SQLite in Python
sqlite3 模块的使用与性能分析
版本 1.0

制作人：

卢熠辉 516030910135

2019-5

文档所用语言：

中文

目录

1. 引言

1.1. 文档目的

本文的目的是为了阐述如何在 Python 代码中调用 SQLite 命令，从而利用 SQLite 提供的功能为 Python 程序提供数据持久化方面的支持，实现进程（python）与数据（SQLite）之间的无缝衔接。

更进一步，本文中会对 SQLite 提供的种种数据库核心功能如事务、隔离性进行测试，同时——作为 Python 程序不可避免的话题——本文会对 SQLite 提供的数据库操作的性能进行测试，希望能够为 Python+SQLite 的开发提供更多帮助。

1.2. 重要名词解释

1.2.1. SQLite[1]

SQLite 是一个 C 语言库，它实现了一个简单、快速、自足（self-contained，指对其他库的依赖少）、高可靠的（high-reliability）、拥有全部特性（full-featured）的 SQL 数据库引擎。SQLite 是世界上使用最广泛的数据库引擎，他被安装于所有的移动手机和绝大多数的计算机中，并且与无数其他各种人们日常使用的应用程序相绑定。

SQLite 文件形式拥有稳定、跨平台且向后兼容的特性，且开发人员会保证它的这些特性至少保持到 2050 年。SQLite 数据库文件被广泛地作为容器应用于跨系统地传递丰富内容以及作为数据的长期归档保存形式。现在有超过十亿的 SQLite 数据库正在被使用。

1.2.2. Python

Python 是一种计算机程序设计语言。是一种面向对象的动态类型语言，最初被设计用于编写自动化脚本(shell)，随着版本的不断更新和语言新功能的添加，越来越多被用于独立的、大型项目的开发。[2]

根据 Python 解释器本身的实现方法的不同，可被分为 CPython、JPython、以及 PyPy。其中 CPython 的解释器使用 C 语言开发，是官网下载安装后的默认解释器，JPython 是运行在 Java 平台上的 Python 解释器，可以直接把 Python 代码编译成 Java 字节码执行，PyPy 采用 JIT 技术，对 Python 代码进行动态编译（注意不是解释），以提高 Python 代码的执行速度。

在本文中默认使用 CPython。

[1] 翻译自 https://www.sqlite.org/index.html，个人英语水平有限，如有问题请以原文为准

[2] https://baike.baidu.com/item/Python/407313

2. 准备工作

2.1. 实验环境

WIN10 + Python 3.7.3

2.2. 安装与使用 Python

请参考 https://www.runoob.com/python3/python3-install.html，在此不做赘述

2.3. 使用 SQLite

sqlite3 已经作为一个模块内置于 python 中 3，所以无需额外配置即可直接调用。

[3] https://github.com/python/cpython/tree/3.7/Lib/sqlite3

3. 基本 API 与说明

3.1. 控制接口

这一部分将会介绍 sqlite3 模块中定义的用于实现基于数据库的一些控制功能的接口，如数据库连接、修改提交、事务隔离等。

3.1.1. 连接数据库

接口形式：

sqlite3.connect(database[, timeout, detect_types, isolation_level, check_same_thread, factory, cached_statements, uri]

接口功能：

连接 SQLite 数据库 database。默认返回 Connection 对象，除非使用了自定义的 factory 参数。

参数说明：

database	准备打开的数据库文件的路径（绝对路径或相对于当前目录的相对路径），它是 path-like object。也可以用 ":memory:" 在内存中打开一个数据库。
timeout	指定了这个连接等待锁释放[4]的超时时间，超时之后会引发一个异常。这个超时时间默认是 5.0（5 秒）。
detect_types	设置 SQLite 接口的类型检测，默认为 0（关闭类型检测），通过设置相应的转换器可以实现原生 5 中数据类型以外的类型支持。
isolation_level	用于设置返回的 connection 对象的 isolation_level 参数。表示自动提交模式的 None 以及 "DEFERRED", "IMMEDIATE" 或 "EXCLUSIVE" 其中之一。
check_same_thread	设置当前连接是否被当前进程独占。默认为 True，只有当前的线程可以使用该连接。 如果设置为 False，则多个线程可以共享返回的连接。当多个线程使用同一个连接的时候，用户应该把写操作进行序列化，以避免数据损坏。
factory	设置返回的连接类，默认为 Connection 类，也可以是用户自己实现的 Connection 的子类。
cached_statements	显式设置当前连接可以缓存的语句数，以调整 SQL 解析开销。默认为 100 条语句
uri	如果 uri 为真，则 database 被解释为 URI。

3.1.2. 从连接获取游标

Connection 类方法：cursor(factory=Cursor)

这个方法接受一个可选参数 factory，如果要指定这个参数，它必须是一个可调用对象，而且必须返回 Cursor 类的一个实例或者子类。

3.1.3. 提交事务

Connection 类方法：commit()

[4] 当一个数据库被多个连接访问的时候，如果其中一个进程修改这个数据库，在这个事务提交之前，这个 SQLite 数据库将会被一直锁定。

这个方法提交当前事务。如果没有调用这个方法，那么从上一次提交 commit() 以来所有的变化在其他数据库连接上都是不可见的。

3.1.4. 回滚事务

Connection 类方法：rollback()
这个方法回滚从上一次调用 commit() 以来所有数据库的改变。

3.1.5. 关闭连接

Connection 类方法：close()
关闭数据库连接。注意，它不会自动调用 commit() 方法。如果在关闭数据库连接之前没有调用 commit()，那么修改将会丢失。

3.2. 执行接口

这部分介绍如何通过 sqlite3 模块执行 SQLite 的各种功能性语句，包括增删查改等。

3.2.1. 执行 SQL 语句

Cursor 类方法：execute(sql[, parameters])
参数是字符串形式的 SQL 语句，也可以是参数化 SQL 语句（即，在 SQL 语句中使用占位符）[5]。sqlite3 模块支持两种占位符：问号（qmark 风格）和命名占位符（命名风格），例如：

```
cur.execute("insert into people values (?, ?)", (who, age))
cur.execute("select * from people where name_last=:who and age=:age", {"who": who, "age": age})
```

