SQLite数据库文件格式

知 zhuanlan.zhihu.com/p/132694244

Coldwind程序员

本文翻译自SQLite官网文档SQLite File Format

Database File Format www.sqlite.org/fileformat2.html

1 名词解释

数据库中的概念

Table：数据库中的表，下文称“table”或者“表”。

Column：表中的各个字段，下文称“column”或者“列”或者“字段”。

Row：表中的各条记录，下文称“row”或者“行”

Index：表中的索引，用户可以建立索引以便加速搜索，但是用户无法直接使用索引，下文称“index”或者“索引”。

View：数据库中的视图，一种由实际的表导出的可视化的表，并不实际存储。

Virtual table：虚拟表是一种表现得像表的对象，从SQL语句的角度看，虚表可以和表或者view一样操作，但是对虚拟表的查询或者修改操作会调用注册在虚拟表上的回调函数，虚拟表机制使程序可以提供类似于SQL的表的接口供SQL语句操作。隐藏在虚拟表下的数据结构可能是内存中的数据，或者通过即时运算得出的结果，或者是磁盘上的文件（比如CSV）。下文称“virtual table”或者“虚拟表”。

Shadow table：FTS（全文搜索）中所使用的每个virtual table，都有3-5个真实的数据库的table（分别名为%_content、%_segdir、%_segment、 %_stat、%_docsize，%是FTS virtual table的名字）来在实现，这些table被称为shadow table。

Trigger：数据库中的触发器，由修改数据库的事件触发的存储过程，下文称“触发器”或者“trigger”。

Schema：SQLite数据库的结构（有哪些table/index/view/trigger，分别有哪些字段），下文称“schema”。

Rowid：rowid是SQLite中的表隐含的一个column，是其内部id，在该表中唯一，是SQLite中的元数据。

Statement：SQL语句。

Prepared statement：经过“预备”的SQL语句，所谓“预备”类似编译，可以再多次执行同一语句的时候加速（跳过“预备”过程）。

sqlite_master：sqlite数据库中维护的系统表，该表的b-tree的根页号永远为1，有5个列，分别是类型（table, view, index,trigger，四者之一）、名称、所在表名、根页号、SQL语句。

Journal：日志

Transaction：事务是用户定义的一系列数据库操作，要么全部执行，要么全部不执行。

Magic string：类似“魔数/幻数”，SQLite数据库文件特征头。

Fraction

Auto-vacuum：自动清空

Incremental-vacuum

BLOB：Binary Large OBject

关于b-tree的概念

Key：b-tree中的键值，在一个b-tree中是唯一的，在SQLite中的实现为rowid，如果是WITHOUT ROWID表，则整个row被视为一个key（通过primary key排序）。请注意这个key和SQL中的primary key的区别！

Data：b-tree中的数据

Pointer：指向子页的指针（页号）

Interior page：内部页，有子页的b-tree页

Root page：根页，b-tree中没有父节点的interior页。

Leaf page：没有子页的b-tree页

Table b-tree：存放rowid表（即普通的表）的b-tree，由算法中的B*-tree实现。

Index b-tree：存放WITHOUT ROWID表和index的b-tree，由算法中的B-tree实现。

Cell：页中的一个单元，根据b-tree类型的不同和页的不同可能由Key、Data、Pointer中的1-3种组成。

Payload：有效载荷，cell中变长的部分，Index b-tree中是key，table b-tree中是data，对应table或者index中的一行。

Record：payload在底层的组成格式

数据库文件

一个完整状态的SQLite数据库通常是一个叫做“main database file （主数据库文件）”的磁盘文件。在一个事务（transaction）中，SQLite会将事务处理信息保存在一个叫做“回滚日志（rollback journal）”的文件中，如果SQLite处于WAL模式，则会保存在一个预写日志文件（WAL文件，write-ahead log文件）。如果在一个transaction完成之前，应用或者主机崩溃，那么rollback journal或者write-ahead log里保存的信息必须要将主数据库文件回复到一个一致、完整的状态。如果一个rollback journal/write-ahead log里包含了恢复数据库所必要的信息，则被称为“热日志（hot journal）”或者“热WAL文件”。本文档定义了rollback journal和WAL文件的格式，但是主要讲的是主数据库文件本身。

页

主数据库文件包括了1个以上的页。页的大小是2的幂，可以从512到65536.同一个数据库中的所有页的大小是一样的。页大小在数据库文件偏移16个字节的地方定义。

页的序号从1开始，最大的页号是2147483646 (231 - 2)。SQLite数据库最小是512字节（只有一个512的页）。最大可以有2147483646 个65536的页，也就是140,737,488,224,256 字节(大约140TB)，不过通常文件系统的限制会小于这个值。通常一个数据库的大小从几K到几G。

在任何时候，主数据库中的任意一个页只能是下面几种之一：

- lock-byte页
- freelist页
    - freelist trunk页
    - freelist leaf页
- b-tree页
    - table b-tree interior页
    - table b-tree leaf页
    - index b-tree interior页

    - index b-tree leaf页

- payload溢出页
- pointer map页

所有对数据库的读/写都是以页为单位进行的，只有一种情况例外，就是当打开数据库时，对数据库文件前100个字节（数据库头）的读取是小于页大小的。

在任何包含信息的页被修改之前，页中未经修改的原始内容会写入到rollback journal中。如果一个transaction被打断并且需要回滚，rollback journal就可以用于将数据库回复到之前的状态。Freelist leaf页中不包含信息，因此在修改前不会被写入journal，以便减小磁盘I/O。

数据库头

最前面的100个字节组成了数据库文件头✦，以下是各个字段的定义，所有的多字节字段均以big-endian存储：

数据库头格式

偏移	长度	描述
0	16	Magic头字符串 "SQLite format 3\000"
16	2	数据库页大小，必须是2的幂，范围从 512到 32768（闭区间），或者为1表示65536。
18	1	文件格式写版本，1为legacy（journal）；2为WAL
19	1	文件格式读版本，1为legacy（journal），2为WAL
20	1	每页尾部保留的空间大小，通常是0。
21	1	最大embedded payload fraction，必须是64.
22	1	最小embedded payload fraction，必须是32.
23	1	Leaf payload fraction，必须是32.
24	4	文件修改次数
28	4	数据库总页数
32	4	第一个freelist trunk页的页号
36	4	所有freelist页的总数
40	4	schema cookie.
44	4	Schema格式版本号. 当前支持的schema格式版本号包括1, 2, 3, and 4.
48	4	默认页缓存大小
52	4	在auto-vacuum或者incremental-vacuum模式时候，最大的b-tree根页号；否则为0
56	4	数据库文本编码格式。 1：UTF-8 2：UTF-16 little endian 3：UTF-16 big endian
60	4	用户版本号
64	4	非0时表示 incremental-vacuum模式，否则为0
68	4	应用程序ID
72	20	保留空间，必须为0
92	4	version-valid-for number.
96	4	最近写过数据库文件的SQLite库版本号

知乎 @ Jun Zheng

Magic头字符串

每个合法的SQLite数据库+都以以下16个字节开头(hex): 53 51 4c 69 74 65 20 66 6f 72 6d 61 74 20 33 00。即"SQLite format 3"

页大小

偏移16开始的2个字节表示页大小，对于3.7.0.1以及之前的SQLite，该值是一个big-endian的整数，必须是2的幂，从512（包含）到32768（包含）。从3.7.1（2010-08-23）开始，65536个字节的页也被支持，用0x00 0x01表示。

文件格式版本号

位于偏移18以及19的文件格式读版本及写版本有同样的定义：为1表示rollback jouranl模式，为2表示WAL journal模式。如果遇到一个数据库文件的读版本为1或2，而写版本大于2，则该数据库文件必须被视为只读。如果读版本大于2，则该数据库文件不能被读写。

页保留空间大小

SQLite可以保留每页最后的一小段空间用作其他用途，比如在加密的SQLite文件中，可以用作nonce和/或checksum。该大小在偏移20，表示页最后的保留大小，可以为奇数。用页大小（偏移16）减去保留大小，即为“可用大小”，可用大小不得小于480，换句话说，如果页大小为512，保留大小必须小于等于32.

Payload fraction

最大和最小embedded payload fraction以及leaf payload fraction必须为64，32,32。在设计之初，这几个值是被设想成可调节的，以便调整b-tree算法的存储格式，但是实际实现中并没有支持，并且当前也没有任何计划去支持它，因此这三个值是固定的。

文件修改次数

文件修改次数是一个位于偏移24的4字节、big-endian的整数，每次数据库文件的修改并解锁之后都会加1。这样当多个进程同时读取同一个数据库文件时，每个进程都可以监控文件是否被修改。如果一个进程修改了数据库文件，其他进程通常需要重新刷新它们的页缓存。

在WAL模式中，对数据库的修改会通过wal-index体现出来，因此在WAL模式中，每次transaction文件修改次数可以不做增加。

数据库大小

位于偏移28的4字节big-endian整数代表数据库文件的大小（页数），如果这个值无效（见下段），则数据库大小会通过实际的数据库文件大小来计算。老版的SQLite忽略该字段而仅仅使用实际文件大小。新版的则会使用该字段，但当字段无效时使用实际文件大小。

只有当文件修改次数（偏移24）和version-valid-for number（位于偏移92）相同，并且数据库大小不为0的时候，数据库大小才有效。当数据库只被新版（3.7.0及之后）的SQLite修改过时，数据库大小总是有效的。老版的SQLite对数据库文件做修改后，数据库大小不会被更新，因此此时数据库大小是错误的，但是同样老版的SQLite也不会修改version-valid-for number，所以这个值会和文件修改数不匹配，因此新版的SQLite可以发现这个错误。

空闲页列表

数据库文件中的空闲页被保存在一个空闲页表freelist中，位于偏移32的4字节big-endian整数代表freelist的第一个页的页号。如果该值为0则表示freelist为空。位于偏移36的4字节big-endian整数则表示了freelist中所有空闲页的总数。

