ENHANCEMENTS TO SQLITE LIBRARY TO IMPROVE PERFORMANCE ON MOBILE PLATFORMS

A Thesis

by

SHYAM SAMBASIVAN RAMACHANDRAN

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	A.L. Narasimha Reddy
Committee Members,	Riccardo Bettati
	Paul V. Gratz
Head of Department,	Chanan Singh

August 2013

Major Subject: Computer Engineering

ABSTRACT

This thesis aims to present solutions to improve the performance of SQLite library on mobile systems. In particular, two approaches are presented to add light-weight locking mechanisms to the SQLite library and improve concurrency of the SQLite library on Android operating system. The impact on performance is discussed after each section.

Many applications on the Android operating system rely on the SQLite library to store ordered data. However, due to heavy synchronization primitives used by the library, it becomes a performance bottleneck for applications which push large amount of data into the database. Related work in this area also points to SQLite database as one of the factors for limiting performance. With increasing network speeds, the storage system can become a performance bottleneck, as applications download larger amounts of data.

The following work in this thesis addresses these issues by providing approaches to increase concurrency and add light-weight locking mechanisms. The factors determining the performance of Application Programming Interfaces provided by SQLite are first gathered from IO traces of common database operations. By analyzing these traces, opportunities for improvements are noticed. An alternative locking mechanism is added to the database file using byte-range locks for fine-grained locking. Its impact on performance is measured using SQLite benchmarks as well as real applications. A multi-threaded benchmark is designed to measure the performance of fine grained locking in multi-threaded applications using common database operations.

Recent versions of SQLite use write ahead logs for journaling. We see that writes to this sequential log can occur concurrently, especially in flash drives. By adding a sequencing mechanism for the write ahead log, the writes can proceed simultaneously. The performance of this method is also analyzed using the synthetic benchmarks and multi-threaded benchmarks. By using these mechanisms, the library is observed to gain significant performance for concurrent writes.

DEDICATION

This thesis is dedicated to my parents who have been instrumental in providing unlimited support in all my endeavors and inspired me to achieve more.

ACKNOWLEDGMENTS

I would like to thank Dr. Reddy, my committee chair, for his invaluable support and guidance through the course of research work. I am grateful for the opportunities he gave for constant learning and development. I would like to thank my committee members, Dr. Bettati and Dr. Gratz, for taking their precious time to review my thesis and providing valuable feedback. I would also like to thank the members of Dr. Reddy's research group for the great suggestions and discussions during the time I spent with the group. Finally, I am thankful to my family members and friends who extended their support and help when it was needed.

TABLE OF CONTENTS

Page

ABSTRACT....ii

DEDICATION....iv

ACKNOWLEDGMENTS....v

TABLE OF CONTENTS....vi

LIST OF TABLES....viii

LIST OF FIGURES....ix

1. INTRODUCTION 1

1.1 Background....1
1.2 Related Work....2
1.3 Android System Overview....3
1.4 SQLite Database....5

2. IMPROVED CONCURRENCY THROUGH BYTE-RANGE LOCKING....9

2.1 Locking Mechanisms in SQLite....9
2.2 Implementation of Byte-Range Locks....12
2.2.1 Control Flow of SQLite Read and Write Transactions....13
2.2.2 Changes to Implement Byte-Level Locking 15

3. EVALUATION OF BYTE-RANGE LOCKING....20

3.1 Experimental Setup....20
3.2 Benchmarks and Results....21
3.2.1 Androbench....22
3.2.2 RL Benchmark....23
3.2.3 Browser Benchmark 24
3.2.4 DB Bench....28
3.3 Conclusion of Enhancement using Byte-Range Locking....30

4. ENHANCEMENT TO WRITE AHEAD LOGGING....31

4.1 Overview of Write Ahead Logging....31

4.2 WAL Implementation in SQLite [3]..32
4.3 Enhancements to Write Ahead Log...35

5. EVALUATION OF CHANGES TO WRITE AHEAD LOGGING.........................38

5.1 DB Bench Benchmark...38
5.2 Comparison between SSD and HDD...40
5.3 Maximum Performance Gain Analysis - DB Bench..42
5.4 Sandisk and Intel SSD – A Comparison..43
5.5 Comparison with Berkeley DB..46

6. CONCLUSION AND FUTURE WORK...47

REFERENCES..49

LIST OF TABLES

Page

Table 1: Locking Mechanisms in SQLite 10

Table 2: Byte-Range Calculation...18

Table 3: Header of the Write Ahead Log...32

Table 4: Frame Header of the Log...33

LIST OF FIGURES

Page

Figure 1: Sub-Systems in Android Architecture [5]..4

Figure 2: Components of SQLite [8]..6

Figure 3: Page Cache..7

Figure 4: Exclusive Locking for Writes...11

Figure 5: Byte-Range Locks for Pages..16

Figure 6: Androbench : Transaction Per Second for Various SQLite Operations 22

Figure 7: Androbench : Time Taken for Various SQLite Operations..............................23

Figure 8: RL Bench : Time Taken for Various Operations..24

Figure 9: Firefox Benchmark : Download Times for Tabs 1-4 : Size 128KB.................26

Figure 10: Firefox Benchmark : Download Times for Tabs 1-4 : Size 256KB................26

Figure 11: Firefox Benchmark : Download Times for Tabs 1-4 : Size 512KB................27

Figure 12: Firefox Benchmark : Download Times for Tabs 1-4 : Size 1MB...................27

Figure 13: DB Bench : 2 Threads 28

Figure 14: DB Bench : Time Taken for Overlapping vs Non-Overlapping Updates.......29

Figure 15: Write Ahead Log : Locking Mechanism and Appending Mechanism............34

Figure 16: Write Ahead Log : Locking Mechanism with Sequencer...............................36

Figure 17: DB Bench - 10K Inserts on Intel SSD..39

Figure 18: DB Bench - 25K Updates on Intel SSD..39

Figure 19: DB Bench : Comparison of SSD and HDD - Vanilla Code............................40

Figure 20: DB Bench : Comparison of SSD and HDD - Delayed Checkpoint................41

Figure 21: DB Bench : Comparison of SSD and HDD - Sequencer...............................41

Figure 22: DB Bench : Comparison of SSD and HDD - Both Enhancements.................42

Figure 23: DB Bench : Performance Comparison for Various Enhancements.................43

Figure 24: DB Bench : Comparison of Flash Drives - Vanilla Code...............................44

Figure 25: DB Bench : Comparison of Flash Drives - Delayed Checkpoint....................44

Figure 26: DB Bench : Comparison of Flash Drives - Sequencer...................................45

Figure 27: DB Bench : Comparison of Flash Drives - Both Enhancements....................45

Figure 28: Comparison of SQLite and Berkeley DB..46

1. INTRODUCTION

This section covers the background, related work and motivation for the work presented in this thesis.