本方法只能执行单个的 SQL 语句，如果试图执行多个语句，将会产生警告。

3.2.2. 执行多条 SQL 语句

Cursor 类方法：executemany(sql, seq_of_parameters)
以 seq_of_parameters 中的元素为参数，多次执行 sql 语句。

[5] 使用 Python 的字符串操作来创建查询语句容易遭受注入攻击，见 https://xkcd.com/327/

4. 代码实例分析

这部分将通过实例说明 sqlite3 模块的使用方法。考虑到文本的简洁性，在文档中仅针对重点部分进行举例，详细说明与注释请见代码（src/4_example/）。

4.1. 常规操作

展示基本操作的使用方法，如：建立连接、获取游标、执行语句、提交修改、关闭连接。在执行语句中，分为创建表、插入、查询、删除和修改五个部分，其中插入根据插入的数量可分为单个插入和批量插入两部分，查询也可以根据游标的使用方法分为四部分。

4.1.1. 创建表

```
c.execute('''CREATE TABLE STUDENT
    (ID INT PRIMARY KEY NOT NULL,
    NAME CHAR(30) NOT NULL,
    AGE INT,
    GENDER CHAR,
    MAJOR CHAR(10) );''')
```

4.1.2. 单值插入

```
c.execute("INSERT INTO STUDENT (ID,NAME,AGE,GENDER,MAJOR) \
    VALUES (1, 'QYF', 22, 'M', 'EE' )")
c.execute("INSERT INTO STUDENT (ID,NAME,AGE,GENDER,MAJOR) \
    VALUES (?, ?, ?, ?, ? )", (5, 'Laurie', 21, 'F', 'CS'))
```

4.1.3. 批量插入

```
studentList2 = [
    [7, 'Hornby', 17, 'F', 'CS'],
    [8, 'Zephaniah', 20, 'M', 'EE'],
]
c.executemany("INSERT INTO STUDENT (ID,NAME,AGE,GENDER,MAJOR) \
    VALUES (?, ?, ?, ?, ? )", studentList2)
```

4.1.4. 查询

```
cursor = c.execute("SELECT id, name, GENDER, MAJOR FROM STUDENT")
### 将获取到的游标作为迭代器使用
for row in cursor:
    print(row)

### 使用 fetchone()方法逐行获得查询结果，每次获得一个元组
while True:
    row = c.fetchone()
    if row == None:
```

```
        break
    else:
        print(row)

### 使用 fetchall()方法获得全部剩余查询结果，获得一个元组的元组
rows = c.fetchall()
for row in rows:
    print(row)

### 使用 fetchmany(size=cursor.arraysize)方法获得(不大于）指定长度的剩余结果，获得一个元组的元组
fetchNum = 3 # 使用一个不能整除 8 的数，测试当剩余数少于参数时的效果
while True:
    rows = c.fetchmany(fetchNum)
    if rows == []:
        break
    else:
        for row in rows:
            print(row)
```

4.1.5. 删除

```
c.execute("DELETE from STUDENT where MAJOR=?", ('SE',))
# 注意第二个参数提供的必须是一个可枚举类型（列表或元组）
```

4.1.6. 更新

```
c.execute("UPDATE STUDENT set MAJOR = 'SE' where MAJOR is NULL")
# 注意虽然空值在 SQLite 中显示是 None，但判定时仍是 NULL
```

4.2. 基于连接的快捷操作

在 4.1 节中 Python 通过 cursor 对象的 execute 方法调用 SQLite 命令，这就要求在连接建立后获取一个 cursor 对象，在实际使用中可以省去这一步，直接调用 connection 对象的同名快捷方法，它会调用 cursor() 方法来创建一个游标对象，并使用给定的 parameters 参数来调用游标对象的 execute() 方法，最后返回这个游标对象，例如：

```
conn.execute("INSERT INTO STUDENT (ID,NAME,AGE,GENDER,MAJOR) \
    VALUES (1, 'QYF', 22, 'M', 'EE' )")
```

4.3. 脚本执行

前两节的方法都需要逐条执行，即使是允许批量操作也要求其操作类型必须相同（例如都是 INSERT）。通过使用 executescript 方法，我们可以让数据库自动的运行多条 SQL 命令，如：

```
c.executescript('''
    CREATE TABLE STUDENT(
```

```
        ID INT PRIMARY KEY NOT NULL,
        NAME CHAR(30) NOT NULL,
        AGE INT,
        GENDER CHAR,
        MAJOR CHAR(10)
    );

    INSERT INTO STUDENT (ID,NAME,AGE,GENDER,MAJOR) \
        VALUES (1, 'QYF', 22, 'M', 'EE' );
    INSERT INTO STUDENT (ID,NAME,AGE,GENDER,MAJOR) \
        VALUES (2, 'LZQ', 21, 'M', 'CS');

    DELETE from STUDENT where MAJOR='CS';

''')
```

创建表以及对表的操作一气呵成，但是要注意在 executescript 方法中不能使用占位符，这就意味着这个方法更适合用于程序员明确定义的大量系统操作，而不适合用于与用户输入相关的交互功能。
同时脚本中每一条 SQL 命令的末尾都必须添加分号。

4.4. 事物的回滚与提交

在本节中将考察数据库 commit 与 rollback 的特性，设计实验流程与结果如下：

阶段	操作	结果
Stage 1	初始化数据库数据并提交	数据库表现为 Stage1 的状态
Stage 2	修改数据库但不提交	数据库表现为 Stage2 的状态
	回滚	数据库表现为 Stage1 的状态
Stage 3	修改数据库但不提交	数据库表现为 Stage3 的状态
Stage 4	使用 executescript 方法修改数据库	数据库表现为 Stage4 的状态
	回滚	数据库表现为 Stage4 的状态

由上述实验可以看出，数据库一旦回滚将会回到上一次 commit 的状态，调用普通的 execute、executemany 方法都不会自动 commit 修改。但是调用 executescript 方法将会在方法调用和结束之前自动 commit[6]，初步分析是因为默认 executescript 方法执行的 SQL 脚本较长可能会发生风险需要自动 commit 修改。