Schema cookie

偏移40处的4字节big-endian整数代表Schema cookie，相当于该数据库schema的版本号，每次数据库schema发生改变都是使该值增加。每条prepared statement都是针对某个版本的数据库schema的，当schema发生改变时，statement必须重新prepare。每当一个prepared statement运行时，会首先检查当前schema cookie是否和该statement prepare的时候的值一致，如果不一致则重新prepare并重新运行该语句，或者返回SQLITE_SCHEMA错误。

Schema格式版本号

位于偏移44的4字节big-endian整数表示Schema格式版本好。Schema格式版本号有点类似文件格式读版本/写版本（偏移18、19）。Schema格式版本号代表上层的SQL格式，而不是底层的b-tree格式+。当前的四个schema格式版本号分别代表以下含义：

1. 格式1：被3.0.0（2004-06-18）之后所有的SQLite版本支持
2. 格式2：增加了对增减数据库表中列的支持，以便支持ALTER TABLE…ADD COLUMN语句。SQLite 3.1.3（2005-02-20）增加了对格式2的读写支持。
3. 格式3：增加了对ALTER TABLE…ADD COLUMN语句非空缺省值的支持，从SQLite 3.1.4（2005-03-11）开始支持格式3。
4. 格式4：增加了index声明时对DESC关键字的支持（之前的版本1,2,3都忽略了DESC关键字），此外格式4还增加了对2种boolean记录类型的支持(serial types 8 and 9)，SQLite 3.3.0（2006-01-10）开始支持格式4.

当前的SQLite在新建数据库时缺省使用格式4，也可以使用legacy_file_format pragma（老版文件格式pragma）以使得SQLite在新建数据库文件的时候使用格式1。在编译期间设置SQLITE_DEFAULT_FILE_FORMAT=1，可以使得缺省的schema格式版本号被设置为1。

缺省缓存大小

位于偏移48的4字节big-endian整数表示的是缺省的数据库文件缓存大小（单位为页）。该值是个建议值，也就是说SQLite并非必须要遵守这个值。可以通过default_cache_size pragma语句来设置这个值。

Incremental vacuum设置

位于偏移52和64的两个4字节big-endian整数用于管理auto-vacuum模式和incremental-vacuum模式。如果位于偏移52的值为0，那么pointer-map（ptrmap）页将被省略，并且auto-vacuum和incremental-vacuum模式都不被支持。如果偏移52的值非0，则该值表示数据库中的最大根页（root page）号，数据库文件中将包含ptrmap页，并且数据库为auto-vacuum（偏移64的值为false）或者incremental-vacuum模式（偏移64的值为true）。如果偏移52的值为0，则偏移64的值必须为0。

文本编码

位于偏移56的4字节big-endian整数表示所有字符串的编码格式。值为1表示UTF-8，值为2表示UTF-16 little-endian，值为3表示UTF-16 big-endian，其他值均为非法。在头文件sqlite3.h中定义了宏SQLITE_UTF8=1，SQLITE_UTF16LE为2，SQLITE_UTF16BE为3。

用户版本号

位于偏移60的4字节big-endian整数表示用户版本号，用户版本号供用户程序使用，SQLite本身并不使用这个值，通过user_version pragma语句读取和设置这个值。

应用程序ID

位于偏移68的4字节big-endian整数为应用程序ID，可以用PRAGMA application_id语句设置和读取。该值用于标示数据库文件属于哪个应用程序。应用ID标示了使用该数据库文件的应用所定义的文件格式（application file-format），类Unix系统中的file命令可以通过这个值来判断数据库文件属于某个特定的应用（而不仅仅给出“SQLite3 Database”这个结果）。已分配的应用ID可以通过SQLite源文件中的magic.txt文件来查看。

写库的版本号和version-valid-for number

位于偏移96的4字节big-endian整数表示最近修改过该数据库文件的SQLite库的版本号。位于偏移92的4字节big-endain整数和文件修改次数（偏移24）的值相同（在新版本的SQLite对数据库文件做修改后，旧版本的SQLite不会修改这个值）。

保留空间

所有没在上面定义的字节（从偏移70开始的20个字节）为保留空间，为之后可能的用途准备，必须被设置为0。

Lock-Byte页

Lock-byte页是一个包含数据库文件偏移1073741824到1073742335的页。小于1073741824的数据库没有lock-byte页，大于1073741824的数据库有且仅有一个lock-byte页。

SQLite中某些特别的操作系统VFS（SQLite的操作系统接口）实现会用Lock-byte页来对数据库文件进行锁操作。SQLite核心（操作系统无关）不使用这个页，并且永远不会读/写该页。SQLite中Unix和Win32的VFS实现不会写这个页，但是第三方实现的其他操作系统的VFS也许会用到这个页。

Lock-byte页的出现是因为对Win95的支持，Win95只有强制文件锁。所有的已知的现代操作系统都支持非强制的文件锁，因此现在lock-byte页已经不再有意义，但是仍然保留以便向前兼容。

Freelist页

数据库文件可能包含若干未使用的空闲页。页中的信息被删除可能会导致出现空闲页。空闲页存储在空闲页列表freelist中，当需要额外的页时，空闲页会被重新使用。

freelist是一个freelist trunk页的链表，每个freelist trunk页中包含了0个或以上freelist leaf页的页号。

Freelist trunk页由4字节big-endian整数的数组组成，这个数组的长度由页中可用空间的大小（参考1.2.4）决定。最小的页可用空间大小是480个字节，所以数组最少会有120个元素。Freelist trunk页中的第一个4字节整数表示的是下一个freelist trunk页的页号，如果为0则表示该页是freelist中最后一个trunk页。第二个整数表示的是leaf页指针的数量，如果该整数L大于0，则每个下标为2到L+1之间（闭区间，即第3到L+2个，共L个）的整数均表示一个freelist leaf页的页号。

Freelist trunk页结构

下一个 freelist trunk page页号 （0表示最后一个） （4字节）	该trunk页中 leaf页指针的数量 （4字节）	第一个 freelist leaf页 的页号 （4字节）	第二个 freelist leaf页 的页号 （4字节）	.....	最后一个 freelist page页 的页号 （4字节）	页保留区

Freelist leaf页中没有任何信息。SQLite不会读写freelist leaf✦页，以避免额外的磁盘I/O。

SQLite 3.6.0之前版本中有一个bug：如果freelist trunk页中最后6个元素中的任何一个不为0，则会导致数据库文件被认为是错的。之后的版本修正了这个问题。然而，新版本的SQLite仍然避免使用freelist trunk页的最后6个元素，以免当数据库文件被旧版本的SQLite使用的时候发生错误。

Freelist页的数量保存在数据库头（即文件头）的偏移36处，第一个freelist trunk页的页号则保存在数据库头偏移32处。

B-tree页

B-tree算法提供了key/data在按页分配的存储设备上的保存方法，这些key/data对具有唯一且有序的key（关于b-tree的背景知识请参考Knuth的The Art Of Computer Programming, 第3卷 "Sorting and Searching", 第471-479页）。

SQLite中使用了两种b-tree。书中被Knuth叫做“B*-tree✦”的算法将所有的data保存在树的leaf节点中。SQLite管这个b-tree的变种叫做“table b-tree”。书中被Knuth简单叫做“b-tree”的算法将key和data保存在leaf页和interior页中。在SQLite实现中，这种原始的b-tree算法仅保存key，而不保存data。SQLite管这个b-tree叫做“index b-tree”。

一个b-tree页要么是interior页，要么是leaf页。Leaf页中保存了key，如果是table b-tree，每个key都有对应的data。Interior页保存了K个key和K+1个指向子b-tree页的指针。这里的“指针”其实就是31-bit的子页页号。

两种b-tree、key/data/指针、四种页、index、table、payload、cell、record等之间的关系总结如下：

```
Table b-tree（B*-tree）：用于保存普通表（rowid table）
      Interior page：（cell）
            Key：rowid
            指针：子页的页号
      Leaf page：（cell）
            Key：rowid
            Data：表中的行/记录（payload/record）
Index b-tree（B-tree，无 data）：用于保存 index/WITHOUT ROWID 表
      Interior page：（cell）
            Key：（payload/record）
                Index：index 中的行
                WITHOUT ROWID 表：表中的行/记录
            指针：子页的页号
      Leaf page：（cell）
            Key：（payload/record）
                Index：index 中的行
                WITHOUT ROWID 表：表中的行/记录
```

知乎 @Jun Zheng

b-tree leaf页的深度被定义为1，任意interior页的深度都是其最大子页深度+1。在一个组织良好的数据库中，一个interior b-tree的所有子节点具有同样的深度。

在一个interior b-tree页中，指针和key交替出现，最左和最右都是指针 （形似Pointer1 Key1 Pointer2 Key2 … PointerN，这里的左右概念指的是逻辑上的而非物理上的，真实的物理布局略复杂，请参考后面章节）。同一页中的key具有唯一的值，并且从左往右升序（这是个逻辑概念，而非物理概念，页中的key的实际位置是任意的），对于任一key X，在X左边的指针指向的页中所有的key均小于等于X，在X右边的指针指向的页中所有的key均大于等于X。

在一个interior b-tree页中，每个key和紧邻它左边的指针形成了一个叫做“cell”的结构，页中最右边的指针则单独存放。Leaf b-tree页中没有指针，但是index b-tree中的leaf页仍然使用cell结构来存放key，table b-tree leaf页也仍然使用cell结构来存放key和data。Data也被存放在cell中。

每个b-tree页最多有一个父b-tree页。一个没有父页的b-tree页被称为root页。一个root页及其所有的子孙页形成了一个完整的b-tree。一个b-tree可以由单个页组成（事实上这种情况很常见），该页既是root页也是leaf页。因为页中有从父页指向子页的指针，因此一个b-tree可以仅

