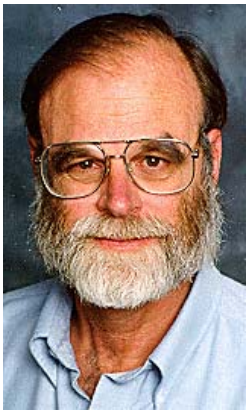


个人简介
专业打杂程序员
联系方式
新浪微博 腾讯微博

昵称：YY哥
园龄：7年2个月
粉丝：342
关注：2
+加关注

< 2009年2月 >

日	一	二	三	四	五	六
25	26	27	28	29	30	31
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
1	2	3	4	5	6	7

搜索
找找看
谷歌搜索

常用链接
我的随笔
我的评论
我的参与
最新评论
我的标签
更多链接

随笔分类
c/c++(9)
Linux相关(24)
MySQL(11)
Others(2)
Web技术(12)
数据结构与算法(15)
数据库技术(30)
系统相关(3)
云计算与虚拟化(3)

随笔档案
2014年7月 (4)

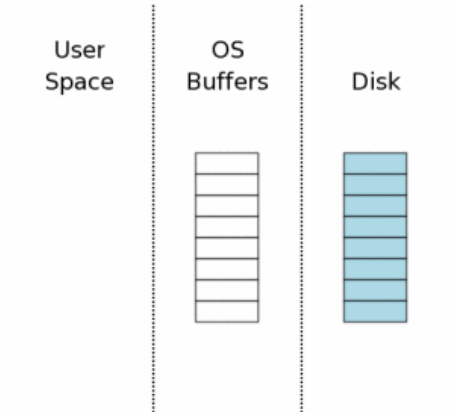
SQLite入门与分析(四)---Page Cache之事务处理(1)

写在前面：从本章开始，将对SQLite的每个模块进行讨论。讨论的顺序按照我阅读SQLite的顺序来进行，由于项目的需要，以及时间关系，不能给出一个完整的计划，但是我会先讨论我认为比较重要的内容。本节讨论SQLite的事务处理技术，事务处理是DBMS中最关键的技术，对SQLite也一样，它涉及到并发控制，以及故障恢复，由于内容较多，分为两节。好了，下面进入正题。

本节通过一个具体的例子来分析SQLite原子提交的实现（基于Version 3.3.6的代码）。
CREATE TABLE episodes(id integer primary key,name text, cid int)；
插入一条记录：insert into episodes(name,cid) values("cat",1) ;
它经过编译器处理后生成的虚拟机代码如下：

```
sqlite> explain insert into episodes(name,cid) values("cat",1);
0|Trace|0|0|0|explain insert into episodes(name,cid) values("cat",1);|00|
1|Goto|0|12|0||00|
2|SetNumColumns|0|3|0||00|
3|OpenWrite|0|2|0||00|
4|NewRowid|0|2|0||00|
5|Null|0|3|0||00|
6|String8|0|4|0|cat|00|
7|Integer|1|5|0||00|
8|MakeRecord|3|3|6|dad|00|
9|Insert|0|6|2|episodes|0b|
10|Close|0|0|0||00|
11|Halt|0|0|0||00|
12|Transaction|0|1|0||00|
13|VerifyCookie|0|1|0||00|
14|Transaction|1|1|0||00|
15|VerifyCookie|1|0|0||00|

16|TableLock|0|2|1|episodes|00|

17|Goto|0|2|0||00|
```

1、初始状态（Initial State）
当一个数据库连接第一次打开时，状态如图所示。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数据。图中中间区域是操作系统的磁盘缓冲区。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。因为这个数据库连接刚刚打开，所以还没有任何数据记录被读入，所以这些内存也是空的。