[6] 调用结束后自动 commit 可由实验结果分析得到，调用之前自动 commit 为官方文档所描述

5. 性能测试与比较

5.1. 文件数据库与内存数据库

5.1.1. 实验说明

如 3.1.1 节所说，sqlite3 库即允许以文件作为数据库载体也允许用户创建内存数据库，显然两者之间体现的是持久性与性能之间的权衡，为了充分考虑这两者之间的性能差异，在这一节将比较文件数据库与内存数据库在处理不同规模的插入操作时所花费的时间，以便于在日常的使用中更好选择数据库的存储介质。

5.1.2. 实验设计

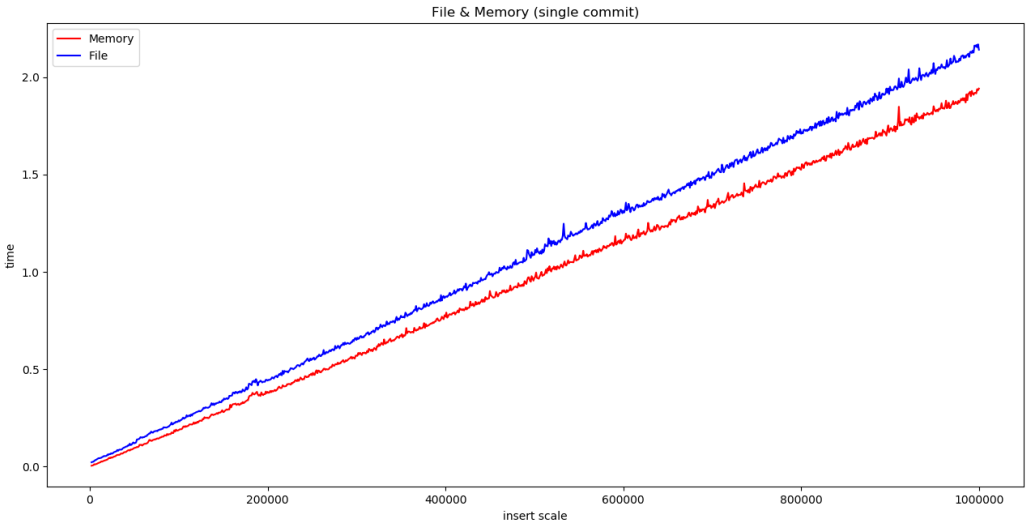
实验比较的对象为文件与内存两种数据库处理相同操作时的性能，实验中只考虑执行插入操作的时间，打开数据库、初始化表、生成插入数据的过程不被记录，每次运行完成后将关闭并清空数据库，下一次插入前重新打开并初始化，每次执行的插入数据规模递增，每一个规模的数据执行十次取平均值以消除误差。

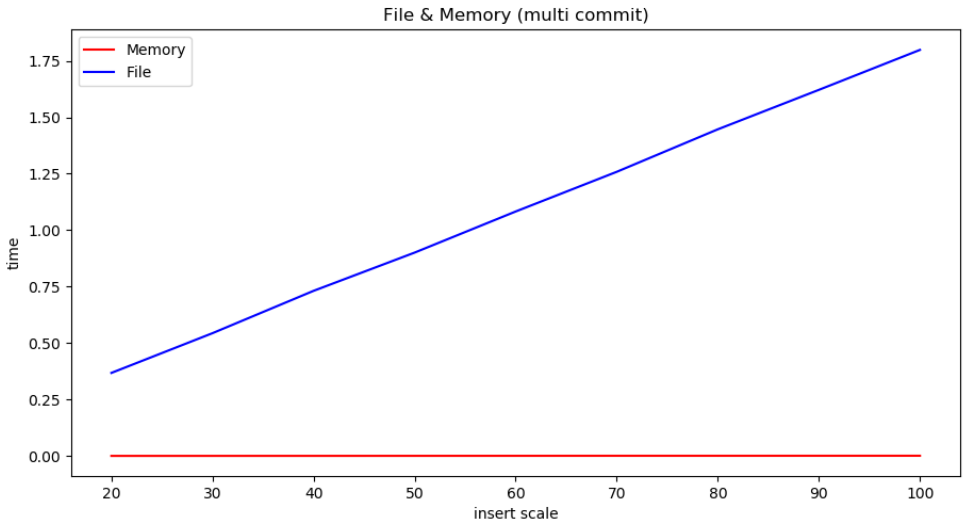
实验共分为两部分：

第一部分操作为 executemany - commit，即插全部数据，提交一次

第二部分操作为 execute - commit，即每次插入一组数据后立即提交，循环多次直至数据全部插入

5.1.3. 实验结果





5.1.4. 结果分析

显然在两种情况下内存数据库相比文件数据库都有着性能优势，在多次 commit 的情况下性能优势尤为明显，但是单次 commit 的情况下文件数据库同样具有较好的性能。

初步分析是由于 SQLite 为数据库修改提供了缓存功能，即数据库的修改都在缓存中进行，仅有 commit 时才被真正写入文件（这一点在 6.2 节中也有体现），所以即使使用文件数据库真正的文件访问次数也仅有一次，在大规模下逐渐与内存产生的差距的来源在于这一次文件访问所写的数据的量的增加，不过由于现在计算机存储访问性能的提升（我使用的是 Samsung SSD 850 Pro），因此这一部分增加的开销并不显著。相对的，在多次提交的过程中，文件访问次数与插入规模相同，这一部分开销被显著放大，造成了性能上的瓶颈，使得文件数据库与内存数据库之间产生了巨大的性能差距。

5.1.5. 实验结论

sqlite3 模块有良好的操作缓存功能，即使数据规模巨大也能够提供良好的处理性能。针对文件数据库，修改的大部分开销来源于 commit 语句的执行，因此设计程序时要充分考虑到性能与可靠性的权衡——在保证数据库能够应对突发异常的情况下减少 commit 的次数充分利用缓存机制。