1.1 Background

Mobile systems are extensively used today and the technology behind such hand-held devices has been constantly improving. With increase in network connectivity speeds, the software and hardware stack on these devices must see improvements to provide a good user experience. From the storage perspective, all of these systems have an internal flash memory with option for an external flash storage. The storage sub-system in particular plays a major role in determining the performance of applications. The application developer has access to a set of application programming interfaces to store persistent-data related to the application. In mobile systems based on Android, this functionality is provided by the SQLite [1] database library. The SQLite database layer provides an interface to store structured data on the persistent storage. Most applications use the SQLite library to store data as well as meta-data for its use.

Recent studies [2] have shown that optimizations to the SQLite layer can offer significant improvements in performance. The study finds a correlation between application performance and the underlying storage performance. The reason for such a correlation is identified to be because of poor flash device performance, random I/O from application databases and use of synchronous writes. While the performance of flash device can be improved by replacing the hardware, we see a potential in making changes to the SQLite layer for performance.

The processor speeds on cell phones and tablets are increasing. Additionally, most of these devices are powered by multi-core processors. Such hardware promotes multi-threaded applications to be written for such devices. Enhancements in the storage stack can reasonably improve the overall performance. Some enhancements to improve concurrent access to the SQLite database have been discussed in sections 2 and 4 of the thesis.

In the work described in this thesis, we look into some methods that can improve performance at the database layer. The SQLite library is also enhanced to allow efficient writes to the database for applications which are multi-threaded by design.

1.2 Related Work

The choice of SD cards and their impact on applications on mobile systems has been extensively studied by Kim et. al. [2]. They also propose a set of pilot solutions to address the performance issues on mobile devices. It involves employing a small phase change memory (PCM) to store performance critical data, using a log-structured file system for SQLite databases or making changes to the fsync code-path in order to considerably reduce the synchronization primitives. The pattern of I/O to the SQLite database has been studied and it is seen that it mostly remains random and with some writes being mostly updates to the same block address. The default synchronization setting of writes to the database is full synchronous writes. While the use of synchronous writes can be disabled by the application, it can risk the durability of the database and therefore, most applications do not disable it.

Some methods like placing the SQLite database on RAM or changing the application interface to the SQLite database have also been discussed in [2]. The work in this thesis focuses on using light-weight locking solutions and improvements to the write ahead log [3] of SQLite.

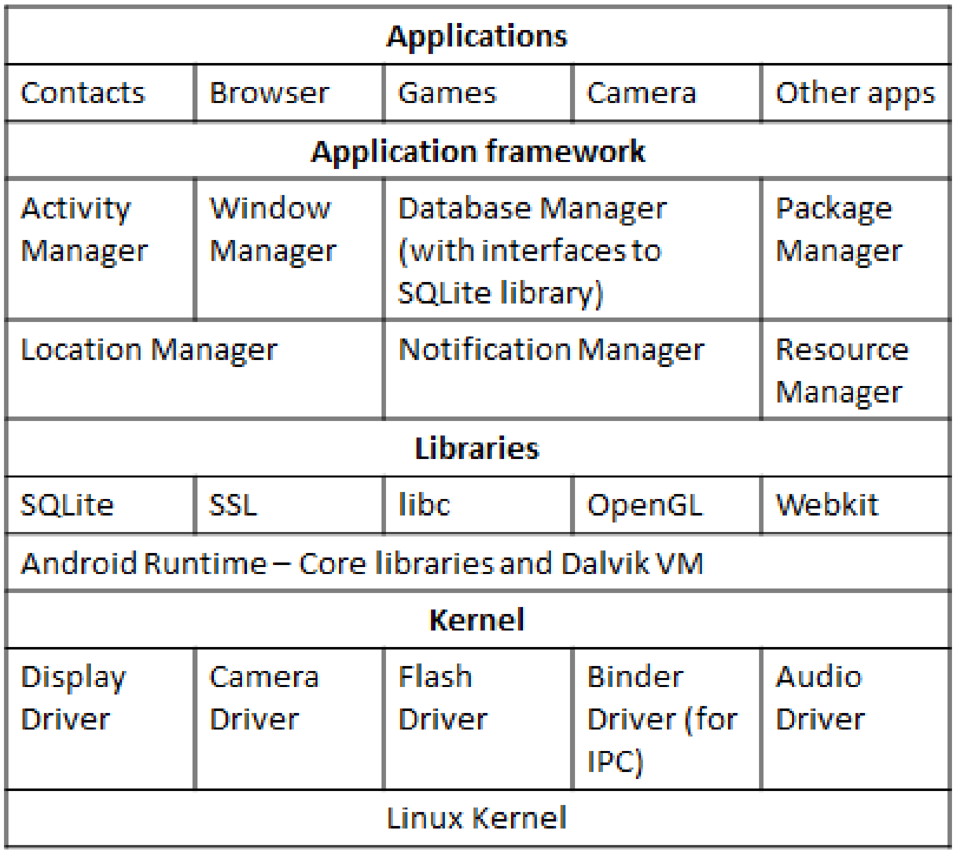
1.3 Android System Overview

The Android operating system is developed on top of the Linux Kernel [4]. The Android version used throughout this work is Android 4.0, code-named ice-cream sandwich. Linux 3.0 kernel is used in this version of Android.

Android is an operating system which includes more than just the Linux kernel. Some of the additions (Figure 1) include a binder driver for inter-process communication and many native libraries like libc, SQLite, SSL, Dalvik VM and other core libraries. Android applications come in the apk format containing the dex byte code. This byte code can be understood by the Dalvik VM. It is essentially used to sandbox each application and run as a separate process. Android also provides a whole lot of APIs for application developers to access the native libraries. The application framework is mostly written in JAVA. The native libraries are written in C or C++.

