用了和之前的版本不兼容的数据库格式和 API. SQLite3 是为了满足以下的需求而开发的:

- 支持 UTF-16 编码.
- 用户自定义的文本排序方法.
- 可以对 BLOBs 字段建立索引.

因此为了支持这些特性我改变了数据库的格式,建立了一个与之前版本不兼容的 3.0 版. 至于其他的兼容性的改变,例如全新的 API 等等,都将在理论介绍之后向你说明,这样可以使你最快的一次性摆脱兼容性问题.

3.0 版的和 2.X 版的 API 非常相似,但是有一些重要的改变需要注意. 所有 API 接口函数和数据结构的前缀都由"sqlite_"改为了"sqlite3_". 这是为了避免同时使用 SQLite 2.X 和 SQLite 3.0 这两个版本的时候发生链接冲突.

由于对于 C 语言应该用什么数据类型来存放 UTF-16 编码的字符串并没有一致的规范. 因此 SQLite 使用了普通的 void* 类型来指向 UTF-16 编码的字符串. 客户端使用过程中可以把 void*映射成适合他们的系统的任何数据类型.

2.0 C/C++ 接口

SQLite 3.0 一共有 83 个 API 函数,此外还有一些数据结构和预定义(#defines). (完整的 API 介绍请参看另一份文档.) 不过你们可以放心,这些接口使用起来不会像它的数量所暗示的那么复杂. 最简单的程序仍然使用三个函数就可以完成: sqlite3_open(), sqlite3_exec(), 和 sqlite3_close(). 要是想更好的控制数据库引擎的执行,可以使用提供的 sqlite3_prepare()函数把 SQL 语句编译成字节码,然后在使用 sqlite3_step()函数来执行编译后的字节码. 以 sqlite3_column_开头的一组 API 函数用来获取查询结果集中的信息. 许多接口函数都是成对出现的,同时有 UTF-8 和 UTF-16 两个版本. 并且提供了一组函数用来执行用户自定义的 SQL 函数和文本排序函数.

2.1 如何打开关闭数据库

```
typedef struct sqlite3 sqlite3;
int sqlite3_open(const char*, sqlite3**);
int sqlite3_open16(const void*, sqlite3**);
int sqlite3_close(sqlite3*);
const char *sqlite3_errmsg(sqlite3*);
const void *sqlite3_errmsg16(sqlite3*);
int sqlite3_errcode(sqlite3*);
```

sqlite3_open() 函数返回一个整数错误代码,而不是像第二版中一样返回一个指向 sqlite3 结构体的指针. sqlite3_open() 和 sqlite3_open16() 的不同之处在于 sqlite3_open16() 使用 UTF-16 编码(使用本地主机字节顺序)传递数据库文件名. 如果要创建新数据库, sqlite3_open16() 将内部文本转换为 UTF-16 编码, 反之 sqlite3_open() 将文本转换为 UTF-8 编码.

打开或者创建数据库的命令会被缓存,直到这个数据库真正被调用的时候才会被执行. 而且允许使用 PRAGMA 声明来设置如本地文本编码或默认内存页面大小等选项和参数.

sqlite3_errcode() 通常用来获取最近调用的 API 接口返回的错误代码. sqlite3_errmsg() 则用来得到这些错误代码所对应的文字说明. 这些错误信息将以 UTF-8 的编码返回,并且在下一次调用任何 SQLite API 函数的时候被清除. sqlite3_errmsg16() 和 sqlite3_errmsg() 大体上相同,除了返回的错误信息将以 UTF-16 本机字节顺序编码.

SQLite3 的错误代码相比 SQLite2 没有任何的改变,它们分别是:

```
#define SQLITE_OK           0   /* Successful result */
#define SQLITE_ERROR        1   /* SQL error or missing database */
#define SQLITE_INTERNAL     2   /* An internal logic error in SQLite */
#define SQLITE_PERM         3   /* Access permission denied */
#define SQLITE_ABORT        4   /* Callback routine requested an abort */
#define SQLITE_BUSY         5   /* The database file is locked */
#define SQLITE_LOCKED       6   /* A table in the database is locked */
#define SQLITE_NOMEM        7   /* A malloc() failed */
#define SQLITE_READONLY     8   /* Attempt to write a readonly database */
#define SQLITE_INTERRUPT    9   /* Operation terminated by sqlite_interrupt() */
#define SQLITE_IOERR       10   /* Some kind of disk I/O error occurred */
#define SQLITE_CORRUPT     11   /* The database disk image is malformed */
#define SQLITE_NOTFOUND    12   /* (Internal Only) Table or record not found */
#define SQLITE_FULL        13   /* Insertion failed because database is full */
#define SQLITE_CANTOPEN    14   /* Unable to open the database file */
#define SQLITE_PROTOCOL    15   /* Database lock protocol error */
#define SQLITE_EMPTY       16   /* (Internal Only) Database table is empty */
#define SQLITE_SCHEMA      17   /* The database schema changed */
#define SQLITE_TOOBIG      18   /* Too much data for one row of a table */
#define SQLITE_CONSTRAINT  19   /* Abort due to contraint violation */
#define SQLITE_MISMATCH    20   /* Data type mismatch */
#define SQLITE_MISUSE      21   /* Library used incorrectly */
#define SQLITE_NOLFS       22   /* Uses OS features not supported on host */
#define SQLITE_AUTH        23   /* Authorization denied */
#define SQLITE_ROW         100  /* sqlite_step() has another row ready */
#define SQLITE_DONE        101  /* sqlite_step() has finished executing */
```

2.2 执行 SQL 语句

```
typedef int (*sqlite_callback)(void*,int,char**, char**);
int sqlite3_exec(sqlite3*, const char *sql, sqlite_callback, void*, char**);
```

sqlite3_exec 函数依然像它在 SQLite2 中一样承担着很多的工作. 该函数的第二个参数中可以编译和执行零个或多个 SQL 语句. 查询的结果返回给回调函数. 更多地信息可以查看 API 参考.

在 SQLite3 里,sqlite3_exec 一般是被准备 SQL 语句接口封装起来使用的.

```
typedef struct sqlite3_stmt sqlite3_stmt;
int sqlite3_prepare(sqlite3*, const char*, int, sqlite3_stmt**, const char**);
int sqlite3_prepare16(sqlite3*, const void*, int, sqlite3_stmt**, const void**);
int sqlite3_finalize(sqlite3_stmt*);
int sqlite3_reset(sqlite3_stmt*);
```

sqlite3_prepare 接口把一条 SQL 语句编译成字节码留给后面的执行函数. 使用该接口访问数据库是当前比较好的的一种方法.

sqlite3_prepare() 处理的 SQL 语句应该是 UTF-8 编码的. 而 sqlite3_prepare16() 则要求是 UTF-16 编码的. 输入的参数中只有第一个 SQL 语句会被编译. 第四个参数则用来指向输入参数中下一个需要编译的 SQL 语句存放的 SQLite statement 对象的指针, 任何时候如果调用 sqlite3_finalize() 将销毁一个准备好的 SQL 声明. 在数据库关闭之前,所有准备好的声明都必须被释放销毁. sqlite3_reset() 函数用来重置一个 SQL 声明的状态，使得它可以被再次执行.

SQL 声明可以包含一些型如"?" 或 "?nnn" 或 ":aaa"的标记， 其中"nnn" 是一个整数，"aaa" 是一个字符串. 这些标记代表一些不确定的字符值（或者说是通配符），可以在后面用 sqlite3_bind 接口来填充这些值. 每一个通配符都被分配了一个编号（由它在 SQL 声明中的位置决定，从 1 开始），此外也可以用 "nnn" 来表示 "?nnn" 这种情况. 允许相同的通配符在同一个 SQL 声明中出现多次, 在这种情况下所有相同的通配符都会被替换成相同的值. 没有被绑定的通配符将自动取 NULL 值.

```
int sqlite3_bind_blob(sqlite3_stmt*, int, const void*, int n, void(*)(void*));
int sqlite3_bind_double(sqlite3_stmt*, int, double);
int sqlite3_bind_int(sqlite3_stmt*, int, int);
int sqlite3_bind_int64(sqlite3_stmt*, int, long long int);
int sqlite3_bind_null(sqlite3_stmt*, int);
int sqlite3_bind_text(sqlite3_stmt*, int, const char*, int n, void(*)(void*));
int sqlite3_bind_text16(sqlite3_stmt*, int, const void*, int n, void(*)(void*));
int sqlite3_bind_value(sqlite3_stmt*, int, const sqlite3_value*);
```

以上是 sqlite3_bind 所包含的全部接口，它们是用来给 SQL 声明中的通配符赋值的. 没有绑定的通配符则被认为是空值. 绑定上的值不会被 sqlite3_reset()函数重置. 但是在调用了 sqlite3_reset()之后所有的通配符都可以被重新赋值.

在 SQL 声明准备好之后(其中绑定的步骤是可选的), 需要调用以下的方法来执行:

```
int sqlite3_step(sqlite3_stmt*);
```

如果 SQL 返回了一个单行结果集，sqlite3_step() 函数将返回 SQLITE_ROW ， 如果 SQL 语句执行成功或者正常将返回 SQLITE_DONE ， 否则将返回错误代码. 如果不能打开数据库文件则会返回 SQLITE_BUSY . 如果函数的返回值是 SQLITE_ROW, 那么下边的这些方法可以用来获得记录集行中的数据:

```
const void *sqlite3_column_blob(sqlite3_stmt*, int iCol);
int sqlite3_column_bytes(sqlite3_stmt*, int iCol);
```

```
int sqlite3_column_bytes16(sqlite3_stmt*, int iCol);
int sqlite3_column_count(sqlite3_stmt*);
const char *sqlite3_column_decltype(sqlite3_stmt *, int iCol);
const void *sqlite3_column_decltype16(sqlite3_stmt *, int iCol);
double sqlite3_column_double(sqlite3_stmt*, int iCol);
int sqlite3_column_int(sqlite3_stmt*, int iCol);
long long int sqlite3_column_int64(sqlite3_stmt*, int iCol);
const char *sqlite3_column_name(sqlite3_stmt*, int iCol);
const void *sqlite3_column_name16(sqlite3_stmt*, int iCol);
const unsigned char *sqlite3_column_text(sqlite3_stmt*, int iCol);
const void *sqlite3_column_text16(sqlite3_stmt*, int iCol);
int sqlite3_column_type(sqlite3_stmt*, int iCol);
```

sqlite3_column_count()函数返回结果集中包含的列数. sqlite3_column_count() 可以在执行了 sqlite3_prepare()之后的任何时刻调用. sqlite3_data_count()除了必需要在 sqlite3_step()之后调用之外,其他跟 sqlite3_column_count()大同小异. 如果调用 sqlite3_step() 返回值是 SQLITE_DONE 或者一个错误代码, 则此时调用 sqlite3_data_count()将返回 0 , 然而 sqlite3_column_count() 仍然会返回结果集中包含的列数.

返回的记录集通过使用其它的几个 sqlite3_column_***() 函数来提取, 所有的这些函数都把列的编号作为第二个参数. 列编号从左到右以零起始. 请注意它和之前那些从 1 起始的参数的不同.

sqlite3_column_type()函数返回第 N 列的值的数据类型. 具体的返回值如下:

```
#define SQLITE_INTEGER  1
#define SQLITE_FLOAT    2
#define SQLITE_TEXT     3
#define SQLITE_BLOB     4
#define SQLITE_NULL     5
```

sqlite3_column_decltype() 则用来返回该列在 CREATE TABLE 语句中声明的类型. 它可以用在当返回类型是空字符串的时候. sqlite3_column_name() 返回第 N 列的字段名. sqlite3_column_bytes() 用来返回 UTF-8 编码的 BLOBs 列的字节数或者 TEXT 字符串的字节数. sqlite3_column_bytes16() 对于 BLOBs 列返回同样的结果，但是对于 TEXT 字符串则按 UTF-16 的编码来计算字节数. sqlite3_column_blob() 返回 BLOB 数据. sqlite3_column_text() 返回 UTF-8 编码的 TEXT 数据. sqlite3_column_text16() 返回 UTF-16 编码的 TEXT 数据. sqlite3_column_int() 以本地主机的整数格式返回一个整数值. sqlite3_column_int64() 返回一个 64 位的整数. 最后, sqlite3_column_double() 返回浮点数.

不一定非要按照 sqlite3_column_type()接口返回的数据类型来获取数据. 数据类型不同时软件将自动转换.

2.3 用户自定义函数

可以使用以下的方法来创建用户自定义的 SQL 函数:

```
typedef struct sqlite3_value sqlite3_value;
int sqlite3_create_function(
    sqlite3 *,
    const char *zFunctionName,
    int nArg,
```

```
    int eTextRep,
    void*,
    void (*xFunc)(sqlite3_context*,int,sqlite3_value**),
    void (*xStep)(sqlite3_context*,int,sqlite3_value**),
    void (*xFinal)(sqlite3_context*)
);
int sqlite3_create_function16(
    sqlite3*,
    const void *zFunctionName,
    int nArg,
    int eTextRep,
    void*,
    void (*xFunc)(sqlite3_context*,int,sqlite3_value**),
    void (*xStep)(sqlite3_context*,int,sqlite3_value**),
    void (*xFinal)(sqlite3_context*)
);
#define SQLITE_UTF8         1
#define SQLITE_UTF16       2
#define SQLITE_UTF16BE   3
#define SQLITE_UTF16LE   4
#define SQLITE_ANY           5
```

nArg 参数用来表明自定义函数的参数个数. 如果参数值为 0，则表示接受任意个数的参数. 用 eTextRep 参数来表明传入参数的编码形式. 参数值可以是上面的五种预定义值. SQLite3 允许同一个自定义函数有多种不同的编码参数的版本. 数据库引擎会自动选择转换参数编码个数最少的版本使用.

普通的函数只需要设置 xFunc 参数，而把 xStep 和 xFinal 设为 NULL. 聚合函数则需要设置 xStep 和 xFinal 参数，然后把 xFunc 设为 NULL. 该方法和使用 sqlite3_create_aggregate() API 一样.

sqlite3_create_function16()和 sqlite_create_function()的不同就在于自定义的函数名一个要求是 UTF-16 编码，而另一个则要求是 UTF-8.

请注意自定函数的参数目前使用了 sqlite3_value 结构体指针替代了 SQLite version 2.X 中的字符串指针. 下面的函数用来从 sqlite3_value 结构体中提取数据:

```
const void *sqlite3_value_blob(sqlite3_value*);
int sqlite3_value_bytes(sqlite3_value*);
int sqlite3_value_bytes16(sqlite3_value*);
double sqlite3_value_double(sqlite3_value*);
int sqlite3_value_int(sqlite3_value*);
long long int sqlite3_value_int64(sqlite3_value*);
const unsigned char *sqlite3_value_text(sqlite3_value*);
const void *sqlite3_value_text16(sqlite3_value*);
int sqlite3_value_type(sqlite3_value*);
```

上面的函数调用以下的 API 来获得上下文内容和返回结果:

```
void *sqlite3_aggregate_context(sqlite3_context*, int nbyte);
void *sqlite3_user_data(sqlite3_context*);
void sqlite3_result_blob(sqlite3_context*, const void*, int n, void(*)(void*));
void sqlite3_result_double(sqlite3_context*, double);
void sqlite3_result_error(sqlite3_context*, const char*, int);
void sqlite3_result_error16(sqlite3_context*, const void*, int);
void sqlite3_result_int(sqlite3_context*, int);
void sqlite3_result_int64(sqlite3_context*, long long int);
void sqlite3_result_null(sqlite3_context*);
void sqlite3_result_text(sqlite3_context*, const char*, int n, void(*)(void*));
void sqlite3_result_text16(sqlite3_context*, const void*, int n, void(*)(void*));
void sqlite3_result_value(sqlite3_context*, sqlite3_value*);
void *sqlite3_get_auxdata(sqlite3_context*, int);
void sqlite3_set_auxdata(sqlite3_context*, int, void*, void (*)(void*));
```

2.4 用户自定义排序规则

下面的函数用来实现用户自定义的排序规则:

```
sqlite3_create_collation(sqlite3*, const char *zName, int eTextRep, void*,
      int(*xCompare)(void*,int,const void*,int,const void*));
sqlite3_create_collation16(sqlite3*, const void *zName, int eTextRep, void*,
      int(*xCompare)(void*,int,const void*,int,const void*));
sqlite3_collation_needed(sqlite3*, void*,
      void(*)(void*,sqlite3*,int eTextRep,const char*));
sqlite3_collation_needed16(sqlite3*, void*,
      void(*)(void*,sqlite3*,int eTextRep,const void*));
```

sqlite3_create_collation() 函数用来声明一个排序序列和实现它的比较函数. 比较函数只能用来做文本的比较. eTextRep 参数可以取如下的预定义值 SQLITE_UTF8, SQLITE_UTF16LE, SQLITE_UTF16BE, SQLITE_ANY，用来表示比较函数所处理的文本的编码方式. 同一个自定义的排序规则的同一个比较函数可以有 UTF-8, UTF-16LE 和 UTF-16BE 等多个编码的版本. sqlite3_create_collation16()和 sqlite3_create_collation() 的区别也仅仅在于排序名称的编码是 UTF-16 还是 UTF-8.

可以使用 sqlite3_collation_needed() 函数来注册一个回调函数，当数据库引擎遇到未知的排序规则时会自动调用该函数. 在回调函数中可以查找一个相似的比较函数，并激活相应的 sqlite_3_create_collation()函数. 回调函数的第四个参数是排序规则的名称，同样 sqlite3_collation_needed 采用 UTF-8 编码. sqlite3_collation_need16() 采用 UTF-16 编码.

如何在 VS 2003 下编译 SQLite

下载

下载 sqlite_source.zip 文件并解压缩. 注意不要下载 .tar.gz 格式压缩的文件，因为它缺少 Windows 平台所需要的预编译头文件.

创建一个 DLL 工程

1. 文件 > 新建 > 工程

2. 选择创建一个 Visual C++ Win32 工程.

3. 选择 "Win32 Project" 工程模版.

4. 输入工程的名字，然后点击完成.

5. 在工程的向导页中选择工程的类型为 "Win32 DLL", 并且把创建一个空项目的复选框钩上. 单击完成，这样你就创建了一个空的 DLL 工程.

把 SQLite 的源文件添加到工程当中去

7. 工程 > 添加现有项

8. 把你解压出来的除去 tclsqlite.c 和 shell.c 两个文件之外的所有 .c 和 .h 文件添加到工程中去.
注意：如果你添加了 tclsqlite.c 和 shell.c 文件，则你必须要添加预定义宏 NO_TCL:
a) 点击工程 -> 属性, 转到 C/C++ 项然后选择预处理器
b) 在预处理器定义中加上 NO_TCL 并以分号分隔

Make a .DEF file

9. 把源代码包中的 .def 文件放到工程所在的目录中. 该文件在 2.x.x 版的时候叫 sqlite.def. 在 3.0.x 版之后改为 sqlite3.def.

10. 把 sqlite[3].def 添加到工程中.

11. 选择 工程 > 属性中的链接器，然后找到"输入"这一项. 在 "模块定义文件" 中输入 sqlite[3].def. 注意: 你需要在 Debug 和 Release 中都输入该项才行.

注意: 为了生成其它程序链接 sqlite[3].dll 的 lib 文件,你需要添加一个生成后事件. 在工程的属性中找到生成事件这一项，然后在生成后事件中的命令行里加上"LIB /DEF:<path>\sqlite[3].def", 记得这也是要在 debug 和 release 中都要写上的, 命令行中的 <path> 就是 sqlite[3].def 所在的绝对路径.

在编译 3.3.7 版的时候(也许其它的版本也需要), 还需要做一个额外的步骤:
把工程的当前目录添加到项目的附加包含目录当中, 具体的做法是:
在工程 > 属性的 C/C++ 选项中的 "常规", 然后在 "附加包含目录中" 输入 "."(一个点表示当前目录).

12. **编译!**

VS 2003 的工程文件包

如何编译 SQLITE.EXE 命令行程序

如果想编译出 sqlite.exe 命令行程序,则需要做一些设置上的改变. 首先创建工程的时候应该选择 "Win32 控制台程序". 然后在往工程里添加文件的时候把 shell.c 也添加进去. 并且不要在加入 .DEF 文件.

sqlite2 的导出文件.

```
  EXPORTS
   sqlite_open
   sqlite_close
   sqlite_exec
   sqlite_last_insert_rowid
   sqlite_error_string
   sqlite_interrupt
   sqlite_complete
   sqlite_busy_handler
   sqlite_busy_timeout
   sqlite_get_table
   sqlite_free_table
   sqlite_mprintf
   sqlite_vmprintf
   sqlite_exec_printf
   sqlite_exec_vprintf
   sqlite_get_table_printf
   sqlite_get_table_vprintf
   sqlite_freemem
   sqlite_libversion
   sqlite_libencoding
   sqlite_changes
   sqlite_create_function
   sqlite_create_aggregate
   sqlite_function_type
   sqlite_user_data
   sqlite_aggregate_context
   sqlite_aggregate_count
   sqlite_set_result_string
   sqlite_set_result_int
```

sqlite_set_result_double
sqlite_set_result_error
sqliteMalloc
sqliteFree
sqliteRealloc
sqlite_set_authorizer
sqlite_trace
sqlite_compile
sqlite_step
sqlite_finalize
sqlite_progress_handler
sqlite_reset
sqlite_last_statement_changes

SQLite 常见问题解答

Frequently Asked Questions

---

(1) 如何建立自动增长字段?