5.2. 插入操作与删除操作

5.2.1. 实验说明

插入（INSERT）与删除（DELETE）操作是数据库修改中的常用操作，这部分中将比较这两种指令之间执行时性能上的差异。

5.2.2. 实验设计

实验中只考察插入、删除操作执行的时间，同时确保每次插入的数据规模与被删除的数据规模以及执行删除时数据库中的数据量规模相同。

考察对象如下：

1、使用 executemany 接口执行的 SQL 插入。

2、使用 execute 接口执行的单条 SQL 删除语句，该删除语句确保删除数据库中的所有数据，通过将 WHERE 条件设置为所有元组的共同特征实现。

3、使用 executemany 接口执行的多条 SQL 删除语句，每条删除语句确保仅删除数据库中的一个元组，且所有语句执行完成后将删除数据库中的全部元组，同时每一条语句的 WHERE 条件将指定为该表的键属性的性质，删除是顺序的。

4、使用 executemany 接口执行的多条 SQL 删除语句，特性与 3 相同，但此时 WHERE 条件不指定为键属性，而是普通属性。

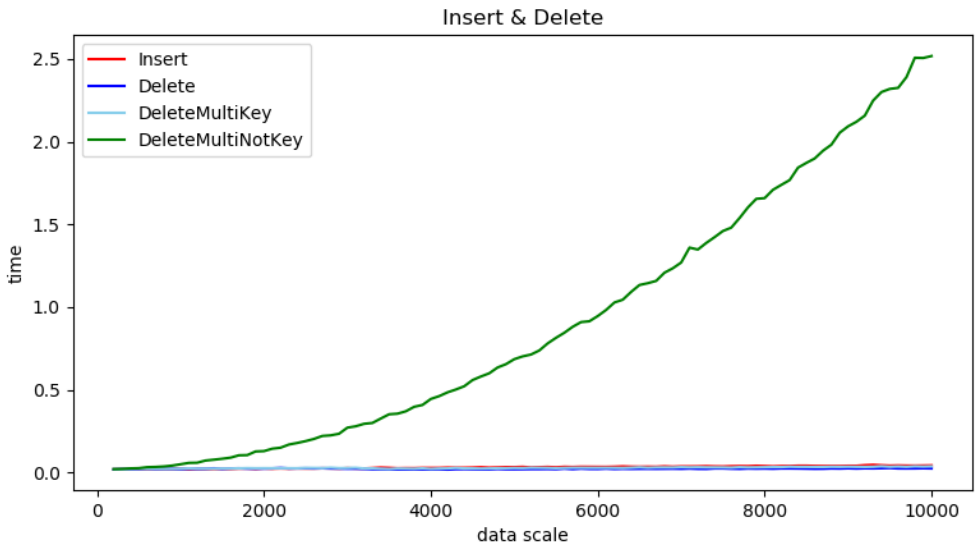
5、使用 executemany 接口执行的多条 SQL 删除语句，特性与 3 相同，但此时 WHERE 条件指定键属性是为逆序。

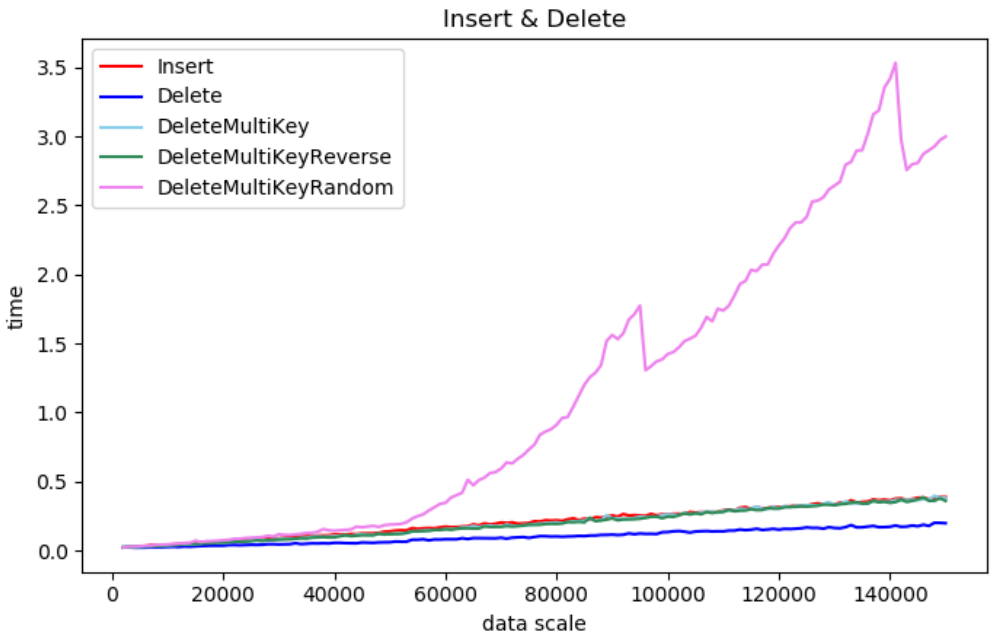
6、使用 executemany 接口执行的多条 SQL 删除语句，特性与 3 相同，但此时 WHERE 条件指定键属性是为乱序。