At the root directory level, some of the main directories that are mounted on start up are /data, /system, /dev and /sdcard. Android likes to keep the application and its data separately. This is enabled by creating separate directories for each application in the

/data directory. Each application directory in turn contains 'cache', 'databases', 'lib' directories. The 'databases' directory contains all the related databases that are created by the application.



Applications				
Contacts	Browser	Games	Camera	Other apps
Application framework				
Activity Manager	Window Manager	Database Manager (with interfaces to SQLite library)		Package Manager
Location Manager		Notification Manager		Resource Manager
Libraries				
SQLite	SSL	libc	OpenGL	Webkit
Android Runtime – Core libraries and Dalvik VM				
Kernel				
Display Driver	Camera Driver	Flash Driver	Binder Driver (for IPC)	Audio Driver
Linux Kernel				

Figure 1: Sub-Systems in Android Architecture [5]

An application can access the data only within its directory. The /system directory contains the native libraries and frameworks. The shared library libsqlite is present in '/system/lib'. SQLite is written in C and so, a Java Native Interface (JNI) must be created for applications to access the library. The libsqlite_jni shared library is present in '/system/lib/'.

Whenever an application wants to access the functionality provided by the low level library, it uses the interfaces provides by the application framework. The package “android.database.sqlite” [6] contains all the classes and interfaces related to creating and managing an SQLite database. In this way, Android provides a convenient way for applications to store structured data. The SQLiteDatabase class among others provides most of the public methods required to create a database, making configuration changes, executing statements, transaction processing and enabling/disabling write-ahead logging. These methods result in calls to libsqlite which is the core of storing and retrieving data. An application may choose to create as many databases in its /databases directory and have as many threads accessing these databases.

In this work, we focus on the SQLite library and make enhancements to the locking mechanism and write ahead logging. An overview of SQLite is provided in section 1.4.

1.4 SQLite Database

SQLite is widely used on mobile systems since it is light-weight as well as most suited for embedded applications. It is a transaction based relational database system [7]. The library contains three main components namely the SQL compiler, core processing methods and the back-end interface. The core processing methods have an interface, SQLite command processor and a virtual machine generator. The interfaces in this unit is frequently accessed by the external applications.

The main functionality of the library is in present in the virtual machine generator and back-end interface (Figure 2). The virtual machine is also called the Virtual Database Back-End (VDBE). When an sqlite command is executed, a new VDBE is created, which is nothing but a state machine to execute the correct set of back-end methods to achieve the desired task.

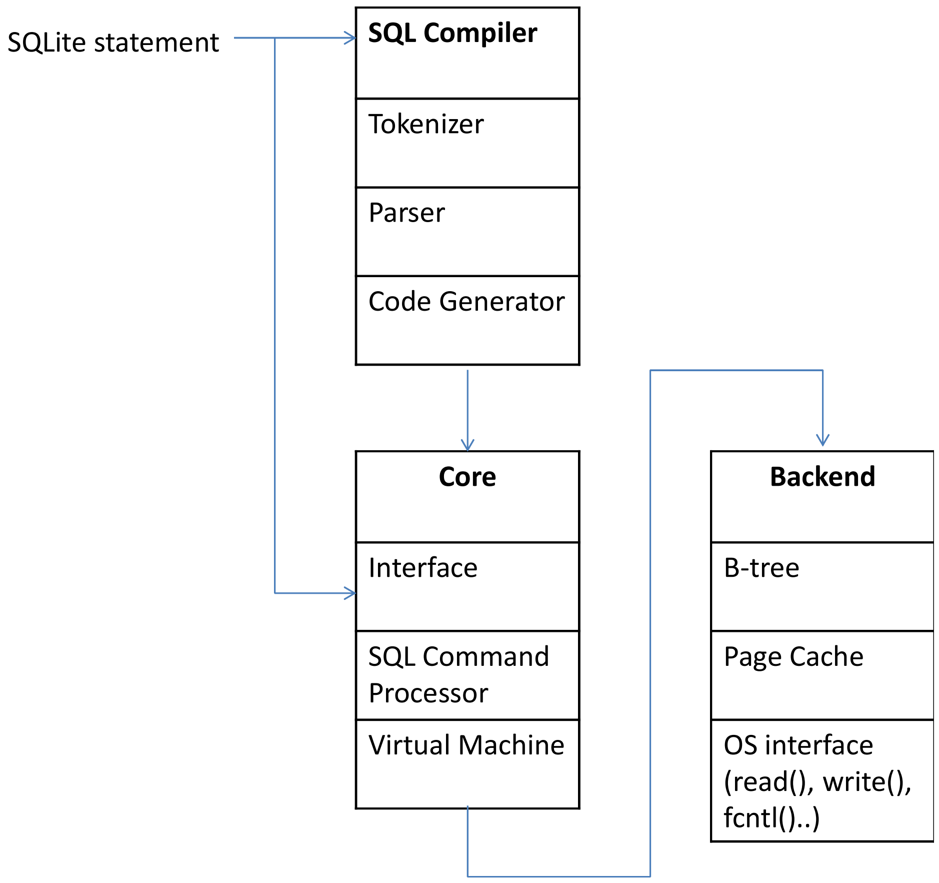


Figure 2: Components of SQLite [8]

The Back-End itself consists of three main modules. The first is the B-tree data structure to maintain the indices in memory for the database. Each table has a separate B-tree associated with it. The Pager module or the page cache is at the core of every transaction. The SQLite database is a single file on disk divided into a number of pages. The page cache among others contains a list of all the dirty pages that have to be written to the disk. If the cache is full, the pages are replaced in a variant of least recently used fashion. A logical view of a page cache is shown in Figure 3.