简短回答: 声明为 INTEGER PRIMARY KEY 的列将会自动增长。

长一点的答案: 如果你声明表的一列为 INTEGER PRIMARY KEY, 那么, 每当你在该列上插入一 NULL 值时, NULL 自动被转换为一个比该列中最大值大 1 的一个整数, 如果表是空的, 将会是 1。(如果是最大可能的主键 9223372036854775807, 那个, 将键值将是随机未使用的数。) 如, 有下列表:

```
CREATE TABLE t1(
  a INTEGER PRIMARY KEY,
  b INTEGER
);
```

在该表上, 下列语句

```
INSERT INTO t1 VALUES(NULL,123);
```

在逻辑上等价于:

```
INSERT INTO t1 VALUES((SELECT max(a) FROM t1)+1,123);
```

有一个新的 API 叫做 sqlite3_last_insert_rowid(), 它将返回最近插入的整数值。

注意该整数会比表中该列上的插入之前的最大值大 1。 该键值在当前的表中是唯一的。但有可能与已从表中删除的值重叠。 要想建立在整个表的生命周期中唯一的键值，需要在 INTEGER PRIMARY KEY 上增加 AUTOINCREMENT 声明。那么，新的键值将会比该表中曾能存在过的最大值大 1。 如果最大可能的整数值在数据表中曾经存在过，INSERT 将会失败， 并返回 SQLITE_FULL 错误代码。

---

(2)SQLite 支持何种数据类型？

参见 http://www.sqlite.org/datatype3.html.

---

(3)SQLite 允许向一个 integer 型字段中插入字符串！

这是一个特性，而不是一个 bug。SQLite 不强制数据类型约束。 任何数据都可以插入任何列。你可以向一个整型列中插入任意长度的字符串， 向布尔型列中插入浮点数，或者向字符型列中插入日期型值。 在 CREATE TABLE 中所指定的数据类型不会限制在该列中插入任何数据。 任何列均可接受任意长度的字符串（只有一种情况除外： 标志为 INTEGER PRIMARY KEY 的列只能存储 64 位整数， 当向这种列中插数据除整数以外的数据时，将会产生错误。

但 SQLite 确实使用声明的列类型来指示你所期望的格式。所以， 例如你向一个整型列中插入字符串时，SQLite 会试图将该字符串转换成一个整数。 如果可以转换，它将插入该整数；否则，将插入字符串。 这种特性有时被称为 类型或列亲和性(type or column affinity).

---

(4)为什么 SQLite 不允许在同一个表不同的两行上使用 0 和 0.0 作主键？

主键必须是数值类型，将主键改为 TEXT 型将不起作用。

每一行必须有一个唯一的主键。对于一个数值型列， SQLite 认为 '0' 和 '0.0' 是相同的， 因为他们在作为整数比较时是相等的(参见上一问题)。 所以，这样值就不唯一了。

---

(5)多个应用程序或一个应用程序的多个实例可以同时访问同一个数据库文件吗？

多个进程可同时打开同一个数据库。多个进程可以同时进行 SELECT 操作，但在任一时刻，只能有一个进程对数据库进行更改。

SQLite 使用读、写锁控制对数据库的访问。（在 Win95/98/ME 等不支持读、 写锁的系统下，使用一个概率性的模拟来代替。）但使用时要注意： 如果数据库文件存放于一个 NFS 文件系统上，这种锁机制可能不能正常工作。 这是因为 fcntl() 文件锁在很多 NFS 上没有正确的实现。 在可能有多个进程同时访问数据库的时候，应该避免将数据库文件放到 NFS 上。 在 Windows 上，Microsoft 的文档中说：如果使用 FAT 文件系统而没有运行 share.exe 守护进程，那么锁可能是不能正常使用的。那些在 Windows 上有很多经验的人告诉我： 对于网络文件，文件锁的实现有好多 Bug，是靠不住的。如果他们说的是对的， 那么在两台或多台 Windows 机器间共享数据库可能会引起不期望的问题。

我们意识到，没有其它*嵌入式的* SQL 数据库引擎能象 SQLite 这样处理如此多的并发。SQLite 允许多个进程同时打开一个数据库， 同时读一个数据库。当有任何进程想要写时，它必须在更新过程中锁住数据库文件。 但那通常只是几毫秒的时间。其它进程只需等待写进程干完活结束。 典型地，其它嵌入式的 SQL 数据库引擎同时只允许一个进程连接到数据库。

但是，Client/Server 数据库引擎（如 PostgreSQL, MySQL, 或 Oracle） 通常支持更高级别的并发，并且允许多个进程同时写同一个数据库。 这种机制在 Client/Server 结构的数据库上是可能的， 因为总是有一个单一的服务器进程很好地控制、协调对数据库的访问。 如果你的应用程序需要很多的并发，那么你应该考虑使用一个 Client/Server 结构的数据库。但经验表明，很多应用程序需要的并发，往往比其设计者所想象的少得多。

当 SQLite 试图访问一个被其它进程锁住的文件时，缺省的行为是返回 SQLITE_BUSY。 可以在 C 代码中使用 sqlite3_busy_handler() 或 sqlite3_busy_timeout() API 函数调整这一行为。

(6)SQLite 线程安全吗？

线程是魔鬼（Threads are evil）。 避免使用它们。

SQLite 是线程安全的。由于很多用户会忽略我们在上一段中给出的建议， 我们做出了这种让步。但是，为了达到线程安全，SQLite 在编译时必须将 SQLITE_THREADSAFE 预处理宏置为 1。在 Windows 和 Linux 上， 已编译的好的二进制发行版中都是这样设置的。 如果不确定你所使用的库是否是线程安全的，可以调用 sqlite3_threadsafe() 接口找出。

在 3.3.1 版本之前，一个 **sqlite3** 结构只能被用于调用 sqlite3_open 创建的同一线程。你不能在一个线程中打开数据库， 然后将数据库句柄传递给另外一个进程使用。 这主要是由于在好多通用的线程实现（如 RedHat9）中的限制引起的（是 Bug 吗？）。 特别的，在有问题的系统上，一个进程创建的 fcntl() 锁无法被其它线程清除或修改。 所以，由于 SQLite 大量使用 fcntl() 锁做并发控制， 如果你在不同的线程间移动数据库连接，就可能会出现严重的问题。

在 3.3.1 版本上，关于在线程间移动数据库连接的限制变得宽松了。 在它及以后的版本中，只要连接没有持有 fcntl() 锁，在线程间移动句柄是安全的。 如果没有未决的事务，并且所有的语句都已执行完毕， 你就可以安全的假定不再持有任何锁。

在 UNIX 中，在执行 fork() 系统调用时不应携带已打开的数据库进入子进程。 那样做将会有问题。

---

(7)在 SQLite 数据库中如何列出所有的表和索引？

如果你运行 **sqlite3** 命令行来访问你的数据库，可以键入 “**.tables**”来获得所有表的列表。或者，你可以输入 “**.schema**” 来看整个数据库模式，包括所有的表的索引。 输入这些命令，后面跟一个 LIKE 模式匹配可以限制显示的表。

在一个 C/C++ 程序中（或者脚本语言使用 Tcl/Ruby/Perl/Python 等） 你可以在一个特殊的名叫 **SQLITE_MASTER** 上执行一个 SELECT 查询以获得所有 表的索引。每一个 SQLite 数据库都有一个叫 SQLITE_MASTER 的表， 它定义数据库的模式。 SQLITE_MASTER 表看起来如下：

```
CREATE TABLE sqlite_master (
```

```
  type TEXT,
  name TEXT,
  tbl_name TEXT,
  rootpage INTEGER,
  sql TEXT
);
```

对于表来说，**type** 字段永远是 **'table'**，**name** 字段永远是表的名字。所以，要获得数据库中所有表的列表， 使用下列 SELECT 语句：

```
SELECT name FROM sqlite_master
WHERE type='table'
ORDER BY name;
```

对于索引，**type** 等于 **'index'**，**name** 则是索引的名字，**tbl_name** 是该索引所属的表的名字。 不管是表还是索引，**sql** 字段是原先用 CREATE TABLE 或 CREATE INDEX 语句创建它们时的命令文本。对于自动创建的索引（用来实现 PRIMARY KEY 或 UNIQUE 约束），**sql** 字段为 NULL。

SQLITE_MASTER 表是只读的。不能对它使用 UPDATE、INSERT 或 DELETE。 它会被 CREATE TABLE、CREATE INDEX、DROP TABLE 和 DROP INDEX 命令自动更新。

临时表不会出现在 SQLITE_MASTER 表中。临时表及其索引和触发器存放在另外一个叫 SQLITE_TEMP_MASTER 的表中。SQLITE_TEMP_MASTER 跟 SQLITE_MASTER 差不多， 但它只是对于创建那些临时表的应用可见。如果要获得所有表的列表， 不管是永久的还是临时的，可以使用类似下面的命令：

```
SELECT name FROM
   (SELECT * FROM sqlite_master UNION ALL
    SELECT * FROM sqlite_temp_master)
WHERE type='table'
ORDER BY name
```

---

(8) SQLite 数据库有已知的大小限制吗？

对有关 SQLite 限制的详细讨论，见 limits.html 。

---

(9) 在 SQLite 中，VARCHAR 字段最长是多少？

SQLite 不强制 VARCHAR 的长度。 你可以在 SQLITE 中声明一个 VARCHAR(10)，SQLite 还是可以很高兴地允许你放入 500 个字符。 并且这 500 个字符是原封不动的，它永远不会被截断。

---

(10) SQLite 支持二进制大对象吗？

SQLite 3.0 及以后版本允许你在任何列中存储 BLOB 数据。 即使该列被声明为其它类型也可以。

---

(11) 在 SQLite 中，如何在一个表上添加或删除一列？

SQLite 有有限地 ALTER TABLE 支持。你可以使用它来在表的末尾增加一列，可更改表的名称。 如果需要对表结构做更复杂的改变，则必须重新建表。 重建时可以先将已存在的数据放到一个临时表中，删除原表， 创建新表，然后将数据从临时表中复制回来。

如，假设有一个 t1 表，其中有 "a", "b", "c" 三列， 如果要删除列 c ，以下过程描述如何做:

```
BEGIN TRANSACTION;
CREATE TEMPORARY TABLE t1_backup(a,b);
INSERT INTO t1_backup SELECT a,b FROM t1;
DROP TABLE t1;
CREATE TABLE t1(a,b);
INSERT INTO t1 SELECT a,b FROM t1_backup;
DROP TABLE t1_backup;
COMMIT;
```

---

(12) 我在数据库中删除了很多数据，但数据库文件没有变小，是 Bug 吗？

不是。当你从 SQLite 数据库中删除数据时， 未用的磁盘空间将会加入一个内部的“自由列表”中。当你下次插入数据时，这部分空间可以重用。磁盘空间不会丢失， 但也不会返还给操作系统。

如果删除了大量数据，而又想缩小数据库文件占用的空间，执行 VACUUM 命令。 VACUUM 将会从头重新组织数据库。这将会使用数据库有一个空的“自由链表”， 数据库文件也会最小。但要注意的是，VACUUM 的执行会需要一些时间 （在 SQLite 开发时，在 Linux 上，大约每 M 字节需要半秒种），并且， 执行过程中需要原数据库文件至多两倍的临时磁盘空间。

对于 SQLite 3.1 版本，一个 auto-vacumm 模式可以替代 VACUUM 命令。 可以使用 auto_vacuum pragma 打开。

---

(13) 我可以在商业产品中使用 SQLite 而不需支付许可费用吗？

是的。SQLite 在 public domain。对代码的任何部分没有任何所有权声明。你可以使用它做任何事。

---

(14) 如何在字符串中使用单引号(')？

SQL 标准规定，在字符串中，单引号需要使用逃逸字符，即在一行中使用两个单引号。在这方面 SQL 用起来类似 Pascal 语言。 SQLite 尊循标准。如：

```
INSERT INTO xyz VALUES('5 O''clock');
```

---

(15) SQLITE_SCHEMA error 是什么错误？为什么会出现该错误？

当一个准备好的(prepared)SQL 语句不再有效或者无法执行时，将返回一个 SQLITE_SCHEMA 错误。发生该错误时，SQL 语句必须使用 sqlite3_prepare() API 来重新编译. 在 SQLite 3 中，一个 SQLITE_SCHEMA 错误只会发生在用 sqlite3_prepare()/sqlite3_step()/sqlite3_finalize() API 执行 SQL 时。而不会发生在使用 sqlite3_exec()时。 在版本 2 中不是这样。

准备好的语句失效的最通常原因是：在语句准备好后， 数据库的模式又被修改了。另外的原因会发生在：

- 数据库离线：DETACHed.
- 数据库被 VACUUMed
- 一个用户存储过程定义被删除或改变。
- 一个 collation 序列定义被删除或改变。
- 认证函数被改变。

在所有情况下，解决方法是重新编译并执行该 SQL 语句。 因为一个已准备好的语句可以由于其它进程改变数据库模式而失效， 所有使用 sqlite3_prepare()/sqlite3_step()/sqlite3_finalize() API 的代码都应准备处理 SQLITE_SCHEMA 错误。下面给出一个例子：

```
int rc;
sqlite3_stmt *pStmt;
char zSql[] = "SELECT .....";

do {
  /* Compile the statement from SQL. Assume success. */
  sqlite3_prepare(pDb, zSql, -1, &pStmt, 0);

  while( SQLITE_ROW==sqlite3_step(pStmt) ){
    /* Do something with the row of available data */
  }

  /* Finalize the statement. If an SQLITE_SCHEMA error has
  ** occured, then the above call to sqlite3_step() will have
  ** returned SQLITE_ERROR. sqlite3_finalize() will return
  ** SQLITE_SCHEMA. In this case the loop will execute again.
  */
  rc = sqlite3_finalize(pStmt);
} while( rc==SQLITE_SCHEMA );
```

---

(16) 为什么 ROUND(9.95,1) 返回 9.9 而不是 10.0？ 9.95 不应该圆整（四舍五入）吗？

SQLite 使用二进制算术，在二进制中， 无法用有限的二进制位数表示 9.95 。使用 64-bit IEEE 浮点 （SQLite 就是使用这个）最接近 9.95 的二进制表示是 9.94999999999999928945726423989981412887573242187 5。 所在，当你输入 9.95 时，SQLite 实际上以为是上面的数字， 在四舍五入时会舍去。

这种问题在使用二进制浮点数的任何时候都会出现。 通常的规则是记住很多有限的十进制小数都没有一个对应的二进制表示。 所以，它们只能使用最接近的二进制数。它们通常非常接近， 但也会有些微小的不同，有些时候也会导致你所期望的不同的结果。

SQLite 的原子提交原理

摘要：

本文源自：http://www.sqlite.org/atomiccommit.html，*2007/11/28 的版本*

本人正在做一个项目，在项目中定义了自己的文件格式，为了做到停电或程序崩溃不损坏这些文件原有的数据，故针对操作的原子性做一些思考，后来看到 sqlite 的这篇文章，与自己的实现方式作了一些对比。故顺手在研究此文章的时候将大意译成了中文。毕竟只是一时顺手之作，应该存在不少的误读与错误，请多多包涵，此文章的原始地址在 http://chensheng.net/p/sqlite/auto_commit_zh_cn.html，本文可以转载，但请保留出处，以便他人能够方便找到我在修改此文可能的错误之后重新发布的版本。如果发现错误请发 mail 给我 erehw#163.com。

本文描述了 sqlite 为保证数据库文件不被损坏而采取的种种手段，对于一些小型应用值得借鉴。其实在我看来，我自己实现这种方式似乎都有些不必要，也可以直接利用 sqlite 或者 berkeley db 即可。

2008-1-29 于杭州，时日江南一片暴雪，众多机场车站都处于凝滞状态。感谢众多在此次雪灾之中作出贡献的人们。

1.0 简介

“原子提交”是 SQLite 这种支持事务的数据库的一个重要特性。原子提交意味着某个事务中数据库的变化会完整完成或者根本不完成。原子提交意味着不同的写入分别写入到数据库的不同部分就似同时发生在同一个时间点一样。

实际上硬件会连续的写到海量存储器中，只是写一个扇区所用的时间非常少。所以，同时或瞬间写入到数据文件的不同部分成为可能。SQLite 的原子提交逻辑会使得一个事务中的变化就象同时发生的一样。

事务的原子是 SQLite 的重要特性，即使事务由于操作系统出错或掉电发生中断也能保持其原子性。

本文描述了 SQLite 实现原子操作的技术。

2.0 硬件设定

在这往篇文章中，我们把海量存储特指定为“硬盘”，即使它可能是 flash memory.

我们假定硬盘是以扇区为单位进行整块写入的。我们不能单独修改硬盘的小于扇区的部分。如果需要修改硬盘小于扇区的部分，你也必须整个读入此部分所在扇区，对此扇区进行修改，然后将整个扇区写回硬盘。

在传统的 Spinning disk 中,扇区是最小的传输单元---无论是读还是写。然而，对于 flash memory，每次读的最小数目通常都远小于最小写操作数目。SQLite 只关心写操作的最小数目，因此在本文中，当我们说“扇区”的时候，就

是指单次写入的最少字节总数。

SQLite 3.3.14 以前的版本，我们假定任何情况下，一个扇区是 512 字节。这是一个编译时设定的值，而且从没针对更大数进行测试过。当磁盘驱动器内部使用的是以 512 字节为单位的扇区时，512 字节的假定显得非常合理。然而，现在的磁盘都已经发展到 4k 每扇区了。同样， flash memory 的扇区大小通常都大于 512 字节。因此，从 3.3.14 版本开始，SQLite 有一个函数去获取文件系统的扇区真实大小。在当前的实现中(3.5.0)，这个函数仍然简单的返回 512—因为在 win32 及 unix 环境下，没有标准方法去取得扇区的真实大小。但这个方法在人们需要针对他们应用进行调整的时候是非常有意义的。

SQLite 并不假 定扇区写操作是原子的。然而，我们假定扇区写操作是线性的。所谓“线性”是指，当开始扇区写操作时，硬件从前一个扇区的结束点开始，然后一字节一字节的写 入，直到此扇区的结束点。这个写操作可能是从尾向头写，也可能是从头向尾写。如果在一个扇区写入操作时发生掉电故障，这个扇区可能会一部分已经修改完成， 还有一部分还没来得及进行修改。SQLite 的关键设定是这样的：如果一个扇区的任何部分发生修改，那么不是它开始的部分发了变化，就是它结束部分发生了变化。所以硬件从来都不会从一个扇区的中间部分开始写入。我们不知道这个假定是否总是真实的，但无论如何，看起来还是蛮合理的。

上段中，SQLite 并没有假定扇区写操作是原子的。在 SQLite3.5.0 版本中，新增了一个 VFS（虚拟文件系统）接口。SQLite 通过 VFS 与实际的文件系统进行交互。SQLite 已经为 windows 及 unix 编写了一个缺省的 VFS 实现。并且可以让用户在运行时实现一个自定义的 VFS 实现。VFS 接口有一个方法叫： xDeviceCharacteristics.此方法读取实际的文件系统各种特性。xDeviceCharacteristics 方法可以指明扇区写操作是原子的，如果确实指定扇区写是原子的，SQLite 是不会放过这等好处的。但在 windows 及 unix 中，缺省 xDeviceCharacteristics 的实现并没有指明扇区写是原子的，所以这些优化通常会忽略掉了。

SQLite 假定操作系统会对写进行缓冲，因此写入请求返回时，有可能数据还没有真实的写入到存储中。SQLite 同时还假定这种写操作会被操作系统记录。因此， SQLite 需要在关键点做"flush" 或 "fsync" 函数调用。SQLite 假定 flush 或 fsync 在数据没有真实的写入到硬盘之前是不会返回的。不幸的是，我们知道在一些 windows 及 unix 版本中，缺少 flush 或 fsync 的真正实现。这使得 SQLite 在写入一个提交发生掉电故障后数据文件得到损坏。然而，这不要紧，SQLite 能够做一些测试或补救。SQLite 假定操作系统会是广告中那样漂亮运行。如果这些都不是问题，那么剩下的只期望你家的电源不要间歇性的休息。

