4-16-2013

SQLite Page Caching Algorithm

Jason V. Ma
Ryerson University, jason.ma@ryerson.ca

SQLite Page Caching algorithm Modification

Jason Ma
jason.ma@ryerson.ca
April 16, 2013.

Contents

Introduction

SQLite is a database which can be easily embedded inside an application written in the C programming language. One common use is inside the Mozilla Firefox web browser. It supports a subset of the SQL language but was not intended to be a replacement for a multi-user database such as MySQL, Oracle or Microsoft SQL Server.

The entire database is integrated into one *.c file and the command shell for the command line interface is inside another file. Thus to embed the database a developer just has to copy and paste the source code into their database. It also comes in another format with separate files for each component but is not recommended for use due to slower performance. SQLite's developers indicate there is a 5-10% performance improvement when using the amalgamation due to compiler optimizations for a single file [3].

SQLite supports connections via multiple threads, but it does not support intra-query parallelism. For example commercial databases can divide a large query so that it is run concurrently across separate CPUs. SQLite has a very coarse locking level. Since the database is located inside one file, the entire file is locked for writing. Concurrent reads can still occur on the file. It does not support finer levels of locking such as table or row locks.

Project Objective

Modify SQLite's page caching algorithm so that it will be able to determine which page cache to evict based on the number of historical memory references to a page.

SQLite Page Cache Overview

Illustration 1 displays the overall page cache architecture for SQLite. It implements two linked lists. The first is the dirty page list which is a linked list of pointers pointing to the actual data in the LRU list. The dirty page list keeps track of transient pages which have been marked as dirty. Once the number of memory references reaches zero, the page is moved from the dirty list to the LRU list.

The source code (pcache.h, struct PGroup) documents this list as Least Recently Used (LRU) list, however from studying the source code it operates in a simple (First In First Out) FIFO mode. There is no indication in the data structures (struct PGroup, PgHdr1, PgHdr, PCache and struct PCache1) that it keeps track of how often a page was used in the past, nor is there any sort of order.

When a page needs to be evicted the first item pointed to by the LRU list is removed. In our case this would be the page pointed to by the head pointer.

The cache can operate in two modes:

- Shared: There is a global Page Group which can contain multiple caches. Each cache belongs to only one Page Group.
- Separate: Each Page Group has only one cache. Each cache belongs to only one Page Group.

SQLite also permits a third party to implement their own page caching algorithm for the LRU list via the function pointers seen below. A developer can write their own implementation and point the function pointers below to their own implementation. SQLite does not provide an interface for developers to modify the Dirty Page list.

```
typedef struct sqlite3_pcache_methods2 sqlite3_pcache_methods2;
struct sqlite3_pcache_methods2
{
  int iVersion;
  void *pArg;
  int (*xInit)(void*);
  void (*xShutdown)(void*);
  sqlite3_pcache *(*xCreate)(int szPage, int szExtra, int bPurgeable);
  void (*xCachesize)(sqlite3_pcache*, int nCachesize);
  int (*xPagecount)(sqlite3_pcache*);
/* Truncated list of functions */
}
```

SQLite's documentation also explicitly states that "The cache *must not perform any reference counting*. A single call to xUnpin() unpins the page regardless of the number of prior calls to xFetch()" [1]

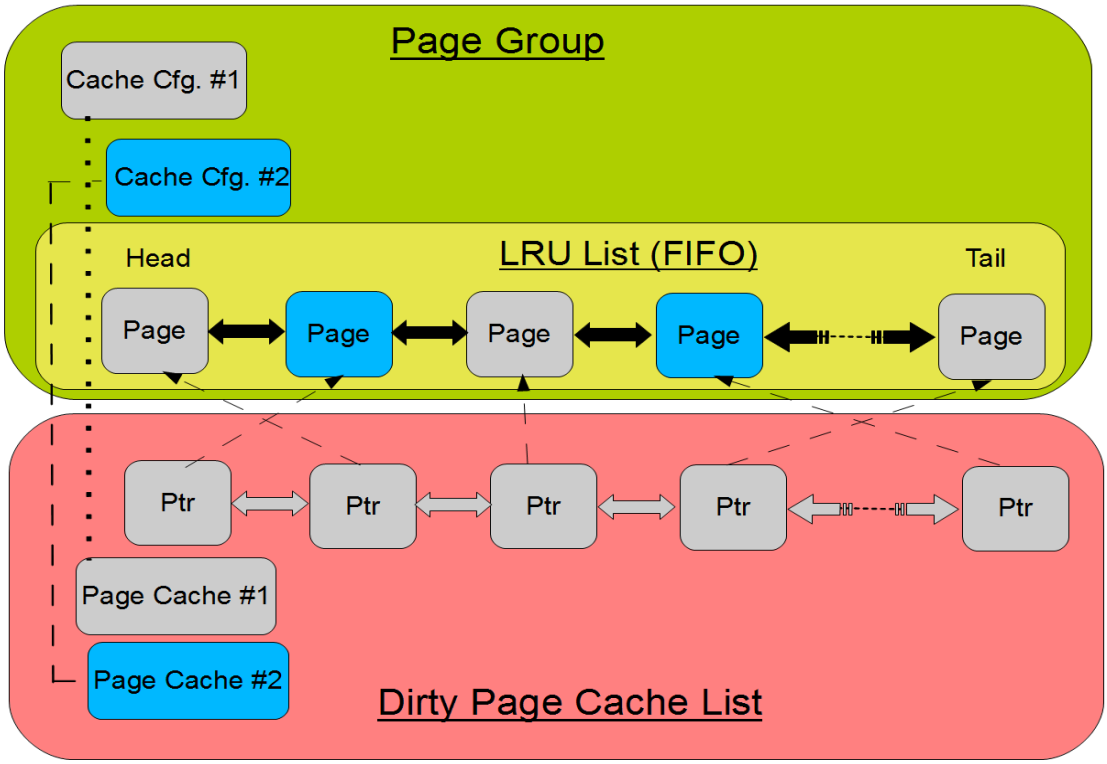


Illustration 1: SQLite Page Cache Overview. Note each page in the LRU list can belong to different caches.