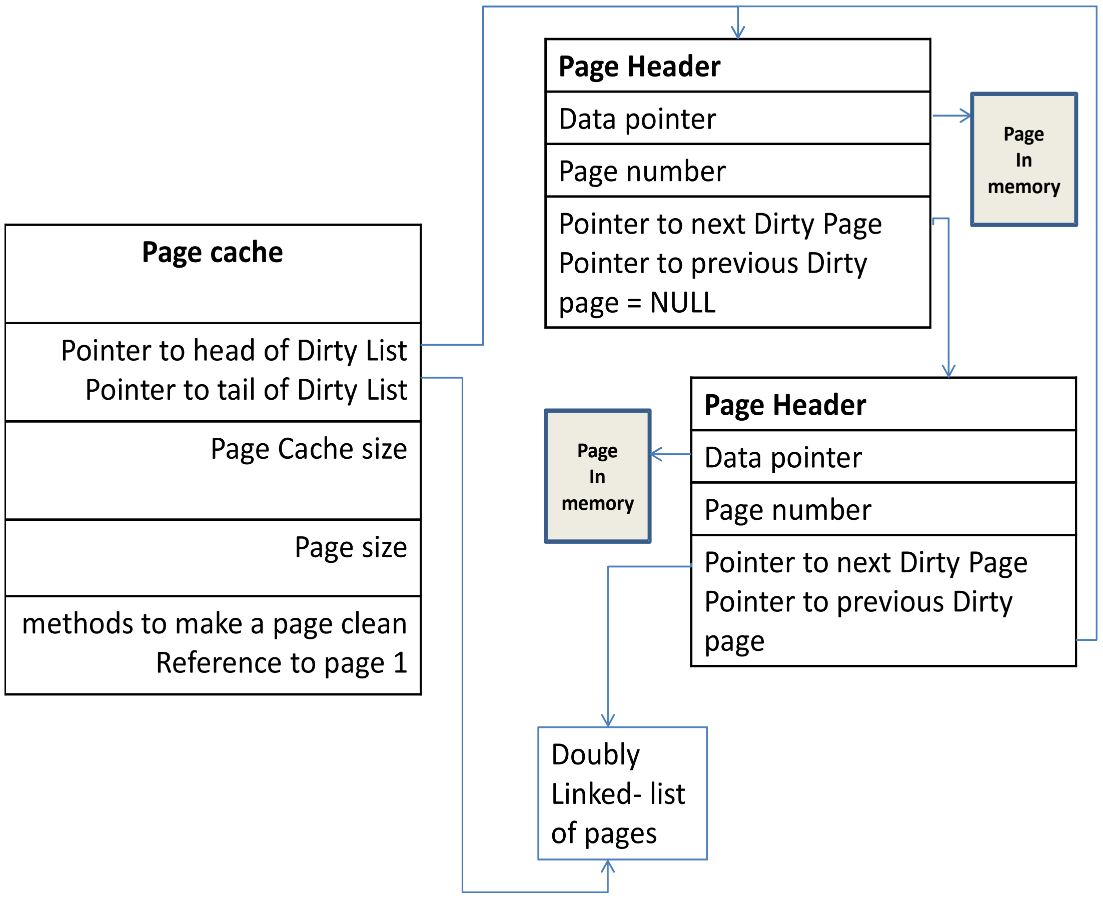

Page cache
Pointer to head of Dirty List
Pointer to tail of Dirty List
Page Cache size
Page size
methods to make a page clean
Reference to page 1
Page Header
Data pointer
Page number
Pointer to next Dirty Page
Pointer to previous Dirty page = NULL
Page In memory
Page Header
Data pointer
Page number
Pointer to next Dirty Page
Pointer to previous Dirty page
Page In memory
Doubly Linked- list of pages

Figure 3: Page Cache

The OS interface module functions as a wrapper over various system calls such as read, write and fcntl in a portable way. Most of the locking primitives come into play here. SQLite uses exclusive file level locking for writing to the database file.

There is an additional module in SQLite for viewing the contents of the database from the command line shell. This can be obtained by executing the 'sqlite3' command on the busy-box. It is a helper module to view the schema of the database, perform unit tests or execute SQL statements over the command line. It can also be used to collect traces and for debugging purposes.

2. IMPROVED CONCURRENCY THROUGH BYTE-RANGE LOCKING

This section describes the locking mechanisms in SQLite, the changes to implement byte-range locks and how it reduces the time for a transaction is discussed. The SQLite library with and without the byte-lock changes are compared using various benchmarks and the results are analyzed.

2.1 Locking Mechanisms in SQLite

The SQLite database is stored as a single file on disk. The database file is organized as pages, each of size 512 bytes or a higher multiple of 2. All the pages that are read from the database or written to the database reside in a page cache. The page cache is essentially a portion of memory allocated by the SQLite library in the heap of the process address space. For the page cache to be consistent, the writes to the page cache must be serialized. Moreover, the writes from the page cache to the database file must take place under an exclusive lock, so that the page cache does not get corrupted in the middle of a data transfer. To make this work, SQLite currently has four main levels of locking [9] as described in Table 1.

SHARED	This lock allows just reading the database. There can be as many processes / threads holding a SHARED lock simultaneously. However, even if a single SHARED lock is held by any process / thread, the database cannot be written to by acquiring an EXCLUSIVE lock. The use of SHARED lock increases concurrency, mainly by allowing multiple readers.
RESERVED	When a thread acquires a RESERVED lock, it indicates that the thread is about to write to the database in the near future. It should be noted that the thread is still doing only reads. Only a single thread can acquire a RESERVED lock at a time.
PENDING	This lock indicates that the thread is ready to write to the database and is waiting for the other SHARED locks to clear. The difference between RESERVED and PENDING is that new SHARED locks cannot be acquired while there is a PENDING lock. Existing SHARED locks are however allowed to move forward.
EXCLUSIVE	The EXCLUSIVE lock must be obtained to commit to the database file. Only one EXCLUSIVE lock is allowed per

Table 1: Locking Mechanisms in SQLite

	database file (Figure 4). To maximize concurrency, SQLite works to minimize the amount of time that EXCLUSIVE locks are held.

Table 1 Continued

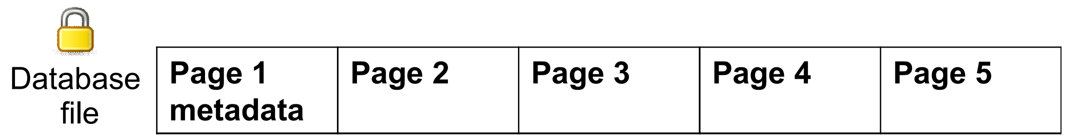


Figure 4: Exclusive Locking for Writes

The multiple levels of locking was introduced to allow for transaction level concurrency. SQLite allows multiple transactions from different database connections. This is enabled by a mode called Shared Cache mode [10], where the the Pager cache is shared between two or more connections. The library takes care of serializing the writes to the database. It supports table-level locking and so writes to two tables can go simultaneously. SQLite does not allow multiple transactions from a single thread, i.e., we cannot have a 'Begin Transaction' within another 'Begin Transaction'. Moreover, multiple reads can also co-exist.