SQLite 假定文件增长方式是指新分配的文件空间，刚分配的时候是随机内容，后来才被填入实际的数据。换而言之，文件先变大，然后再填充其内容。这是一悲观假定，因而 SQLite 不得不做一些额外的操作来防止因断电发生的破坏数据文件—发生在文件大小已经增大，而文件内容还没完全填入之间的掉电。VFS 的 xDeviceCharacteristics 可以指明文件系统是否总是先写入数据然后才更变文件大小的。(这就是那个：SQLITE_IOCAP_SAFE_APPEND 属性，如果你想查看代码的话) 当 xDeviceCharacteristics 方法指示了文件内容先写入然后才改变文件大小的话，SQLite 会减少一些相当的数据保护及错误处理过程，这将大大减少一个提交磁盘 IO 操作。然而在当前的版本，windows 及 unix 的 VFS 实现并没有这样假定。

SQLite 假定文件删除从用户进程角度来讲是原子的。也就说当 SQLite 要 求删除一个文件，也在这删除的过程中间，断电了，一旦电源恢复，只有下列二种情况之一分发生：文件仍然存在，所有内容都没有发生变化；或者文件已经被删除 掉了。如果电源恢复之后，文件只发生了部分删除，或者部分内容发生了变化或清除，或者文件只是清空，那么数据库还有用才怪呢。

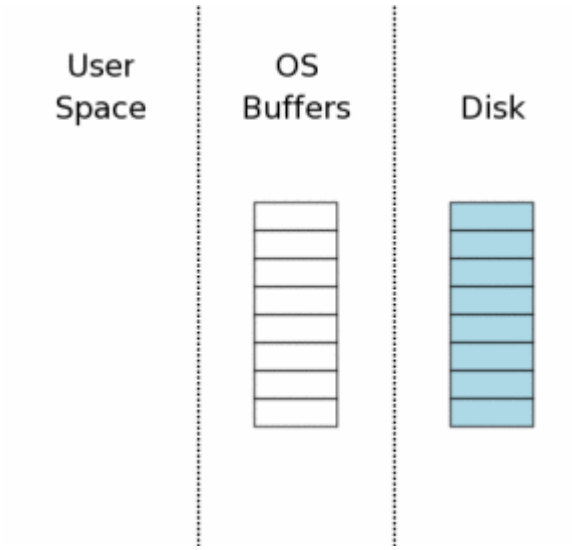
SQLite 假定发现或修改由于宇宙射线，热噪声，量子波动，设备驱动 bug 等等其他可能所引发的错误，都由操作系统或硬件来完成。SQLite 并不为此类问题增加任何数据冗余处理。SQLite 假定在写入之后去读取所获得的数据，是与写入的数据完全一致的！

3.0 单个文件提交

我们着手观察 SQLite 在针对一个数据库文件时，为保证一个原子提交所采取的步骤。关于在多个数据库文件之间为防止电源故障损坏数据库及保证提交的原子性所采用的技术及具体的文件格式在下一节进行讨论。

3.1 实始状态

当一个数据库第一次打开时计算机的状态示意图如右图所示。图中最右边（”Disk”标注）表示保存在存储设备中的内容。每个方框代表一个扇区。蓝色的块表示这个扇区保存了原始资料。图中中间区域是操作系统的磁盘缓冲区。在我们的案例开始的时候，这些缓存是还没有被使用—因此这些方框是空白的。图中左边区域显示 SQLite 用户进程的内存。因为这个数据库联接刚刚打开，所以还没有任何数据记录被读入，所以这些内存也是空的。

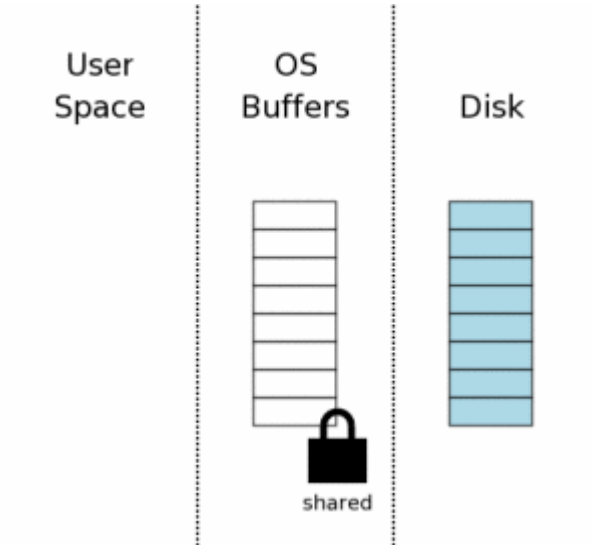


3.2 申请一个共享锁

SQLite 在可以写数据库之前，它必须先读这个数据库，看它是否已经存在了。即使只是只是增加添加新的数据，SQLit 仍然必须从 sqlite_master 表中读取数据库格式，这样才知道如何分析 INSERT 语句，知道在哪儿保存新的信息。

为了从数据库文件读取，第一步是获得一个数据库文件的共享锁。一个“共享”锁允许多个数据库联接在同一时刻从这个数据库文件中读取信息。“共享”锁将不允许其他联接针对此数据库进行写操作。这是必然的，如果一个联接在向数据库写入数据的同时，我们去读到信息，也可能读到的一部分数据是修改之前的，而另一部分数据是修改之后的。这将使得另外联接的修改操作看起来是非原子的。

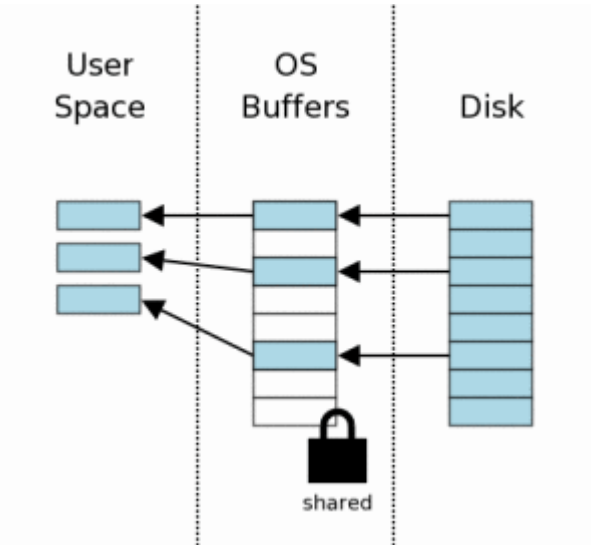
请注意共享锁只是针对操作系统的磁盘缓存，并非磁盘本身。通常文件锁只是操作系统内核的一些标识（详情要根据具体的操作系统）。因此，锁会立即消失一旦操作系统崩溃或者停电。当然创建该锁的进程消失，该锁也会随之而去。



3.3 从数据库里面读取信息

当 共享锁取得之后，我们就可以开始从数据库文件中读取信息了。在当前环节，我们已经假定了系统缓存是空的，所以信息必须首先从读硬盘读取到系统缓存中去，然 后从系统缓存中传递到用户空间。针对之后的读取，部分或者全部数据都可能可以从操作系统缓存中取得，所以只需要传递到用户空间即可。

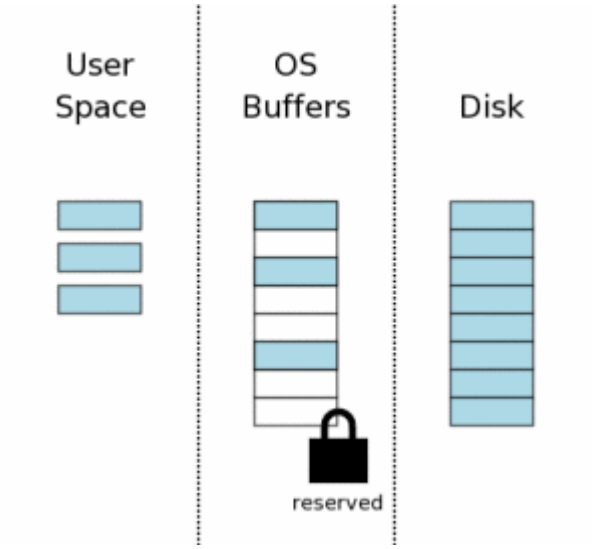
一般的，数据库文件只有部分被读取。这个例子中，8 页中只有 3 页被读取。一个典型应用中，一个数据库文件拥有成千上万页，一个查询通常读取到的页码数量只占总数一个很小的百分比。



3.4 申请一个 Reserved Lock

在修改一个数据库之前，SQLite 首先得拥有一个针对数据库文件的“Reserved” 锁。Reserved 锁类似于共享锁，它们都允许其他数据库联接读取信息。单个 Reserved 锁能够与其他进程的多个共享锁一起协作。然后一个数据库文件同时只能存在一个 Reserved 。因此只能有一个进程在某一时刻尝试去写一个数据库文件。

Reserved 锁的存在是宣告一个进程将打算去更新数据库文件，但还没有开始。因为还没有开始修改，因此其他进程可以读取数据，其他进程不应该去尝试修改该数据库。

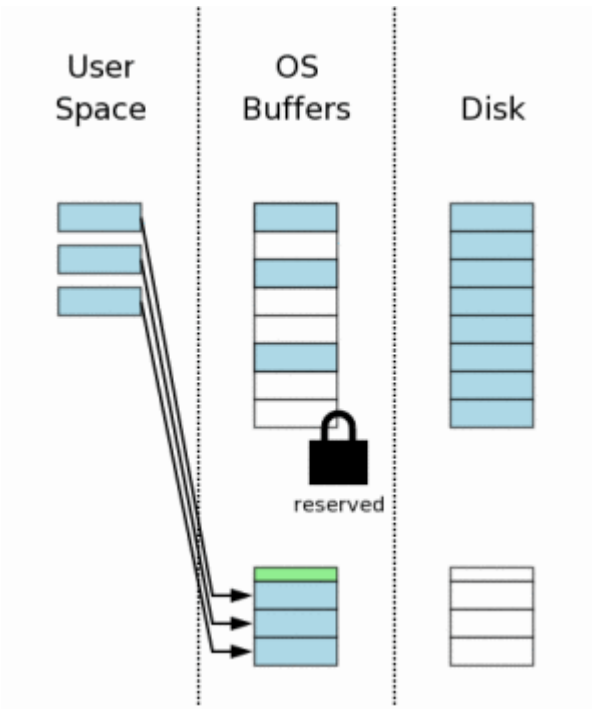


3.5 生成一个回滚日志文件

在修改数据库文件之前，SQLite 会生成一个单独的回滚日志文件，并在其中写进将会被修改的页的原始数据。回滚日志文件意味它将包含了所有可以将数据库文件恢复到原始状态的数据。

回滚日志文件有一个小的头部（图中绿色标记部分）记录了数据库文件的原始大小。因此，如果一旦即使数据库文件变大，我们还是会知道它原始大小。数据库文件中被修改的页码及他们的内容都被写进了回滚日志文件中。

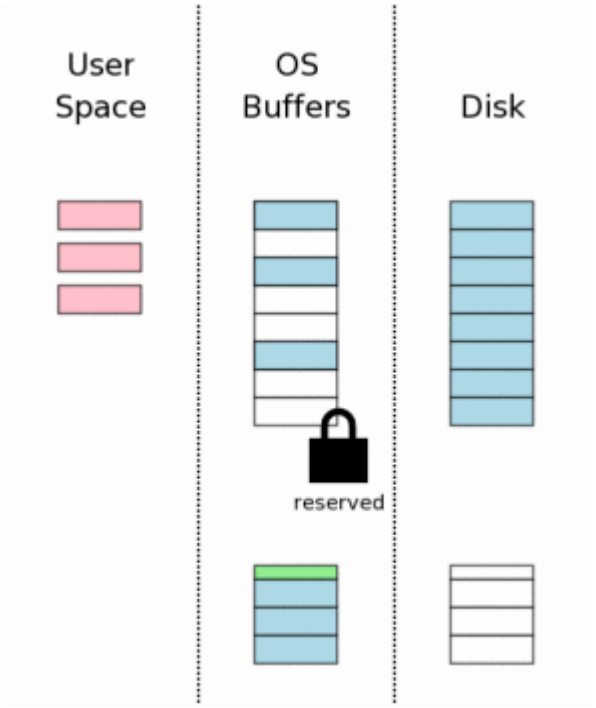
当一个新文件刚被创建，大部分的桌面操作系统（windows,linux,macOSX）实际并不会马上写入数据到硬盘。此文件还只是存在于操作系统磁盘缓存中。这个文件还不会立即写到存储设备中，一般都会有一些延迟，或者到操作系统相当空闲的时候。用户的对于文件生成感觉是要远远快（先）于其真实的发生磁盘 I/O 操作。右图中我们用图例说明了这一点，当新的回滚日志文件创建之后，它还只是出现在操作系统磁盘缓存之中，还没真实在写入到硬盘之上。



3.6 修改用户进程中的数据页

当原始的数据已经被保存到回滚日志文件中之后，用户内存的数据就可以被修改了。任何一个数据库联接都有其他私有用户内存空间，所以用户内存空间发生的变化只有当前数据库联接才可见。

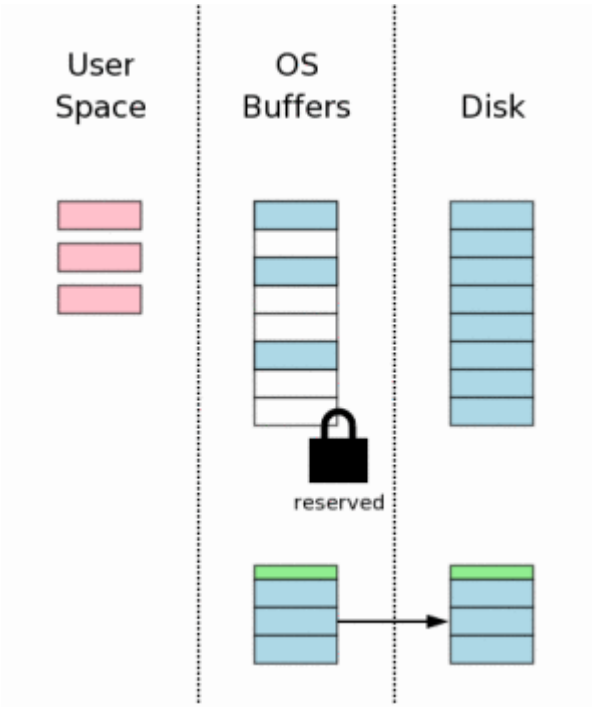
其他数据库联接仍然可以读取那些存在于操作系统磁盘缓存中还没有被修改的数据。所以即使一个联接忙于某些修改，其他进程还可以读取原始数据到它们各自的空间中去



3.7 刷新回滚日志文件到存储设备中

接下来的步骤是将回滚日志文件刷新到硬盘中去。接下来我们会看到，这是一个紧要步骤用来保证我们可以从突然掉电中救回数据。这个步骤将要花费大量的时间—因为通常写入到硬盘是一个耗时操作。.

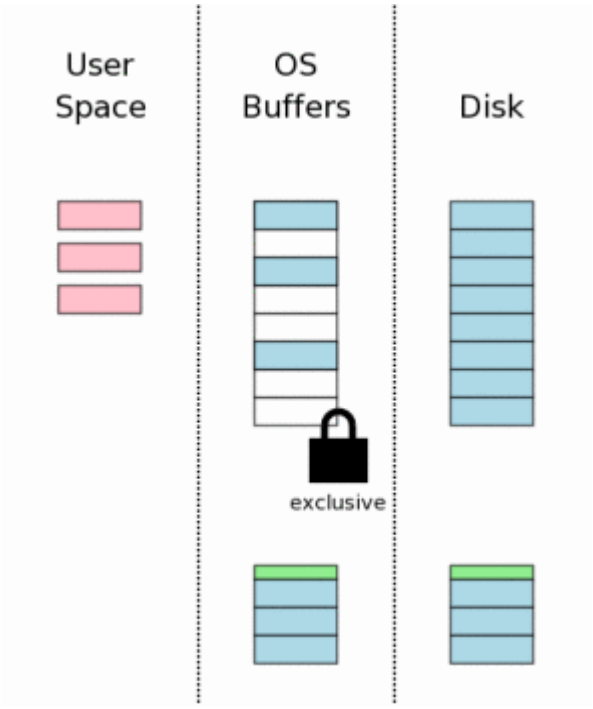
这个步骤通常要比简单的直接刷新这个回滚文件到硬盘要复杂一些。在大部分的操作系统中，二个单独的 flush 是必须的。第一个 flush 处理日志文件的内容部分。接下来，将日志文件的页码总数写入到日志文件头部，然后将日志头部 flsuh 到硬盘中。至少为什么我们要做一个头部修改及做一个额外的 flush 操作的原因我们会在后面的章节解释。



3.8 获得一个独享锁

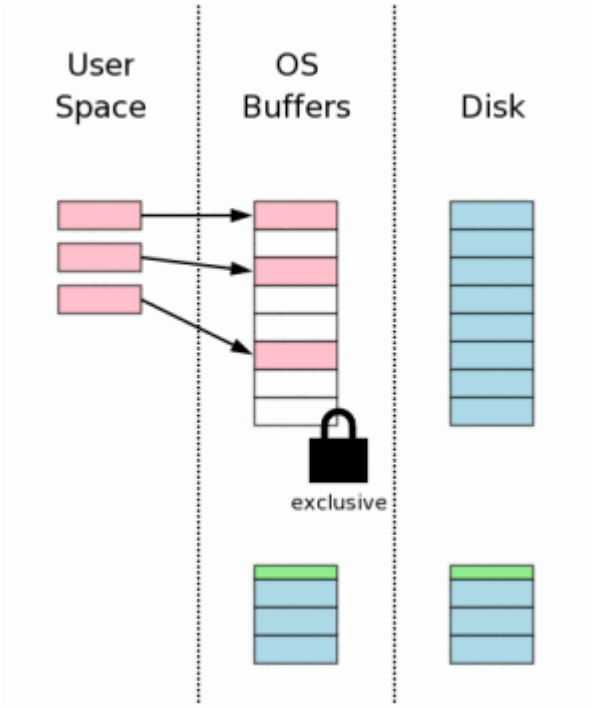
在修改数据库文件本身之前，我们必须取得一个针对此数据库文件的独享锁。取得此锁的过程是分二步走的。首先 SQLite 取得一个“临界”锁，然后将此锁提升成一个独享锁。

一 个临界锁允许其他所有已经取得一个共享锁的进程从数据库文件中继续读取数据。但是它会阻止新的共享锁的生成。也就说，临界锁将会防止因大量连续的读操作而 无法获得写入的机会。这些读取者可能有一打，也可能上百，甚至于上千。任何一个读取者在开始读取之前都要申请一个共享锁，然后开始读取它需要的数据，然后 释放共享锁。然而存在这样一种可能：如果有太多的进程来读取同一个数据文件，在老的进程释放它的共享锁之前总是会有新的进程申请共享锁，因此不会存在某一 时刻这个数据库文件上没有共享锁的存在，也因此写入者不会拥有取得一个独享锁的机会。临界锁的概念可以使现有的读取者完成他们的读取，同时阻止新的读取者 读取，最后所有的读取者都读完之后，这个临界锁就可以被提升为独享锁了。.