仅通过root页的页号来确定和遍历。

一个b-tree页要么是table b-tree页，要么是index b-tree页。同一个b-tree中的所有页都是同一种类型的：table或index。数据库schema✦中的每一个rowid table对应一个数据库文件中的table b-tree，包括系统table比如sqlite_master。数据库schema中的每一个index，对应于数据库文件中的一个index b-tree，包括因为唯一性而创建的隐含index。Virtual table并不对应任何b-tree，某些特殊的virtual table可能会使用其下的shadow table来存放数据。没有rowid的table（WITHOUT ROWID）使用index b-tree而非table b-tree，所以每个WITHOUT ROWID table都有一个对应的index b-tree。sqlite_master系统表所对应的b-tree是一个table b-tree✦，并且其root page页号永远是1。sqlite_master表中包含了数据库文件中其他table和index的root page页号。

Table b-tree中的每一个entry由一个64位的符号整数的key和最多2147483647字节的任意长度data组成。（table b-tree中的key对应于SQL表中的rowid）。Interior table b-tree页中只有key和指向其子页的指针，所有的data都存放在table b-tree leaf页中。

Index b-tree中的每一个entry由一个任意长度的key（长度最多为2147483647字节）构成，没有data。

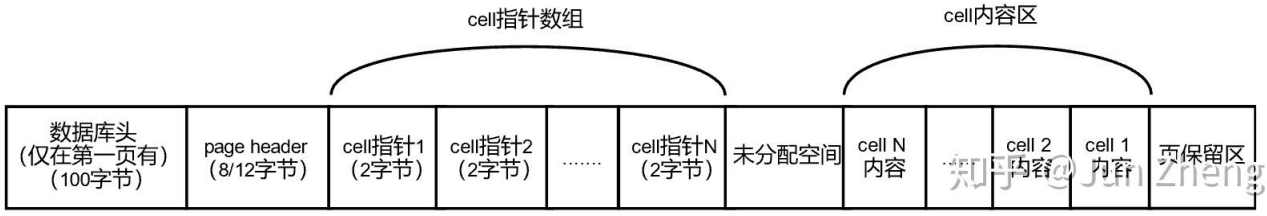
一个cell中的payload指的是这个cell中任意长度的部分。对于一个index b-tree来说，因为key的长度是任意的，因此key就是payload。在interior table b-tree页中没有任意长度的部分，因此这些cell就没有payload。Table b-tree leaf页中包含任意长度的的内容，因此那些cell中的payload就是这些内容。

当一个cell中的payload超过一个特定的阀值的时候，只有最前面的部分会被保存在这个b-tree页中，其它部分会被保存在一个由溢出页构成的链表中。

一个b-tree页被按顺序分成以下几个部分：

1. 100个字节的数据库文件头 （只有在第1页中有）
2. 8或者12字节的页头
3. Cell指针数组
4. 未分配空间
5. Cell内容区
6. 保留空间

b-tree页格式



知乎 @ Jun Zheng

100个字节的数据库文件头只会在第1页中存在，这个页永远是个table b-tree页。数据库文件中所有其他的b-tree页都不会有这个100字节的文件头。

保留空间是一段每个页中都会有的未使用的区域（除了lock-byte页），某些扩展程序可以用它来保存一些与页相关的信息。这个保留空间的大小由数据库文件头偏移20的无符号整数决定（参见1.2.4）。通常来说，这个保留空间的长度为0。

b-tree leaf页头的长度为8个字节，interior页头的长度为12个字节。页头中所有超过一个字节的数都是big-endian的。B-tree页头由以下几个部分组成：

B-tree页头格式

偏移	大小	描述
0	1	偏移0处的单字节标志表示了b-tree页的类型： • 2（0x02，0010）表示该页是index b-tree的interior页 • 5（0x05，0101）表示该页是table b-tree的interior页 • 10（0x0a，1010）表示该页是index b-tree的leaf页 • 13（0x0d，1101）表示该页是table b-tree的leaf页 所有其他值都是非法的
1	2	这个双字节整数表示的是页中第一个空闲块的位置，如果为0则表示没有空闲块。
3	2	这个双字节整数表示的是页中cell的数量
5	2	这个双字节整数表示的是cell内容区的位置，如果为0表示位于偏移65536
7	1	表示cell内容区中的空闲碎片的字节数
8	4	这个4字节整数表示的是（逻辑上）最右边的指针所指向的页号。只有在interior b-tree页中才有该值。

b-tree页中的Cell指针数组紧接着b-tree页头，设K为页中cell的个数，则cell指针数组中包含着K个指向cell内容的指针，每个cell指针为2字节整数。Cell指针以key顺序排列，最左边的（最小key）cell最先，最右边（最大key）的cell最后。

Cell内容保存在b-tree页中的cell内容区。SQLite尽量将cell内容放在b-tree页的最后，这样可以空出空间以便cell指针数组将来的增长。最后一个cell指针和第一个cell内容之间的区域为空闲区。

如果一个页没有cell（只有当该页是一个空表的root页才可能），那么cell内容区的偏移就是页大小减去保留空间的大小。如果数据库使用65536字节的页大小（最大可能）并且保留空间为0（通常都为0），那么在该页为空的情况下，cell内容区的偏移就是65536，然而该值无法存放在一个2字节无符号整数中（最大只能表示65535），因此此时用0来表示偏移65536。

空闲块freeblock用于定位b-tree页中未分配的空间。Freeblock是一个链表，空闲块中的头2个字节是一个big-endian整数，用于表示b-tree页中下一个空闲块的偏移，为0则表示该空闲块是链表中最后一个空闲块。空闲块中第3个和第4个字节组成的big-endian整数表示这个空闲块的长度（字节数，包括前4个字节）。Freeblock链表总是以偏移从小到大排列。B-tree页头中的

第2个字段表示的freeblock链表中第一个空闲块的偏移，为0则表示该页中没有空闲块。在一个组织良好的b-tree页中，应该至少有一个cell在第一个空闲块之前（否则该空闲块应该被并入cell指针数组和cell内容区中的未分配空间中）。

一个空闲块需要至少4个字节（2字节用于存放下一个空闲块指针，2字节用于存放本空闲块长度），cell内容区中的1、2或者3个字节长的未使用空间形成了碎片。所有碎片的长度总和（以字节为单位）保存在b-tree页头的第5个字段中。在一个组织良好的b-tree页中，这个值不应该超过60。

一个b-tree页中空闲空间的总大小由（cell指针数组和cell内容区之间的）未分配空间大小、所有空闲块的总大小、碎片总大小三个部分组成。SQLite会时不时的整理b-tree页，这样页中就没有空闲块和碎片，所有的空闲空间都包括在未分配中间中，而所有的cell都紧密的挤在页的最后。这个整理过程被称作“defragmenting”（去碎片化）。

Varint表示一个变长整数，是霍夫曼编码✦的64位补码整数，可以用更小的空间占用来表示较小的正值。Varint的长度介于1到9个字节之间。Varint有两种格式：一种由若干个（0个或以上）最高位为1的字节，及跟在后面的1个最高位为0的字节组成；另一种由9个字节组成，哪个短就用哪种格式。前8个字节的低7位和第9个字节的所有8位组成了一个64位的补码整数。Varint是big-endian✦的：从前面的字节取出的bit排在高位，而从后面的字节取出的位在低位。

Cell的格式跟其所在b-tree页的类型有关，下表是构成cell的元素：

Table B-Tree Leaf Cell （页头标志0x0d）：

- Varint，表示了payload的总字节数，包括所有的溢出。
- Varint，整数key（也就是rowid）
- Payload中没有溢出（从而放到溢出页存储）的部分
- 4字节big-endian整数，表示第一个溢出页的页号，如果没有溢出页则省略

Table B-Tree Interior Cell （页头标志0x05）：

- 4字节big-endian整数，表示左孩子节点页的页号
- Varint，整数key

Index B-Tree Leaf Cell（页头标志0x0a）：

- Varint，表示了payload的总字节数，包括所有的溢出
- Payload中没有溢出（从而放到溢出页存储）的部分
- 4字节big-endian整数，表示第一个溢出页的页号，如果没有溢出页则省略

Index B-Tree Interior Cell（页头标志0x0a）：

- 4字节big-endian整数，表示左孩子节点页的页号
- Varint，表示了payload的总字节数，包括所有的溢出
- Payload中没有溢出（从而放到溢出页存储）的部分

- 4字节big-endian整数✦，表示第一个溢出页的页号，如果没有溢出页则省略

以上信息可以总结为下表

B-tree Cell 格式

类型	出现在…				描述
	Table Leaf (0x0d)	Table Interior (0x05)	Index Leaf (0x0a)	Index Interior (0x02)	
4字节整数		✔		✔	左孩子节点页的页号
varint	✔		✔	✔	Payload字节数
varint	✔	✔			Rowid
数组	✔		✔	✔	Payload
4字节整数	✔		✔	✔	首个溢出页的页号

知乎 @Jun Zheng

溢出到溢出页的payload✦的数量也依赖于页的类型。我们设：U是页的可用大小（页大小减去页保留空间大小），P是payload大小，X表示能存储在b-tree页而不溢出的最大的payload的大小，M表示存储在b-tree页上的payload的最小值。

Table B-Tree Leaf Cell:

Let X be U-35. If the payload size P is less than or equal to X then the entire payload is stored on the b-tree leaf page. Let M be and let K be M+((P-M)%(U-4)). If P is greater than X then the number of bytes stored on the table b-tree leaf page is K if K is less or equal to X or M otherwise. The number of bytes stored on the leaf page is never less than M.

Table B-Tree Interior Cell:

Interior pages of table b-trees have no payload and so there is never any payload to spill。

Index B-Tree Leaf或者Interior Cell:

Let X be ((U-12)*64/255)-23). If the payload size P is less than or equal to X then the entire payload is stored on the b-tree page. Let M be ((U-12)*32/255)-23 and let K be M+((P-M)%(U-4)). If P is greater than X then the number of bytes stored on the index b-tree page is K if K is less than or equal to X or M otherwise. The number of bytes stored on the index page is never less than M.