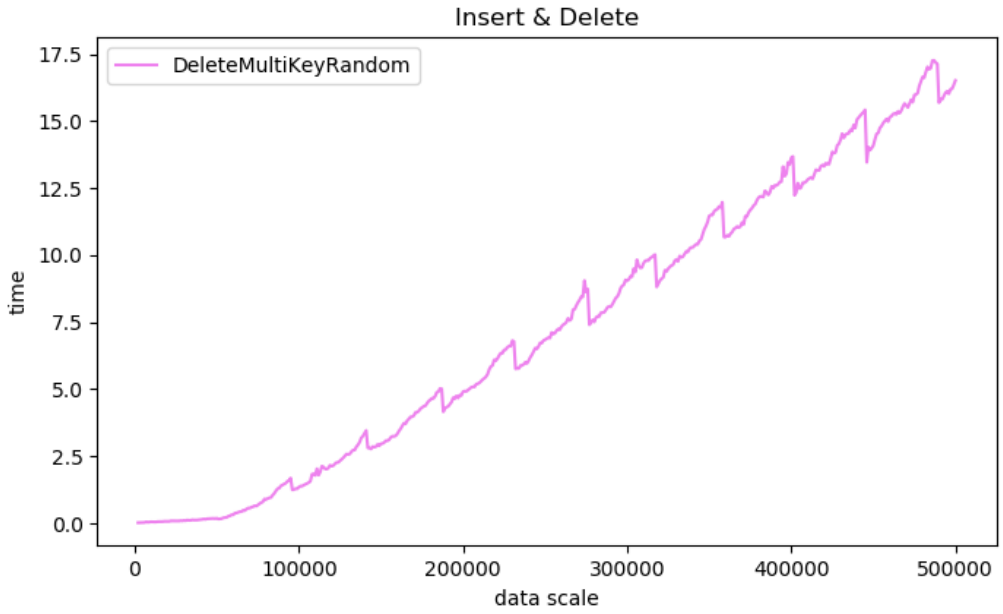
*针对 3、5、6 说明：

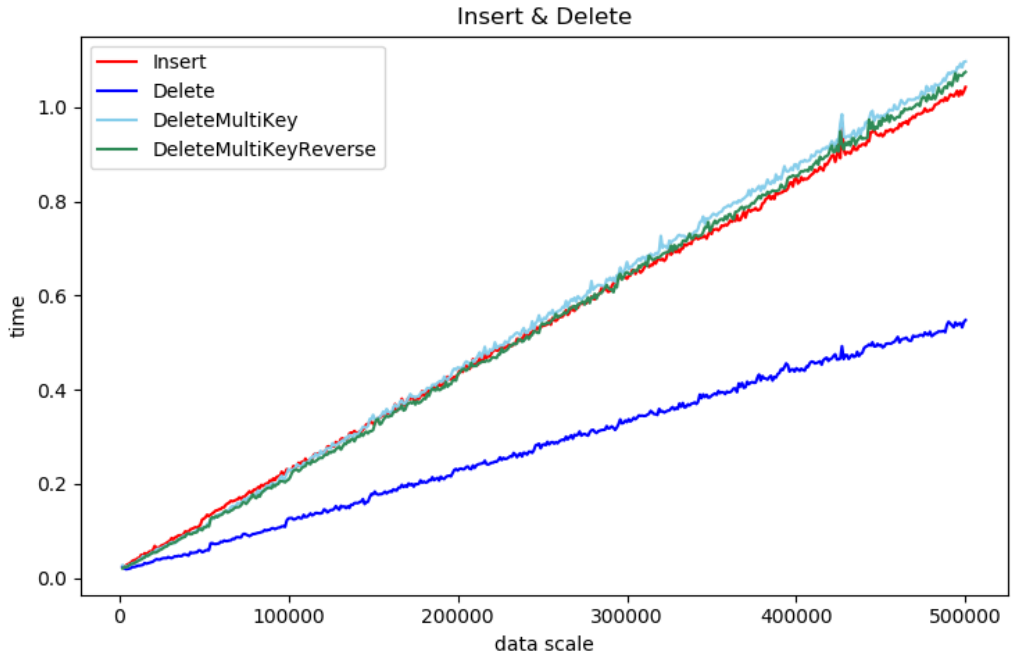
假设表中有 id 分别为 1、2、3、4 的元组，则 3 中删除顺序为[1, 2, 3, 4]，5 中删除顺序为[4, 3, 2, 1]，6 中删除顺序任意，可能为[2, 4, 1, 3]等。

5. 2. 3. 实验结果









5.2.4. 结果分析

注意到最耗费时间的是针对非键属性的多次删除操作，其开销在仅仅为六千的数据规模下就已经超过了一秒，远非其他任意一种情况所能比拟。开销第二大的是键属性的随机顺序多次删除操作，在八万的数据规模下延迟超过一秒，且可以注意到它的曲线随着规模的增长呈现锯齿形。插入语句、多次键属性删除和逆序多次键属性删除的开销基本相同。单次全部删除的开销最小，仅有前面三者的一半左右。

初步分析如下：

针对非键属性和键属性的多次删除，前者每次删除都需要扫描整张表进行逐项匹配，这一点上难以采取任何优化策略，因此开销最大，而后者每次只需要匹配到一项满足就可以停止，因此开销较小。

针对不同顺许下键属性的多次删除，我认为是因为空间局部性的关系，不论是顺序还是逆序都有良好的空间局部性，因此开销较小。相对而言随机顺序全无空间局部性，因此开销较大，同时其阶梯状的开销曲线也十分满足空间局部性的特征。

针对单次删除语句，虽然使用的是非键属性，但其仅需要扫描一遍表就可以完成，因此效率最高。

针对键属性删除语句和插入语句，正如此前空间局部性的分析，在良好的局部性和缓存机制下删除语句几乎仅有单语句的开销（而不是针对整张表的扫描），因此和多次单语句的插入性能相仿，仅仅是多了一点点的开销，这也与局部性推理相契合。

5.3. execute 语句与 executemany 语句

5.3.1. 实验说明

sqlite3 模块提供了 execute 接口允许用户执行单条的 SQL 语句，同时也提供了 executemany 接口允许用户通过提供一组可迭代类型的参数来多次执行具有相同模式和不同参数的 SQL 语句（详见 4.1.3 节）。

在这一节中将考察通过这两种接口向数据库中插入相同规模的数据时所体现出的性能差异，以便于在实际使用中正确的选择合适的接口。

5.3.2. 实验设计

针对同一组待插入数据，先使用 executemany 语句一次性插入，再多次使用 execute 语句单条插入，两者均在插入完成后提交（针对每次 execute 后 commit 的情况，实验中性能劣化太过严重，没有参考价值，具体可见 5.1 节的分析，或取消本实验代码中的相关注释自己体验一下）。

5.3.3. 实验结果

5.3.4. 结果分析

相比于多次的 execute 语句，相同规模的 executemany 语句具有相对较高的执行效率（约

为前者的 1.25 倍）。

初步分析这是因为从 python 环境进入 SQLite 命令执行的环境有一个上下文切换的开销，execute 语句每次执行都有这个开销，而 executemany 仅有一次这个开销，因此后者效率更高。至于为什么随着数据规模的增加两者的效率倍率并没有发生变化，推测这是因为 executemany 虽然减少了上下文切换的开销，但是在执行多条 SQL 语句时任存在一定的开销，而不是前后两句无缝衔接，因此两者的性能倍率相对固定。