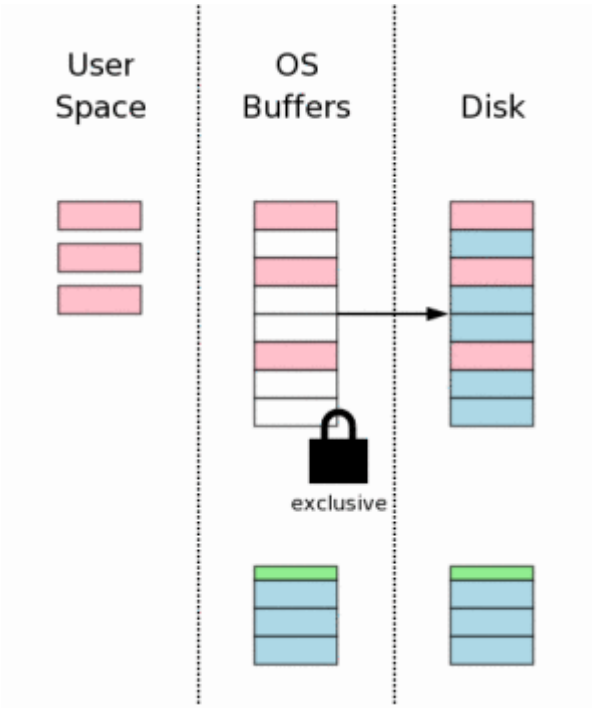
3.9 将变更写入到数据库文件中

一旦独享锁在手，我们知道再也没有其他进程在读取此数据库文件了，此时修改此文件是安全的了。通常，这些变更只会发生在操作系统磁盘缓存中，并不会全部写入到磁盘中去。



3.10 刷新变更到存储

一个附加的 flush 操作是必要的，这样才可以保证针对此文件的变化真正的写入到永久存储器中。这也是一个重要的步骤，将可以保证数据在掉电之后也将是完整无损的。然而，因为写入到磁盘所固有的慢，这个步骤同上面 3.7 节将日志文件 flush 到磁盘中一样，占据了 SQLIite 事务提交操作的绝大部分时间。

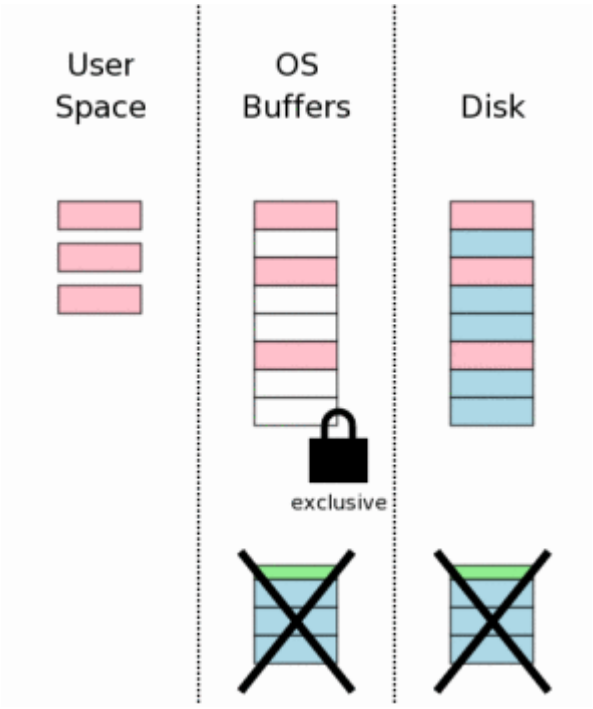


3.11 删除回滚日志文件

当数据变更已经安全的写入到硬盘之后，回滚日志文件就没有必要再存在了，因此立即删除之。如果在删除之前又掉电了或者系统崩溃了，恢复进程（在后面将会提到）会将日志文件的内容写回到数据库文件中—即使这个数据库没有发生变化。如果删除之后系统崩溃或者又停电了，看起来好象所有变化都已经写入到磁盘。因此，SQLite 判断数据库文件是否完成了变更是依赖于回滚日志文件是否存在。

删除一个文件实际上不是一个原子操作，但从用户进程的角度来看，它是一个原子操作。一个进程总是可以向操作系统询问某个文件存在否，而它得到的答案只有“YES”和“NO”二种。在一个事务提交的中间，系统崩溃或又停了，之后，SQLite 会向操作系统咨询回滚日志文件存在与否，如果存在，则这个事务是没有完成，被中断了，需要对数据库文件进行回滚。如果日志文件不存在，意味着事务已经提交 ok 了。.

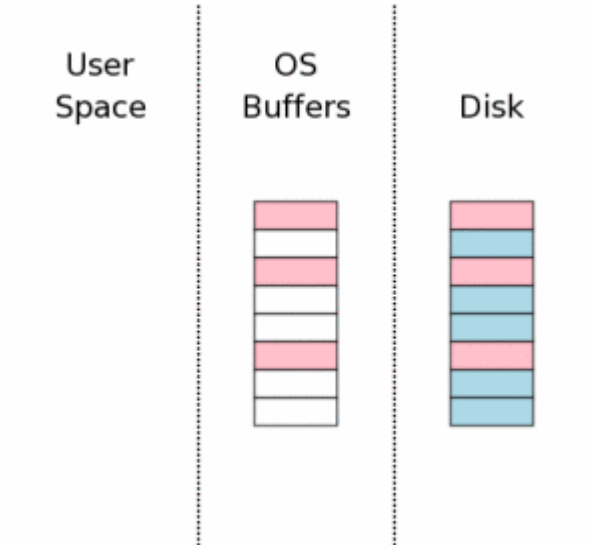
事务存在的可能性依赖于是否有回滚日志文件。删除一个文件对于一个用户进程来说是原子性的。因此，整个事务看起来也是一个原子操作。.



3.12 释放锁

事务提交最后一个步骤是释放独享锁，其他进程就又可以立即访问数据库文件了。

右图中，我们指明了当锁被释放的时候用户空间所拥有的信息已经被清空了.对于老版本的 SQLite 你可这么认为。但最新的 SQLite 会保存些用户空间的缓存不会被清空—万 一下一个事务开始的时候，这些数据刚好可以用上呢。重新利用这些内存要比再次从操作系统磁盘缓存或者硬盘中读取要来得轻松与快捷得多，何乐而不为呢？在再 次使用这些数据之前，我们必须先取得一个共享锁，同时我们还不得不去检查一下，保证还没有其他进程在我们拥有共享锁之前对数据库文件进行了修改。数据库文 件的第一页中有一个计数器，数据库文件每做一次修改，这个计数器就会增长一下。我们可以通过检查这个计数器就可得知是否有其他进程修改过数据库文件。如果 数据库文件已经被修改过了，那么用户内存空间的缓存就不得不清空，并重新读入。大多数情况下，这种情况不大会发生，因此用户空间的内存缓存将是有效的，这 对于性能提高来说作用是显著的。



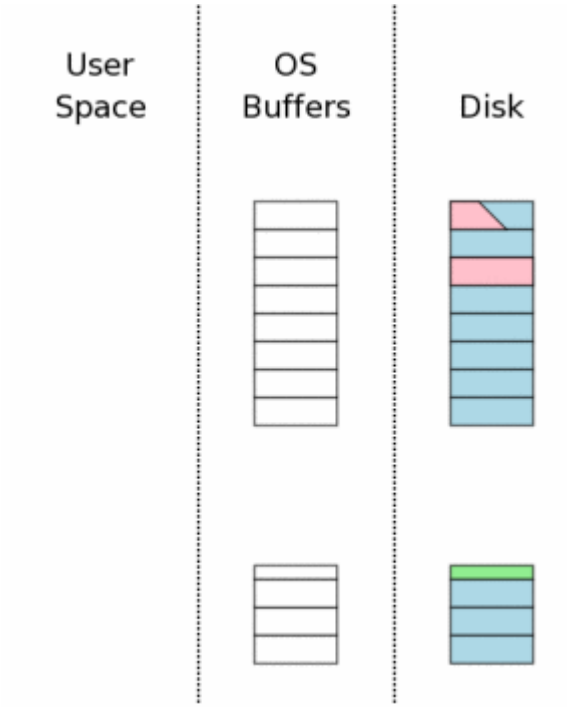
4.0 回滚

原子提交被设定是瞬间发生的。但上面的描述已经指出了其实这个过程是要花费不少时间的。如果在上面的提交过程中，计算机的电源被拉掉的情况下，为了保证变更是瞬间发生的事情，我们将“回滚”这些变化，将数据库文件恢复到事务开始之前的状态。

4.1 出事了，出事了!!!

假设掉电发生在上面 3.10 步骤中。电源恢复之后，当前的状态可能如右图所示。我们打算修改数据库文件中的三页但只有一页被成功写入，其他一页只部分写入，还有一页根本就没有写入。

这时，回滚日志文件是完整的。这是关键因素。上面 3.7 步骤做 flush 操作的理由是将任何变更写入到数据库文件之前要绝对保证回滚日志文件已经安全、完整的写入到了永久存储中。



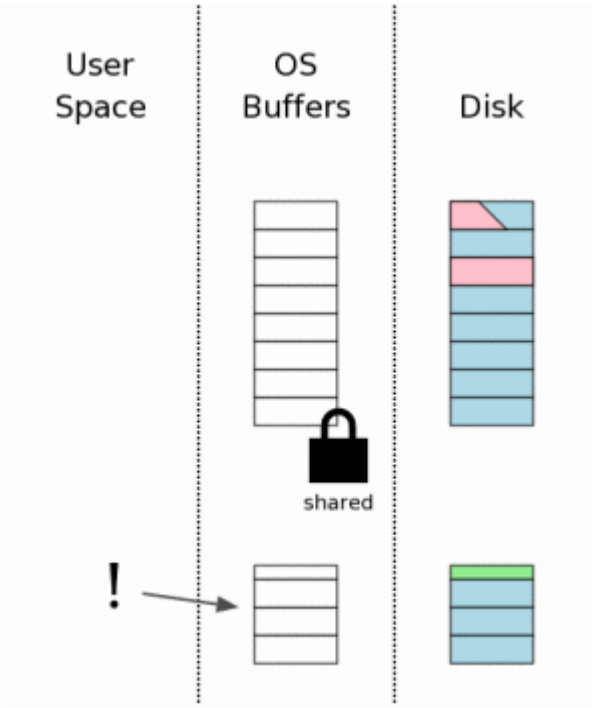
4.2 Hot Rollback Journals

上面 3.2 节已经描述了，所有 SQLite 进程尝试访问数据库文件之前，都得必须取得一个共享锁。但现在却被告知有一个回滚日志文件存在。SQLite 会进行检查看这个日志文件是否是一个”hot journal”。A hot journal 是指需要被用来进行处理以使数据库回复到健壮的初始状态的。hot journal 的存在意味着早先的进程在一个事务中间发生了系统崩溃或掉电故障。

回滚日志是一个”hot”的先天条件

- 回滚日志文件存在
- 回滚日志非空
- 这时数据库文件没有独享锁

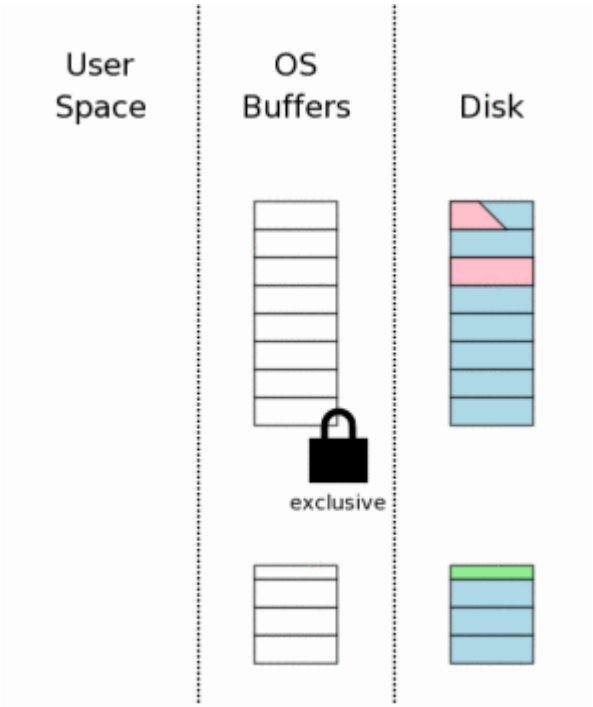
- 回滚日志文件头部没有包括主日志文件的名字（5.5 节）或者包含了主日志文件名称而且主日志文件存在



“hot”日志文件存在指明先前的进程尝试去提交一个事务，但由于种种原因在完成提交以前，事务被中止了。同时指明了数据库文件的状态是需要通过回滚来修复的，修复之后才可以被正常使用。

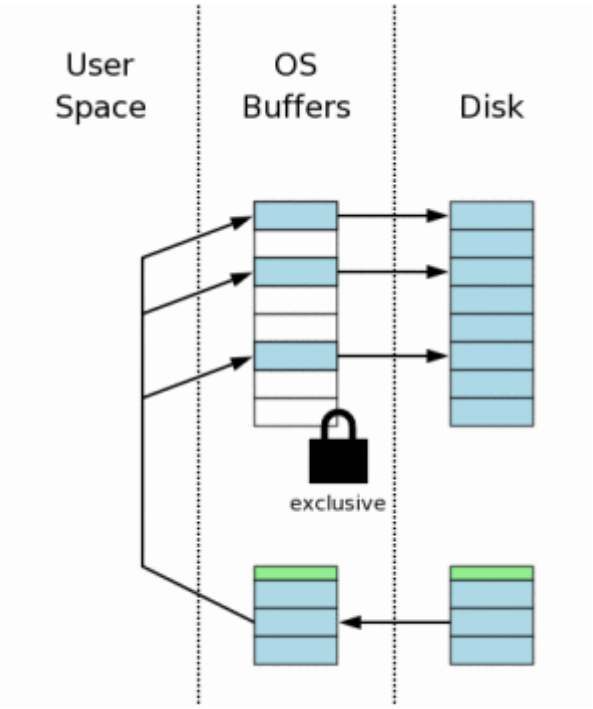
4.3 取得数据库的一个独享锁

为了处理“hot”日志文件首先是要取得一个数据库的独享锁。这将防止 2 个或多个进程在同一时刻来尝试回滚同一个“hot”日志文件。



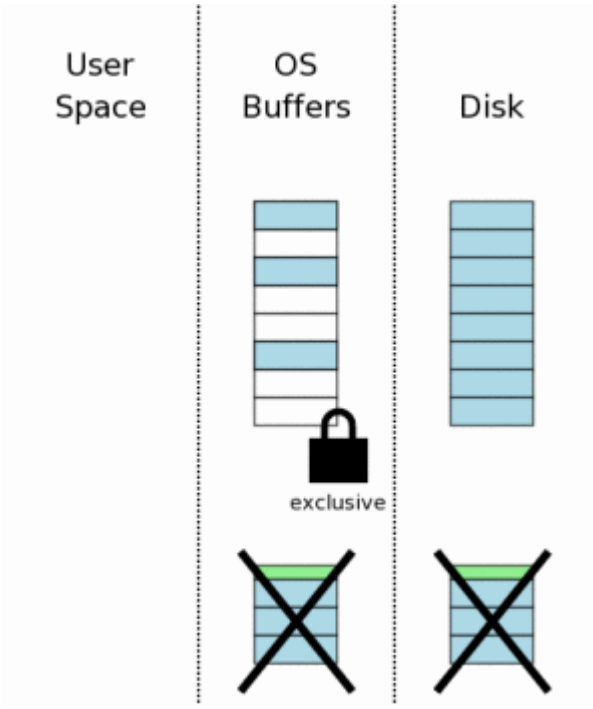
4.4 回滚没有完成的变更

一旦进程获得一个独享锁，它就被允许更新数据库文件。然后从日志文件中读取原始的内容，并写回到数据库文件中。是否还记得在这个被中止的事务的开始的时候，数据库文件原始大小已经被写进了日志文件的头部。SQLite 使用这些信息来截断数据库文件，让文件恢复到原始大小—如果这个没有完成的事务使得数据库变大了。最后，数据库文件大小及内容肯定与这个被中断事务开始之前是一样的了。



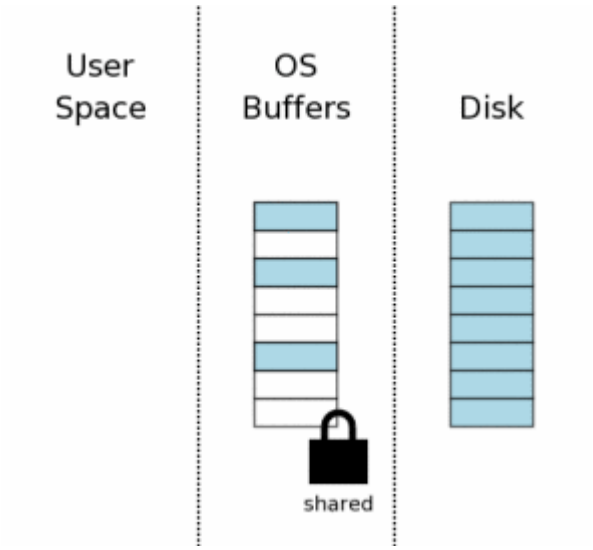
4.5 删除 hot 日志文件

当日志文件中的所有数据都被放回至数据库文件之后（并且做了 flush），此日志文件就可以被删除了。



4.6 如果一切正常，没有什么未完成的写操作

恢复过程最后的步骤就是将独享锁降格成共享锁。一旦到了这里，数据库已经回得被中断事务开始的时状态。既然这个恢复操作已经完成，自动，自然而又透明，似乎被中断的事务从没有发生过一样！☺



5.0 多文件提交

SQLite 允许单个数据库联接通过使用 ATTACH DATABASE 命令同时与 2 个或多个数据库文件交互。在一个事务中，多个数据库文件被修改，所有文件的更新是原子性的。换而言之，要么所有文件被修改好了，要么什么也没有发生。针对多个文件的原子提交是要比仅针对单个文件的处理复杂一些。本节描述 SQLite 是如何完成这有魔术色彩的工作的。

5.1 每个数据库文件单独拥有日志文件

当一个事务涉及到多个数据库文件时，每个数据库文件都会有其相应的独立的回滚日志文件，并且每个数据库都是分别加锁的。下图显示了某个事务中修改了三个不同的数据库文件。这种情况与 3.6 步骤处理单个文件的事务还是有一些类似的。每个数据库文件有一个独享锁。针对每个数据库，要被修改页的原始内容被写入它们相应的回滚日志文件，但日志文件的内容还没有被 flush 到硬盘中。这时针对数据库的变更还没有发生，虽然有可能用户空间的数据已经发生了变化。

简 单的说，下图已经简化了它们之前的状态。蓝色仍然指明是原始内容，而粉红是新的内容。但日志文件及数据库单独的页我们没有指示出来，同时我们也没有指明信 息在操作系统磁盘缓存与硬盘中信息的差异。所有这些因素在一个多文件提交的场合下仍然起作用。这些因素会占据图中许多位置，但并没有增加新的信息，因此它 们在此图中被省略掉了。

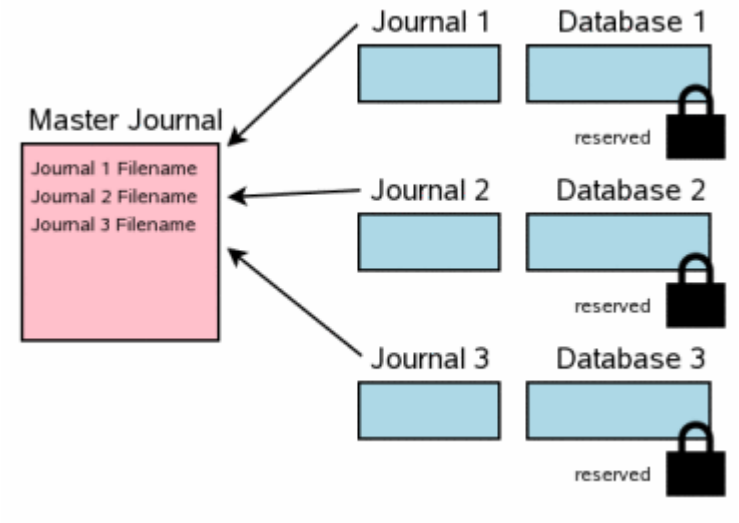
5.2 主日志文件

多文件提交的下一步是生成“主日志”文件。主日志文件的名称是与原始的数据库文件名 (数据库指的是用 sqlite3_open 的，而不是 ATTACHed 等辅助数据库)，再加上文本"**-mj***HHHHHHHH*"。附加的 *HHHHHHHH* 是一个随机 32 位 16 进数。每一个新的主日志文件会有一个变化的随机数 *HHHHHHHH* 后缀。

（注意：前面计算主日志文件名的算法是与 SQLite3.5.0 是一致的，但这不是 SQLite 规范的一部分，或许在新版本中发生变化）

不同于回滚日志文件，主日志文件并不包含任何数据库文件的页的原始内容。主日志文件包含了此事务所涉及的数据库的回滚日志文件的全路径。

当主日志文件已经创建完成之后，它会被立即 flush 到硬盘，这个操作早于任何其他操作。在 unix 下面，这个主日志文件所在目录也被同步到了硬盘，保证掉电以后主日志文件显示在此目录中。

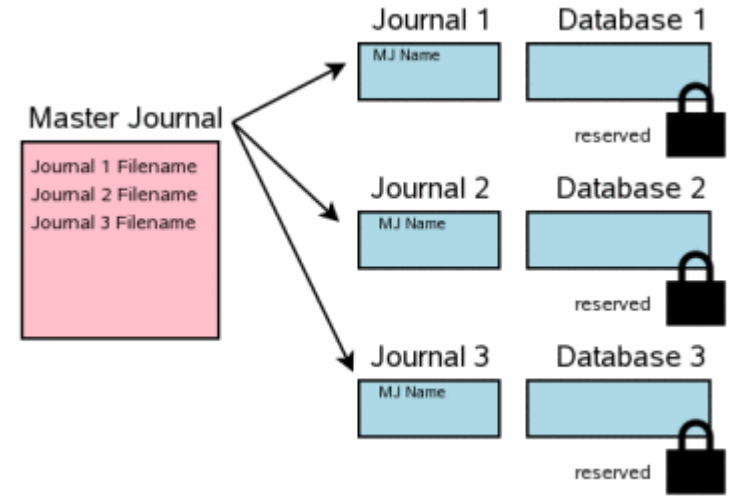


5.3 更新回滚日志文件头

接下来在每一个回滚日志文件的头部需要记录主日志文件的全路径。当一个回滚日志文件被创建时，用来存储主日志文件名的空间已经被保留在每一个日志文件的开始部分。

在主日志文件名写入到日志名头部之前与之后都要进行一次 Flush 日志文件内容到硬盘。做二回 flush 很重要。幸运的是第二次的 flush 相对而言代价不是那么昂贵，因为一般的日志文件只有一页发生变化（第一页）