以下是上述说法的另一种形式:

- X = U-35（table btree leaf页）或者 ((U-12)*64/255)-23 （index页）。
- M = ((U-12)*32/255)-23。
- 设K为M+((P-M)%(U-4))。
- 如果P<=X 那么payload可以全部保存在b-tree页中。

- 如果P>X并且K<=X，那么P中前K个字节保存在b-tree页中，而剩下的P-K个字节保存在溢出页中。
- 如果P>X 并且 K>X，那么P中前M个字节保存在b-tree页中，而剩下的P-M个字节保存在溢出页中。

溢出阀值给出了index b-tree上最小的payload大小，以确保record头可以全部保存在b-tree页中，而不用转去溢出页读取。从事后经验来看，SQLite b-tree逻辑✦的设计者意识到这些阀值的计算方式其实可以简单很多，可惜的是，没法在兼容之前格式的情况下更改阀值的计算方式。不过现在的这种计算方式也可以正常工作，虽然略微复杂了一点。

Cell溢出页

当一个b-tree cell太大而无法装入一个b-tree页的时候，多余部分会放入溢出页。溢出页会形成一个链表，每个溢出页的头4个字节是链表中下一个溢出页的页号（big-endian），为0则表示是链表中最后一个溢出页。从第5个字节开始的所有有效字节（参考1.2.4）都用于保存溢出的内容。

Cell溢出页格式

Ptrmap页

指针map或ptrmap页是数据库中额外的页，这些页可以使auto_vacuum和incremental_vacuum模式更加高效。数据库中其它类型的页通常都会有父页到子页的指针，比如interior b-tree页中有指向子页的指针，溢出页链表✦中前面的页有指向后一个溢出页的指针。Ptrmap页中包含了相反方向的连接信息——从子页到父页，这样当需要释放一个页的时候，可以迅速通过ptrmap找到并修改它的父页。

在数据库文件头偏移52处的最大根页号如果不为0，则该数据库必须有ptrmap页。如果最大根页号为0，则该数据库不得有ptrmap页。

在一个有ptrmap页的数据库文件中，第一个ptrmap页的页号是2。Ptrmap页由长度为5的元素的数组组成，如果J是页可用空间中允许存在的最大的该数组的元素个数（或者说，J=U/5）。那么第一个ptrmap页将会包含页号从3到J+2的J个页的反向指针。第二个ptrmap页的页号则是J+3，其中将会包含页号从J+4到2*J+3的J个页的反向指针。以此类推。

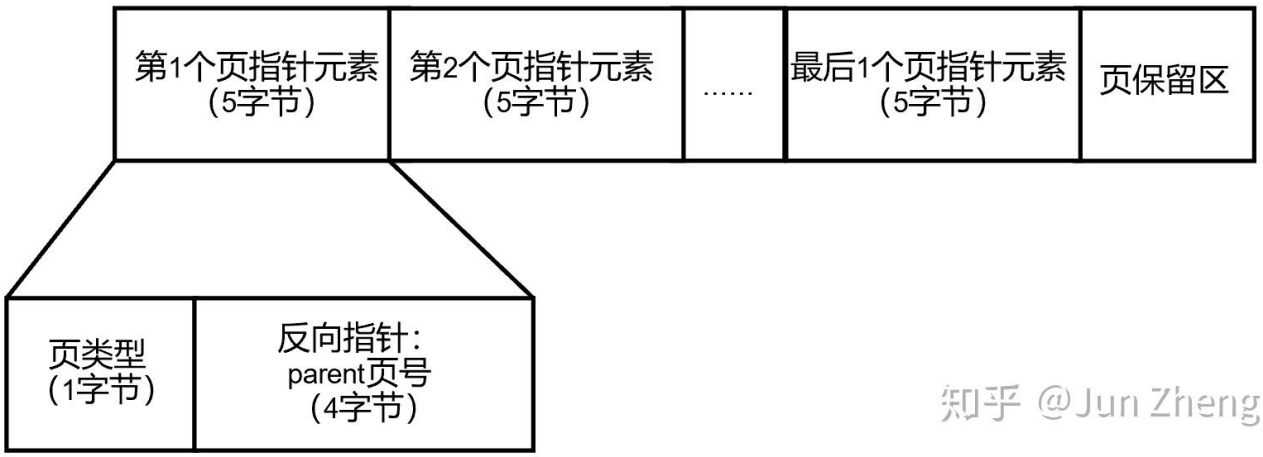
在一个有ptrmap页的数据库中，所有由上述计算方式得出的页必须是ptrmap页，除此之外的其它页不能为ptrmap页。唯一的例外是，当byte-lock页正好落在ptrmap页上，则ptrmap页的位置需要向后移动一页。

Ptrmap页中的每一个5字节数组元素包含了紧跟在该ptrmap页之后的某页的反向指针。如果页B是一个ptrmap页，那么页B+1的反向指针在B的第一个元素中，页B+2的反向指针在B的第二个元素里，以此类推。

每个ptrmap中的的5字节数组元素由1个字节的“页类型”和其后的4字节big-endian页号组成。有5种页类型：

1. B-tree根页。后面的4字节页号应该为0。
2. Freelist页。后面的4字节页号应该为0。
3. Cell payload溢出页链表中的第一个页。后面的4字节页号是指向该链表的b-tree页（即溢出的页）页号。
4. Cell payload溢出页链表中的其它页。后面的4字节页号是链表中前一个页的页号。
5. 一个非root的b-tree页。后面的4字节页号是其父页页号。

Ptrmap页结构



在包含ptrmap页的数据库里，所有的b-tree root页必须在任何非root页、cell溢出页或者freelist页之前。这个限定确保了在auto-vacuum或者incremental-vacuum模式下，root页永远不会被移动。Auto-vacuum并不知道该如何更新sqlite_master表中的root页字段，因此必须确保root页在auto-vacuum过程中不会被移动以保证sqlite_master表的正确性。Root页通过CREATE TABLE、CREATE INDEX、DROP TABLE和DROP INDEX操作被移动到数据库的前面。

Schema层

以下内容描述了SQLite的底层文件格式，b-tree机制提供了用于访问大量数据的高效方式，本章描述了下层的b-tree层如何实现上层SQL的各项功能。

record格式

Table b-tree leaf页中的data，和index b-tree页中的key，被描述为任意长度的数据。之前的章节中提到key之间的大小比对，但并没有定义这里“大小”的意思。本节将会讨论这些。

Payload——或者是table b-tree中的数据，或者是index b-tree中的key——被组织成record的格式。Record格式定义了一系列对应于table或者index中的column的数据。Record格式描述了column的个数，每个column的数据类型，以及每个column的内容。

Record格式使用了变长整数varint来表示64位符号整数。

一个record包括头和正文，record头以一个varint开始，这个varint表示了整个record头的长度（包括这个描述大小的varint本身）。紧接着长度的是若干个varint+，每个column一个。这些varint被称作“serial type”，用于描述column的数据类型及长度：

Record格式中的Serial Type

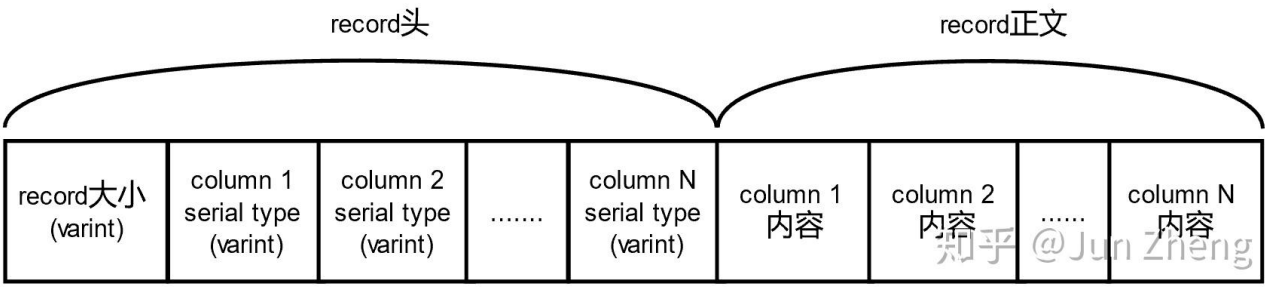
Serial Type	Column 内容长度	含义
0	0	Column内容为NULL
1	1	Column内容是big-endian的8位补码整数
2	2	Column内容是big-endian的16位补码整数
3	3	Column内容是big-endian的24位补码整数
4	4	Column内容是big-endian的32位补码整数
5	6	Column内容是big-endian的48位补码整数
6	8	Column内容是big-endian的64位补码整数
7	8	Column内容是big-endian IEEE 754-2008 64位浮点数
8	0	Column内容是整数0（只有schema版本4以上支持）
9	0	Column内容是整数1（只有schema版本4以上支持）
10,11		*保留未使用*
N≥12 的偶数	(N-12)/2	Column内容是BLOB（Binary Large OBject），且长度为(N-12)/2个字节
N≥13 的奇数	(N-13)/2	Column内容是字符串（格式参考1.2.13），长度为(N-13)/2字节。字符串结尾的NULL不保存。

知乎@Laurence Cheng

描述record头大小和serial type的varint通常只有一个字节。用于描述长字符串和BLOB的serial type的varint可能会扩展到2-3个字节，但是通常来说这是个例。描述record头，Varint是个非常有效的格式。

紧接着record头，record中每个column的值组成了record的正文，对于serial type为0、8、9、12和13的几种类型的column来说，column内容的长度为0。如果所有column都属于以上类型，则整个record正文长度为0。