The locking structure creates a problem of deadlock and starvation despite improving concurrency [11]. For example, if two threads read the database after acquiring a SHARED lock and subsequently make some changes and decide to write to the database. Only one of the two threads can acquire a RESERVED lock to go on to obtain an EXCLUSIVE lock. However, EXCLUSIVE lock cannot be obtained now because the other thread is holding a SHARED lock. SQLite solves this deadlock problem by allowing the thread requesting a RESERVED lock to retry a fixed number of times. It would not succeed and hence return an error code to the application, which has to be handled by the application. The thread with the EXCLUSIVE lock will proceed, making way for the other thread to acquire the SHARED, RESERVED and EXCLUSIVE locks to commit. It can be seen that this creates a problem of starvation.

While SQLite is not designed for high performance concurrent applications which cannot tolerate starvation, the number of locking steps between SHARED lock and EXCLUSIVE lock mean more calls to the operating system. Moreover, the two threads may be writing to two different pages of the database file. This scenario could benefit from using light-weight locking mechanisms and fine-grained locks for the database file.

2.2 Implementation of Byte-Range Locks

In this implementation, the granularity of locking is modified from database file-level locks to page-level locks. By doing so, exclusive locking of the database file can be avoided when multiple threads are writing to non-overlapping pages of the database file. Whenever a commit occurs, a set of pages are written to the database. Before writing this list of pages, they are sorted based on the page numbers, a byte-level lock is obtained from the starting page to the last page in the page list and then the write call is invoked. Since the page-list is sorted before acquiring the byte-level lock, there is no chance that two threads acquire write locks on overlapping pages, thereby avoiding the situation of a deadlock.

This section is divided into two sub-sections: 2.2.1 describes the Control flow of SQLite read and write transactions and section 2.2.2 describes the changes to implement byte-level locking.

2.2.1 Control Flow of SQLite Read and Write Transactions

Common SQLite statements such as INSERT, UPDATE, DELETE and REPLACE result in data being manipulated and eventually end as sqlite read or write calls to the actual database file. The SQLite exec method is a wrapper around the prepare, step and finalize methods.

The prepare method checks for the integrity of the Database schema version number, by comparing the integer value read from the database file with the integer value of the in-memory schema representation.

The step method starts a Virtual DataBase Engine (VDBE) [12] which works on the parameters passed to start a state machine. The state machine's job is essentially to figure out the right set of operations to perform on the SQLite statement. When it is time to begin a read or write transaction to the database, the VDBE calls the Btree's begin transaction method. Every database has an associated B-tree data structure. The B-tree is an on-disk representation of the database, with a separate B-tree for each table and index in the database. The B-tree methods access the lower Pager sub-system to perform reads and writes to the database. The I/O occurs in fixed block/page sizes of 512 bytes or a higher power of 2. This is called the page size. When the pager sub-system is ready to begin a transaction, it calls sqlite3PagerBegin() method. Until now, the lock on the database file was a SHARED lock. Within the call to sqlite3PagerBegin(), the lock level is raised to RESERVED lock. The pages are read from the database file and placed in the page cache. The VDBE state machine performs appropriate actions on the page based on the type of statement. If the page has to be written back to the database, the page is marked writable and the rollback journal is updated. The steps in changing the Journal file involves opening the Journal, reading the Journal header and writing the previous version of the page out to the Journal. At a point when the transaction has come to an end, the VDBE commit state kicks in. At this stage, the data from the page cache must be destaged to the database file. The VDBE commit method acquires an Exclusive lock on the database file, syncs the Journal just in case the the data was still held in the OS buffers, and performs two phases of commit. The first phase writes out the pages to the database file and the second phase decrements the lock level from EXCLUSIVE to RESERVED and then to SHARED. If required, the Journal file is truncated since all the data has been committed.

The finalize method is used to free up the memory allocated to the VDBE and delete the instance of the VDBE.

The control flow of the read and write was obtained by adding traces to functions and dumping them to a file.

The locking primitives on the database file is implemented by the unixLock() method which accepts the file descriptor and lock-type as arguments. The method first does a sanity check to see if the request is valid. A valid request is one in which a higher lock is requested while holding a lower level lock. A thread will be able to obtain a

RESERVED only if it holds a SHARED lock. The section 2.2.2 provides details on the changes made to implement byte-range locking.

2.2.2 Changes to Implement Byte-Level Locking

In the original code which had 4 levels of locking, the reserved and pending locks were required to avoid write starvation. Byte-range locks enable locking portions of the file exclusively. With such locks in place, it is possible to eliminate reserved and pending locking levels. This is because, the entire file is not locked for a write operation and so portions of a file can still be read using shared locks. In the modified approach, the byte-range locks are implemented at a page level. A page is the smallest unit that can be read or written to the database file. The locking state of each page is maintained in memory. In this way, it is possible to read portions of file, still writing to other pages in the database file.

A locking method named unixLockByte() is created, which takes the start_byte and end_byte of the file to be locked in addition to the file descriptor and lock-type. The locking is internally implemented using POSIX fcntl() system calls. The fcntl() call [13] [14] takes the file descriptor, a command parameter which tells the system call to get/set record locks and the third argument is a pointer to an flock structure. The flock structure contains mainly the following:

1. Lock type – Read lock, write lock or Unlock
2. start_byte – the starting offset in bytes relative to 'whence'

3. whence – it can be either SEEK_SET, SEEK_CUR or SEEK_END. The starting byte offset will be set from the beginning of the file, current position of the file descriptor or from the end respectively.

4. length or the range or bytes to lock from start_byte

5. pid of the process holding the lock in case it is already held by another process.

The unlock method is also implemented in a similar fashion. The unixUnlockByte() method takes the file descriptor, type of lock to downgrade to and the start_byte and end_byte. The fcntl() call is used to decrement the locking level by specifying lock type of flock structure as F_UNLCK. The unlock methods are called by the phase-two commit methods since SQLite follows a two-phase locking mechanism to ensure the isolation property of the database is maintained. Locks are acquired until the phase-one commit but never released. During the phase-two commit, the locks are released in the reverse order.