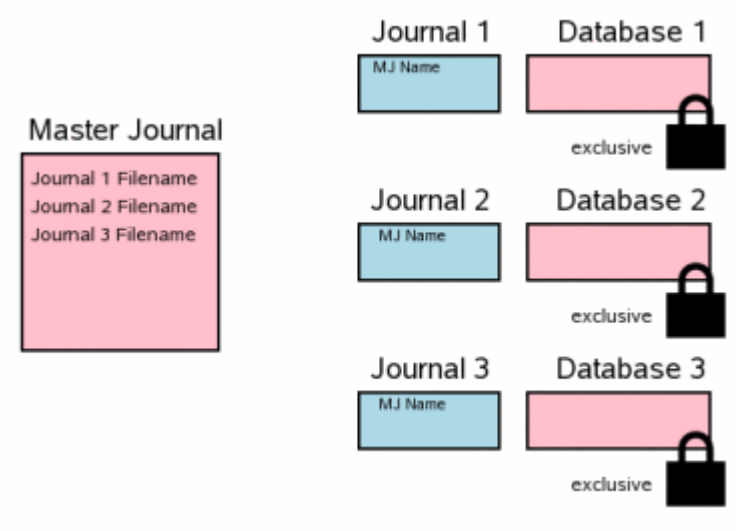
这一步与 3.7 节的单个文件事务提交场景类似。



5.4 修改数据库文件

一旦所有的回滚日志文件已经 flush 到了硬盘中，就已经很安全的进行数据库文件更新了。我们在修改数据库文件之前必须得到所有数据库的独享锁。当所有的修改都完成的时候，flsuh 数据库文件到硬盘是非常重要的。这将防止因系统崩溃或掉电而导致数据库损坏。

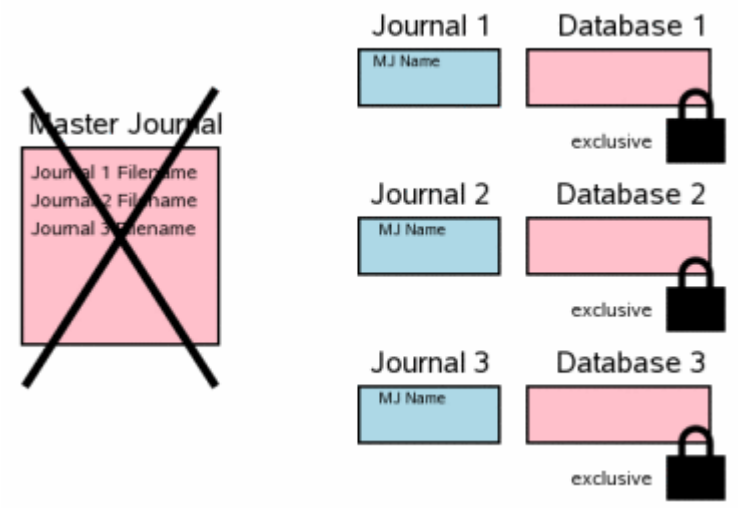
这个步骤与单个文件提交过程中 3.8,3.9 及 3.10 步骤是一致的。



5.5 删除主日志文件

接下来的步骤是删除主日志文件。对于多文件事务提交，这是一个要点。这个步骤与上面 3.11 中单个文件的事务提交场景是相呼应的。

这时，如果发生系统崩溃或者又停电了，当系统重新运行的时候，即使回滚日志文件存在，这个事务不会被回滚。不同点在于回滚日志文件中主日志文件路径。当系统重启的时候，如果回滚日志文件没有主日志文件名(针对于单文件提交)或者主日志文件仍然存在的时候，SQLite 才会将这些日志文件视为”hot”，并将回滚日志文件的内容放回到数据库文件中去。

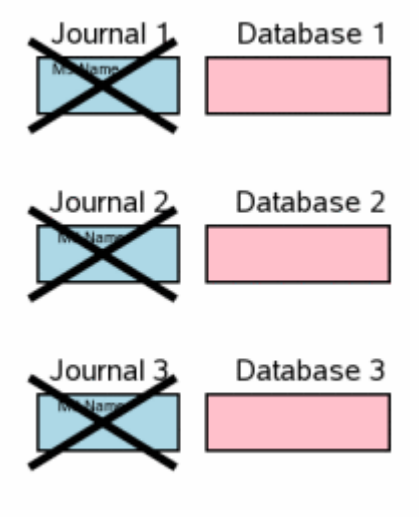


5.6 清除回滚日志

多文件事务提交的最后一步是删除单独的回滚日志文件，释放数据库文件的独享锁，其他进程就可以看到数据库的

变化；这与上面 3.12 是相一致的。

这时事务已经提交完成了。所以删除日志的时间点并不是很紧急。当前的实现是删除某个回滚日志文件，并释放相应的数据库锁，然后处理另一个日志文件。以后有可能改为删除所有日志文件之后才释放所有的锁。日志文件删除只要是在其相对应的锁释放之前就没有任何问题。



6.0 原子操作的一些实现细节

3.0 节大致描述了 SQLite 中原子提交是如何工作的。但它略过了许多重要的细节。下面的这些部分将尝试补充说明这些地方。

6.1 总是记录整个扇区

当数据库文件的原始代码被写入到日志文件时（参见 3.5 节），SQLite 总是写入完整的扇区，即使数据文件页大小是小于一个扇区。由于历史上的原因，SQLite 的扇区大小原先是固定为 512 字节，此外由于最小的页大小是 512 字节，因此这从来都不是一个问题。自 SQLite3.3.14 版本以来，SQLite 便有可能使用最小扇区大于 512 字节的海量存储设备。所以，自从 3.3.14 版本开始，只要一个扇区中的任何一页被写进到回滚日志文件中，那么同一扇区中的所有节都会写入到日志文件中去。

将扇区中的所有页都写入日志文件中去是很重要的，它将可以防止因为在写一个扇区时发生掉电故障而导致数据库损坏。假充页 1，2，3，4 都是保存扇区 1 中，页 2 被修改了。为了将这种变更写回到页 2 中，实际的硬件设备将也会同时重写页 1，3 及 4 的内容—这是因为硬件必须以扇区为单元作写操作。如果一个写操作正在进行的时候，由于电源的原因，发生了中断，这样，页 1，3，4 中会有 1 页或者多页数据是不完整，不正确的。因此为了防止这种损坏，数据库文件的同一扇区中的所有页都必须写入到日志文件中去。

6.2 写日志文件时垃圾的处理

当向一个日志文件追加数据时,QLite 总是悲观的假定文件会首先变大，变大的部分会填之一些无效的垃圾数据，在此之后正确的数据才会取代这些垃圾。换而言之，SQLite 假定文件先改变大小，然后内容才会写进来。如果在文件大小增大之后，在内容还没有写完之前发生掉电故障，那么这些日志文件就会留下一些垃圾数据在其中。下次当电源恢复，另一个 SQLite 进程就会看到这些保存了垃圾数据的日志文件，并同时会把这些垃圾数据回滚到数据库文

件中去，然后整个数据库就玩完了。

SQLite 使用了二种方式来预防这种问题。首先，SQLite 会记录日志文件中的页数量。这个数量被初始化成 0。所以在尝试回滚一个不完整（可能不正确的）回滚日志文件时，处理回滚的进程会看到日志只包含 0 个页面，那么它就会不对数据库作任何改变。提交之后，日志文件会被 flush 到硬盘中用来确保所有的内容都同步到硬盘，同时没有任何垃圾内容保留在文件中。同时，只有在此之后日志文件头部的页总数值才会置成真实有效的数据（原先数值是 0）。日志文件的头部总是与任何数据页处于不同的扇区中。所以它可被修改并且单独 flush，因此即使发生掉电，也不会给页面数据带来任何风险。请注意，日志文件被单独 flush 二回：第一次写页数据（其实也把头部给 flush 了）第二次是将页面数量写入(flush)到文件头部中。

前面的章节描述了当 synchronous pragma 设置成”full”发生的事情。

PRAGMA synchronous=FULL;

缺省的 synchronous 设置是“full”，所以上面描述是通常会发生的情形。然而，如果 synchronous 设置成“normal”，那么 SQLite 会只会在页面数量写入之后，flush 日志文件一次。这将意味着一个数据破损的风险。因为有可能被修改的页面数量（非 0）会比所有的数据早一些写入到硬盘之中。数据部分的写入虽然是更早调用的，但 SQLite 假定实际的文件系统会重新调整写入顺序。所以有可能页面数量会更早的记录到磁盘中，即使是它的写请求是发生在最后。所以作为第二个预防手段，SQLite 会为日志文件中的每一个页记录一个个 32 位的校验值。当一个日志回滚进程回滚数据时（节 4.4），这些值用来指示这些页是否有效。如果发现一个不正确的校验时，那么回滚就会放弃。要注意的是，由于校验值比较小，所以校验值并不确保页面数据百分百的正确。但也不用过于担心，如果数据损坏了，检验值仍然正确的概率实在很小。所以校验值还是能够有一定作用的。

要知道，如果 synchronous 设置成 full 时校验时不是必须的。只有当 synchronous 设置成 normal 时，我们才使用这些校验值。尽管这样，校验数据是无害的，所以无论 synchronous 设是什么，他们都保存在日志文件了。

6.3 提交前缓存溢出

节 3.0 描述的提交过程都假定了所有变更的数据都保存在用户内存中，直到真正提交。这是通常会出现的状况。但有时一个非常大或（多）的修改会超出用户空间的内存缓存大小。在这种情况下，一个事务完成之前，缓存不得不将数据先写入到数据库中。

在一个缓存发生溢出之前，这个数据库联接的状态如 3.6 节。数据库原始的内容已经被写入回滚日志文件中了，页面修改部分还保存在用户内存中。要处理这种缓存溢出，SQLite 会执行 3.7 节到 3.9 节。换而言之，回滚日志被 flush 到硬盘，独享锁已经申请到，修改已经被写入到数据库了。但剩余的步骤会延迟，直到这个事务被真正提交。一个新的日志文件头会追加到回滚日志文件尾部（处于它自己单独的扇区中），独享锁仍然保留，但其他处理则回到节 3.6.当这个事务提交时，或者另外的缓存溢出发生，节 3.7 及节 3.9 会再次发生（节 3.8 在第二次或以后过程中被省略掉，因为独享锁已经拿到了）。

一次缓存溢会使数据库的临界锁提升到独享锁。这将减少并发。一次缓存溢出也将导致额外的硬盘 flush(fsync)操作，并将导致这些操作变慢，因此缓存溢出将严重降低性能。由于这些因素，缓存溢出现象应该尽量避免。

7.0 优化

在大部分的操作系统，大多数的工作环境下面，性能指标指示 SQLite 主要费时在磁盘操作上面。如果我们能够减少磁盘 IO 数量就会显著的提高 SQLite 的性能。本节将描述 SQLite 在不影响提交原子性的前提下，为减少磁盘 IO

数量所采用的一些技术。

7.1 在事务间保存缓存

事务提交处理过程中，节 3.12 指 出一旦共享锁被释放，用户空间所有的缓存的数据库内容镜像都必须得抛弃。这是因为如果没有一个共享锁，其他进程就可以修改数据库的内容，所以用户空间所缓 存的数据库数据就会过期无效。因此，每一个新的事务会尝试去读取它以前读取过的数据。这似乎并不是太糟糕，因为第一次读取过的数据还可能存在于操作系统的 磁盘缓存中。所以这个读实际上只是只一次从内核空间到用户空间的复制。但尽管是这样，这还是需要占用 cpu 时间的。

自从 SQLite3.3.14 开始，新增了一个机制用来减少一些不必要的数据重复读取操作。最新的 SQLite 中，用户空间的页面缓存在用户锁释放之后仍然保留。之后，当要开始一个新事务，在取得一个共享锁之后，SQLite 会尝试检查在此期间是否有进程对数据进行了修改。如果在锁释放这段时间，数据库发生任何的变化，那么用户空间的缓存就会被释放。但通常情况下，数据文件是没有被修改过的，因此用户空间的缓存因而得到保留，一些不必要的读取操作从而得到了减免。

为了判断数据库文件是否被修改过，SQLite 使用了一个计数器，存于数据库文件头部（处于字节 24～27），每针对数据库做一次修改，就会对此值进行一回增长。SQLite 会在释放一个锁之前记录一份这个值的。当下回取得锁之后，就会去与原先保存的值进行比较。如果值不一致，则必须清除这些缓存，反之缓存可以重新使用。

7.2 独享访问模式

SQLite 从 3.3.14 版本之后增加一个“独享访问模式”概念。当处于独享访问模式时，SQLite 会在一个事务完成之后仍然保留独享锁。这将阻止其他进程访问这个数据库；由于大部分的开发都只有一个进程访问数据库，所以大部分情况下这不是一个严重的问题。独享访问模式的好处可以在三个方面减少磁盘 IO 数量：

1) 不再需要为每个事务完成之后修改文件头部的变更计数器。这可以为回滚日志及数据库文件减少一次页写入。

2) 没有其他进程会修改数据库，所以不必在一个事务开始的时候去检查变更计数器或者清除掉用户空间的缓存。

3) 当一个事务完成之后，可以采用清空日志文件的方式，而不必去删除这个文件。在大多数的操作系统中，清空文件要远快于删除一个文件。

上述的第三点优化，清空而不是删除回滚日志文件，不再要求一直取得一个独享锁。在理论上，我们可以在任何时刻做这项优化，并不是只有在独享访问模式时。在将来的 SQLite 版本中我们或许可能这么做。但当前的版本（3.5.0）回滚日志文件清空优化只发生在独享访问模式。

7.3 不必将空闲页写进日志

SQLite 数据库的信息被删除之后，这些被删除的数据所使用的页会被加入到空页链表之中。后来的插入操作会尽量先使用空页链表中的页。

一些空白页包含紧要数据：特别是其他空百页的位置。但是大多数的空白页并不包含有用信息。这类页被称之为“叶子”页。我们可以随意修改这些叶子页的内容而不会影响数据库。

因为叶子页的内容是不重要的，SQLite 避免保存这些叶子页的内容到回滚日志文件中去（3.5 节）。如果一个叶子页的内容被修改了，那么在事务恢复过程中这些针对叶子页的修改并不会回滚。这不会对数据库产生伤害。同样的，新的空页链表的内容也从不会在节 3.9 中写回到到数据库，也不会在节 3.3 从数据库读入。当针对数据库文件的变化包含有空白**页时**，这种优化可以大量的减少磁盘 io 操作总数

7.4 单页更新及扇区原子写

从 3.5.0 开始，新的 VFS 接口包含了一个新的方法： xDeviceCharacteristics ，它能够读取实际的文件系统可能有的特性。xDeviceCharacteristics 会报告是否文件系统能够支持扇区写原子操作。

回想前面，在一般情况下 SQLite 假定扇区写是线性的，但是非原子的。线性写从另一个扇区结束点开始一字节一字节进行修改，直到扇区的结束点。如果在写一个扇区时，线性写会将修改一个扇区的一部分，而另一部分是没有变动的。在一个扇区原子写的情况下，要么整个扇区被重写了，要么扇区没有发生变化。

我们相信大部分现代磁盘驱动器实现了原子写操作。当停电发生时，磁盘驱动器可以利用电容中的电能，同时（或者）利用盘片旋转的角动量来完成正在进行中的任何操作。然而，在系统写调用与磁盘电子器材之间，存在有太多的层次。因此在 unix 及 win32 上面的 VFS 实现比较安全的选择是，我们假定扇区写操作是非原子性的。On the other hand, device manufactures with more control over their filesystems might want to consider enabling the atomic write property of xDeviceCharacteristics if their hardware really does do atomic writes.

当一个扇区写是原子性的，并且扇区大小与页大小是相同，并且一次数据库的变化只是某一个单独的页发生变化时，SQLite 会跳过整个日志记录过程，直接简单地将被修改过的数据写回到数据库文件。数据库首页中的变更计数器将会被独立进行修改—因为不会对数据库产生任何影响—即使在计数器更新以前发生停电。.

7.5 Filesystems With Safe Append Semantics

SQLite3.5.0 中介绍的另一个优化是利用实际磁盘的“安全追加”行为。回想上面，SQLite 假定为一个文件追加数据时（特别是针对回滚日志文件），会先增大文件的大小，之后才会把数据内容写入。所以在文件的大小已经变化，而内容还没有写完的情况下发生掉电，那么文件新增部分将会有一些无效的垃圾数据。VFS 的 xDeviceCharacteristics 可以用来指示文件系统是否实现了“安全追加”语义。这意味着在文件大小变大之前会先写入文件内容。这就防止当系统崩溃或掉电后，垃圾数据出现在回滚日志文件中

当文件系统有安全追加特性时，SQLite 总是保存一个特别的值：-1 来标明日志文件中页总数。页面数量为-1 告诉任何尝试进行回滚操作程序页面数量需要从日志文件大小计算得来。同时，这-1 值会从不进行修改。所以，在一个提交过程中，我们节省一个 flsuh 操作及日志文件首页的扇区写入操作。此外，当发生缓存溢出时，也不必要在日志文件后面增加一个新的日志头。我们能够简单的在一个现有的日志文件中添加一些新的页。

8.0 原子提交行为测试

SQLite 的开发者对 SQLite 在面对电源故障及系统崩溃时所拥有健壮性具有足够的自信。因为自动化的测试过程做

了大量的面对模拟的电源故障的 SQLite 恢复能力测试。我们称之为“崩溃测试”。

SQLite 的崩溃测试是使用一个修改过的 VFS，它能够模拟种种发生掉电或系统崩溃时文件系统发生的损坏。崩溃测试用的 VFS 能够模拟未完成的扇区写操作，未完成的写操作造成的页面垃圾，还有无序写操作，一个测试场景中各种种各样的变化。崩溃测试不停地执行事务，让模拟的掉电或系统崩溃发生在不同的各种时刻,造成不同的数据损坏。在模拟的事件之后，任何一次测试重新打开数据库之后，会检测事务是否完成或者没有完成，数据库状态是否正常。

SQLite 的这些崩溃测试发现恢复机制的大量细微的 BUG（现在都已经修复了）。其中一些 BUG 是非常模糊的，如果只是单单观察、分析代码所不能发现的。通通过这试验，SQLite 的开发者感觉很自信，因为其他的数据库没有采用类似的崩溃测试，很可能他们都包含一些没有被检测出的 bug，在一次掉电或者系统崩溃之后会导致数据库损坏。

9.0 会导致完蛋的事情

SQLite 的原子提交机制已经被证明是健壮的。但它也可能被一些不完整的操作系统实现所陷害。本节描述一些会在掉电或系统崩溃下会导致 SQLite 数据损坏的情形

9.1 缺乏文件锁实现

SQLite 通过文件系统的锁来实现在同一时刻只有一个进程及一个数据库联接能够修改数据库。文件锁机制由 VFS 层实现，不同的操作系统具有不同的实现方式。SQLite 依赖于这种实现的正确性。如果在某种情况下，二个或更多进程能够在同一时间写同一个数据库文件，这将会没有什么好果子吃的。

我们已经接收到报告说 windows 的网络文件系统及 NFS 的锁存在一些微妙的缺陷。我们不能验证这些报告。但是因为网络文件系统本身实现锁很困难，所以我们没有理由怀疑这些报告。首先，既然性能不足，建议你不要在网络文件系统中使用 SQLite。但是如果你不得不使用一个网络文件来保存 SQLite 的数据文件，那们考虑采用其他的锁机制来防止本身的文件锁机制出错时发生多个进程同时写一个数据文件的现象。

苹果 MacOSX 预装的 SQLite 版本已经扩展拥有一种可供选择的锁策略可以工作在苹果支持的所有网络文件系统上。这些苹果使用的扩展在多个进程在同时访问数据库文件时工作得很好。不幸的是，这些锁机制并不互相排斥，如果一个进程使用 AFP 锁去访问文件，而另一个进程（或许是另一台机器）使用 dot-file 锁去访问这个文件，那么这二个进程可能发生冲突，因为 AFP 锁并不排斥 dot-file 锁，反之亦然。

9.2 不完整的磁盘刷新

SQLite 在 unix 使 fsysnc,在 win32 下面使用 FlushFileBuffers,用来将文件内容同步到磁盘中（节 3.7 及节 3.10）。不幸的是，我们也收到报告，在许多平台上，这二者都没有象广告中宣称的那样工作。我们听说 FlushFileBuffersc 在一些 windows 版本中，可以通过修改注册表，能够完全禁止其工作。我们也被告之，Linux 的一些早先版本，他们的一些文件系统中的 fsync 完全是一个空操作。即使是 FlushFileBuffers 及 fsync 被告之可以工作的系统中，IDE 硬盘经常会撒谎说数据已经写入到盘片中，其实还只是存在状态可变的磁盘控制器缓存中。