2、获取读锁(Acquiring A Read Lock)
在SQLite写数据库之前，它必须先从数据库中读取相关信息。比如，在插入新的数据时，SQLite会先从sqlite_master表中读取数据库模式(相当于数据字典)，以便编译器对INSERT语句进行分析，确定数据插入的位置。
在进行读操作之前，必须先获取数据库的共享锁(shared lock)，共享锁允许两个或更多的连接在同一时刻读取数据库。但是共享锁不允许其它连接对数据库进行写操作。
shared lock存在于操作系统磁盘缓存，而不是磁盘本身。文件锁的本质只是操作系统的内核数据结构，当操作系统崩溃或掉电时，这些内核数据也会随之消失。

2014年3月 (1)
2013年9月 (1)
2013年8月 (1)
2013年2月 (1)
2012年11月 (4)
2012年1月 (1)
2011年12月 (1)
2011年10月 (1)
2011年3月 (1)
2010年9月 (1)
2010年8月 (1)
2010年7月 (3)
2010年6月 (2)
2010年5月 (7)
2010年4月 (1)
2010年3月 (1)
2010年1月 (1)
2009年12月 (2)
2009年10月 (2)
2009年9月 (14)
2009年8月 (4)
2009年6月 (14)
2009年5月 (3)
2009年4月 (1)
2009年3月 (3)
2009年2月 (11)
2008年10月 (7)
2008年8月 (5)
2008年7月 (1)
2008年6月 (2)
2008年5月 (2)
2008年4月 (5)

kernel

kernel中文社区
LDN
The Linux Document Project
The Linux Kernel Archives

manual

cppreference
gcc manual
mysql manual

sites

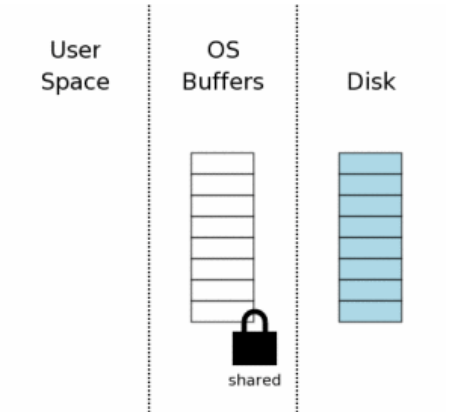
Database Journal
Fedora镜象
highscalability
KFUPM ePrints
Linux docs
Linux Journal
NoSQL
SQLite

技术社区

apache
CSDN
IBM-developerworks
lucene中国
nutch中国
oldlinux
oracle's forum

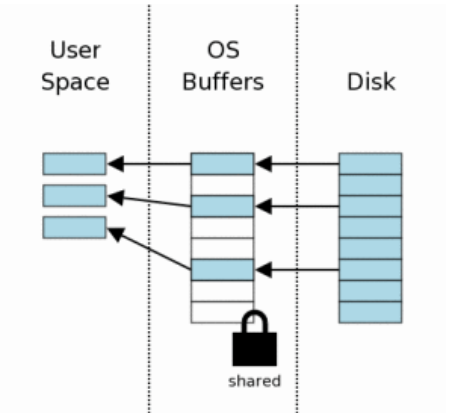
最新评论

1. Re:理解MySQL——架构与概念
我试验了下.数据 5 9 10 13 18begin;select * from asf_execution where num> 5 and num 5 and INSTANCE_ID_<18 lock in share mode;会有 1.行锁 2.间隙所 [5 18)插



3、读取数据

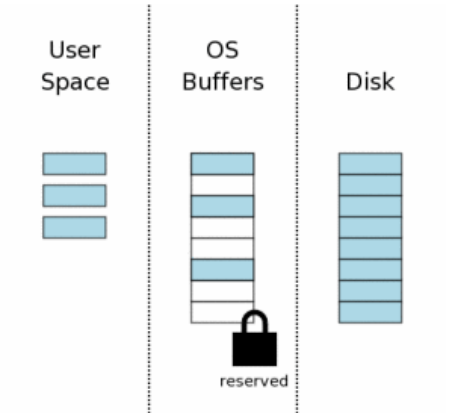
一旦得到shared lock，就可以进行读操作。如图所示，数据先由OS从磁盘读取到OS缓存，然后再由OS移到用户进程空间。一般来说，数据库文件分为很多页，而一次读操作只读取一小部分页面。如图，从8个页面读取3个页面。