SQLite Page Cache Modifications

Two changes were made to the LRU a algorithm in two stages:

1. Sort the LRU list so that the head contains the least frequently referenced memory page and the tail contains the most frequently used page.
2. The array histLookup was added to improve insertion speed into the LRU list.

The first change is self explanatory. To keep track of the historical references to the page, the function sqlite3PCacheSynchHist() was added to synchronize the historical reference (histRef member) between the structures PgHdr (dirty list structure) and PgHdr1 (LRU structure). Additional changes were made to the function pcache1Unpin.

Change #2 was made to try and improve on the slow performance after making modification #1 above. The goal was to improve insertion performance to reduce the length taken to insert a new item into the LRU list. The idea is to add an index to the linked list so that an insert goes quickly without having to traverse the entire linked list to perform an inserted sorted by histRef value.

This was implemented by adding an array with a size the same as the largest number of histRef counts. For example if histRef wraps around when it reaches 30, the histLookup[] array is the same size. Illustration 2 displays the change.

A detailed list of changes such as functions change is in the Appendix section.

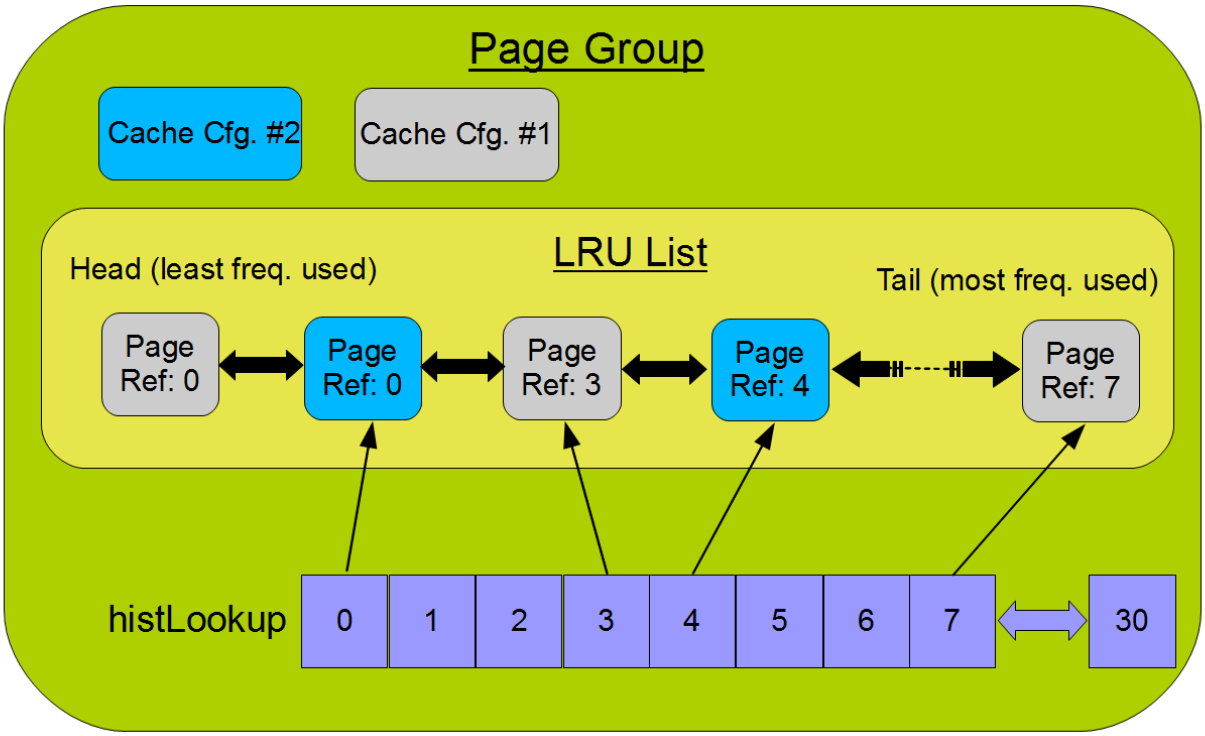


Illustration 2: Modifications made to LRU algorithm. A reference count was added (e.g. Ref:0) and the index histLookup to permit quick inserts. histLookup always points to the last element with the same histRef so that an insert will be made immediately after the node it points to. After the insert histLookup is updated to point to the new node.

Performance Benchmarks

SQLite includes its own set of benchmarks under the test directory which is called from a Tcl script. The test run was speed1.test which contains many other sub tests.

The tests require that development tcl libraries and headers be installed.

1. Add -g (symbols) to TCC and BCC makefile variables
2. Add -ltcl8.5 to LIBTCL makefile variable
3. make test
4. ./testfixture test/speed1.test

Illustration 3 displays the output. Orig is the unmodified SQLite, *Mod. LRU* is the first modification in which the LRU list is sorted. *Opt. LRU* is when the histLookup index was added to try and improve insert performance into the list. The number behind it is the maximum histRef count before it wraps around. The histLookup array for each test was set to the same size as the maximum histRef. The initial modifications have disappointingly slowed the performance of the database down by 3 to 5 times compared to the unmodified SQLite.