在 Mac 你可设置下面项：

PRAGMA fullfsync=ON;

在 Mac 上设置 fullfsync 能够保证数据通过 flush 会真实的写入到盘片中。但 fullfsync 会导致磁盘控制进行重设。这并不是一般意义上的慢，它还会导致其他磁盘 IO 降速，所以此项配置并不推荐。

9.3 文件部分地删除

SQLite 假设从用户进程角度来看是一个原子操作。当删除过程中发生掉电，当电源恢复之后，SQLite 希望看到文件要么完整的存在，要么根本找不到了。如果操作系统不能做到这一点，那事务就可能不是原子性的了。

9.4 写入到文件中的垃圾

SQLite 的数据文件是一种普通的磁盘文件，可以由普通用户进行读写。一些流氓进程可能会打开一个 SQLite 文件，并在其中写入一些混乱的数据。混乱的数据也可能由于操作系统的 BUG 而写入到一个 SQLite 的数据文件中。对于这些情况，SQLite 无能为力。

9.5 删除掉或更名了“hot”日志文件

如果掉电或系统崩溃导致留下了一个”hot”日志文件在磁盘上。实际上，原来的数据文件再加上留下来的“hot“日志文件， 是 SQLite 下回打开时发生回滚使用的，这可以恢复 SQLite 数据的正常状态（节 4.2）。SQLite 会在数据库所在同一目录下用打开的文件名来寻找可能存在的”hot”日志文件。如果数据文件或者日志文件被移动或者改名，或者删除掉了，那么这些日志文件将不会被回滚，数据库也就可能损坏，无法使用了。

我们常怀疑 SQLite 发生的恢复失败的例子是这样的：停电了，之后电又恢复了。一个好心的用户或者系统管理管理员开始查看磁盘损坏。他们看到名为"important.data"数据库文件，或许类似的文件。但由于停电，这里也同样有一个日志文件名为"important.data-journal".这个用户删除了这个“hot”日志文件，认为他是清理系统。那于这种情况，除了进行用户培训，没有其他办法。

如果有多个联接（硬或者符号联接）指向一个数据文件，这个日志文件会以被打开的联接文件名相关来创建的。如果系统崩溃之后，数据库以一个新的联接重新打开，这个“hot”日志文件就不会被找到，数据也不会发生回滚。

有时，电源问题会导致文件系统出现毛病，如最新修改的文件名被丢失了，并会转移至类似于"/lost+found"这样的目录中。当这种情况发生的时候，这个 hot 日志文件就不会被找到，同样恢复也不会发生。SQLite 在同步一个日志文件时通过打开并同步日志文件所在目录来尝试阻止这类事件发生。然后，转移文件到"/lost+found"可能会由不相关的其他进程在相同的目录中产生与主数据库文件名相同的不相关文件。既然这都是 SQLite 所无法控制，所以 SQLite 没有什么好办法。如果你运行在一种易导致名称空间冲突的文件系统上，那么你最好把每一个 SQLite 的数据文件放在你私有的子目录中。

10.0 总结及未来的路

即使到了现在，还是有人发现了一些关于原子提交机制失败模式，开发者不得不为此做一些补丁。这样的事情发生得越来越少了，失败模型也变得越来越模糊了。但如果就认为 SQLite 的原子提交逻辑是没有任何 bug，那是相当愚昧的。开发者承诺将尽可能快的修复被发现的 bug。

开发者同时在考虑新的优化提交机制的办法。当前的 linux,macOSX,win32 的 VFS 实 现使用这些系统之上的一些悲观设定。或许在与一些了解这些系统如何工作的专家交流之后，我们或许可能放松一些这些系统上的设定，使其跑

得更快些。特别的， 我们怀疑的大部分现代文件系统现在已经展现安全追加特性，或许他们都已经支持了扇区的原子操作。但是除非这些得到明确，SQLite 仍将采用更安全、保守的方法，作最坏的打算。

SQLite 的查询优化

SQLite 是个典型的嵌入式 DBMS，它有很多优点，它是轻量级的，在编译之后很小，其中一个原因就是在查询优化方面比较简单，它只是运用索引机制来进行优化的，经过对 SQLite 的查询优化的分析以及对源代码的研究，我将 SQLite 的查询优总结如下：

一、影响查询性能的因素：

1. 对表中行的检索数目，越小越好

2. 排序与否。

3. 是否要对一个索引。

4. 查询语句的形式

二、几个查询优化的转换

1. 对于单个表的单个列而言，如果都有形如 T.C=expr 这样的子句，并且都是用 OR 操作符连接起来，形如： x = expr1 OR expr2 = x OR x = expr3 此时由于对于 OR，在 SQLite 中不能利用索引来优化，所以可以将它转换成带有 IN 操作符的子句：x IN(expr1,expr2,expr3)这样就可以用索引进行优化，效果很明显，但是如果在都没有索引的情况下 OR 语句执行效率会稍优于 IN 语句的效率。

2. 如果一个子句的操作符是 BETWEEN，在 SQLite 中同样不能用索引进行优化，所以也要进行相应的等价转换： 如：a BETWEEN b AND c 可以转换成：(a BETWEEN b AND c) AND (a>=b) AND (a<=c)。在上面这个子句中， (a>=b) AND (a<=c)将被设为 dynamic 且是(a BETWEEN b AND c)的子句，那么如果 BETWEEN 语句已经编码,那么子句就忽略不计,如果存在可利用的 index 使得子句已经满足条件，那么父句则被忽略。

3. 如果一个单元的操作符是 LIKE，那么将做下面的转换：x LIKE ‘abc%’，转换成：x>=‘abc’ AND x<‘abd’。因为在 SQLite 中的 LIKE 是不能用索引进行优化的，所以如果存在索引的话，则转换后和不转换相差很远，因为对 LIKE 不起作用，但如果不存在索引，那么 LIKE 在效率方面也还是比不上转换后的效率的。

三、 几种查询语句的处理（复合查询）

1. 查询语句为：<SelectA> <operator> <selectB> ORDER BY <orderbylist> ORDER BY
执行方法： is one of UNION ALL, UNION, EXCEPT, or INTERSECT. 这个语句的执行过程是先将 selectA 和 selectB 执行并且排序，再对两个结果扫描处理，对上面四种操作是不同的，将执行过程分成七个子过程：

outA: 将 selectA 的结果的一行放到最终结果集中

outB: 将 selectA 的结果的一行放到最终结果集中(只有 UNION 操作和 UNION ALL 操作，其它操作都不放入最终结果集中)

AltB: 当 selectA 的当前记录小于 selectB 的当前记录

AeqB: 当 selectA 的当前记录等于 selectB 的当前记录

AgtB: 当 selectA 的当前记录大于 selectB 的当前记录

EofA: 当 selectA 的结果遍历完

EofB: 当 selectB 的结果遍历完

下面就是四种操作的执行过程：

执行顺序	UNION ALL	UNION	EXCEPT	INTERSECT
AltB:	outA, nextA	outA, nextA	outA,nextA	nextA
AeqB:	outA, nextA	nextA	nextA	outA, nextA
AgtB:	outB, nextB	outB, nextB	nextB	nextB
EofA:	outB, nextB	outB, nextB	halt	halt
EofB:	outA, nextA	outA, nextA	outA,nextA	halt

2. 如果可能的话，可以把一个用到 GROUP BY 查询的语句转换成 DISTINCT 语句来查询，因为 GROUP BY 有时候可能会用到 index，而对于 DISTINCT 都不会用到索引的 。

四、子查询扁平化

例子：SELECT a FROM (SELECT x+y AS a FROM t1 WHERE z<100) WHERE a>5

对这个 SQL 语句的执行一般默认的方法就是先执行内查询，把结果放到一个临时表中，再对这个表进行外部查询，这就要对数据处理两次，另外这个临时表没有索引，所以对外部查询就不能进行优化了，如果对上面的 SQL 进行处理后可以得到如下 SQL 语句：SELECT x+y AS a FROM t1 WHERE z<100 AND a>5，这个结果显然和上面的一样，但此时只需要对

数据进行查询一次就够了，另外如果在表 t1 上有索引的话就避免了遍历整个表。

运用 flatten 方法优化 SQL 的条件：

1.子查询和外查询没有都用集函数

2.子查询没有用集函数或者外查询不是个表的连接

3.子查询不是一个左外连接的右操作数

4.子查询没有用 DISTINCT 或者外查询不是个表的连接

5.子查询没有用 DISTINCT 或者外查询没有用集函数

6.子查询没有用集函数或者外查询没有用关键字 DISTINCT

7.子查询有一个 FROM 语句

8.子查询没有用 LIMIT 或者外查询不是表的连接

9.子查询没有用 LIMIT 或者外查询没有用集函数

10.子查询没有用集函数或者外查询没用 LIMIT

11.子查询和外查询不是同时是 ORDER BY 子句

12.子查询和外查询没有都用 LIMIT

13.子查询没有用 OFFSET

14.外查询不是一个复合查询的一部分或者子查询没有同时用关键字 ORDER BY 和 LIMIT

15.外查询没有用集函数子查询不包含 ORDER BY

16.复合子查询的扁平化：子查询不是一个复合查询，或者他是一个 UNION ALL 复合查询，但他是都由若干个非集函数的查询构成，他的父查询不是一个复合查询的子查询，也没有用集函数或者是 DISTINCT 查询，并且在 FROM 语句中没有其它的表或者子查询，父查询和子查询可能会包含 WHERE 语句，这些都会受到上面 11、12、13 条件的限制。

例：

```
SELECT a+1 FROM (
                    SELECT x FROM tab
                    UNION ALL
                    SELECT y FROM tab
                      UNION ALL
                      SELECT abs(z*2) FROM tab2
                ) WHERE a!=5 ORDER BY 1
```

转换为：

```
SELECT x+1 FROM tab WHERE x+1!=5
     UNION ALL
     SELECT y+1 FROM tab WHERE y+1!=5
     UNION ALL
     SELECT abs(z*2)+1 FROM tab2 WHERE abs(z*2)+1!=5
     ORDER BY 1
```

17.如果子查询是一个复合查询，那么父查询的所有的 ORDER BY 语句必须是对子查询的列的简单引用

18.子查询没有用 LIMIT 或者外查询不具有 WHERE 语句

子查询扁平化是由专门一个函数实现的，函数为：

```
static int flattenSubquery(
  Parse *pParse,              /* Parsing context */
  Select *p,                      /* The parent or outer SELECT statement */
  int iFrom,                      /* Index in p->pSrc->a[] of the inner subquery */
  int isAgg,                      /* True if outer SELECT uses aggregate functions */
  int subqueryIsAgg       /* True if the subquery uses aggregate functions */
)
```

它是在 Select.c 文件中实现的。显然对于一个比较复杂的查询，如果满足上面的条件时对这个查询语句进行扁平化处理后就可以实现对查询的优化。如果正好存在索引的话效果会更好！

五、连接查询

在返回查询结果之前，相关表的每行必须都已经连接起来，在 SQLite 中，这是用嵌套循环实现的，在早期版本中，最左边的是最外层循环，最右边的是最内层循环，连接两个或者更多的表时，如果有索引则放到内层循环中，也就是放到 FROM 最后面，因为对于前面选中的每行，找后面与之对应的行时，如果有索引则会很快，如果没有则要遍历整个表，这样效率就很低，但在新版本中，这个优化已经实现。

优化的方法如下：

对要查询的每个表，统计这个表上的索引信息，首先将代价赋值为 SQLITE_BIG_DBL（一个系统已经定义的常量）：

1) 如果没有索引，则找有没有在这个表上对 rowid 的查询条件：

1. 如果有 Rowid=EXPR，如果有的话则返回对这个表代价估计，代价计为零，查询得到的记录数为 1，并完成对这个表的代价估计，

2. 如果没有 Rowid=EXPR 但有 rowid IN (...)，而 IN 是一个列表，那么记录返回记录数为 IN 列表中元素的个数，估计代价为 NlogN,

3. 如果 IN 不是一个列表而是一个子查询结果，那么由于具体这个子查询不能确定，所以只能估计一个值，返回记录数为 100，代价为 200。

4. 如果对 rowid 是范围的查询，那么就估计所有符合条件的记录是总记录的三分之一，总记录估计为 1000000，并且估计代价也为记录数。

5．如果这个查询还要求排序，则再另外加上排序的代价 NlogN

6．如果此时得到的代价小于总代价，那么就更新总代价，否则不更新。

2)　　如果 WHERE 子句中存在 OR 操作符，那么要把这些 OR 连接的所有子句分开再进行分析。

1．　如果有子句是由 AND 连接符构成，那么再把由 AND 连接的子句再分别分析。

2．　如果连接的子句的形式是 X<op><expr>，那么就再分析这个子句。

3．　接下来就是把整个对 OR 操作的总代价计算出来。

4．　如果这个查询要求排序，则再在上面总代价上再乘上排序代价 NlogN

5．　如果此时得到的代价小于总代价，那么就更新总代价，否则不更新。

3)　　如果有索引，则统计每个表的索引信息，对于每个索引：

1．　先找到这个索引对应的列号，再找到对应的能用到（操作符必须为=或者是 IN（…））这个索引的 WHERE 子句，如果没有找到，则退出对每个索引的循环，如果找到，则判断这个子句的操作符是什么，如果是=，那么没有附加的代价，如果是 IN（sub-select），那么估计它附加代价 inMultiplier 为 25，如果是 IN（list），那么附加代价就是 N（N 为 list 的列数）。

2．　再计算总的代价和总的查询结果记录数和代价。

3．　nRow = pProbe->aiRowEst[i] * inMultiplier;/*计算行数*/

4．　cost = nRow * estLog(inMultiplier);/*统计代价*/

5．　如果找不到操作符为=或者是 IN（…）的子句，而是范围的查询，那么同样只好估计查询结果记录数为 nRow/3，估计代价为 cost/3。

6．　同样，如果此查询要求排序的话，再在上面的总代价上加上 NlogN

7．　如果此时得到的代价小于总代价，那么就更新总代价，否则不更新。

4)　　通过上面的优化过程，可以得到对一个表查询的总代价（就是上面各个代价的总和），再对第二个表进行同样的操作，这样如此直到把 FROM 子句中所有的表都计算出各自的代价，最后取最小的，这将作为嵌套循环的最内层，依次可以得到整个嵌套循环的嵌套顺序，此时正是最优的，达到了优化的目的。

5)　　所以循环的嵌套顺序不一定是与 FROM 子句中的顺序一致，因为在执行过程中会用索引优化来重新排列顺序。

六、索引

在 SQLite 中，有以下几种索引：

1)　　单列索引

2)　　多列索引

3)　　唯一性索引

4)　　对于声明为：INTEGER PRIMARY KEY 的主键来说，这列会按默认方式排序，所以虽然在数据字典中没有对它生成索引，但它的功能就像个索引。所以如果在这个主键上在单独建立索引的话，这样既浪费空间也没有任何好处。

运用索引的注意事项：

1)　　对于一个很小的表来说没必要建立索引

2)　　在一个表上如果经常做的是插入更新操作，那么就要节制使用索引

3)　　也不要在一个表上建立太多的索引，如果建立太多的话那么在查询的时候 SQLite 可能不会选择最好的来执行查询，一个解决办法就是建立聚簇索引

索引的运用时机：

1)　　操作符：=、>、<、IN 等

2)　　操作符 BETWEEN、LIKE、OR 不能用索引，

如 BETWEEN：SELECT * FROM mytable WHERE myfield BETWEEN 10 and 20;

这时就应该将其转换成：

SELECT * FROM mytable WHERE myfield >= 10 AND myfield <= 20;

此时如果在 myfield 上有索引的话就可以用了，大大提高速度

再如 LIKE：SELECT * FROM mytable WHERE myfield LIKE 'sql%';

此时应该将它转换成：

SELECT * FROM mytable WHERE myfield >= 'sql' AND myfield < 'sqm';

此时如果在 myfield 上有索引的话就可以用了，大大提高速度

再如 OR：SELECT * FROM mytable WHERE myfield = 'abc' OR myfield = 'xyz';

此时应该将它转换成：

SELECT * FROM mytable WHERE myfield IN ('abc', 'xyz');

此时如果在 myfield 上有索引的话就可以用了，大大提高速度

3)　　有些时候索引都是不能用的，这时就应该遍历全表（程序演示）

```
SELECT * FROM mytable WHERE myfield % 2 = 1;

SELECT * FROM mytable WHERE substr(myfield, 0, 1) = 'w';

SELECT * FROM mytable WHERE length(myfield) < 5;
```

SQLITE3 使用总结

前序：

Sqlite3 的确很好用。小巧、速度快。但是因为非微软的产品，帮助文档总觉得不够。这些天再次研究它，又有一些收获，这里把我对 sqlite3 的研究列出来，以备忘记。

这里要注明，我是一个跨平台专注者，并不喜欢只用 windows 平台。我以前的工作就是为 unix 平台写代码。下面我所写的东西，虽然没有验证，但是我已尽量不使用任何 windows 的东西，只使用标准 C 或标准 C++。但是，我没有尝试过在别的系统、别的编译器下编译，因此下面的叙述如果不正确，则留待以后修改。

下面我的代码仍然用 VC 编写，因为我觉得 VC 是一个很不错的 IDE，可以加快代码编写速度（例如配合 Vassist ）。下面我所说的编译环境，是 VC2003。如果读者觉得自己习惯于 unix 下用 vi 编写代码速度较快，可以不用管我的说明，只需要符合自己习惯即可，因为我用的是标准 C 或 C++ 。不会给任何人带来不便。

一、 版本

从 www.sqlite.org 网站可下载到最新的 sqlite 代码和编译版本。我写此文章时，最新代码是 3.3.17 版本。

很久没有去下载 sqlite 新代码，因此也不知道 sqlite 变化这么大。以前很多文件，现在全部合并成一个 sqlite3.c 文件。如果单独用此文件，是挺好的，省去拷贝一堆文件还担心有没有遗漏。但是也带来一个问题：此文件太大，快接近 7 万行代码，VC 开它整个机器都慢下来了。如果不需要改它代码，也就不需要打开 sqlite3.c 文件，机器不会慢。但是，下面我要写通过修改 sqlite 代码完成加密功能，那时候就比较痛苦了。如果个人水平较高，建议用些简单的编辑器来编辑，例如 UltraEdit 或 Notepad 。速度会快很多。

二、 基本编译

这个不想多说了，在 VC 里新建 dos 控制台空白工程，把 sqlite3.c 和 sqlite3.h 添加到工程，再新建一个 main.cpp 文件。在里面写：

```
extern "C"
{
    #include "./sqlite3.h"
};
int main( int , char** )
{
    return 0;
}
```

为什么要 extern “C” ？如果问这个问题，我不想说太多，这是 C++的基础。要在 C++ 里使用一段 C 的代码，必须要用 extern “C” 括起来。C++跟 C 虽然语法上有重叠，但是它们是两个不同的东西，内存里的布局是完全不同的，在 C++编译器里不用 extern “C”括起 C 代码，会导致编译器不知道该如何为 C 代码描述内存布局。

可能在 sqlite3.c 里人家已经把整段代码都 extern “C” 括起来了，但是你遇到一个 .c 文件就自觉的再括一次，也没什么不好。

基本工程就这样建立起来了。编译，可以通过。但是有一堆的 warning。可以不管它。

三、 SQLITE 操作入门

sqlite 提供的是一些 C 函数接口，你可以用这些函数操作数据库。通过使用这些接口，传递一些标准 sql 语句（以 char * 类型）给 sqlite 函数，sqlite 就会为你操作数据库。

sqlite 跟 MS 的 access 一样是文件型数据库，就是说，一个数据库就是一个文件，此数据库里可以建立很多的表，可以建立索引、触发器等等，但是，它实际上得到的就是一个文件。备份这个文件就备份了整个数据库。

sqlite 不需要任何数据库引擎，这意味着如果你需要 sqlite 来保存一些用户数据，甚至都不需要安装数据库(如果你做个小软件还要求人家必须装了 sqlserver 才能运行，那也太黑心了)。

下面开始介绍数据库基本操作。

（1） 基本流程

i.1 关键数据结构

sqlite 里最常用到的是 sqlite3 * 类型。从数据库打开开始，sqlite 就要为这个类型准备好内存，直到数据库关闭，整个过程都需要用到这个类型。当数据库打开时开始，这个类型的变量就代表了你要操作的数据库。下面再详细介绍。

i.2 打开数据库

```
int sqlite3_open( 文件名, sqlite3 ** );
```

用这个函数开始数据库操作。

需要传入两个参数，一是数据库文件名，比如：c:\\DongChunGuang_Database.db。

文件名不需要一定存在，如果此文件不存在，sqlite 会自动建立它。如果它存在，就尝试把它当数据库文件来打开。

sqlite3 ** 参数即前面提到的关键数据结构。这个结构底层细节如何，你不要关它。

函数返回值表示操作是否正确，如果是 SQLITE_OK 则表示操作正常。相关的返回值 sqlite 定义了一些宏。具体这些宏的含义可以参考 sqlite3.h 文件。里面有详细定义（顺便说一下，sqlite3 的代码注释率自称是非常高的，实际上也的确很高。只要你会看英文，sqlite 可以让你学到不少东西）。