4、获取Reserved Lock

在对数据进行修改操作之前，先要获取数据库文件的Reserved Lock，Reserved Lock和shared lock的相似之处在于，它们都允许其它进程对数据库文件进行读操作。Reserved Lock和Shared Lock可以共存，但是只能是一个Reserved Lock和多个Shared Lock——多个Reserved Lock不能共存。所以，在同一时刻，只能进行一个写操作。

Reserved Lock意味着当前进程(连接)想修改数据库文件，但是还没开始修改操作，所以其它的进程可以读数据库，但不能写数据库。



5、创建恢复日志(Creating A Rollback Journal File)

在对数据库进行写操作之前，SQLite先要创建一个单独的日志文件，然后把要修改的页面的原始数据写入日志。回滚日志包含一个日志头(图中的绿色)——记录数据库文件的原始大小。所以即使数据库文件大小改变了，我们仍知道数据库的原始大小。

从OS的角度来看，当一个文件创建时，大多数OS(Windows,Linux,Mac OS X)不会向磁盘写入数据，新创建的文件此时位于磁盘缓存中，之后才会真正写入磁盘。如图，日志文件位于OS磁盘缓存中，而不是位于磁盘。

入INSERT I.....

--麒麟飞

2. Re:理解MySQL——架构与概念

例1-5
insert into t(i) values(1);
这句话应该是可以插入的.
不会被阻塞

--麒麟飞

3. Re:理解MySQL——架构与概念

注：SELECT ... FOR UPDATE仅在自动提交关闭(即手动提交)时才会对元组加锁，而在自动提交时，符合条件的元组不会被加锁。

这个是错误的.自动提交的,也会尝试获取排它锁.
你可以试验下.

--麒麟飞

4. Re:浅谈mysql的两阶段提交协议

YY哥 偶像啊!细腻文笔 配有说服力的代码和图 我崇拜你 !!!
之前sqlite的深入分析帮了我大忙..现在做mysql相关 有来你的博客找东西 哈哈哈!!

--hark.perfe

5. Re:(i++)+(i++)与(++i)+(++i)

@arrowcat
这类语句本身没什么意义，但是楼主思考的角度让我豁然开朗。

--HJWAJ

阅读排行榜

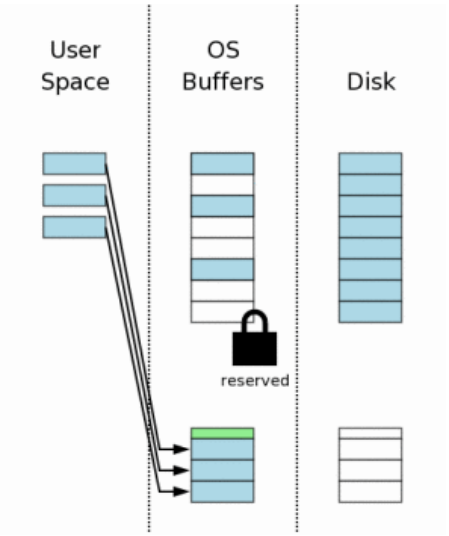
1. 理解MySQL——索引与优化(77627)
2. SQLite入门与分析(一)---简介(48610)
3. 理解MySQL——复制(Replication)(26209)
4. libevent源码分析(19048)
5. SQLite入门与分析(二)---设计与概念(16977)

评论排行榜

1. (i++)+(i++)与(++i)+(++i)(40)
2. SQLite入门与分析(一)---简介(30)
3. 浅谈SQLite——实现与应用(20)
4. 一道算法题,求更好的解法(18)
5. 理解MySQL——索引与优化(16)