Record格式



Record的column数量可能会有少于所在表的column数量，比如在使用ALTER TABLE … ADD COLUMN语句之后，表的column数量增加，但是实际存储的record并没变化，缺失的值会用该column的缺省值✦代替。

record排序顺序

index b-tree中的key的顺序由key所代表的record的顺序决定。Record间的比较具体是由左到右的各个column的比较：从左数起第一个不相同的column之间的顺序决定了record的顺序。Column间的比较规则如下：

1. 空值（serial type 0）排最先。
2. 整数值（serial types 1 到 9） 排在空值之后，整数间按数值排序
3. 文本（serial type 大于等于13的奇数） 排在整数之后，文本间的顺序由column比对函数决定。
4. BLOB（serial type 大于等于12的偶数）排在最后，BLOB间的顺序由memcmp()函数决定。

SQLite定义了3种用于比对文本字段的函数：

BINARY

内嵌的BINARY函数通过C库的memcmp()函数按字节比较字符串的大小

NOCASE

NOCASE比对类似BINARY，但是忽略大小写（将大写字母转换成小写字母），但仅仅针对ASCII字符，NOCASE不实现针对unicode的大小写无关比对。

RTRIM

RTRIM类似BINARY，但是字符串尾部的空格不参与比较，换句话说，如果两个字符串的区别仅仅是尾部空格的长度，那么这两个字符串被视为一样大小。

应用程序可以通过sqlite3_create_collation()自己向SQLite添加自定义的比对函数。

文本字段缺省的比对函数是BINARY，如果想使用其他的比对函数，可以通过CREATE TABLE语句中的COLLATE从句来指定。如果一个column被索引了，那么给index排序的函数就是在CREATE TABLE中指定的对比函数，或者可以在创建索引的CREATE INDEX语句中用COLLATE从句覆盖原先的比对函数✦。

SQL表的存储结构

每个数据库schema中普通的（有rowid）SQL表在磁盘上都是通过table b-tree来表示的。Table b-tree中的每个entry对应于SQL表中的一行。Entry中的64位符号整数key对应于rowid。

首先将各个column组成一个record格式的字节数组，数组中record中各个值的排列顺序和各个column在表中的排列顺序一致，接着将该数组作为payload存放在table b-tree的entry中。如果一个SQL表包含INTEGER PRIMARY KEY的column（这样定义的column就是rowid，从而替代原先隐含的rowid），那么这个column在record中的值为空值，SQLite在访问INTEGER PRIMARY KEY的column的时候，总是使用table b-tree的key替代。

如果一个column的affinity（建议类型）是REAL类型，并且包含一个可以被转换为整数的值（没有小数部分，并且没有大到溢出整数所能表示范围）那么该值在record中可能被存储为整数。当将之从record中提取出来的时候，SQLite会自动转换为浮点数✦。

WITHOUT ROWID表的存储结构

如果一个SQL表在创建的时候用了“WITHOUT ROWID”从句，那么该表就是一个WITHOUT ROWID表，在磁盘上的存储格式也和普通SQL表不同。WITHOUT ROWID表使用index b-tree而非table b-tree来进行存储。这个index b-tree中各个entry的key就是record，这些record由PRIMARY KEY 修饰的column打头（WITHOUT ROWID表必须要有primary key，用于替代rowid的作用），其它column在其后面。

所以说，WITHOUT ROWID表的的内容编码和普通的rowid表编码是一样的，区别在于：

1. 将PRIMARY KEY的column提前到最前
2. 将整个record（比较的时候就看PRIMARY KEY，值唯一且不能为空）作为index b-tree中的key，而非table b-tree中的data。

普通rowid表中对REAL建议类型存储的特殊规则，在WITHOUT ROWID表中也同样适用。

Index的存储结构

所有的SQL index，无论是通过CREATE INDEX语句显式创建的，还是通过UNIQUE或者PRIMARY KEY约束隐式创建的，都对应于数据库文件中的一个index b-tree。Index b-tree中的每个entry对应于相应SQL表的一行，index b-tree的key是一个record，这个record由对应表

中被索引的column（一个或者若干）以及对应表的key组成。对于普通rowid表来说，这个key就是rowid，对于WITHOUT ROWID表来说，这个key是PRIMARY KEY，无论哪种情况，这个key都是表中唯一的（请注意这里的key指的不是b-tree的key）。

在普通的index中，原table中的行，和相应的index中的entry是一一对应的，但是在一个partial index（创建index的时候加WHERE从句，对表中部分row索引）中，index b-tree只包含WHERE从句为真的行对应的entry。Index和table b-tree中对应的行使用同样的rowid或primary key，并且所有被索引的column都具有和原表中column同样的值。

上层结构	底层结构	B-tree Key	B-tree data
普通表	Table b-tree	rowid	所有列
WITHOUT ROWID表	Index b-tree	Primary key列 + 其他列	无
INDEX	Index b-tree	Index列 + 表的B-tree key	无
Primary key（普通表）	Index b-tree	Primary key列 + rowid	无
Unique列	Index b-tree	Unique列 + 表的B-tree key	无

知乎 @Jun Zheng

WITHOUT ROWID表中index的冗余抑制

在一个WITHOUT ROWID表中，如果PRIMARY KEY中的一个或多个column与index的column重合，那么这些被索引的column不会重复出现在index record后部的table-key中（index record的前半部分是index的column，后半部分是rowid/primary key）。举例如下：

CREATE TABLE ex25(a,b,c,d,e,PRIMARY KEY(d,c,a)) WITHOUT rowid;

CREATE INDEX ex25ce ON ex25(c,e);

CREATE INDEX ex25acde ON ex25(a,c,d,e);

Index ex25ce中每行的record由如下column构成：c、e、d、a。前面2个column（c和e）是被index的列，后面的column（d和a）是ex25表中的primary key的剩余部分。通常来说，primary key应该是d、c和a，但是因为column c已经在index中了，因此被record中后面的key部分省略了。

在极端情况中：如果所有的PRIMARY KEY列都被索引了，index中将会只包含被索引的列。上面的ex25acde就是例子，每个ex25acde中的entry按顺序只包含a、c、d、e四个column。

以上这种对冗余列的抑制只出现在WITHOUT ROWID表的index里。在一个普通的rowid表中，index的entry永远以rowid结尾，即便INTEGER PRIMARY KEY列也是被index的列之一。

SQL数据库schema的存储

数据库文件中的第1页是名为“sqlite_master”（如果是TEMP数据库就是“sqlite_temp_master”）的系统表的根页，该表保存了整个数据库的结构。这个表的结构相当于使用了以下SQL语句创建：

CREATE TABLE sqlite_master(

type text,

name text,

tbl_name text,

rootpage integer,

sql text

);

Sqlite_master表的每个row代表了数据库schema中的一个table、index、view或者trigger（统称为对象），sqlite_master表本身则不在其中。除了应用程序或者开发者所定义的对象之外，sqlite_master表中页还包含了内部schema对象。

sqlite_master.type列的取值为以下字串之一：“table”、“index”“view”和“trigger”。“table”用于表示普通表和virtual table。

sqlite_master.name列中将存放对象的名字。UNIQUE和PRIMARY KEY约束将会使得SQLite自动创建名为“sqlite_autoindex_TABLE_N”的内部index，名字中的“TABLE”是表名，而“N”是自动创建的index的序号。WITHOUT ROWID表的PRIMARY KEY在sqlite_master中是没有对应的内部index的。

sqlite_master.tbl_name列中存放的是该对象所在的table或者view的名字。对于table和view来说，tbl_name字段的值和name字段一样。对index来说，tbl_name字段放的是其所索引的table的名字。对trigger来说，tbl_name字段存放的是触发该trigger动作的table或者view的名字。

sqlite_master.rootpage列存放的是table和index的b-tree的根页页号。对于view、trigger和virtual table来说，该字段为0或者NULL。

sqlite_master.sql字段存放的而是描述该对象的SQL语句，是“CREATE TABLE”、“CREATE VIRTUAL TABLE”、“CREATE INDEX”、“CREATE VIEW”或者“CREATE TRIGGER”其中之一。该语句通常是创建对象的原始语句的复制，但是会有一些标准化的修改：

- 语句开头的CREATE, TABLE, VIEW, TRIGGER, 和INDEX关键字会被替换为大写。
- CREATE关键字之后的TEMP或者TEMPORARY关键字会被移除。

- 对象名称前的任何数据库名修饰词（database_name.table_name里的database_name）都会被移除。
- 开头的空格会被移除。
- 最前2个关键字（即CREATE XXX）之后的所有空格会被转换为一个空格。

除了这些标准化的修改之外，ALTER TABLE语句给对象的修改也会体现在sqlite_master.sql字段中。如果对象是由UNIQUE或PRIMARY KEY约束所自动创建的内部index，那么该字段为空。

内部schema对象

除了用户和程序显式创建的table/index/view/trigger之外，sqlite_master表✦可能还包含SQLite自己创建的内部schema对象（internal schema object）。内部schema对象总是以“sqlite_”开头，任何以此开头的table/index/view/trigger都是内部schema对象，SQLite页不允许程序或者用户创建以“sqlite_”开头的对象。

SQLite所使用的内部schema对象包括以下：

- 形似“sqlite_autoindex_TABLE_N”的index，由UNIQUE和PRIMARY KEY约束创建。
- 名为“sqlite_sequence”的表用于追踪使用了AUTOINCREMENT约束的表中INTEGER PRIMARY KEY的历史最大值。
- 形似“sqlite_statN”的表，N为整数。这些表保存了用ANALYZE命令得出的数据库统计值，query planner可以利用这些统计数据来选择最佳的查询策略。

在将来，也许会有更多的以“sqlite_”开头的内部schema对象，加入到SQLite文件格式中。

sqlite_sequence表

sqlite_sequence表是一个用于帮助实现AUTOINCREMENT的内部表。每当有任何带有AUTOINCREMENT integer primary key约束的表被创建时，sqlite_sequence表就自动被建立了。一旦建立之后，sqlite_sequence就用会一直续存在sqlite_master表中，并且无法被drop。sqlite_sequence的结构相当于使用以下语句创建：