下面介绍关闭数据库后，再给一段参考代码。

i.3 关闭数据库

```
int sqlite3_close(sqlite3 *);
```

前面如果用 sqlite3_open 开启了一个数据库，结尾时不要忘了用这个函数关闭数据库。

下面给段简单的代码：

```
extern "C"
{
    #include "./sqlite3.h"
};
int main( int , char** )
{
        sqlite3 * db = NULL; //声明 sqlite 关键结构指针
        int result;

    //打开数据库
```

```
    //需要传入 db 这个指针的指针，因为 sqlite3_open 函数要为这个指针分配内存，还要让 db 指针指向这个内存区
        result = sqlite3_open( “c:\\Dcg_database.db”, &db );
        if( result != SQLITE_OK )
        {
            //数据库打开失败
          return -1;
        }
        //数据库操作代码
        //…

        //数据库打开成功
        //关闭数据库
        sqlite3_close( db );
        return 0;
    }
```

这就是一次数据库操作过程。

（2） SQL 语句操作

本节介绍如何用 sqlite 执行标准 sql 语法。

i.1 执行 sql 语句

```
int sqlite3_exec(sqlite3*, const char *sql, sqlite3_callback, void *,   char **errmsg );
```

这就是执行一条 sql 语句的函数。

第 1 个参数不再说了，是前面 open 函数得到的指针。说了是关键数据结构。

第 2 个参数 const char *sql 是一条 sql 语句，以\0 结尾。

第 3 个参数 sqlite3_callback 是回调，当这条语句执行之后，sqlite3 会去调用你提供的这个函数。（什么是回调函数，自己找别的资料学习）

第 4 个参数 void * 是你所提供的指针，你可以传递任何一个指针参数到这里，这个参数最终会传到回调函数里面，如果不需要传递指针给回调函数，可以填 NULL。等下我们再看回调函数的写法，以及这个参数的使用。

第 5 个参数 char ** errmsg 是错误信息。注意是指针的指针。sqlite3 里面有很多固定的错误信息。执行 sqlite3_exec 之后，执行失败时可以查阅这个指针（直接 printf(“%s\n”,errmsg)）得到一串字符串信息，这串信息告诉你错在什么地方。sqlite3_exec 函数通过修改你传入的指针的指针，把你提供的指针指向错误提示信息，这样 sqlite3_exec 函数外面就可以通过这个 char*得到具体错误提示。

说明：通常，sqlite3_callback 和它后面的 void * 这两个位置都可以填 NULL。填 NULL 表示你不需要回调。比如你做 insert 操作，做 delete 操作，就没有必要使用回调。而当你做 select 时，就要使用回调，因为 sqlite3 把数据查出来，得通过回调告诉你查出了什么数据。

i.2 exec 的回调

```
typedef int (*sqlite3_callback)(void*,int,char**, char**);
```

你的回调函数必须定义成上面这个函数的类型。下面给个简单的例子：

```
//sqlite3 的回调函数
// sqlite 每查到一条记录，就调用一次这个回调
        int LoadMyInfo( void * para, int n_column, char ** column_value, char ** column_name )
{
```

```
                        //para 是你在 sqlite3_exec 里传入的 void * 参数
                        //通过 para 参数，你可以传入一些特殊的指针（比如类指针、结构指针），然后在这里面强制转换成对应的类型（这里面是 void*类型，必须强制转换成你的类型才可用）。然后操作这些数据
                        //n_column 是这一条记录有多少个字段 (即这条记录有多少列)
                        // char ** column_value 是个关键值，查出来的数据都保存在这里，它实际上是个 1 维数组（不要以为是 2 维数组），每一个元素都是一个 char * 值，是一个字段内容（用字符串来表示，以\0 结尾）
                        //char ** column_name 跟 column_value 是对应的，表示这个字段的字段名称

                        //这里，我不使用 para 参数。忽略它的存在.

                        int i;
        printf( "记录包含 %d 个字段\n", n_column );
        for( i = 0 ; i < n_column; i ++ )
        {
                printf( "字段名:%s    <-> 字段值:%s\n",    column_name[i], column_value[i] );
        }
        printf( "------------------\n" );
        return 0;
    }

    int main( int , char ** )
    {
                    sqlite3 * db;
                     int result;
                     char * errmsg = NULL;

                     result = sqlite3_open( "c:\\Dcg_database.db", &db );
           if( result != SQLITE_OK )
           {
                //数据库打开失败
            return -1;
        }
        //数据库操作代码
        //创建一个测试表，表名叫 MyTable_1，有 2 个字段： ID 和 name。其中 ID 是一个自动增加的类型，以后 insert 时可以不去指定这个字段，它会自己从 0 开始增加
        result = sqlite3_exec( db, "create table MyTable_1( ID integer primary key autoincrement, name nvarchar(32) )", NULL, NULL, errmsg );
        if(result != SQLITE_OK )
        {
                printf( "创建表失败，错误码:%d，错误原因:%s\n", result, errmsg );
        }
        //插入一些记录
        result = sqlite3_exec( db, "insert into MyTable_1( name ) values ( '走路' )", 0, 0, errmsg );
        if(result != SQLITE_OK )
        {
                printf( "插入记录失败，错误码:%d，错误原因:%s\n", result, errmsg );
        }
```

```
        result = sqlite3_exec( db, "insert into MyTable_1( name ) values ( '骑单车' )", 0, 0, errmsg );
        if(result != SQLITE_OK )
        {
                printf( "插入记录失败，错误码:%d，错误原因:%s\n", result, errmsg );
        }
        result = sqlite3_exec( db, "insert into MyTable_1( name ) values ( '坐汽车' )", 0, 0, errmsg );
        if(result != SQLITE_OK )
        {
                printf( "插入记录失败，错误码:%d，错误原因:%s\n", result, errmsg );
        }


        //开始查询数据库
        result = sqlite3_exec( db, "select * from MyTable_1", LoadMyInfo, NULL, errmsg );

        //关闭数据库
        sqlite3_close( db );
        return 0;
    }
```

通过上面的例子，应该可以知道如何打开一个数据库，如何做数据库基本操作。

有这些知识，基本上可以应付很多数据库操作了。

i.3 不使用回调查询数据库

上面介绍的 sqlite3_exec 是使用回调来执行 select 操作。还有一个方法可以直接查询而不需要回调。但是，我个人感觉还是回调好，因为代码可以更加整齐，只不过用回调很麻烦，你得声明一个函数，如果这个函数是类成员函数，你还不得不把它声明成 static 的（要问为什么？这又是 C++基础了。C++成员函数实际上隐藏了一个参数：this，C++调用类的成员函数的时候，隐含把类指针当成函数的第一个参数传递进去。结果，这造成跟前面说的 sqlite 回调函数的参数不相符。只有当把成员函数声明成 static 时，它才没有多余的隐含的 this 参数）。

虽然回调显得代码整齐，但有时候你还是想要非回调的 select 查询。这可以通过 sqlite3_get_table 函数做到。

```
int sqlite3_get_table(sqlite3*, const char *sql, char ***resultp, int *nrow, int *ncolumn, char **errmsg );
```

第 1 个参数不再多说，看前面的例子。

第 2 个参数是 sql 语句,跟 sqlite3_exec 里的 sql 是一样的。是一个很普通的以\0 结尾的 char *字符串。

第 3 个参数是查询结果，它依然一维数组（不要以为是二维数组，更不要以为是三维数组）。它内存布局是：第一行是字段名称，后面是紧接着是每个字段的值。下面用例子来说事。

第 4 个参数是查询出多少条记录（即查出多少行）。

第 5 个参数是多少个字段（多少列）。

第 6 个参数是错误信息，跟前面一样，这里不多说了。

下面给个简单例子:

```
int main( int , char ** )
{
                sqlite3 * db;
                int result;
                char * errmsg = NULL;
    char **dbResult; //是 char ** 类型，两个*号
```

```
                    int nRow, nColumn;
                     int i , j;
                     int index;

                     result = sqlite3_open( “c:\\Dcg_database.db”, &db );
           if( result != SQLITE_OK )
           {
                 //数据库打开失败
            return -1;
        }
        //数据库操作代码
        //假设前面已经创建了 MyTable_1 表
        //开始查询，传入的 dbResult 已经是 char **，这里又加了一个 & 取地址符，传递进去的就成了 char ***
        result = sqlite3_get_table( db, “select * from MyTable_1”, &dbResult, &nRow, &nColumn, &errmsg );
        if( SQLITE_OK == result )
        {
                 //查询成功

               index = nColumn; //前面说过 dbResult 前面第一行数据是字段名称，从 nColumn 索引开始才是真正的数据
                 printf( “查到%d 条记录\n”, nRow );
                 for(   i = 0; i < nRow ; i++ )
                 {
                         printf( “第 %d 条记录\n”, i+1 );
                         for( j = 0 ; j < nColumn; j++ )
                         {
                                  printf( “字段名:%s    <-> 字段值:%s\n”,    dbResult[j], dbResult [index] );
                                  ++index; // dbResult 的字段值是连续的，从第 0 索引到第 nColumn - 1 索引都是字段名称，从第 nColumn 索引开始，后面都是字段值，它把一个二维的表（传统的行列表示法）用一个扁平的形式来表示
                         }
                         printf( “-------\n” );
                 }
        }

        //到这里，不论数据库查询是否成功，都释放 char** 查询结果，使用 sqlite 提供的功能来释放
        sqlite3_free_table( dbResult );

        //关闭数据库
        sqlite3_close( db );
        return 0;
    }
```

到这个例子为止，sqlite3 的常用用法都介绍完了。

用以上的方法，再配上 sql 语句，完全可以应付绝大多数数据库需求。

但有一种情况，用上面方法是无法实现的：需要 insert、select 二进制。当需要处理二进制数据时，上面的方法就没办法做到。下面这一节说明如何插入二进制数据

（2） 操作二进制

sqlite 操作二进制数据需要用一个辅助的数据类型：sqlite3_stmt * 。

这个数据类型记录了一个“sql 语句”。为什么我把 “sql 语句” 用双引号引起来？因为你可以把 sqlite3_stmt * 所表示的内容看成是 sql 语句，但是实际上它不是我们所熟知的 sql 语句。它是一个已经把 sql 语句解析了的、用 sqlite 自己标记记录的内部数据结构。

正因为这个结构已经被解析了，所以你可以往这个语句里插入二进制数据。当然，把二进制数据插到 sqlite3_stmt 结构里可不能直接 memcpy ，也不能像 std::string 那样用 + 号。必须用 sqlite 提供的函数来插入。

i.1 写入二进制

下面说写二进制的步骤。

要插入二进制，前提是这个表的字段的类型是 blob 类型。我假设有这么一张表：

```
create table Tbl_2( ID integer, file_content   blob )
```

首先声明

```
    sqlite3_stmt * stat;
```

然后，把一个 sql 语句解析到 stat 结构里去：

```
    sqlite3_prepare( db, “insert into Tbl_2( ID, file_content) values( 10, ? )”, -1, &stat, 0 );
```

上面的函数完成 sql 语句的解析。第一个参数跟前面一样，是个 sqlite3 * 类型变量，第二个参数是一个 sql 语句。

这个 sql 语句特别之处在于 values 里面有个 ? 号。在 sqlite3_prepare 函数里，?号表示一个未定的值，它的值等下才插入。

第三个参数我写的是-1，这个参数含义是前面 sql 语句的长度。如果小于 0，sqlite 会自动计算它的长度（把 sql 语句当成以\0 结尾的字符串）。

第四个参数是 sqlite3_stmt 的指针的指针。解析以后的 sql 语句就放在这个结构里。

第五个参数我也不知道是干什么的。为 0 就可以了。

如果这个函数执行成功（返回值是 SQLITE_OK 且 stat 不为 NULL ），那么下面就可以开始插入二进制数据。

```
    sqlite3_bind_blob( stat, 1, pdata, (int)(length_of_data_in_bytes), NULL ); // pdata 为数据缓冲区，length_of_data_in_bytes 为数据大小，以字节为单位
```

这个函数一共有 5 个参数。

第 1 个参数：是前面 prepare 得到的 sqlite3_stmt * 类型变量。

第 2 个参数：?号的索引。前面 prepare 的 sql 语句里有一个?号，假如有多个?号怎么插入？方法就是改变 bind_blob 函数第 2 个参数。这个参数我写 1，表示这里插入的值要替换 stat 的第一个?号（这里的索引从 1 开始计数，而非从 0 开始）。如果你有多个?号，就写多个 bind_blob 语句，并改变它们的第 2 个参数就替换到不同的?号。如果有?号没有替换，sqlite 为它取值 null。

第 3 个参数：二进制数据起始指针。

第 4 个参数：二进制数据的长度，以字节为单位。

第 5 个参数：是个析够回调函数，告诉 sqlite 当把数据处理完后调用此函数来析够你的数据。这个参数我还没有使用过，因此理解也不深刻。但是一般都填 NULL，需要释放的内存自己用代码来释放。

bind 完了之后，二进制数据就进入了你的“sql 语句”里了。你现在可以把它保存到数据库里：

```
    int result = sqlite3_step( stat );
```

通过这个语句，stat 表示的 sql 语句就被写到了数据库里。

最后，要把 sqlite3_stmt 结构给释放：

```
    sqlite3_finalize( stat ); //把刚才分配的内容析构掉
```

i.2 读出二进制

下面说读二进制的步骤。

跟前面一样，先声明 sqlite3_stmt * 类型变量：

```
sqlite3_stmt * stat;
```

然后，把一个 sql 语句解析到 stat 结构里去：

```
sqlite3_prepare( db, “select * from Tbl_2”, -1, &stat, 0 );
```

当 prepare 成功之后（返回值是 SQLITE_OK ），开始查询数据。

```
int result = sqlite3_step( stat );
```

这一句的返回值是 SQLITE_ROW 时表示成功（不是 SQLITE_OK ）。

你可以循环执行 sqlite3_step 函数，一次 step 查询出一条记录。直到返回值不为 SQLITE_ROW 时表示查询结束。

然后开始获取第一个字段：ID 的值。ID 是个整数，用下面这个语句获取它的值：

```
int id = sqlite3_column_int( stat, 0 ); //第 2 个参数表示获取第几个字段内容，从 0 开始计算，因为我的表的 ID 字段是第一个字段，因此这里我填 0
```

下面开始获取 file_content 的值，因为 file_content 是二进制，因此我需要得到它的指针，还有它的长度：

```
const void * pFileContent = sqlite3_column_blob( stat, 1 );
int len = sqlite3_column_bytes( stat, 1 );
```

这样就得到了二进制的值。

把 pFileContent 的内容保存出来之后，不要忘了释放 sqlite3_stmt 结构：

```
sqlite3_finalize( stat ); //把刚才分配的内容析构掉
```

i.3 重复使用 sqlite3_stmt 结构

如果你需要重复使用 sqlite3_prepare 解析好的 sqlite3_stmt 结构，需要用函数： sqlite3_reset。

```
result = sqlite3_reset(stat);
```

这样， stat 结构又成为 sqlite3_prepare 完成时的状态，你可以重新为它 bind 内容。

（4） 事务处理

sqlite 是支持事务处理的。如果你知道你要同步删除很多数据，不仿把它们做成一个统一的事务。

通常一次 sqlite3_exec 就是一次事务，如果你要删除 1 万条数据，sqlite 就做了 1 万次：开始新事务->删除一条数据->提交事务->开始新事务->… 的过程。这个操作是很慢的。因为时间都花在了开始事务、提交事务上。

你可以把这些同类操作做成一个事务，这样如果操作错误，还能够回滚事务。

事务的操作没有特别的接口函数，它就是一个普通的 sql 语句而已：

分别如下：

```
int result;
result = sqlite3_exec( db, "begin transaction", 0, 0, &zErrorMsg ); //开始一个事务
result = sqlite3_exec( db, "commit transaction", 0, 0, &zErrorMsg ); //提交事务
result = sqlite3_exec( db, "rollback transaction", 0, 0, &zErrorMsg ); //回滚事务
```

四、 给数据库加密

前面所说的内容网上已经有很多资料，虽然比较零散，但是花点时间也还是可以找到的。现在要说的这

个——数据库加密，资料就很难找。也可能是我操作水平不够，找不到对应资料。但不管这样，我还是通过网上能找到的很有限的资料，探索出了给 sqlite 数据库加密的完整步骤。

这里要提一下，虽然 sqlite 很好用，速度快、体积小巧。但是它保存的文件却是明文的。若不信可以用 NotePad 打开数据库文件瞧瞧，里面 insert 的内容几乎一览无余。这样赤裸裸的展现自己，可不是我们的初衷。当然，如果你在嵌入式系统、智能手机上使用 sqlite，最好是不加密，因为这些系统运算能力有限，你做为一个新功能提供者，不能把用户有限的运算能力全部花掉。

Sqlite 为了速度而诞生。因此 Sqlite 本身不对数据库加密，要知道，如果你选择标准 AES 算法加密，那么一定有接近 50%的时间消耗在加解密算法上，甚至更多(性能主要取决于你算法编写水平以及你是否能使用 cpu 提供的底层运算能力，比如 MMX 或 sse 系列指令可以大幅度提升运算速度)。

Sqlite 免费版本是不提供加密功能的，当然你也可以选择他们的收费版本，那你得支付 2000 块钱，而且是 USD。我这里也不是说支付钱不好，如果只为了数据库加密就去支付 2000 块，我觉得划不来。因为下面我将要告诉你如何为免费的 Sqlite 扩展出加密模块——自己动手扩展，这是 Sqlite 允许，也是它提倡的。

那么，就让我们一起开始为 sqlite3.c 文件扩展出加密模块。

i.1 必要的宏

通过阅读 Sqlite 代码（当然没有全部阅读完，6 万多行代码，没有一行是我习惯的风格，我可没那么多眼神去看），我搞清楚了两件事：

Sqlite 是支持加密扩展的；

需要 #define 一个宏才能使用加密扩展。

这个宏就是 SQLITE_HAS_CODEC。

你在代码最前面（也可以在 sqlite3.h 文件第一行）定义：

```
#ifndef SQLITE_HAS_CODEC
#define SQLITE_HAS_CODEC
#endif
```

如果你在代码里定义了此宏，但是还能够正常编译，那么应该是操作没有成功。因为你应该会被编译器提示有一些函数无法链接才对。如果你用的是 VC 2003，你可以在“解决方案”里右键点击你的工程，然后选“属性”，找到“C/C++”，再找到“命令行”，在里面手工添加“/D "SQLITE_HAS_CODEC"”。

定义了这个宏，一些被 Sqlite 故意屏蔽掉的代码就被使用了。这些代码就是加解密的接口。

尝试编译，vc 会提示你有一些函数无法链接，因为找不到他们的实现。

如果你也用的是 VC2003，那么会得到下面的提示：

```
error LNK2019: 无法解析的外部符号 _sqlite3CodecGetKey ，该符号在函数 _attachFunc 中被引用
error LNK2019: 无法解析的外部符号 _sqlite3CodecAttach ，该符号在函数 _attachFunc 中被引用
error LNK2019: 无法解析的外部符号 _sqlite3_activate_see ，该符号在函数 _sqlite3Pragma 中被引用
error LNK2019: 无法解析的外部符号 _sqlite3_key ，该符号在函数 _sqlite3Pragma 中被引用
fatal error LNK1120: 4 个无法解析的外部命令
```

这是正常的，因为 Sqlite 只留了接口而已，并没有给出实现。

下面就让我来实现这些接口。

i.2 自己实现加解密接口函数

如果真要我从一份 www.sqlite.org 网上 down 下来的 sqlite3.c 文件，直接摸索出这些接口的实现，我认为我还没有这个能力。

好在网上还有一些代码已经实现了这个功能。通过参照他们的代码以及不断编译中 vc 给出的错误提示，最终我把整个接口整理出来。

实现这些预留接口不是那么容易，要重头说一次怎么回事很困难。我把代码都写好了，直接把他们按我下面的说明拷贝到 sqlite3.c 文件对应地方即可。我在下面也提供了 sqlite3.c 文件，可以直接参考或取下来使