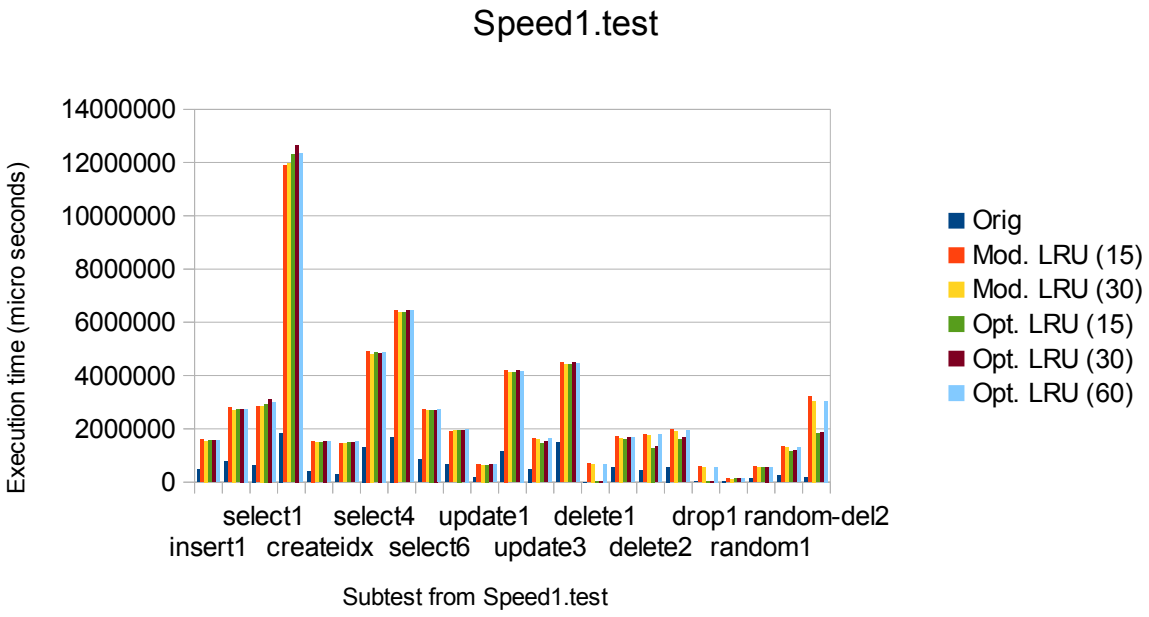
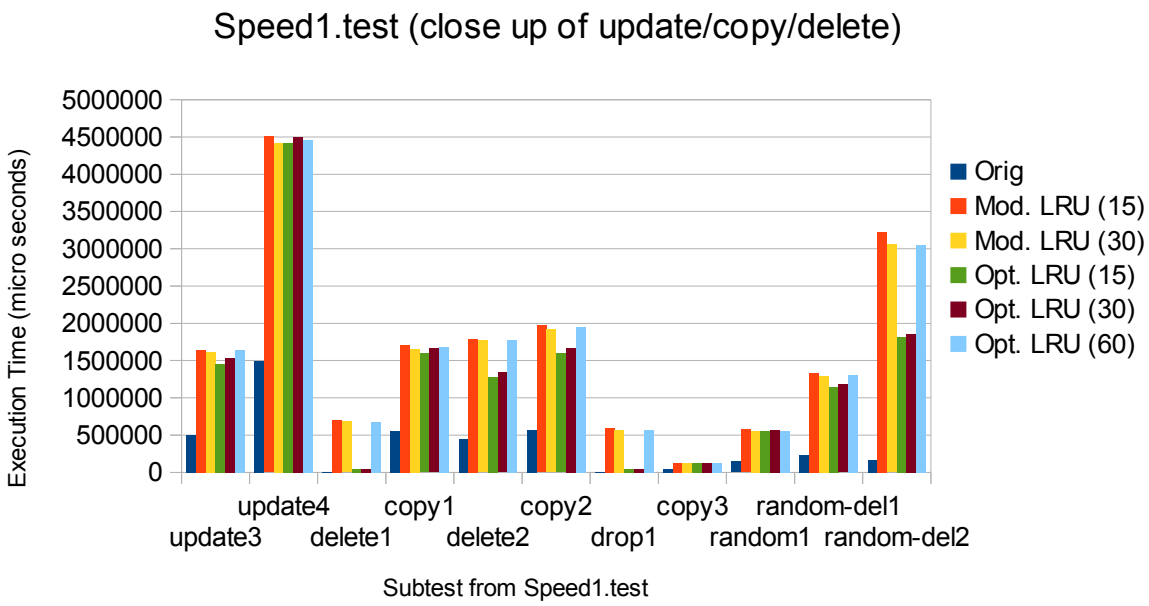


Illustration 3: Mod. is the first modification where the LRU list is sorted. Opt. is when the histLookup index was added to try and improve insert performance.

Illustration 4 is a closeup of some copy/update/drop/delete operations. We can see that after adding

the histLookup array index it improves performance over the original modification of just sorting the LRU list. However it is still much slower than the unmodified SQLite.

Illustration 4: Closeup of update/copy/delete operations since these ran faster then the histLookup index was added.



The Valgrind tool [2] was used to determine where most of the slow down was occurring.

```
valgrind –tool=callgrind –trace-children=yes ./testfixture
../test/speed1.test
```

After making change #1 we see that pcache1Unpin is consuming a lot of time (6.56 CPU execution time). The window on the right shows the source code where the slowdown is occurring. It is the code to traverse the linked list. Illustration 5 shows this output. If Valgrind is run on the unmodified SQLite, pcache1Unpin's CPU usage is only a small fraction and thus very far down the list.