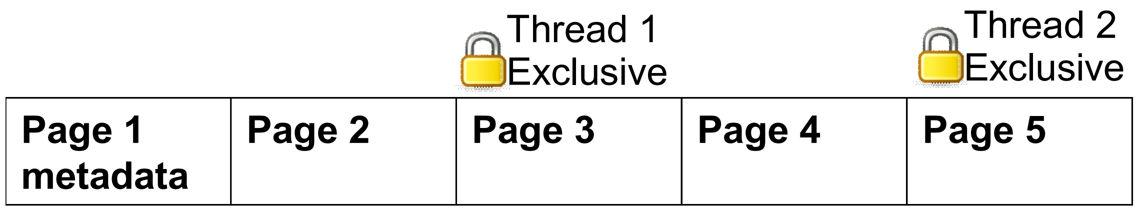


Figure 5: Byte-Range Locks for Pages

With the enhancement of byte-range locks in place at page level (Figure 5), two threads can write to different regions of the database file. The behavior of the record-level or byte-level locks follows the same principle enforced by using SHARED, RESERVED and EXCLUSIVE locks. When there are one or more read locks on a given byte-range, a write lock is denied until all the read locks have been cleared. Only a single thread can hold a write lock for a byte-range. For all other cases, the lock is granted. Therefore, multiple reads can go in parallel in this case too.

Since we want more light-weight locking mechanisms, the number of levels of locking is reduced to using SHARED and EXCLUSIVE locks. The SHARED locks are also obtained as byte-range locks and are used to read the database. Since multiple F_RDLCK (read locks) can co-exist, reads can go in parallel. If two or more threads want to write to the same byte-range, they will be serialized. The problem of starvation still exists, since writes cannot go ahead until all the reads to a byte-range have completed. But we see that it does not make a big difference because applications do not perform heavy concurrent reads and write on the same block. The I/O pattern mostly remains random with some updates to same block addresses [2].

SQLite uses the first page on the database file to store meta-data about the database. There is a 4-byte integer at offset 24 called 'dbFileVers', which is incremented after each database change. While acquiring the SHARED lock, this integer is read from the database file and checked with the in-memory copy. If there is a mismatch, the Page cache is flushed to remove any invalid entries. In our implementation, it is possible that a thread writes to a database file and another thread sees a mismatch of this field, thereby flushing the page cache. However, the library is built with Shared-Cache feature enabled, so that multiple connections to the database from a process can share the page cache. Moreover, in android, each application runs as a process and it has its own database files. Therefore, the entire page cache is shared between multiple threads which open a connection to the database and the pages that were changed are visible to all the threads. As an optimization, the check of 'dbFileVers' is not performed and the page cache is not flushed while acquiring the SHARED lock. It may also be possible to mark as dirty / invalidate the modified pages in the cache of multiple processes if more than one process is accessing the database. But, that feature is not implemented as part of this project.

At the point where phase-one commit is done, the page list to be committed is sorted by the page number. Locks are always acquired for the range of sorted page range and they never should overlap. Locks are also acquired until phase one commit and released in phase two commit. This ensures a deadlock free system.

The equations to calculate the starting and ending byte of the commit is given in Table 2. The variable 'begin' is initialized to the first page number in the commit list and the variable 'end' is initialized to the last page number in the commit list.

Byte Range	**Expression to calculate byte-range**
start_byte	(begin-1) * pageSize
end_byte	(end * pageSize)

Table 2: Byte-Range Calculation

An EXCLUSIVE byte-range lock is obtained on the database file from start_byte to end_byte, the page list is written to the database and the database is unlocked and the lock-level is set to SHARED lock. If an error occurs in any of the operations, the EXCLUSIVE lock is given up and the normal error handling routines are called.

This concludes the implementation of byte-range locks for writing to the database file. The next section describes the evaluation of byte-range locking using different benchmarks.

3. EVALUATION OF BYTE-RANGE LOCKING

This section is split into 3 sub-sections – 3.1 covers the experimental setup, 3.2 covers the benchmarks that were run and 3.3 concludes this section with some comments.

3.1 Experimental Setup

The code was developed for the Android operating system. Android was ported to the x86 platform by the android-x86 project [15]. This enables the use of Android on x86 processors either directly or over Virtual Machines. The virtual machine approach is more suitable because it allows ease of testability and debugging. The android operating system is compiled as an iso image [16] which can be loaded using the Virtual Player. We use Virtual box [17] as a virtual machine player for the experiments. The configuration of the host system and the space allocated for the guest operating system are as follows:

- Android x-86 kernel image on Virtual Box
- Intel i3 – Quad core CPU 2.1GHz
- 6 GB System RAM
- 1 GB RAM allocated to Virtual machine
- The virtual number of cores is set to 4
- 2 GB SATA hard disk for data storage, including apps and data

After the android image is installed as a virtual machine, the image disk is unmounted to enable normal booting of the operating system. The Android-x86 in a virtual box offers the same features as the firmware loaded into a mobile device. Additionally, the command line can be accessed by switching to TTY1. The command line is a busybox shell utility [18] that allows the user to enter frequently used Linux commands. The guest machine can access the host machine's internet connection. The IP address of the guest machine can be assigned by starting the DHCP client daemon [19]. The DNS server address may sometimes not be set properly. It can be set using the 'setprop' command [20]. Now that the internet connection is setup, we can proceed transferring applications from the host machine to the guest machine using SCP file transfer [21]. The APK files [22] can be installed using ADB install command [23] from the busybox shell. On a successful install, the application becomes accessible from the apps screen.

We use four benchmarks to evaluate the performance of using byte-range locks. The discussion on benchmarks and the results are given in section 3.2.

3.2 Benchmarks and Results

The benchmarks used to evaluate the performance of byte-range locks are :

- Androbench
- RL Bench
- Firefox Browser
- DB Bench

3.2.1 Androbench

Androbench [24] is one of the benchmarks available to measure the transactions per second (TPS) and latency of SQLite insert, update and delete operations. The benchmark is run with SQLite insert, update and delete each doing 300 transactions. The Figure 6 gives the TPS for each of the operations and a comparison of vanilla code and with the byte-range lock changes. It can be seen that the modified code can give up to 15% more transactions per second than the vanilla code for insert and delete operations, but no significant gain for update operations. This benchmark is a single threaded benchmark, and so the gain that we see is mainly from the light-weight locking mechanism. In Figure 7, we see a corresponding pattern for latency of insert, delete and update.