CREATE TABLE sqlite_sequence(name,seq);

每一个用了AUTOINCREMENT的表都对应sqlite_sequence表中的一行，sqlite_sequence.name是该表的名字（同该表在sqlite_master.name✦中的名字一样），而该表最大的INTEGER PRIMARY KEY被存储在sqlite_sequence.seq字段，自动产生的AUTOINCREMENT integer primary key都保证大于sqlite_sequence.seq字段，如果该字段的值已经达到了最大整数值（9223372036854775807），那么任何试图给该表添加行的操作都会失败，并返回SQLITE_FULL错误。sqlite_sequence.seq字段会在AUTOINCREMENT表插入行之后自动更新。sqlite_sequence表中的行会在对应的AUTOINCREMENT表被drop之后自

动删除。如果AUTOINCREMENT表更新时，sqlite_sequence表中对应的行不存在，则一个新的行会被自动创建。如果sqlite_sequence.seq的值被手动设置为其他值，然后对应的AUTOINCREMENT表中被插入行或者更新，则结果未知。

应用程序可以修改sqlite_sequence表，增加新的行、删除行、或者修改存在的行。但是如果sqlite_sequence表不存在，应用程序不能创建它。应用程序可以删除sqlite_sequence中所有的行，但是却不能drop这个表。

sqlite_stat1表

sqlite_stat1内部表由ANALYZE命令创建，其中保存了数据库中的table和index的补充信息。应用程序可以对该表进行更新、删除、插入行操作，也可以drop该表，但是不能创建，也不能alter该表（的结构）。该表的结构相当于由以下SQL语句创建：

CREATE TABLE sqlite_stat1(tbl,idx,stat);

通常一个index对应sqlite_stat1✦表中的一行，sqlite_stat1.idx为该index的名字，sqlite_stat1.tbl是该index所属的表，sqilte_stat1.stat是一个字符串，该字符串包含一系列整数值：第一个值是该index中大概的row的数量（index中row的数量和index所属表的row数量一样，partial index除外）；第二个值是index中第1个column具有同样值的行的平均个数；第三个值是index中头两个column具有同样取值的行的平均个数；第N个值（N>1）是index中头N-1个column具有同样取值的行的平均个数。对于一个由K个column的index来说，sqlite_stat1.stat列✦会有K+1个整数值。如果index是UNIQUE约束的，那么列中最后一个整数值必然为1（因为没有任何两个index的所有列完全相同）。

sqlite_stat1.stat中的整数列后面有时会有参数，每个参数都是一串非空格字符，每个参数之前都有一个空格。

- 如果有“unordered”参数，说明该index是未经排序的，因此query planner不能通过该index做排序或者范围查询。
- “sz=NNN”参数表示该表或者index中的row的平均长度。SQLite的query planner可以通过这个值选择更小的table和index，以减小磁盘I/O。
- “noskipscan”参数使得该index不会被用于skip-scan optimization。

将来或许会有更多的字串被作为参数加入到sqlite_stat1.stat列中，为了兼容这些可能出现的新的字串，不认识的字串会被简单的忽略掉而不会报错。

如果sqlite_stat1.idx列为空，那么sqlite_stat1.stat列中仅包含一个整数值，该值是sqlite_stat1.tbl这个表的行数。如果sqlite_stat1.idx和sqlite_stat1.tbl一样，那么该表是个WITHOUT ROWID表，而sqlite_stat1.stat字段包含的是这个表对应的index b-tree的信息。

sqlite_stat2表

Sqlite_stat2表只有当满足以下两个条件时才会创建：

1. 编译SQLite的时候带着SQLITE_ENABLE_STAT2选项
2. SQLite的版本介于3.6.18和3.7.8之间

版本在3.6.18之前和3.7.8之后的SQLite不会读写sqlite_stat2表。sqlite_stat2表中包含了index中key的分布。该表的结构相当于由如下SQL语句创建：

CREATE TABLE sqlite_stat2(tbl,idx,sampleno,sample);

The sqlite_stat2.idx column and the sqlite_stat2.tbl column in each row of the sqlite_stat2 table identify an index described by that row. There are usually 10 rows in the sqlite_stat2 table for each index.

The sqlite_stat2 entries for an index that have sqlite_stat2✦.sampleno between 0 and 9 inclusive are samples of the left-most key value in the index taken at evenly spaced points along the index. Let C be the number of rows in the index. Then the sampled rows are given by

rownumber = (i*C*2 + C)/20

The variable i in the previous expression varies between 0 and 9. Conceptually, the index space is divided into 10 uniform buckets and the samples are the middle row from each bucket.

The format for sqlite_stat2 is recorded here for legacy reference. Recent versions of SQLite no longer support sqlite_stat2 and the sqlite_stat2 table, if is exists, is simply ignored.

The sqlite_stat3 table

The sqlite_stat3 is only used if SQLite is compiled with SQLITE_ENABLE_STAT3 or SQLITE_ENABLE_STAT4 and if the SQLite version number is 3.7.9 or greater. The sqlite_stat3 table is neither read nor written by any version of SQLite before 3.7.9. If the SQLITE_ENABLE_STAT4 compile-time option is used and the SQLite version number is 3.8.1 or greater, then sqlite_stat3 might be read but not written. The sqlite_stat3 table contains additional information about the distribution of keys within an index, information that the query planner can use to devise better and faster query algorithms. The schema of the sqlite_stat3 table is as follows:

CREATE TABLE sqlite_stat3(tbl,idx,nEq,nLt,nDLt,sample);

There are usually multiple entries in the sqlite_stat3 table for each index. The sqlite_stat3.sample column holds the value of the left-most field of an index identified by sqlite_stat3.idx and sqlite_stat3.tbl. The sqlite_stat3.nEq column holds the approximate number of entries in the index whose left-most column exactly matches the sample. The

sqlite_stat3.nLt holds the approximate number of entries in the index whose left-most column is less than the sample. The sqlite_stat3.nDLt column holds the approximate number of distinct left-most entries in the index that are less than the sample.

There can be an arbitrary number of sqlite_stat3 entries per index. The ANALYZE command will typically generate sqlite_stat3 tables that contain between 10 and 40 samples that are distributed across the key space and with large nEq values.

In a well-formed sqlite_stat3 table, the samples for any single index must appear in the same order that they occur in the index. In other words, if the entry with left-most column S1 is earlier in the index b-tree than the entry with left-most column S2, then in the sqlite_stat3 table✦, sample S1 must have a smaller rowid than sample S2.

The sqlite_stat4 table✦

The sqlite_stat4 is only created and is only used if SQLite is compiled with SQLITE_ENABLE_STAT4 and if the SQLite version number is 3.8.1 or greater. The sqlite_stat4 table is neither read nor written by any version of SQLite before 3.8.1. The sqlite_stat4 table contains additional information about the distribution of keys within an index or the distribution of keys in the primary key of a WITHOUT ROWID table. The query planner can sometimes use the additional information in the sqlite_stat4 table to devise better and faster query algorithms. The schema of the sqlite_stat4 table is as follows:

CREATE TABLE sqlite_stat4(tbl,idx,nEq,nLt,nDLt,sample);

There are typically between 10 to 40 entries in the sqlite_stat4 table for each index for which statistics are available, however these limits are not hard bounds. The meanings of the columns in the sqlite_stat4 table are as follows:

tbl:

The sqlite_stat4.tbl column holds name of the table that owns the index that the row describes

idx:

The sqlite_stat4.idx column holds name of the index that the row describes, or in the case of an sqlite_stat4 entry for a WITHOUT ROWID table, the name of the table itself.

sample:

The sqlite_stat4.sample column holds a BLOB in the record format that encodes the indexed columns followed by the rowid for a rowid table or by the columns of the primary key for a WITHOUT ROWID table. The sqlite_stat4.sample BLOB for the WITHOUT ROWID table itself contains just the columns of the primary key. Let the number of columns encoded by

the sqlite_stat4.sample blob be N. For indexes on an ordinary rowid table, N will be one more than the number of columns indexed. For indexes on WITHOUT ROWID tables, N will be the number of columns indexed plus the number of columns in the primary key. For a WITHOUT ROWID table, N will be the number of columns in the primary key.

nEq:

The sqlite_stat4.nEq column holds a list of N integers where the K-th integer is the approximate number of entries in the index whose left-most K columns exactly match the K left-most columns of the sample.

nLt:

The sqlite_stat4.nLt column holds a list of N integers where the K-th integer is the approximate number of entries in the index whose K left-most columns are collectively less than the K left-most columns of the sample.

nDLt:

The sqlite_stat4.nDLt column holds a list of N integers where the K-th integer is the approximate number of entries in the index that are distinct in the first K columns and where the left-most K columns are collectively less than the left-most K columns of the sample.

The sqlite_stat4 is a generalization of the sqlite_stat3 table. The sqlite_stat3 table provides information about the left-most column of an index whereas the sqlite_stat4 table provides information about all columns of the index.

There can be an arbitrary number of sqlite_stat4 entries per index. The ANALYZE command will typically generate sqlite_stat4 tables that contain between 10 and 40 samples that are distributed across the key space✦ and with large nEq values.

In a well-formed sqlite_stat4 table, the samples for any single index must appear in the same order that they occur in the index. In other words, if entry S1 is earlier in the index b-tree than entry S2, then in the sqlite_stat4 table, sample S1 must have a smaller rowid than sample S2.