推荐排行榜

1. SQLite入门与分析(一)---简介(12)
2. 理解MySQL——索引与优化(12)
3. 浅谈SQLite——查询处理及优化(10)
4. 乱谈服务器编程(9)
5. libevent源码分析(6)



上面 5步的代码的实现：

```
//事务指令的实现
//p1为数据库文件的索引号---0为main database;1为temporary tables使用的文件
//p2 不为0, 一个写事务开始
case OP_Transaction: {
    //数据库的索引号
    int i = pOp->p1;
    //指向数据库对应的btree
  Btree *pBt;

  assert( i>=0 && i<db->nDb );
  assert( (p->btreeMask & (1<<i))!=0 );
  //设置btree指针
  pBt = db->aDb[i].pBt;

  if( pBt ){
      //从这里btree开始事务,主要给文件加锁,并设置btree事务状态
    rc = sqlite3BtreeBeginTrans(pBt, pOp->p2);

    if( rc==SQLITE_BUSY ){
      p->pc = pc;
      p->rc = rc = SQLITE_BUSY;
      goto vdbe_return;
    }
    if( rc!=SQLITE_OK && rc!=SQLITE_READONLY /* && rc!=SQLITE_BUSY */ ){
      goto abort_due_to_error;
    }
  }
  break;
}

//开始一个事务,如果第二个参数不为0,则一个写事务开始,否则是一个读事务
//如果wrflag>=2,一个exclusive事务开始,此时别的连接不能访问数据库
int sqlite3BtreeBeginTrans(Btree *p, int wrflag){
  BtShared *pBt = p->pBt;
  int rc = SQLITE_OK;

  btreeIntegrity(p);

  /* If the btree is already in a write-transaction, or it
  ** is already in a read-transaction and a read-transaction
  ** is requested, this is a no-op.
  */
  //如果b-tree处于一个写事务;或者处于一个读事务,一个读事务又请求,则返回SQLITE_OK
  if( p->inTrans==TRANS_WRITE || (p->inTrans==TRANS_READ && !wrflag) ){
    return SQLITE_OK;
  }

  /* Write transactions are not possible on a read-only database */
  //写事务不能访问只读数据库
  if( pBt->readOnly && wrflag ){
    return SQLITE_READONLY;
  }
```