5.3.5. 实验结论

在实际使用中要执行多次 SQL 语句时，尽量使用 executemany 接口，可以提升效率。

5.4. isolation level 对性能的影响

5.4.1. 实验说明

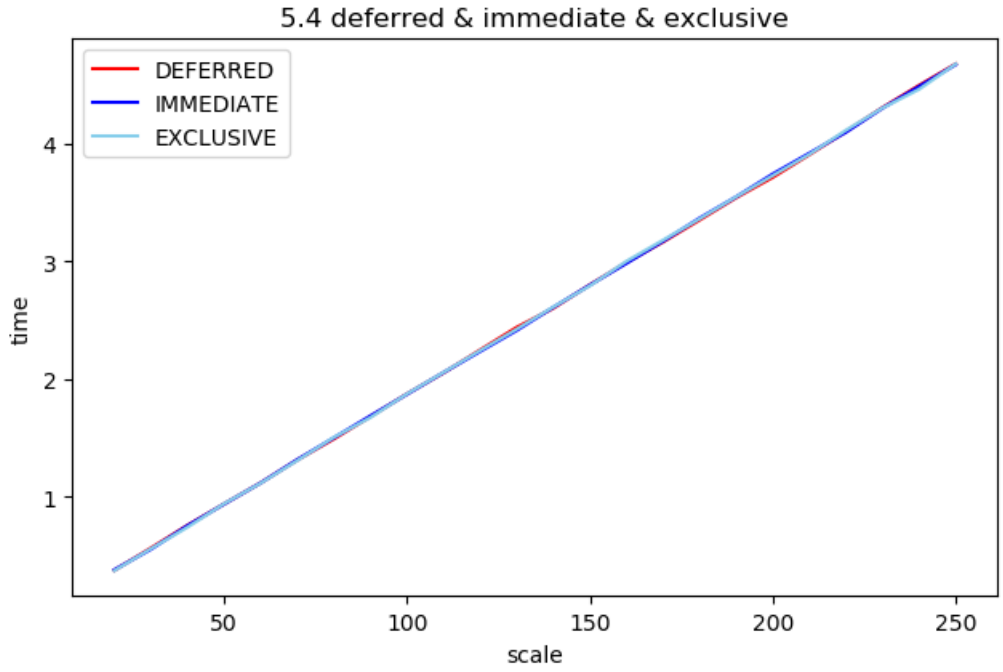
如 3.1.1 节所言，在 sqlite3 模块中连接数据库时需要指定连接的隔离等级（isolation level 参数），该参数会影响此连接获取锁的过程（关于隔离性与锁的关系详情请见 6.1 节），从而间接地影响在此连接上执行的数据库修改操作的性能。

在本节中，将会通过实验比较 DEFERRED、IMMEDIATE、EXCLUSIVE 这三种隔离等级下相同规模的插入操作所需要的时间，从而间接体现不同隔离等级对数据库性能的影响，帮助在日常使用中对安全性和性能之间做出更好的权衡。

5.4.2. 实验设计

针对三个数据库打开三种处于不同隔离层次的连接，每次针对相同规模的数据使用 execute 语句执行多次单条 SQL 插入语句，同时每次插入后执行 commit（因为隔离性的锁与 commit 这个动作关联紧密，所以通过多次 commit 放大不同隔离等级导致的的性能差异）