一些例子

下面让我们看看在执行了某些SQL语句之后，数据库文件✦会出现什么变化，以帮助各位读者复习整个第二章的内容。（注：本节并未出现在原版英文文档中）

CREATE TABLE t1 (c1,c2,c3)；

1. 数据库文件中新增一个表示t1的table b-tree，其key为隐含的rowid，data为c1,c2,c3组成的record。

2. 系统表sqlite_master新增一行，X为table b-tree的根页号：

table|t1|t1|X|CREATE TABLE t1(c1,c2,c3)

CREATE INDEX i11 on t1 (c3)；

1. 数据库文件中新增一个表示i11的index b-tree，其key为c3,rowid组成的record。
2. 系统表sqlite_master新增一行，X为b-tree的根页号：

index|i11|t1|X|CREATE INDEX i11 on t1(c3)

CREATE TABLE t2 (c1 primary key+, c2, c3);

1. 数据库文件中新增一个表示t2的table b-tree，其key为隐含的rowid，data为c1,c2,c3组成的record。
2. 数据库文件中新增一个表示c1（t2的primary key）的index b-tree，其key为c1,rowid组成的record。
3. 系统表sqlite_master新增两行，X和Y分别为两个b-tree的根页号：

table|t2|t2|X|CREATE TABLE tx (c1 primary key ,c2,c3)

index|sqlite_autoindex_t2_1|t2|Y|

CREATE TABLE t3 (c1 integer primary key, c2, c3);

1. 数据库中新增一个表示t3的table b-tree，其key为c1（integer primary key代替隐含的rowid），data为c2,c3组成的record。
2. 系统表sqlite_master新增一行，X为table b-tree的根页号：

table|t3|t3|X|CREATE TABLE t3 (c1 integer primary key, c2,c3)

CREATE TABLE t4 (c1, c2 primary key,c3) without rowid;

1. 数据库中新增一个表示t4的index b-tree，其key为c2,c1,c3组成的record。
2. 系统表sqlite_master新增一行，X为index b-tree的根页号：

table|t4|t4|X|CREATE TABLE t4(c1,c2 primary key,c3) without rowid

CREATE TABE t5 (c1, c2 integer primary, c3) without rowid;

1. 数据库中新增一个表示t5的index b-tree，其key为c2,c1,c3组成的record。
2. 系统表sqlite_master新增一行，X为index b-tree的根页号：

table|t5|t5|X|CREATE TABLE t5(c1,c2 integer primary key,c3) without rowid

CREATE TABLE t6(c1,c2,c3,PRIMARY KEY(c3,c2));

1. 数据库文件中新增一个表示t6的table b-tree，其key为隐含的rowid，data为c1,c2,c3组成的record。
2. 数据库文件中新增一个表示primary key(c3,c2)的index b-tree，其key为c3,c2,rowid组成的record。
3. 系统表sqlite_master新增两行，X和Y分别为两个b-tree的根页号：

table|t6|t6|X|CREATE TABLE t6(c1,c2,c3, PRIMARY KEY(c3,c2))

index|sqlite_autoindex_t6_1|t6|Y|

CREATE TABLE t7(c1 UNIQUE, c2 UNIQUE,c3);

1. 数据库文件中新增一个表示t7的table b-tree，其key为隐含的rowid，data为c1,c2,c3组成的record。
2. 数据库文件中新增一个分别表示c1的index b-tree，其key为c1,rowid组成的record
3. 数据库文件中新增一个分别表示c2的index b-tree，其key为c2,rowid组成的record
4. 系统表sqlite_master新增3行，X、Y和Z分别表示三个b-tree的根页号：

table|T7|T7|X|CREATE TABLE T7(c1 UNIQUE, c2 UNIQUE, c3)

index|sqlite_autoindex_T7_1|T7|Y|

index|sqlite_autoindex_T7_2|T7|Z|

CREATE TABLE t9(c1,c2,c3,primary key(c3,c2), unique(c1,c2), unique(c2,c1)) without rowid;

1. 数据库文件中新增一个表示t9的index b-tree，其key为隐含的c3,c2,c1组成record。
2. 数据库文件中新增一个分别表示（c1,c2）的index b-tree，其key为c1,c2,c3组成的record

3. 数据库文件中新增一个分别表示（c2,c1）的index b-tree，其key为c2,c1,c3组成的record
4. 系统表sqlite_master新增3行，X、Y和Z分别表示三个b-tree的根页号：

table|t9|t9|X|CREATE TABLE t9(c1,c2,c3, primary key(c2,c3),unique(c1,c2),unique(c2,c1)) without rowid

index|sqlite_autoindex_t9_2|t9|Y|

index|sqlite_autoindex_t9_3|t9|Z|

CREATE INDEX i91 ON t9(c1,c3);

1. 数据库文件中新增一个分别表示i91的index b-tree，其key为c1,c3,c2组成的record
2. 系统表sqlite_master中新增一行，X为b-tree的根页号：index|i91|t9|X|CREATE INDEX i91 on t9(c1,c3)

index|i92|t9|X|CREATE INDEX i92 on t9(c2,c3)

CREATE INDEX i2 ON t9(c2,c3);

1. 数据库文件中新增一个分别表示i92的index b-tree，其key为c2,c3组成的record
2. 系统表sqlite_master中新增一行，X为b-tree的根页号： index|i92|t9|X|CREATE INDEX i92 on t9(c2,c3)

回滚日志

Rollback journal是一个跟数据库文件绑定的文件，保存transaction过程中用于恢复该数据库的信息。回滚日志文件总是和数据库文件位于同一目录中，其文件名是对应的数据库文件名加上“-journal”后缀。一个数据库文件只能有一个对应的回滚日志文件，因此对于一个数据库同时只能有一个写操作。

如果由于应用程序崩溃、操作系统崩溃✦或者电源崩坏导致一次事务被中断，那么数据库就会处于一个不一致的状态。下次SQLite打开数据库文件的时候，回滚日志会被检测到，并用于将数据库恢复✦到未完成事务之前的状态。

只有当回滚日志文件存在，并且包含一个有效的头的时候，回滚日志才被视为是有效的。一个事务可以以以下方法提交：

1. 回滚日志文件可以被删除

2. 回滚日志文件✦可以被截成0长度
3. 回滚日志文件的头可以用无效数据覆盖

这三种方式分别对应journal_mode pragma中的DELETE，TRUNCATE，和PERSIST三种设置。

一个合法的回滚日志有以下格式的头：

回滚日志✦头格式

偏移	大小	描述
0	8	头字符串：0xd9, 0xd5, 0x05, 0xf9, 0x20, 0xa1, 0x63, 0xd7
8	4	当前segment（日志段：日志头+若干page record）中的page record的数量，-1表示该segment一直延续到文件末尾。
12	4	nonce （校验值初始随机数）
16	4	事务开始前的数据库页数
20	4	磁盘扇区的大小，由写日志的程序决定
24	4	日志文件的页数

知乎 @Jun Zheng

日志头之后的部分会填充0，直到扇区大小✦（偏移20的值）的位置。日志头本身也在扇区中，因此当写日志头的时候出现掉电时，日志头之后的信息（希望）不会受到影响。

日志头和填充的0之后，是一系列（大于等于0个）page record（页记录）。每个page record存储了数据库文件中一个页的原始内容。同一个页在日志中出现不会超过一次。如果要回滚一个未完成的事务，只需要从头到尾读取rollback journal，然后将其中的page record写入到数据库文件对应的位置。

如果数据库页的大小是N，那么page record的格式如下：

回滚日志Page Record格式

偏移	大小	描述
0	4	数据库文件中的页号
4	N	页中的原始内容
N+4	4	校验值

知乎 @Jun Zheng

校验值是无符号32位整数，计算如下：

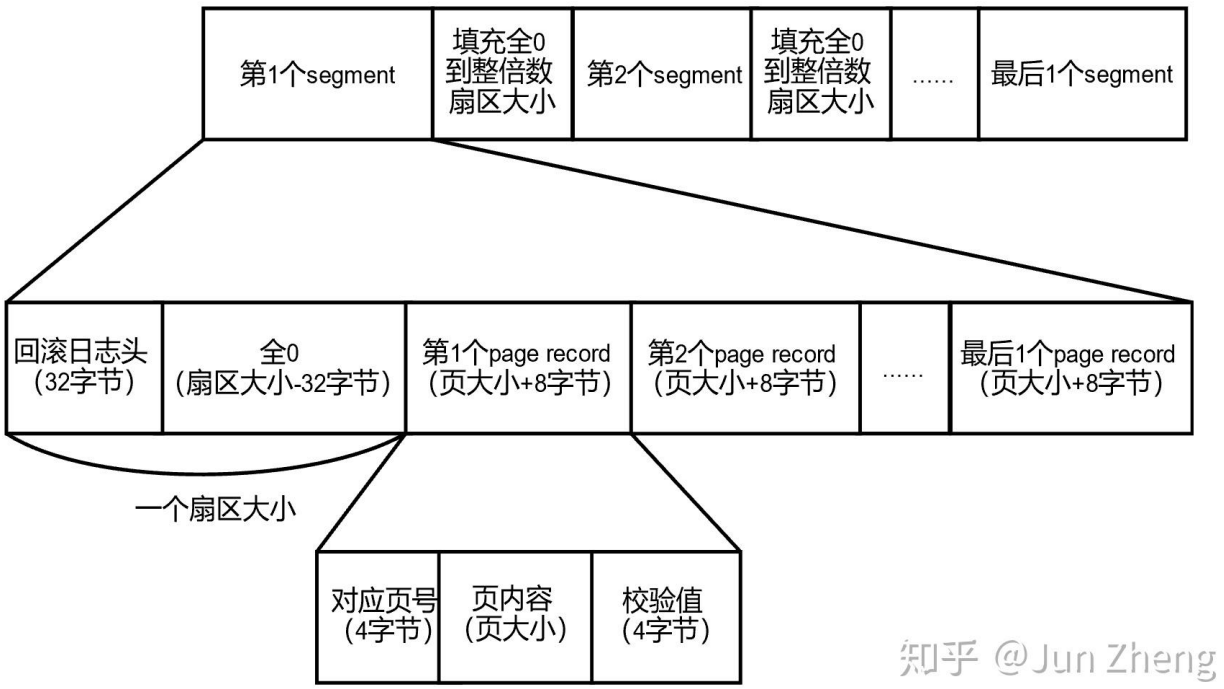
1. 用nonce（日志头偏移12）初始化checksum
2. 设索引X=N-200（N为数据库页的字节数）
3. 将页中位于偏移X的字节视为8位无符号整数，并将该整数累加到checksum中。
4. X=X-200
5. 如果X>=0，回到步骤3