After implementing change #2 by adding the histLookup array index for quick inserts performance improves slightly and we see the overall CPU execution for this time go down from 8th place down to 23rd place in Illustration 6. Valgrind shows the same parts of the source code which is slow. This is a decent improvement but is still slower than the unmodified SQLite. Illustration 6 shows this output.

The algorithm for pcache1Unpin works by:

1. Check if histRef is smaller or equal to the value in the head node of the linked list. If so we can quickly insert it to before the head node.
2. Check if histRef is larger or equal to the value in the tail node of the linked list. If so we can quickly append the new page to the end of the list.
3. Check histLookup index to see if we can quickly insert the value into the list.
4. Traverse list if #3 fails.

Most of the cases should fall into step #1 to step #3, except when the histLookup array is being populated the first time. Additional coding improvements were made by adding the newly inserted page into the histLookup index in step #1 and step #2. After making these changes performance has significantly improved and is close to the original performance of SQLite as seen in Illustration 7

As the maximum number of historical references (histRef) for each page is increased from 15 to 30 there is a slight performance decrease as well. This value does not have any effect on the length of the LRU list.

In Illustration 7 there will still be some overhead during the initial insert into the LRU linked list before the histLookup index array is populated.

SQLite has a hash table for quick lookup of a memory page (**apHash in struct PCache1) when specifying a key (iKey) associated with the page. This is not useful for sorting of the LRU list since it is sorted on histRef instead of iKey.

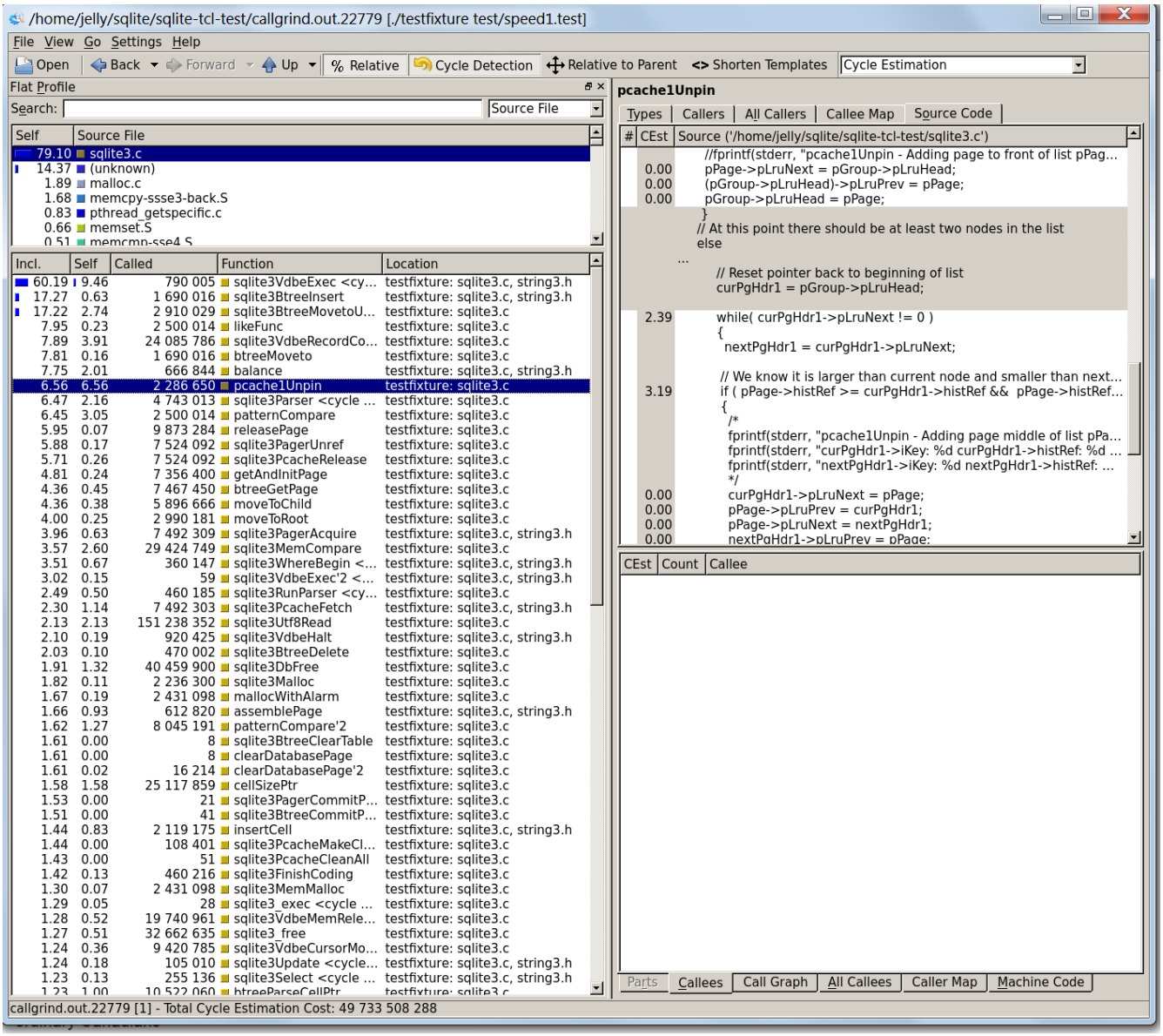


Illustration 5: Valgrind output after making change #1 (sorting LRU list). pcache1Unpin is consuming 6.56 CPU units during execution. Code on right side shows 2.39 CPU units consumed just to traverse the linked list. Note: I'm unable to find how what units are used by Valgrind to measure CPU execution time.