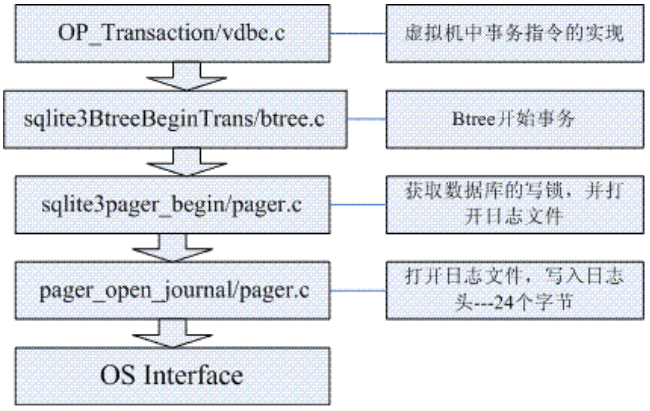
```
  /* If another database handle has already opened a write transaction
  ** on this shared-btree structure and a second write transaction is
  ** requested, return SQLITE_BUSY.
  */
  //如果数据库已存在一个写事务,则该写事务请求时返回SQLITE_BUSY
  if( pBt->inTransaction==TRANS_WRITE && wrflag ){
    return SQLITE_BUSY;
  }

  do {
    //如果数据库对应btree的第一个页面还没读进内存
      //则把该页面读进内存，数据库也相应的加read lock
    if( pBt->pPage1==0 ){
      //加read lock，并读页面到内存
      rc = lockBtree(pBt);
    }

    if( rc==SQLITE_OK && wrflag ){
        //对数据库文件加RESERVED_LOCK锁
      rc = sqlite3pager_begin(pBt->pPage1->aData, wrflag>1);
      if( rc==SQLITE_OK ){
        rc = newDatabase(pBt);
      }
    }

    if( rc==SQLITE_OK ){
      if( wrflag ) pBt->inStmt = 0;
    }else{
      unlockBtreeIfUnused(pBt);
    }
  }while( rc==SQLITE_BUSY && pBt->inTransaction==TRANS_NONE &&
          sqlite3InvokeBusyHandler(pBt->pBusyHandler) );

  if( rc==SQLITE_OK ){
    if( p->inTrans==TRANS_NONE ){
        //btree的事务数加1
      pBt->nTransaction++;
    }
    //设置btree事务状态
    p->inTrans = (wrflag?TRANS_WRITE:TRANS_READ);
    if( p->inTrans>pBt->inTransaction ){
      pBt->inTransaction = p->inTrans;
    }
  }

  btreeIntegrity(p);
  return rc;
}

/*
**获取数据库的写锁,发生以下情况时去除写锁:
**    *  sqlite3pager_commit() is called.
**    *  sqlite3pager_rollback() is called.
**    *  sqlite3pager_close() is called.
**    *  sqlite3pager_unref() is called to on every outstanding page.
**   pData指向数据库的打开的页面,此时并不修改,仅仅只是获取
**   相应的pager,检查它是否处于read-lock状态。
**如果打开的不是临时文件,则打开日志文件.
**如果数据库已经处于写状态，则do nothing
*/
int sqlite3pager_begin(void *pData, int exFlag){
  PgHdr *pPg = DATA_TO_PGHDR(pData);
  Pager *pPager = pPg->pPager;
  int rc = SQLITE_OK;
  assert( pPg->nRef>0 );
  assert( pPager->state!=PAGER_UNLOCK );
  //pager已经处于share状态
  if( pPager->state==PAGER_SHARED ){
    assert( pPager->aInJournal==0 );
    if( MEMDB ){
      pPager->state = PAGER_EXCLUSIVE;
      pPager->origDbSize = pPager->dbSize;
    }else{
        //对文件加 RESERVED_LOCK
      rc = sqlite3OsLock(pPager->fd, RESERVED_LOCK);
      if( rc==SQLITE_OK ){
```

```
        //设置pager的状态
        pPager->state = PAGER_RESERVED;
        if( exFlag ){
          rc = pager_wait_on_lock(pPager, EXCLUSIVE_LOCK);
        }
      }
      if( rc!=SQLITE_OK ){
        return rc;
      }
      pPager->dirtyCache = 0;
      TRACE2("TRANSACTION %d\n", PAGERID(pPager));
      //使用日志,不是临时文件,则打开日志文件
      if( pPager->useJournal && !pPager->tempFile ){
          //为pager打开日志文件,pager应该处于RESERVED或EXCLUSIVE状态
          //会向日志文件写入header
        rc = pager_open_journal(pPager);
      }
    }
  }
  return rc;
}


//创建日志文件,pager应该处于RESERVED或EXCLUSIVE状态
static int pager_open_journal(Pager *pPager){
  int rc;
  assert( !MEMDB );
  assert( pPager->state>=PAGER_RESERVED );
  assert( pPager->journalOpen==0 );
  assert( pPager->useJournal );
  assert( pPager->aInJournal==0 );
  sqlite3pager_pagecount(pPager);
  //日志文件页面位图
  pPager->aInJournal = sqliteMalloc( pPager->dbSize/8 + 1 );
  if( pPager->aInJournal==0 ){
    rc = SQLITE_NOMEM;
    goto failed_to_open_journal;
  }
  //打开日志文件
  rc = sqlite3OsOpenExclusive(pPager->zJournal, &pPager->jfd,
                              pPager->tempFile);
  //日志文件的位置指针
  pPager->journalOff = 0;
  pPager->setMaster = 0;
  pPager->journalHdr = 0;
  if( rc!=SQLITE_OK ){
    goto failed_to_open_journal;
  }
  /*一般来说，os此时创建的文件位于磁盘缓存，并没有实际