5.4.3. 实验结果



5.4.4. 结果分析

在三种 isolation_level 的连接下执行操作并没有任何性能上的差别。
详细分析请见 7.1 节。

5.4.5. 实验结论

在常用操作下，并不需要考虑连接对象的 isolation_level 对性能造成的影响。

6. 隔离性相关测试与分析

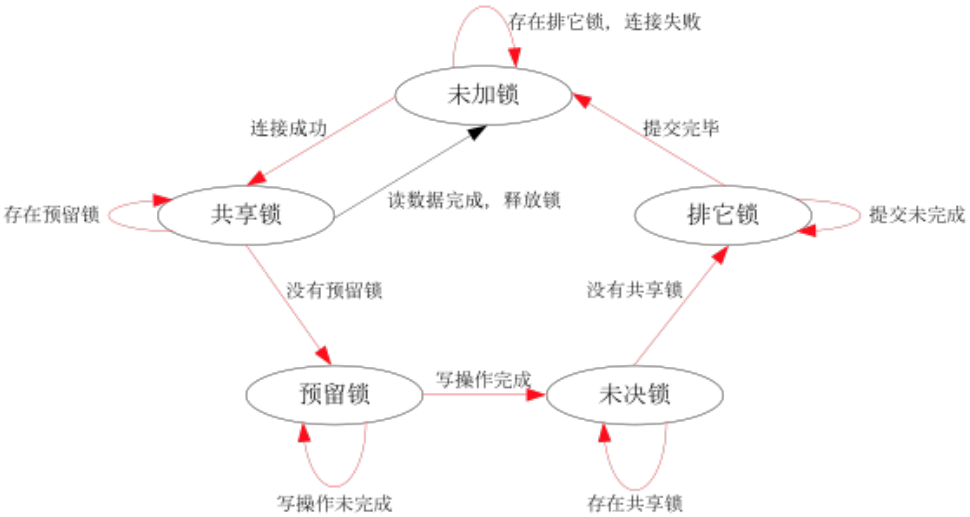
6.1. 不同隔离等级下的连接与锁获取

6.1.1. 实验说明

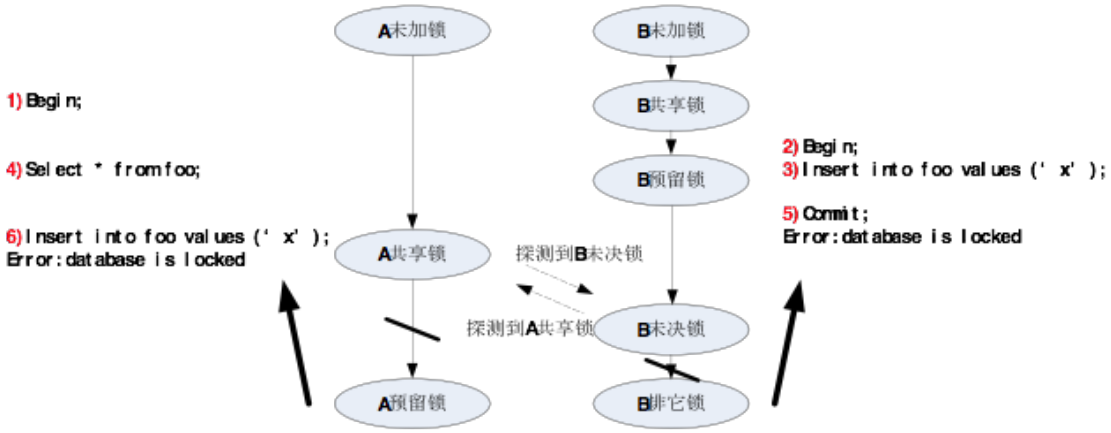
这一节中所说的“隔离性”指 sqlite3 模块中 connection 类型的 isolation_level 属性，虽然名为“隔离性”，但其实和数据库原理中所说的四种隔离性（读未提交、读提交、可重复读、可串行化）并不相同，它更接近于描述一个事务获取数据库锁的策略（真正的隔离性将在 6.2 节讨论）。

在 SQLite 中共有五种类型的锁，对应于不同的操作权限，其名称与转移过程如下：

锁状态	说明
为枷锁-unlocked	未对数据库进行访问（读写）之前
共享锁-shared	对数据库进行读操作
预留锁-reserved	对数据库进行写（缓存）操作
未决锁-pending	等待其它共享锁关闭
排它锁-exclusive	将缓存中的操作提交到数据库



但是上述的锁转移过程在并发访问中可能会产生死锁，例如：



在上图中连接 A 想要修改数据库因此需要从共享锁升级为预留锁，连接 B 完成了写操作获得未决锁，为了执行 commit 操作需要从未决锁升级为排他锁。在这一过程中连接 A 识别到 B 持有未决锁因此获得预留锁失败，连接 B 识别到 A 持有共享锁因此获得排他锁失败，但同时两者都不愿意放弃自己所持有的锁从而造成死锁

为了避免这种情况的发生，SQLite 中使用 isolation_level 在连接启动时就制定获取锁的策略（在这一部分中隔离性、isolation_level、事务、连接等都指 SQLite 中所定义的概念，务必要和数据库原理中的概念辨析开来），这个策略将会在这个链接上的事务启动时被体现。

事务类型	说明
Deferred	直到必须使用时才获取锁
Immediate	在 begin 执行时试图获取预留锁
Exclusive	试图获取排它锁

在这一节中将通过实验来观察以不同的 isolation_level 建立的连接获得锁的过程以及对其他连接的影响。

6.1.2. 实验设计

在实验中使用两个连接打开同一个数据库，并且分别执行以下三种操作：无操作、读操作、写操作，执行过程如下：

连接 A 访问数据库，使用 Isolation A 作为隔离等级

连接 A 执行操作 Operation A （包括 DoNothing 即不执行操作，Read 即只读数据，Uncommitted 即修改数据但不提交，Committed 即修改并提交数据）

连接 B 访问数据库，使用 Isolation B 作为隔离等级

连接 B 执行操作 Operation B（包括 Read 即只读数据，Write_Un 即修改数据但不执行提交操作，Write_Co 即修改数据并执行提交操作）

6.1.3. 实验结果

此处只列出发生异常的情况，完整表格请看“\src\6_isolation\6_1_link&lock\结果.xlsx”文件

Isolation A	Operation A	Isolation B	Operation B	结果
DEFERRED	Uncommitted	DEFERRED	Write_Un	Connection B 执行插入操作失败（database is locked）
DEFERRED	Uncommitted	DEFERRED	Write_Co	Connection B 执行插入操作失败（database is locked）
DEFERRED	Uncommitted	IMMEDIATE	Write_Un	Connection B 执行插入操作失败（database is locked）
DEFERRED	Uncommitted	IMMEDIATE	Write_Co	Connection B 执行插入操作失败（database is locked）
DEFERRED	Uncommitted	EXCLUSIVE	Write_Un	Connection B 执行插入操作失败（database is locked）
DEFERRED	Uncommitted	EXCLUSIVE	Write_Co	Connection B 执行插入操作失败（database is locked）
IMMEDIATE	Uncommitted	DEFERRED	Write_Un	Connection B 执行插入操作失败（database is locked）
IMMEDIATE	Uncommitted	DEFERRED	Write_Co	Connection B 执行插入操作失败（database is locked）

IMMEDIATE	Uncommitted	IMMEDIATE	Write_Un	Connection B 执行插入操作失败（database is locked）
IMMEDIATE	Uncommitted	IMMEDIATE	Write_Co	Connection B 执行插入操作失败（database is locked）
IMMEDIATE	Uncommitted	EXCLUSIVE	Write_Un	Connection B 执行插入操作失败（database is locked）
IMMEDIATE	Uncommitted	EXCLUSIVE	Write_Co	Connection B 执行插入操作失败（database is locked）
EXCLUSIVE	Uncommitted	DEFERRED	Read	Connection B 执行读取操作失败（database is locked）
EXCLUSIVE	Uncommitted	DEFERRED	Write_Un	Connection B 执行插入操作失败（database is locked）
EXCLUSIVE	Uncommitted	DEFERRED	Write_Co	Connection B 执行插入操作失败（database is locked）
EXCLUSIVE	Uncommitted	IMMEDIATE	Read	Connection B 执行读取操作失败（database is locked）
EXCLUSIVE	Uncommitted	IMMEDIATE	Write_Un	Connection B 执行插入操作失败（database is locked）
EXCLUSIVE	Uncommitted	IMMEDIATE	Write_Co	Connection B 执行插入操作失败（database is locked）
EXCLUSIVE	Uncommitted	EXCLUSIVE	Read	Connection B 执行读取操作失败（database is locked）
EXCLUSIVE	Uncommitted	EXCLUSIVE	Write_Un	Connection B 执行插入操作失败（database is locked）
EXCLUSIVE	Uncommitted	EXCLUSIVE	Write_Co	Connection B 执行插入操作失败（database is locked）