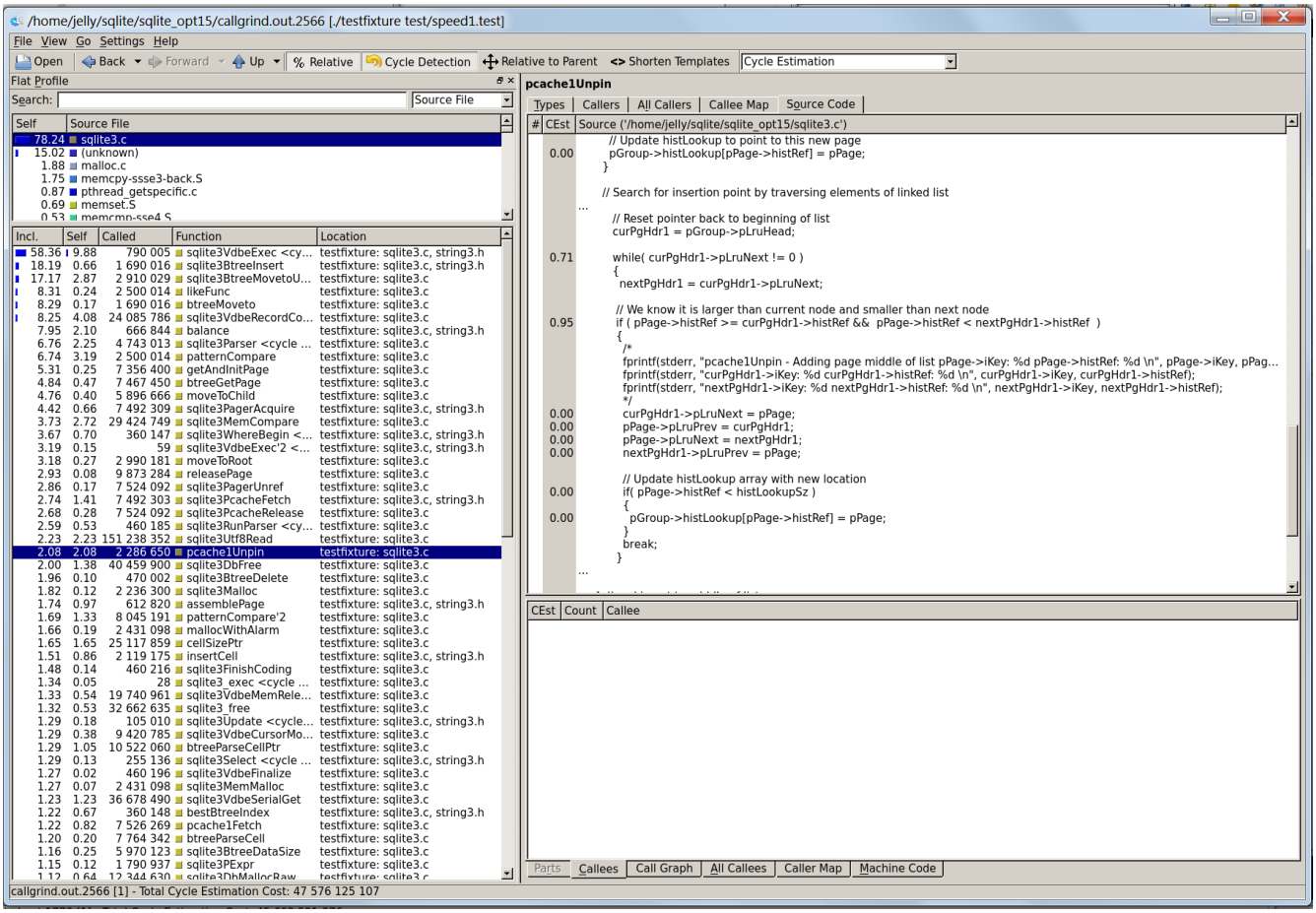


Illustration 6: Valgrind output after adding the histLookup index for quick inserts. Performance is improved but still much slower than the unmodified SQLite.