  **存在于磁盘,下面三个操作就是为了把结果写入磁盘,而对于
  **windows系统来说,并没有提供相应API，所以实际上没有意义.
  */
  //fullSync操作对windows没有意义
  sqlite3OsSetFullSync(pPager->jfd, pPager->full_fsync);
  sqlite3OsSetFullSync(pPager->fd, pPager->full_fsync);
  /* Attempt to open a file descriptor for the directory that contains a file.
  **This file descriptor can be used to fsync() the directory
  **in order to make sure the creation of a new file is actually written  to disk.
  */
  sqlite3OsOpenDirectory(pPager->jfd, pPager->zDirectory);
  pPager->journalOpen = 1;
  pPager->journalStarted = 0;
  pPager->needSync = 0;
  pPager->alwaysRollback = 0;
  pPager->nRec = 0;
  if( pPager->errCode ){
    rc = pPager->errCode;
    goto failed_to_open_journal;
  }
  pPager->origDbSize = pPager->dbSize;
  //写入日志文件的header---24个字节
  rc = writeJournalHdr(pPager);

  if( pPager->stmtAutoopen && rc==SQLITE_OK ){
    rc = sqlite3pager_stmt_begin(pPager);
  }
  if( rc!=SQLITE_OK && rc!=SQLITE_NOMEM ){
```

```
    rc = pager_unwritelock(pPager);
    if( rc==SQLITE_OK ){
      rc = SQLITE_FULL;
    }
  }
  return rc;

failed_to_open_journal:
  sqliteFree(pPager->aInJournal);
  pPager->aInJournal = 0;
  if( rc==SQLITE_NOMEM ){
    /* If this was a malloc() failure, then we will not be closing the pager
    ** file. So delete any journal file we may have just created. Otherwise,
    ** the system will get confused, we have a read-lock on the file and a
    ** mysterious journal has appeared in the filesystem.
    */
    sqlite3OsDelete(pPager->zJournal);
  }else{
    sqlite3OsUnlock(pPager->fd, NO_LOCK);
    pPager->state = PAGER_UNLOCK;
  }
  return rc;
}

/*写入日志文件头
**journal header的格式如下:
** - 8 bytes: 标志日志文件的魔数
** - 4 bytes: 日志文件中记录数
** - 4 bytes: Random number used for page hash.
** - 4 bytes: 原来数据库的大小(kb)
** - 4 bytes: 扇区大小512byte
*/
static int writeJournalHdr(Pager *pPager){
  //日志文件头
  char zHeader[sizeof(aJournalMagic)+16];

  int rc = seekJournalHdr(pPager);
  if( rc ) return rc;

  pPager->journalHdr = pPager->journalOff;
  if( pPager->stmtHdrOff==0 ){
    pPager->stmtHdrOff = pPager->journalHdr;
  }
  //设置文件指针指向header之后
  pPager->journalOff += JOURNAL_HDR_SZ(pPager);

  /* FIX ME:
  **
  ** Possibly for a pager not in no-sync mode, the journal magic should not
  ** be written until nRec is filled in as part of next syncJournal().
  **
  ** Actually maybe the whole journal header should be delayed until that
  ** point. Think about this.
  */
  memcpy(zHeader, aJournalMagic, sizeof(aJournalMagic));
  /* The nRec Field. 0xFFFFFFFF for no-sync journals. */
  put32bits(&zHeader[sizeof(aJournalMagic)], pPager->noSync ? 0xffffffff : 0);
  /* The random check-hash initialiser */
  sqlite3Randomness(sizeof(pPager->cksumInit), &pPager->cksumInit);
  put32bits(&zHeader[sizeof(aJournalMagic)+4], pPager->cksumInit);
  /* The initial database size */
  put32bits(&zHeader[sizeof(aJournalMagic)+8], pPager->dbSize);
  /* The assumed sector size for this process */
  put32bits(&zHeader[sizeof(aJournalMagic)+12], pPager->sectorSize);
  //写入文件头
  rc = sqlite3OsWrite(pPager->jfd, zHeader, sizeof(zHeader));

  /* The journal header has been written successfully. Seek the journal
  ** file descriptor to the end of the journal header sector.
  */
  if( rc==SQLITE_OK ){
    rc = sqlite3OsSeek(pPager->jfd, pPager->journalOff-1);
    if( rc==SQLITE_OK ){
      rc = sqlite3OsWrite(pPager->jfd, "\000", 1);
    }
  }
  return rc;
```

```
}
```