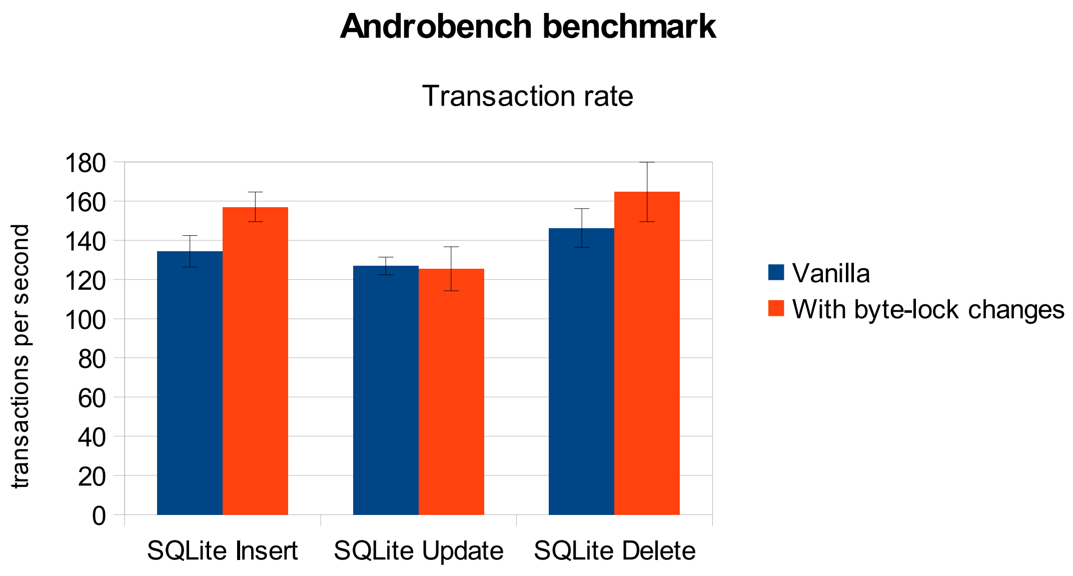


Figure 6: Androbench : Transaction Per Second for Various SQLite Operations

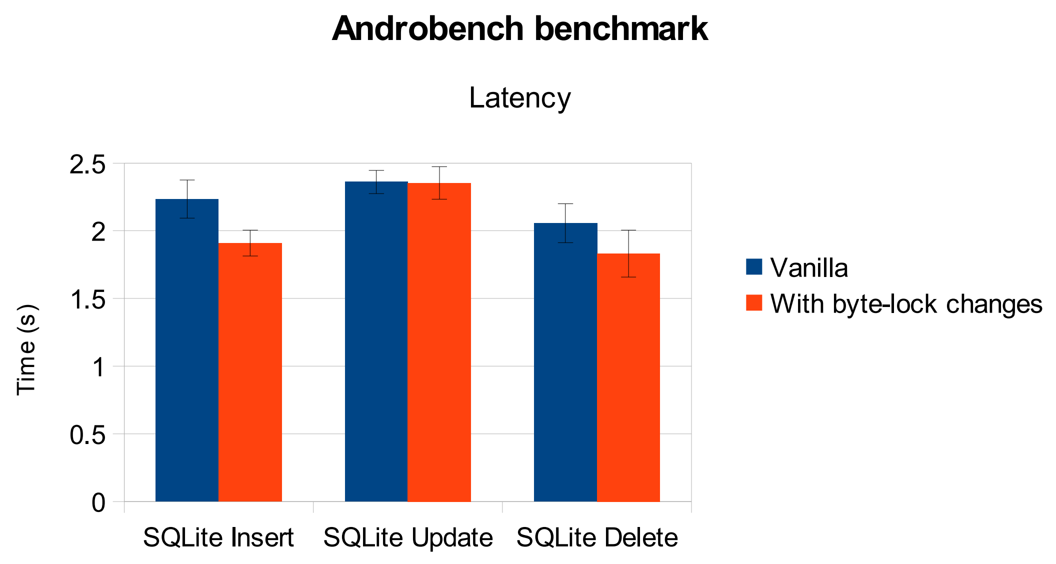


Figure 7: Androbench : Time Taken for Various SQLite Operations

3.2.2 RL Benchmark

The RL benchmark [25] is used to measure the time taken for a variety of database operations. The benchmark starts a predefined set of operations and gives the time taken for individual set of operations as well as the total time for all operations. The Figure 8 gives the data for 1000 inserts, 25000 inserts within a transaction and 25000 inserts into an indexed table inside a transaction. It was noticed that for other operations, there was no significant improvement and hence they are not shown. While the performance of the three types of operations show an improvement, the operations performed within a transaction generally show higher performance gains. This is due to the reduced number of commit operations to the database file. Also, with an indexed table there is about 15% performance gain. This is mainly obtained by avoiding the overhead to iterate through the data structure and find the correct index.

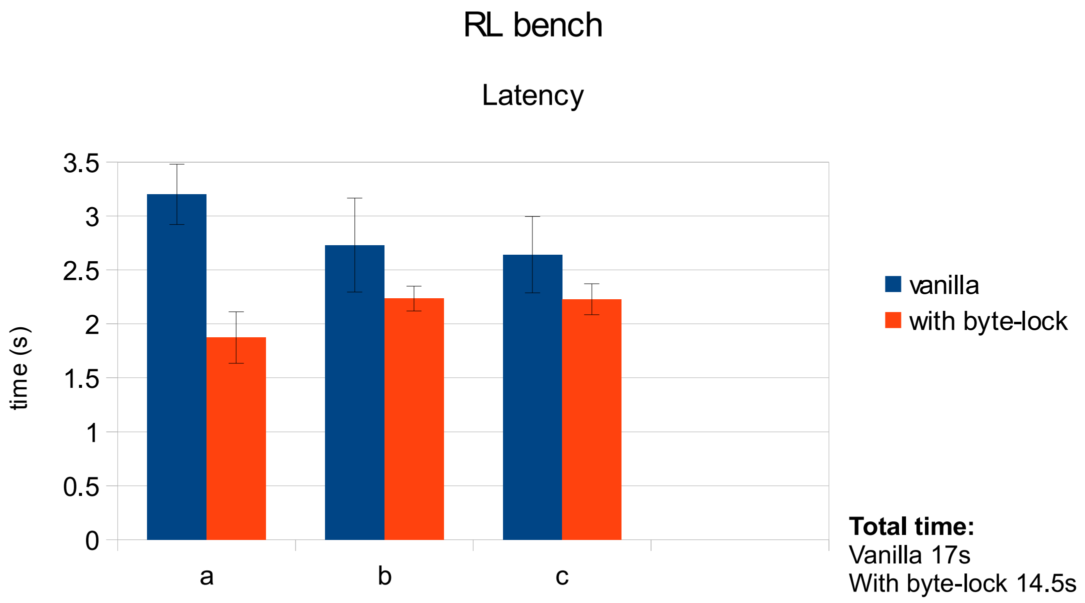


Figure 8: RL Bench : Time Taken for Various Operations

x-axis : (a) 1K inserts (b) 25K inserts in a transaction (c) 25K inserts into an indexed table in a transaction