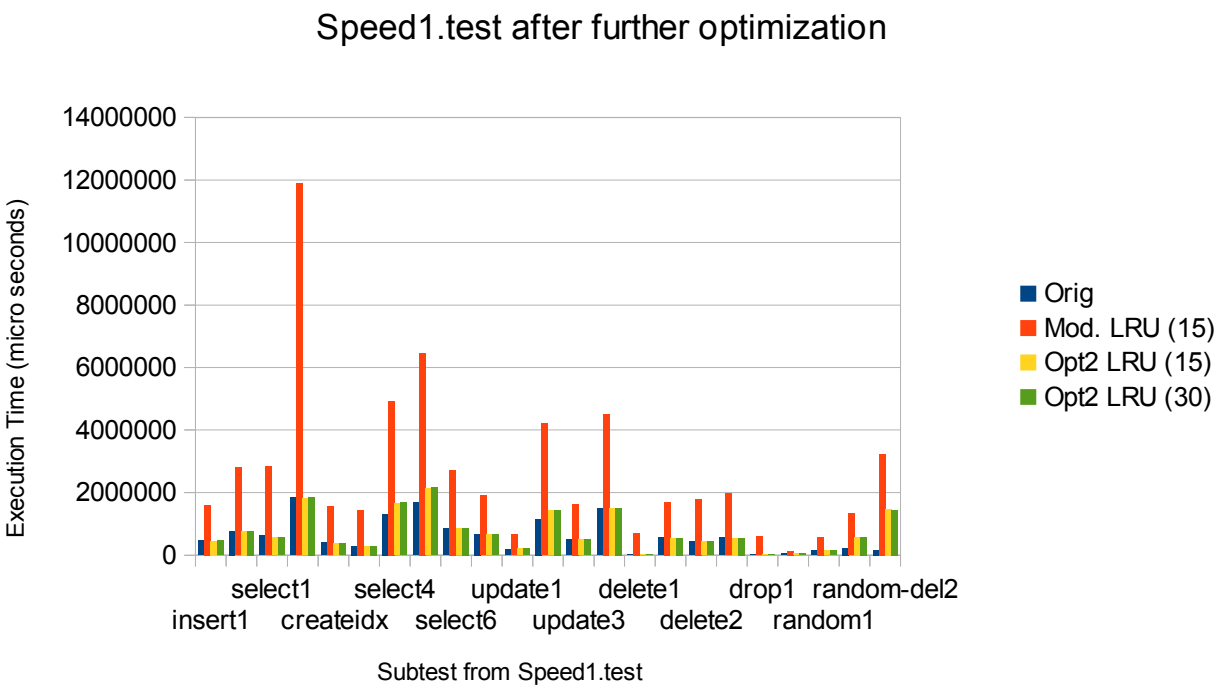


Illustration 7: Significant performance increase after adding original optimizations, performance almost the same as the default SQLite. Opt2 is the new optimization by ensuring histLookup is populated for all cases (see p.9). Opt2 is almost as fast as the unmodified database.

From Illustration 7 and the associated data performance is close a few percentage points within original SQLite for most benchmarks. Illustration 8 shows the numbers used to generate the graph in Illustration 7.

Random deletes (random-del1, random-del2) are still significantly worse by approximately 2 to 8.5x worse than default SQLite. SELECT benchmarks select4 and select5 are 25% and 27% slower compared to Opt2 LRU (15). update1 and delete1 are about 18% and 11% slower than default SQLite respectively.

	Orig	Mod. LRU (15)	Opt2 LRU (15)	Opt2 LRU (30)
insert1	465582	1594615	457555	464109
insert2	773463	2799851	766634	770597
select1	635233	2832418	568456	563469
select2	1838341	11893310	1815030	1858608
createidx	396145	1551221	382539	377332
select3	287580	1434262	287498	285867
select4	1318412	4925507	1650302	1683368
select5	1683413	6467880	2139556	2171434
select6	859303	2719964	849291	863764
vacuum	665195	1910530	658362	658689
update1	180882	658463	213863	216163
update2	1155061	4207034	1420840	1443180
update3	499155	1640951	506391	499641
update4	1498018	4506684	1507706	1511417
delete1	9973	701283	11159	10984
copy1	558177	1704603	543964	540905
delete2	441137	1793114	438189	433722
copy2	569343	1971115	549928	544757
drop1	11848	592399	12612	12603
copy3	43747	129779	41917	42014
random1	145629	574572	141679	140712
random-del1	233280	1334421	582503	573923
random-del2	168091	3225259	1477310	1433234