其实现过程如下图所示：



主要参考：http://www.sqlite.org/atomiccommit.html

分类: 数据库技术

« 上一篇：SQLite入门与分析(三)---内核概述(2)
» 下一篇：SQLite入门与分析(四)---Page Cache之事务处理(2)

posted @ 2009-02-26 10:45 YY哥 阅读(7565) 评论(8) 编辑 收藏

评论列表

#1楼 2009-02-26 11:44 true[未注册用户]

感谢的同时，期待下文

#2楼[楼主] 2009-02-26 16:59 YY哥

@true
不客气

支持(0) 反对(0)

#3楼 2009-02-26 23:56 ajax+u[未注册用户]

继续，最好将一下虚拟机

#4楼 2009-03-05 08:57 Soli

Not bad!

支持(0) 反对(0)

#5楼 2010-06-01 23:43 MichaelMj

您讲的很好，使我受益匪浅。我想请教一个问题，为什么SQLiite要用到OS的BUFFER，而不自己设计缓存管理器？

支持(0) 反对(0)

#6楼[楼主] 2010-06-02 21:32 YY哥

@MichaelMj
一般来说，只有大型通用DBMS才会放弃使用OS的BUFFER，因为它有足够的内存，这样，一方便便于管理，一方面速度更快（节省了从内核态的BUFFER拷贝数据到用户态的BUFFER）。但是，SQLite作为嵌入式数据库，受到它所应用的设备资源。所以，它的缓冲块都是按需从OS申请的，这样对内存的使用更少。这也是SQLite消耗内存少的原因之一。

支持(0) 反对(0)

#7楼 2010-06-03 10:27 MichaelMj

@arrowcat
自己编写缓存管理器也可以实时监测内存的空间，动态的增减缓存的大小。或由用户来确定缓冲区的大小。并能够根据用户的要求或者嵌入式设备的不同来确定缓存替代策略，以达到最少的IO次数。所以我觉得如果SQLite自己设计缓存管理器会提高它的性能（缺点是编程工作量和可执行程序的增大）。

支持(0) 反对(0)

#8楼[楼主] 2010-06-03 22:33 YY哥

@MichaelMj
我想你理解错我的意思了。SQLite并不是没有Buffer管理系统，相反，它做的还比较有意思。但是它为什么还要用OS Buffer?我认为这与它的I/O方式和缓冲块的内存分配有关。一方面，它在系统初始化，并不是分配一大块内存作为缓冲区(而通用DBMS是这样做的)；另一方面，它使用简单的同步I/O。SQLite作为嵌入式数据库，它没有必要把I/O搞那么复杂，而且它本身也是单线程，异步I/O往往需要多线程的支持。如果它不用OS的buffer，它进行I/O时，OS只支读取当前块，而由于它应用的环境内存较少，可以经常涉及页面换进换出，这样带来大量的I/O。如果使用OS buffer，一般来说，OS都会相邻的数据缓存在OS的buffer，这样就能减少实际I/O的次数，从而降低I/O对系统性能的影响。以上纯属个人愚见，慎采纳。

支持(0) 反对(0)