6.1.4. 结果分析

从上述结果中可以注意到仅有当连接 A 执行了写操作且未提交时连接 B 的操作才可能发生异常，推测这是由于 sqlite3 模块中将一条语句作为默认的事务粒度的原因，即执行一条 SQL 语句会被自动执行为一个事物，事务完成后就会立即释放该事务所获得的锁，因此虽然连接 A 和连接 B 同时打开数据库，但是其对数据库的访问事务并不是真正并发的，仅当 A 执行写未提交时虽然相应的事务完成，但是锁仍然没有释放，因此会使 B 的操作被拒绝。

同时我们可以注意到所得获取并不像实验说明中所预测的那样在连接建立时就被自动获取（IMMEDIATE 连接立即获取预留锁，EXCLUSIVE 连接立即获取排他锁），否则 EXCLUVE-XXX-EXCLUSIVE-XXX 都将在连接建立时就被拒绝，推测这是来源于 sqlite3 模块的自动优化调度。首先，针对一个连接，仅当其执行某一个操作时才启动事务并获得锁。其次，其获得的锁并不会满足一定的限度，并不会增加不必要的“安全性”。

可见 sqlite3 模块中的隔离性这一概念不仅和数据库原理中的隔离性相差较大，其自身的定义仍然较为模糊，本次实验也没有真正做到并发情况下的尝试，仅仅只能作为一个经验上的参考，并没有真正窥探到其本质。

6.2. 数据隔离性测试

6.2.1. 实验说明

在 6.1 节中针对 sqlite3 模块中定义的连接的隔离性进行了尝试，这一节中将从传统意义上的“隔离性”这一概念出发，考察 sqlite3 模块提供的接口的行为。

6.2.2. 实验设计

本实验共分为三个部分：

第一部分中，先创建连接 1，并依次执行插入、删除与提交操作，每次操作之后都创建并记录新连接，同时从每个连接进行读操作，比对每次每个连接读取到的内容的区别。

第二部分中，先建立连接 1，再建立连接 2，连接 2 依次执行插入、删除与提交操作，每次操作之后都从连接 1 和连接 2 分别执行读操作，比对每次每个连接读取到的内容的区别。

第三部分中，与第一部分相同，不过此时针对的对象是同一个连接中所创建的不同游标对象。

6.2.3. 实验结果

第一部分实验：

连接 1 创建	
连接 1 读取	1、2、3、4
连接 1 插入 5、6	
连接 2 创建	
连接 1 读取	1、2、3、4、5、6
连接 2 读取	1、2、3、4
连接 1 删除 3、4	
连接 3 创建	
连接 1 读取	1、2、5、6
连接 2 读取	1、2、3、4
连接 3 读取	1、2、3、4
连接 1 提交	
连接 4 创建	
连接 1 读取	1、2、5、6
连接 2 读取	1、2、5、6
连接 3 读取	1、2、5、6
连接 4 读取	1、2、5、6

第二部分实验：

连接 1 创建	
连接 2 创建	
连接 2 插入 5、6	
连接 1 读取	1、2、3、4
连接 2 读取	1、2、3、4、5、6
连接 2 删除 3、4	
连接 1 读取	1、2、3、4
连接 2 读取	1、2、5、6

连接 2 提交	
连接 1 读取	1、2、5、6
连接 2 读取	1、2、5、6

第三部分实验：

连接创建	
游标 1 创建	
游标 1 读取	1、2、3、4
游标 1 插入 5、6	
游标 2 创建	
游标 1 读取	1、2、3、4、5、6
游标 2 读取	1、2、3、4、5、6
游标 1 删除 3、4	
游标 3 创建	
游标 1 读取	1、2、5、6
游标 2 读取	1、2、5、6
游标 3 读取	1、2、5、6
游标 1 提交	
游标 4 创建	
游标 1 读取	1、2、5、6
游标 2 读取	1、2、5、6
游标 3 读取	1、2、5、6
游标 4 读取	1、2、5、6

6.2.4. 结果分析

由上述实验结果可知，sqlite3 中的隔离层次类似于可串行化层次，既没有脏读、不可重复读，也不会产生幻影元组。由于上述实验中连接均创建于 DEFERRED 层级，可想而知 IMMEDIATE 与 EXCLUSIVE 层级均必然满足这一性质，因此在数据隔离性方面 sqlite3 模块始终确保最强的隔离性。

针对同一个连接建立的不同游标，其对数据库的访问是完全共享的，在实验中已经确保这些游标是不同的实例。

7. 总结

本文的目的在于对 python 中 sqlite3 模块的说明，从 API 出发，在第 3 节中先详细解释了一些常用的接口及其参数，确保读者对其有一个清晰的认识。在第 4 节中利用代码实例说明了常规接口的使用方法，并针对功能相似的接口以及同种对象的不同用法特性做了辨析，为日常代码编写的方法与流程提供了参考。接下来在第 5 节中，通过具体实验与图标分析了各种可能会对程序性能造成影响的因素，为代码的性能调优提供了借鉴。最后在第 6 节中通过实验分析了 sqlite3 中提供的连接隔离性与数据隔离性的特性，为实际的操作正确性提供了保障。

在编写本文的过程中我对 sqlite3 模块的认识也在逐步加深，从最初的只能对照官方文档写程序，到后来可以独立编写完整逻辑并利用一些对象的高级特性，再后来对于这个模块中相对底层的性质也有了定性层面的认识——包括功能类似的不同接口的性能差异以及数据库中事务与锁的获取与释放等等。

同时，在 sqlite3 这个模块之外，我在编写本文时体验了自己设计实验、分析结果、修改实验的过程，第 5、6 节中的实验对象与实验过程设计都是我多次修改反复尝试之后得到的结果，我尽力地将更多有可比性的对象归入一组，并将实际上无关的内容去除。

综上所述，本次报告的编写过程既是我对 sqlite3 这个库逐步了解的过程，也是我自身实验能力得到锻炼的过程，同时我希望我在这篇报告中所解释的内容以及完成的实验可以为其他希望学习 sqlite3 这个模块的同学提供一定的帮助。