Illustration 8: Numbers which generated the graph for Illustration 7

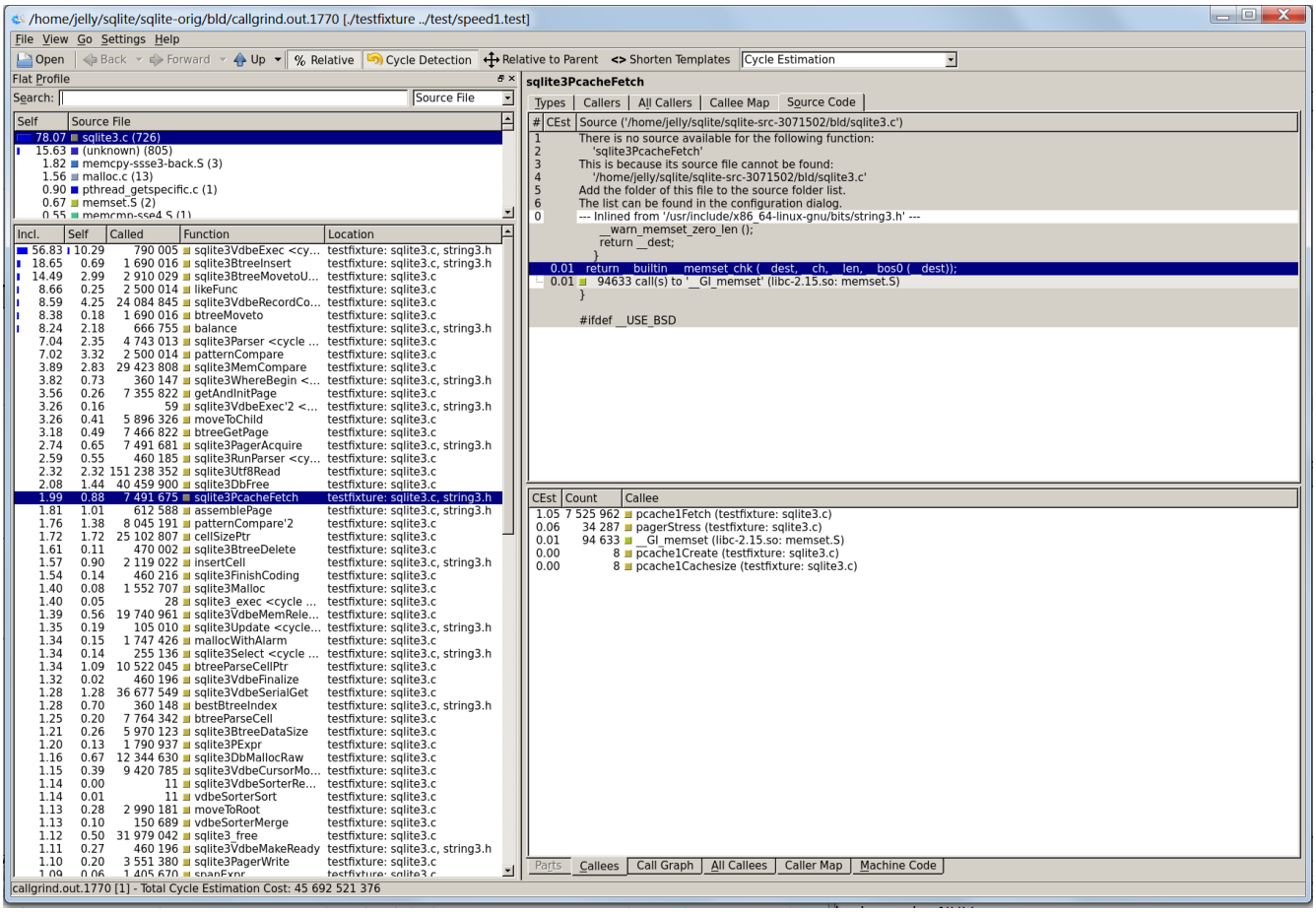


Illustration 9: Valgrind output for unmodified SQLite.

Illustration 9 shows the Valgrind output for unmodified SQLite and Illustration 9 shows the Valgrind output after the final optimization.

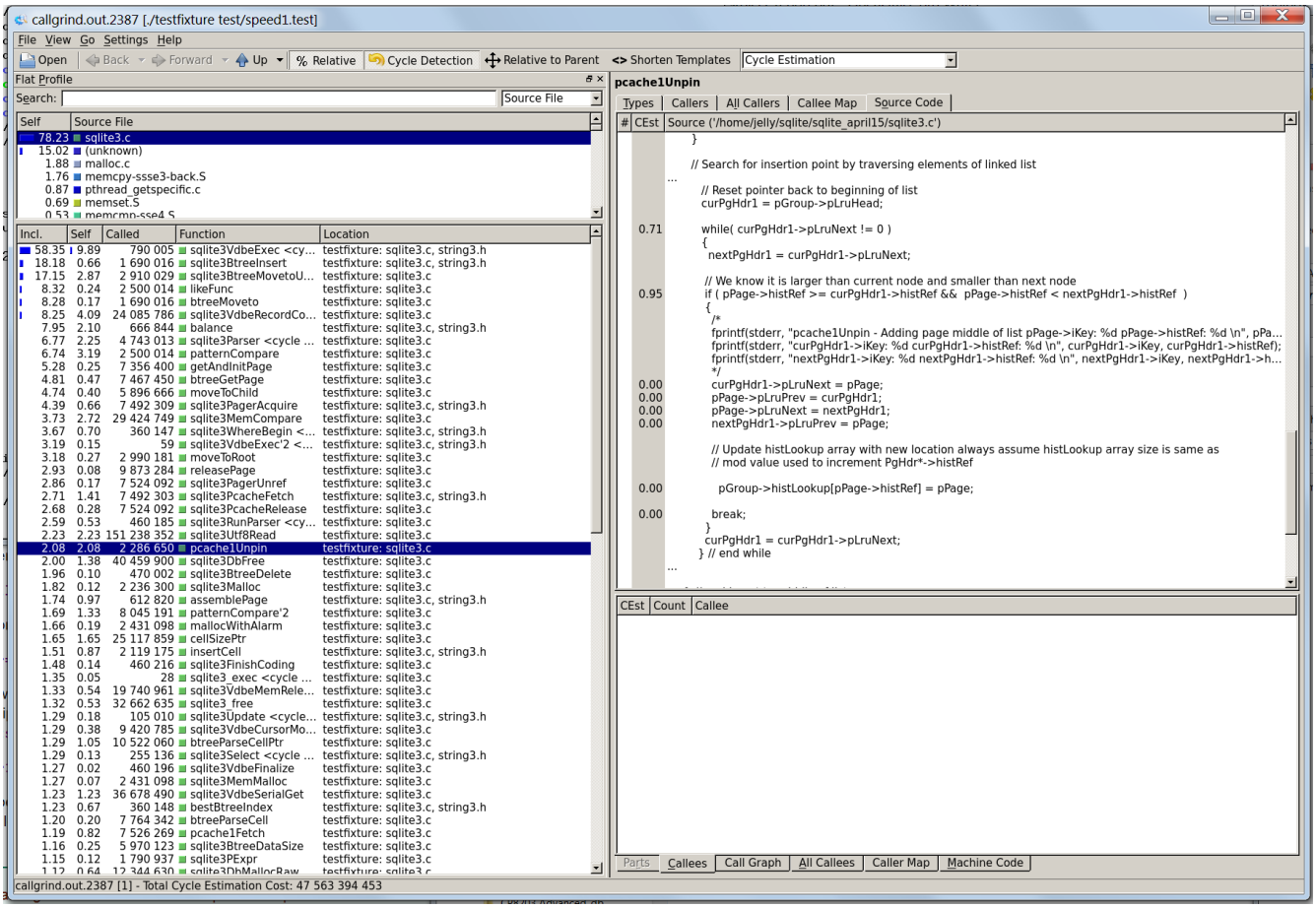


Illustration 10: Valgrind output for all modifications made so that the speed1.test is approximately the same as unmodified SQLite. These correspond to Illustration 7's Opt2 runs.