校验值用于保护对于日志page record的写操作，防止在掉电之后出现的不一致错误。每次开始一个新的事务，就会换一个新的nonce，以防止出现跟上次事务有相同的页。通过更换nonce，老的数据页会产生错误的checksum从而被检测出来。因为效率原因，校验值只使用从page record中稀疏挑出的样本来计算——研究显示计算校验值相当耗费性能。

假设日志头中的页数（偏移8处）为M，如果M大于0，那么在M个page record之后会填充0直到扇区整倍数，接着（可能）是另一个日志头。同一个日志文件中所有的日志头必须有同样的数据库页数和扇区大小。

如果M为-1，则page record的数量是剩余日志文件大小/page record大小。

回滚日志格式



Write-Ahead Log

从版本3.7.0（2010-07-21）开始，SQLite支持另外一种事务控制机制：“write-ahead log”或者“WAL”。当一个数据库处于WAL模式，所有使用该数据库的连接必须使用WAL。某个数据库也许既使用rollback journal也使用WAL，但是不能同时使用这两者。WAL总是和数据库文件处于同一个目录下，跟数据库文件同名并有“-wal”后缀。

WAL文件格式

WAL文件由文件头和之后的N（N>=0）个frame组成。每个frame记录了对应的数据库中的页上所做的修改。事务中所有对数据库所做的修改都被记录在了WAL的frame中。当一个包含commit marker的frame（该frame即commit frame✦）被写入WAL文件时，被视为一个事务被提交。一个WAL可以，而且通常会记录若干事务。每隔一段时间，WAL的内容会被转写到数据库文件中，这个动作被称为“checkpoint”。

一个WAL文件可以被重用若干次。换句话说，WAL里可以有许多frame，然后被checkpoint，然后新的frame会覆盖老的。Frame上附带的校验值和计数器会决定WAL中的哪些frame是有效的，而哪些是之前的checkpoint的。

WAL头大小为32个字节，包含以下8个32位big-endian无符号整数：

WAL头格式

偏移	大小	描述
0	4	魔数，0x377f0682 或 0x377f0683
4	4	文件格式版本，当前是 3007000
8	4	数据库页大小，例如：1024
12	4	Checkpoint序号
16	4	Salt-1：随着每一个checkpoint增加的随机值
20	4	Salt-2：每个checkpoint都不同的随机值
24	4	Checksum-1: First part of a checksum on the first 24 bytes of header
28	4	Checksum-2: Second part of the checksum on the first 24 bytes of header

知乎 @Jun Zheng

在WAL头之后是N个（N>=0）frame，每个frame由24字节的frame头和一个页大小的页数据组成。Frame头则由6个32位big-endian无符号整数值组成：

WAL Frame头格式

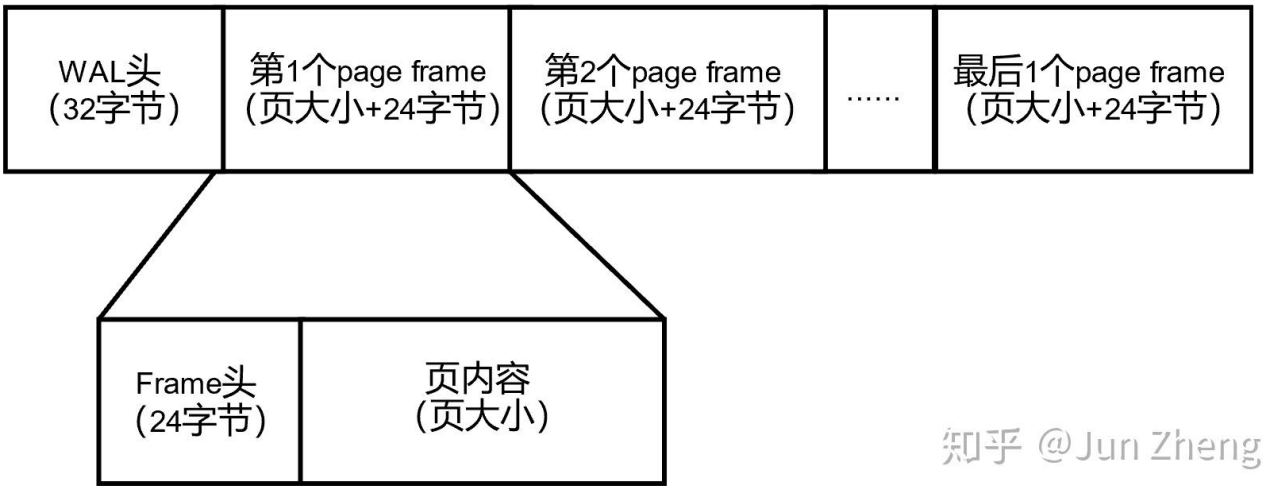
偏移	大小	描述
0	4	页号
4	4	commit marker：对于commit frame，是提交之后数据库的总页数；对于其他记录，该值为0
8	4	从WAL头复制来的salt-1
12	4	从WAL头复制来的salt-2
16	4	Checksum-1: Cumulative checksum up through and including this page
20	4	Checksum-2: Second half of the cumulative checksum.

知乎 @Jun Zheng

一个frame只有在满足以下条件的时候才被认为是有效的：

1. Frame头中的salt-1及salt-2与WAL头中的salt-1及salt-2一致
2. Frame头的最后8个字节，等于WAL头中的前24字节、迄今为止的所有frame的前8个字节及frame内容计算得出的校验值。

WAL文件格式



知乎 @Jun Zheng

校验值算法

The checksum is computed by interpreting the input as an even number of unsigned 32-bit integers: x(0) through x(N). The 32-bit integers are big-endian if the magic number in the first 4 bytes of the WAL header is 0x377f0683 and the integers are little-endian if the magic number is 0x377f0682. The checksum values are always stored in the frame header in a big-endian format regardless of which byte order is used to compute the checksum.

The checksum algorithm only works for content which is a multiple of 8 bytes in length. In other words, if the inputs are x(0) through x(N) then N must be odd. The checksum algorithm is as follows:

s0 = s1 = 0

for i from 0 to n-1 step 2:

s0 += x(i) + s1;

s1 += x(i+1) + s0;

endfor

result in s0 and s1

The outputs s0 and s1 are both weighted checksums✦ using Fibonacci weights in reverse order. (The largest Fibonacci weight occurs on the first element of the sequence being summed.) The s1 value spans all 32-bit integer terms of the sequence whereas s0 omits the final term.

Checkpoint算法

在checkpoint来的时候，首先用VFS的xSync函数将WAL文件刷进非易失性存储介质（比如磁盘，之前WAL缓存在内存中）。然后WAL中的有效内容被转写进数据库文件中。最后用xSync函数将数据库文件刷写回磁盘中。xSync函数是写栅栏——所有在xSync之后的写操作都会等待xSync之前的写操作完全完成之后执行（否则操作系统的缓存机制可能使得其乱序）。

Checkpoint之后，新的写事务会从头覆盖WAL文件。在第一次写事务开始的时候，WAL头中的salt-1会增加，而salt-2✦会取一个随机值。这些更新会使得WAL中已经被checkpoint的老frame失效，以免它们被再次checkpoint。

读算法

如果想从数据库中读取页号为P的页，读者必须先确定WAL中是否包含页P。如果是，那么最后一个有效的页P的frame（在某个commit frame之前或者本身就是commit frame）就是有效的页P的内容。如果WAL中没有页P对应的frame，或者有对应frame但其后没有/本身不是commit frame（说明页P的这个对应frame正处于一个写事务中，而该事务尚未提交）。

在读事务的开始，读者记录下WAL中最后的有效frame的index。后续的读操作中将始终使用这个记录下的值“mxFrame”。新的事务可以被提交到WAL中，但是只要读者使用这个mxFrame，所有后续添加的内容都会被忽略，读者会始终面对一个在开始读的时间点上一致的数据库快照。这个技术确保了多个并发的读者可以同时看到不同版本的数据库内容。

上文中提到的读算法可以正常工作，但是因为页P对应的frame可能存在于WAL文件中任何位置，读程序需要扫描整个文件才能找到正确的frame。如果WAL过大（通常是几M）这个扫描会降低读操作的效率。为了解决这个问题，一个叫做wal-index的数据结构被用于加速对frame的搜索。

WAL-Index格式

从概念定义上来说，wal-index使用共享内存，然而当前VFS的实现使用映射文件来实现wal-index。被映射的文件存于数据库文件的同一目录中，有“-shm”后缀。因为wal-index是共享内存，在网络文件系统下，SQLite不支持WAL模式的日志（journal_mode=WAL），因为各个客户端在在不同的机器上，无法共享同一块内存。

Wal-index存在的目的是为了快速回答以下问题：

给定一个页号P和一个最大WAL frame index M，返回页P所对应的小于M的最大WAL frame index，如果在M之内没有对应于P的frame则返回NULL。

这里的值M就是4.4节中提到的“mxFrame”，确定了整个读操作可以使用的WAL中的最大frame。

WAL-index是易失性的，在崩溃之后，wal-index需要从WAL文件中重建。当最后一个wal-index的连接被关闭之后，VFS必须要清空wal-index头。因为wal-index的易失性，它可以使用体系结构相关的格式，而无需跨平台，因此，不像数据库文件和WAL文件必须是大端的，wal-index可以使用的本地的字节序。

本文档关注的是数据库文件的非易失性格式，因为wal-index是易失性的，因此对此不多做介绍。关于wal-index的更多信息可以从SQLite源代码的注释中找到。

有需要了解WAL的请看官网文档，或者相应的中文翻译。