用。

这里要说一点的是，我另外新建了两个文件：crypt.c 和 crypt.h。

其中 crypt.h 如此定义：

```
#ifndef   DCG_SQLITE_CRYPT_FUNC_
#define   DCG_SQLITE_CRYPT_FUNC_
/***********
董淳光写的 SQLITE 加密关键函数库
***********/

/***********
关键加密函数
***********/
int My_Encrypt_Func( unsigned char * pData, unsigned int data_len, const char * key, unsigned int len_of_key );

/***********
关键解密函数
***********/
int My_DeEncrypt_Func( unsigned char * pData, unsigned int data_len, const char * key, unsigned int len_of_key );

#endif
```

其中的 crypt.c 如此定义：

```
#include "./crypt.h"
#include "memory.h"
/***********
关键加密函数
***********/
int My_Encrypt_Func( unsigned char * pData, unsigned int data_len, const char * key, unsigned int len_of_key )
{
   return 0;
}

/***********
关键解密函数
***********/
int My_DeEncrypt_Func( unsigned char * pData, unsigned int data_len, const char * key, unsigned int len_of_key )
{
   return 0;
}
```

这个文件很容易看，就两函数，一个加密一个解密。传进来的参数分别是待处理的数据、数据长度、密钥、密钥长度。

处理时直接把结果作用于 pData 指针指向的内容。

你需要定义自己的加解密过程，就改动这两个函数，其它部分不用动。扩展起来很简单。

这里有个特点，data_len 一般总是 1024 字节。正因为如此，你可以在你的算法里使用一些特定长度的

加密算法，比如 AES 要求被加密数据一定是 128 位（16 字节）长。这个 1024 不是碰巧，而是 Sqlite 的页定义是 1024 字节，在 sqlite3.c 文件里有定义:

```
# define SQLITE_DEFAULT_PAGE_SIZE 1024
```

你可以改动这个值，不过还是建议没有必要不要去改它。

上面写了两个扩展函数，如何把扩展函数跟 Sqlite 挂接起来，这个过程说起来比较麻烦。我直接贴代码。分 3 个步骤。

首先，在 sqlite3.c 文件顶部，添加下面内容：

```
#ifdef SQLITE_HAS_CODEC
#include "./crypt.h"
/***********
用于在 sqlite3 最后关闭时释放一些内存
***********/
void sqlite3pager_free_codecarg(void *pArg);
#endif
```

这个函数之所以要在 sqlite3.c 开头声明，是因为下面在 sqlite3.c 里面某些函数里要插入这个函数调用。所以要提前声明。

其次，在 sqlite3.c 文件里搜索“sqlite3PagerClose”函数，要找到它的实现代码（而不是声明代码）。

实现代码里一开始是：

```
#ifdef SQLITE_ENABLE_MEMORY_MANAGEMENT
  /* A malloc() cannot fail in sqlite3ThreadData() as one or more calls to
  ** malloc() must have already been made by this thread before it gets
  ** to this point. This means the ThreadData must have been allocated already
  ** so that ThreadData.nAlloc can be set.
  */
  ThreadData *pTsd = sqlite3ThreadData();
  assert( pPager );
  assert( pTsd && pTsd->nAlloc );
#endif
```

需要在这部分后面紧接着插入：

```
#ifdef SQLITE_HAS_CODEC
  sqlite3pager_free_codecarg(pPager->pCodecArg);
#endif
```

这里要注意，sqlite3PagerClose 函数大概也是 3.3.17 版本左右才改名的，以前版本里是叫“sqlite3pager_close”。因此你在老版本 sqlite 代码里搜索“sqlite3PagerClose”是搜不到的。

类似的还有“sqlite3pager_get”、“sqlite3pager_unref”、“sqlite3pager_write”、“sqlite3pager_pagecount”等都是老版本函数，它们在 pager.h 文件里定义。新版本对应函数是在 sqlite3.h 里定义（因为都合并到 sqlite3.c 和 sqlite3.h 两文件了）。所以，如果你在使用老版本的 sqlite，先看看 pager.h 文件，这些函数不是消失了，也不是新蹦出来的，而是老版本函数改名得到的。

最后，往 sqlite3.c 文件下找。找到最后一行：

```
/************** End of main.c ********************************************/
```

在这一行后面，接上本文最下面的代码段。

这些代码很长，我不再解释，直接接上去就得了。

唯一要提的是 DeriveKey 函数。这个函数是对密钥的扩展。比如，你要求密钥是 128 位，即是 16 字节，但是如果用户只输入 1 个字节呢？2 个字节呢？或输入 50 个字节呢？你得对密钥进行扩展，使之符合 16 字节的要求。

DeriveKey 函数就是做这个扩展的。有人把接收到的密钥求 md5，这也是一个办法，因为 md5 运算结果固定 16 字节，不论你有多少字符，最后就是 16 字节。这是 md5 算法的特点。但是我不想用 md5，因为还得为它添加包含一些 md5 的.c 或.cpp 文件。我不想这么做。我自己写了一个算法来扩展密钥，很简单的算法。当然，你也可以使用你的扩展方法，也而可以使用 md5 算法。只要修改 DeriveKey 函数就可以了。

在 DeriveKey 函数里，只管申请空间构造所需要的密钥，不需要释放，因为在另一个函数里有释放过程，而那个函数会在数据库关闭时被调用。参考我的 DeriveKey 函数来申请内存。

这里我给出我已经修改好的 sqlite3.c 和 sqlite3.h 文件。

如果太懒，就直接使用这两个文件，编译肯定能通过，运行也正常。当然，你必须按我前面提的，新建 crypt.h 和 crypt.c 文件，而且函数要按我前面定义的要求来做。

i.3 加密使用方法：

现在，你代码已经有了加密功能。

你要把加密功能给用上，除了改 sqlite3.c 文件、给你工程添加 SQLITE_HAS_CODEC 宏，还得修改你的数据库调用函数。

前面提到过，要开始一个数据库操作，必须先 sqlite3_open 。

加解密过程就在 sqlite3_open 后面操作。

假设你已经 sqlite3_open 成功了，紧接着写下面的代码：

```
    int i;
  //添加、使用密码
    i =   sqlite3_key( db, "dcg", 3 );
    //修改密码
    i =   sqlite3_rekey( db, "dcg", 0 );
```

用 sqlite3_key 函数来提交密码。

第 1 个参数是 sqlite3 * 类型变量，代表着用 sqlite3_open 打开的数据库（或新建数据库）。

第 2 个参数是密钥。

第 3 个参数是密钥长度。

用 sqlite3_rekey 来修改密码。参数含义同 sqlite3_key。

实际上，你可以在 sqlite3_open 函数之后，到 sqlite3_close 函数之前任意位置调用 sqlite3_key 来设置密码。

但是如果你没有设置密码，而数据库之前是有密码的，那么你做任何操作都会得到一个返回值：SQLITE_NOTADB，并且得到错误提示：“file is encrypted or is not a database”。

只有当你用 sqlite3_key 设置了正确的密码，数据库才会正常工作。

如果你要修改密码，前提是你必须先 sqlite3_open 打开数据库成功，然后 sqlite3_key 设置密钥成功，之后才能用 sqlite3_rekey 来修改密码。

如果数据库有密码，但你没有用 sqlite3_key 设置密码，那么当你尝试用 sqlite3_rekey 来修改密码时会得到 SQLITE_NOTADB 返回值。

如果你需要清空密码，可以使用：

```
//修改密码
        i =   sqlite3_rekey( db, NULL, 0 );
```

来完成密码清空功能。

i.4 sqlite3.c 最后添加代码段

```
/***
董淳光定义的加密函数
***/
#ifdef SQLITE_HAS_CODEC

/***
加密结构
***/
#define CRYPT_OFFSET 8
typedef struct _CryptBlock
{
BYTE*         ReadKey;          // 读数据库和写入事务的密钥
BYTE*         WriteKey;        // 写入数据库的密钥
int             PageSize;        // 页的大小
BYTE*         Data;
} CryptBlock, *LPCryptBlock;

#ifndef   DB_KEY_LENGTH_BYTE                  /*密钥长度*/
#define   DB_KEY_LENGTH_BYTE     16     /*密钥长度*/
#endif


#ifndef   DB_KEY_PADDING                          /*密钥位数不足时补充的字符*/
#define   DB_KEY_PADDING             0x33   /*密钥位数不足时补充的字符*/
#endif


/*** 下面是编译时提示缺少的函数 ***/

/** 这个函数不需要做任何处理，获取密钥的部分在下面 DeriveKey 函数里实现 **/
void sqlite3CodecGetKey(sqlite3* db, int nDB, void** Key, int* nKey)
{
return ;
}

/*被 sqlite 和 sqlite3_key_interop 调用, 附加密钥到数据库.*/
int sqlite3CodecAttach(sqlite3 *db, int nDb, const void *pKey, int nKeyLen);

/**
```

```
这个函数好像是 sqlite 3.3.17 前不久才加的，以前版本的 sqlite 里没有看到这个函数
这个函数我还没有搞清楚是做什么的，它里面什么都不做直接返回，对加解密没有影响
**/
void sqlite3_activate_see(const char* right )
{
return;
}

int sqlite3_key(sqlite3 *db, const void *pKey, int nKey);

int sqlite3_rekey(sqlite3 *db, const void *pKey, int nKey);



/***
下面是上面的函数的辅助处理函数
***/

// 从用户提供的缓冲区中得到一个加密密钥
// 用户提供的密钥可能位数上满足不了要求，使用这个函数来完成密钥扩展
static unsigned char * DeriveKey(const void *pKey, int nKeyLen);
//创建或更新一个页的加密算法索引.此函数会申请缓冲区.
static LPCryptBlock CreateCryptBlock(unsigned char* hKey, Pager *pager, LPCryptBlock pExisting);
//加密/解密函数, 被 pager 调用
void * sqlite3Codec(void *pArg, unsigned char *data, Pgno nPageNum, int nMode);
//设置密码函数
int __stdcall sqlite3_key_interop(sqlite3 *db, const void *pKey, int nKeySize);
// 修改密码函数
int __stdcall sqlite3_rekey_interop(sqlite3 *db, const void *pKey, int nKeySize);
//销毁一个加密块及相关的缓冲区,密钥.
static void DestroyCryptBlock(LPCryptBlock pBlock);
static void * sqlite3pager_get_codecarg(Pager *pPager);
void sqlite3pager_set_codec(Pager *pPager,void *(*xCodec)(void*,void*,Pgno,int),void *pCodecArg       );



//加密/解密函数, 被 pager 调用
void * sqlite3Codec(void *pArg, unsigned char *data, Pgno nPageNum, int nMode)
{
LPCryptBlock pBlock = (LPCryptBlock)pArg;
unsigned int dwPageSize = 0;

if (!pBlock) return data;

// 确保 pager 的页长度和加密块的页长度相等.如果改变,就需要调整.
if (nMode != 2)
{
```

```
        PgHdr *pageHeader;
        pageHeader = DATA_TO_PGHDR(data);
        if (pageHeader->pPager->pageSize != pBlock->PageSize)
        {
                CreateCryptBlock(0, pageHeader->pPager, pBlock);
        }
}

switch(nMode)
{
case 0: // Undo a "case 7" journal file encryption
case 2: //重载一个页
case 3: //载入一个页
        if (!pBlock->ReadKey) break;


        dwPageSize = pBlock->PageSize;
        My_DeEncrypt_Func(data, dwPageSize, pBlock->ReadKey, DB_KEY_LENGTH_BYTE );   /*调用我的解密函数*/

        break;
case 6: //加密一个主数据库文件的页
        if (!pBlock->WriteKey) break;

        memcpy(pBlock->Data + CRYPT_OFFSET, data, pBlock->PageSize);
        data = pBlock->Data + CRYPT_OFFSET;


        dwPageSize = pBlock->PageSize;
        My_Encrypt_Func(data , dwPageSize, pBlock->WriteKey, DB_KEY_LENGTH_BYTE ); /*调用我的加密函数*/
        break;
case 7: //加密事务文件的页
        /*在正常环境下, 读密钥和写密钥相同. 当数据库是被重新加密的,读密钥和写密钥未必相同.
        回滚事务必要用数据库文件的原始密钥写入.因此,当一次回滚被写入,总是用数据库的读密钥,
        这是为了保证与读取原始数据的密钥相同.
        */
        if (!pBlock->ReadKey) break;

        memcpy(pBlock->Data + CRYPT_OFFSET, data, pBlock->PageSize);
        data = pBlock->Data + CRYPT_OFFSET;


        dwPageSize = pBlock->PageSize;
        My_Encrypt_Func( data, dwPageSize, pBlock->ReadKey, DB_KEY_LENGTH_BYTE ); /*调用我的加密函数*/
        break;
}

return data;
```

```
}



//销毁一个加密块及相关的缓冲区,密钥.
static void DestroyCryptBlock(LPCryptBlock pBlock)
{
//销毁读密钥.
if (pBlock->ReadKey){
        sqliteFree(pBlock->ReadKey);
}

//如果写密钥存在并且不等于读密钥,也销毁.
if (pBlock->WriteKey && pBlock->WriteKey != pBlock->ReadKey){
        sqliteFree(pBlock->WriteKey);
}

if(pBlock->Data){
        sqliteFree(pBlock->Data);
}

//释放加密块.
sqliteFree(pBlock);
}

static void * sqlite3pager_get_codecarg(Pager *pPager)
{
return (pPager->xCodec) ? pPager->pCodecArg: NULL;
}
// 从用户提供的缓冲区中得到一个加密密钥
static unsigned char * DeriveKey(const void *pKey, int nKeyLen)
{
unsigned char *   hKey = NULL;
int j;

if( pKey == NULL || nKeyLen == 0 )
{
        return NULL;
}

hKey = sqliteMalloc( DB_KEY_LENGTH_BYTE + 1 );
if( hKey == NULL )
{
        return NULL;
}
hKey[ DB_KEY_LENGTH_BYTE ] = 0;
```

```
if( nKeyLen < DB_KEY_LENGTH_BYTE )
{
        memcpy( hKey, pKey, nKeyLen ); //先拷贝得到密钥前面的部分
        j = DB_KEY_LENGTH_BYTE - nKeyLen;
        //补充密钥后面的部分
        memset(   hKey + nKeyLen,   DB_KEY_PADDING, j   );
}
else
{ //密钥位数已经足够,直接把密钥取过来
        memcpy(   hKey, pKey, DB_KEY_LENGTH_BYTE );
}

return hKey;
}


//创建或更新一个页的加密算法索引.此函数会申请缓冲区.
static LPCryptBlock CreateCryptBlock(unsigned char* hKey, Pager *pager, LPCryptBlock pExisting)
{
LPCryptBlock pBlock;

if (!pExisting) //创建新加密块
{
        pBlock = sqliteMalloc(sizeof(CryptBlock));
        memset(pBlock, 0, sizeof(CryptBlock));
        pBlock->ReadKey = hKey;
        pBlock->WriteKey = hKey;
        pBlock->PageSize = pager->pageSize;
        pBlock->Data = (unsigned char*)sqliteMalloc(pBlock->PageSize + CRYPT_OFFSET);
}
else //更新存在的加密块
{
        pBlock = pExisting;
        if ( pBlock->PageSize != pager->pageSize && !pBlock->Data){
                sqliteFree(pBlock->Data);
                pBlock->PageSize = pager->pageSize;
                pBlock->Data = (unsigned char*)sqliteMalloc(pBlock->PageSize + CRYPT_OFFSET);
        }
}


memset(pBlock->Data, 0, pBlock->PageSize + CRYPT_OFFSET);

return pBlock;
}

/*
```

```
** Set the codec for this pager
*/
void sqlite3pager_set_codec(
                                                    Pager *pPager,
                                                    void *(*xCodec)(void*,void*,Pgno,int),
                                                 void *pCodecArg
                                                    )
{
pPager->xCodec = xCodec;
pPager->pCodecArg = pCodecArg;
}


int sqlite3_key(sqlite3 *db, const void *pKey, int nKey)
{
return sqlite3_key_interop(db, pKey, nKey);
}

int sqlite3_rekey(sqlite3 *db, const void *pKey, int nKey)
{
return sqlite3_rekey_interop(db, pKey, nKey);
}

/*被 sqlite 和 sqlite3_key_interop 调用, 附加密钥到数据库.*/
int sqlite3CodecAttach(sqlite3 *db, int nDb, const void *pKey, int nKeyLen)
{
       int rc = SQLITE_ERROR;
       unsigned char* hKey = 0;

       //如果没有指定密匙,可能标识用了主数据库的加密或没加密.
       if (!pKey || !nKeyLen)
       {
              if (!nDb)
              {
                     return SQLITE_OK; //主数据库, 没有指定密钥所以没有加密.
              }
              else //附加数据库,使用主数据库的密钥.
              {
                     //获取主数据库的加密块并复制密钥给附加数据库使用
                                          LPCryptBlock                    pBlock                    =
(LPCryptBlock)sqlite3pager_get_codecarg(sqlite3BtreePager(db->aDb[0].pBt));

                     if (!pBlock) return SQLITE_OK; //主数据库没有加密
                     if (!pBlock->ReadKey) return SQLITE_OK; //没有加密

                     memcpy(pBlock->ReadKey, &hKey, 16);
              }
```

```
	}
	else //用户提供了密码,从中创建密钥.
	{
		hKey = DeriveKey(pKey, nKeyLen);
	}

	//创建一个新的加密块,并将解码器指向新的附加数据库.
	if (hKey)
	{
		LPCryptBlock pBlock = CreateCryptBlock(hKey, sqlite3BtreePager(db->aDb[nDb].pBt), NULL);
		sqlite3pager_set_codec(sqlite3BtreePager(db->aDb[nDb].pBt), sqlite3Codec, pBlock);
		rc = SQLITE_OK;
	}
	return rc;
}

// Changes the encryption key for an existing database.
int __stdcall sqlite3_rekey_interop(sqlite3 *db, const void *pKey, int nKeySize)
{
Btree *pbt = db->aDb[0].pBt;
Pager *p = sqlite3BtreePager(pbt);
LPCryptBlock pBlock = (LPCryptBlock)sqlite3pager_get_codecarg(p);
unsigned char * hKey = DeriveKey(pKey, nKeySize);
int rc = SQLITE_ERROR;


if (!pBlock && !hKey) return SQLITE_OK;

//重新加密一个数据库,改变 pager 的写密钥, 读密钥依旧保留.
if (!pBlock) //加密一个未加密的数据库
{
	pBlock = CreateCryptBlock(hKey, p, NULL);
	pBlock->ReadKey = 0; // 原始数据库未加密
	sqlite3pager_set_codec(sqlite3BtreePager(pbt), sqlite3Codec, pBlock);
}
else // 改变已加密数据库的写密钥
{
	pBlock->WriteKey = hKey;
}

// 开始一个事务
rc = sqlite3BtreeBeginTrans(pbt, 1);

if (!rc)
{
	// 用新密钥重写所有的页到数据库。
	Pgno nPage = sqlite3PagerPagecount(p);
```

```
        Pgno nSkip = PAGER_MJ_PGNO(p);
        void *pPage;
        Pgno n;

        for(n = 1; rc == SQLITE_OK && n <= nPage; n ++)
        {
                if (n == nSkip) continue;
                rc = sqlite3PagerGet(p, n, &pPage);
                if(!rc)
                {
                        rc = sqlite3PagerWrite(pPage);
                        sqlite3PagerUnref(pPage);
                }
        }
}

// 如果成功，提交事务。
if (!rc)
{
        rc = sqlite3BtreeCommit(pbt);
}

// 如果失败，回滚。
if (rc)
{
        sqlite3BtreeRollback(pbt);
}



// 如果成功，销毁先前的读密钥。并使读密钥等于当前的写密钥。
if (!rc)
{
        if (pBlock->ReadKey)
        {
                sqliteFree(pBlock->ReadKey);
        }
        pBlock->ReadKey = pBlock->WriteKey;
}
else// 如果失败，销毁当前的写密钥，并恢复为当前的读密钥。
{
        if (pBlock->WriteKey)
        {
                sqliteFree(pBlock->WriteKey);
        }
        pBlock->WriteKey = pBlock->ReadKey;
}
```

```
// 如果读密钥和写密钥皆为空，就不需要再对页进行编解码。
// 销毁加密块并移除页的编解码器
if (!pBlock->ReadKey && !pBlock->WriteKey)
{
        sqlite3pager_set_codec(p, NULL, NULL);
        DestroyCryptBlock(pBlock);
}

return rc;
}




/***
下面是加密函数的主体
***/
int __stdcall sqlite3_key_interop(sqlite3 *db, const void *pKey, int nKeySize)
{
   return sqlite3CodecAttach(db, 0, pKey, nKeySize);
}


// 释放与一个页相关的加密块
void sqlite3pager_free_codecarg(void *pArg)
{
if (pArg)
        DestroyCryptBlock((LPCryptBlock)pArg);
}

#endif //#ifdef SQLITE_HAS_CODEC
```

五、 后记

写此教程，可不是一个累字能解释。

但是我还是觉得欣慰的，因为我很久以前就想写 sqlite 的教程，一来自己备忘，二而已造福大众，大家不用再走弯路。

本人第一次写教程，不足的地方请大家指出。