Conclusion

SQLite's page caching algorithm was modified so the LRU list is sorted by the number of past historical references so that the most frequently used page is kept in memory, but the least recently used is evicted. SQLite's unmodified LRU algorithm is just a FIFO and exhibits no signs that it keeps track of past references.

An index was also added so that items added to the LRU list do not have to traverse the linked list to to perform a sorted insert. After two stages of optimizations, the final performance is very close to the unmodified SQLite database except for a few benchmarks which are still slow than the unmodified SQLite database.

Appendix

List of functions modified. Each modification has a comment beside it called "Jason" so it can easily be found by using the UNIX/Linux grep function.

pcache.h

Function/Structure	Purpose	Change(s) Made
struct PgHdr	Structure used to keep meta data for dirty list page.	Added histRef member to keep track of how many times the page was referenced in the past.
struct PgHdr1	Structure used to keep meta data for LRU list page.	Added histRef member to keep track of how many times the page was referenced in the past.
struct pcache.h		Moved some function declarations from pcache.c to pcache.h so some functions can be called from both pcache*.c files.

pcache.c

Function/Structure	Purpose	Change(s) Made
sqlite3PCacheFetch	Obtain page from cache.	• Initialization of histRef=0 • Call pcache1HistRefInc
sqlite3PcacheRelease	Decrement reference count on page. This is different than histRef count.	• Call sqlite3PcacheSyncHist
sqlite3PcacheRef	Increase reference count on page. This is different than histRef count.	• Increment PgHdr->histRef and wrap around when it reaches the max e.g. 15
sqlite3PcacheDrop	Remove page from dirty list.	• Call sqlite3PcacheSyncHist
sqlite3PcacheMakeClean	Mark page as clean	• Call sqlite3PcacheSyncHist
sqlite3PCacheSyncHist	Synchronize histRef counts between the dirty list and LRU list.	• New function.

pcache1.c

Function/Structure	Purpose	Changes Made
pcache1HistRefInc	Increment histRef count for LRU page.	• New function
pcache1Fetch	Fetch page by key value. Also	• Initialize histRef=0

	allocates new pages as well.	
pcache1Unpin	Mark page as unpinned (eligible for recycling)	• Inserted page based on histRef order • Created quick look up table for inserts into list based on histRef
pcache1PrintList	Used for debug purposes to print out LRU linked list.	• New function

Debug information was added during testing to ensure the linked list contained histRef values in order. Sample of debug fprintf() messages below.

```
======   pcache1Unpin - Enter  =====

pcache1Unpin - pCache->nRecyclable: 6
pcache1Unpin - pPage->iKey: 15
pcache1Unpin - pPage->histRef: 0
pcache1Unpin - pGroup->pLruHead.iKey: 13
pcache1Unpin - pGroup->pLruHead.histRef: 0
pcache1Unpin - pGroup->pLruTail.iKey: 14
pcache1Unpin - pGroup->pLruTail.histRef: 2


pcache1Unpin - Adding page to front of list pPage->iKey: 15 pPage->histRef:
0
============ Printing Linked list after adding to tail ==============
PgHdr1->iKey 15  PgHdr1->histRef 0
PgHdr1->iKey 13  PgHdr1->histRef 0
PgHdr1->iKey 10  PgHdr1->histRef 0
PgHdr1->iKey 9  PgHdr1->histRef 0
PgHdr1->iKey 11  PgHdr1->histRef 0
PgHdr1->iKey 12  PgHdr1->histRef 1
PgHdr1->iKey 14  PgHdr1->histRef 2
============ Linked list end ==============
inc pCache->nRecyclable 7
======   pcache1Unpin - Exit  =====
pcache1PinPage - Dec. to pPage->pCache->nRecyclable: 6
pcache1PinPage -exit
pcache1PinPage - Dec. to pPage->pCache->nRecyclable: 5
pcache1PinPage -exit
======   pcache1Unpin - Enter  =====

pcache1Unpin - pCache->nRecyclable: 5
pcache1Unpin - pPage->iKey: 15
pcache1Unpin - pPage->histRef: 0
pcache1Unpin - pGroup->pLruHead.iKey: 13
pcache1Unpin - pGroup->pLruHead.histRef: 0
pcache1Unpin - pGroup->pLruTail.iKey: 12
pcache1Unpin - pGroup->pLruTail.histRef: 1
```

```
pcache1Unpin - Adding page to front of list pPage->iKey: 15 pPage->histRef:
0
============ Printing Linked list after adding to tail ==============
PgHdr1->iKey 15  PgHdr1->histRef 0
PgHdr1->iKey 13  PgHdr1->histRef 0
PgHdr1->iKey 10  PgHdr1->histRef 0
PgHdr1->iKey 9  PgHdr1->histRef 0
PgHdr1->iKey 11  PgHdr1->histRef 0
PgHdr1->iKey 12  PgHdr1->histRef 1
============ Linked list end ==============
inc pCache->nRecyclable 6
======   pcache1Unpin - Exit  =====
```

References

[1] SQLite – Application Defined Page Cache
http://www.sqlite.org/c3ref/pcache_methods2.html

[2] Valgrind
http://valgrind.org/

[3] SQLite Amalgamation
http://sqlite.org/amalgamation.html