3.2.3 Browser Benchmark

Next, we look at a browser benchmark using the Firefox web browser [26]. The Firefox browser uses SQLite to store data and meta-data. This was verified by adding traces to the SQLite library and running the Firefox application. By opening multiple tabs simultaneously, the database must be accessed by each tab to commit data. A mechanism is designed to open tabs simultaneously. First, the required number of tabs are opened, then the app is closed. When the browser is restarted, the web-cache is cleared and the previous session is restored. At this time, the data is downloaded for each tab and the database is accessed for each tab. The total duration from opening the set of tabs to the point when the web page is loaded by all the tabs is measured. The page being downloaded is present in the host machine. In this way, we can eliminate the variation in download times of internet pages. The download size represents the size of the web-page downloaded by each tab. The following Figures 9 - 12 give the download times for each set.

The improvement for download size of 128 KB is in the order of 10% on an average for multiple tabs. We also notice that there is no significant improvement in download time for a single tab in download sizes 256, 512 and 1024 KB. For all other cases, there is an average improvement of 15% in download times. The peak improvement is seen with download size of 512 KB. In this case, (Figure 11) we can get around 20% improvement on an average. The download sizes have an impact on determining the performance gain. As download size increases, more data is pushed to be written to the drive. This essentially takes majority of the time and hence we see a lesser improvement in download times. It should be noted that the fsync() calls in the SQLite library are not disabled and hence the library ensures that data is written to the drive cache. The data containing the download time for various data sizes is shown in Figures 9 - 12, along with the standard deviation of each data-set.

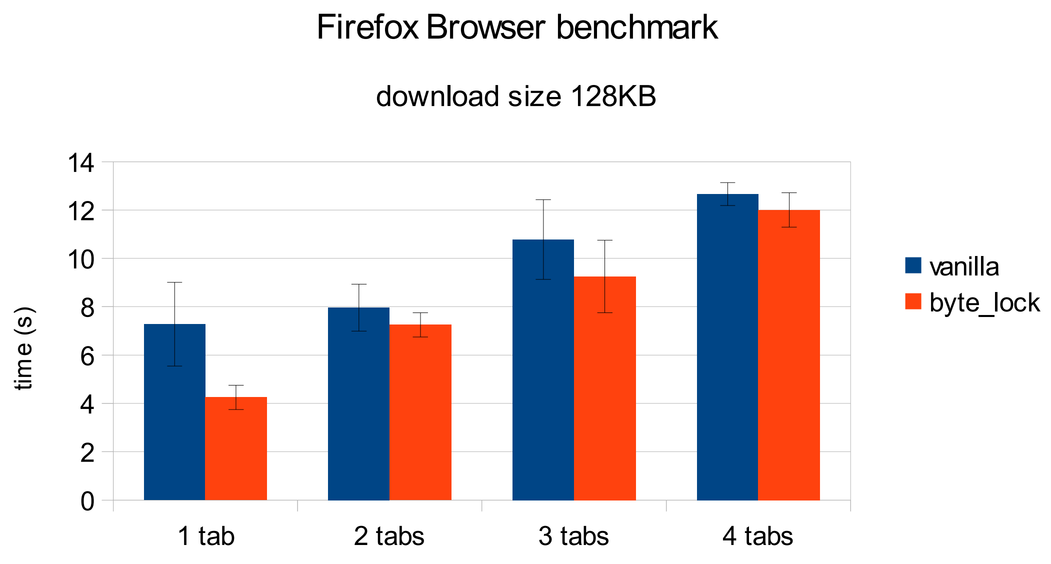


Figure 9: Firefox Benchmark : Download Times for Tabs 1-4 : Size 128KB

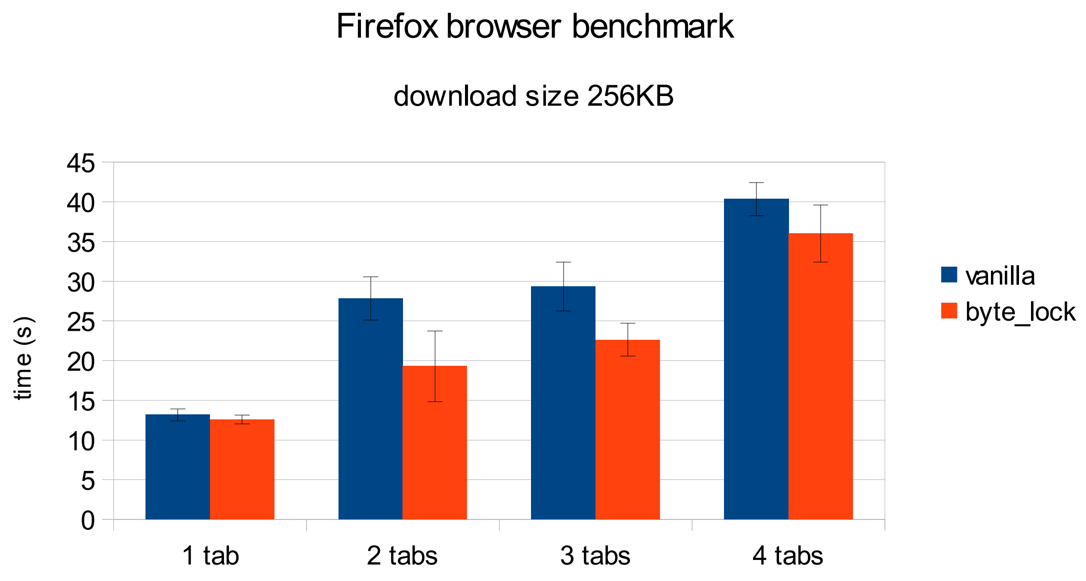


Figure 10: Firefox Benchmark : Download Times for Tabs 1-4 : Size 256KB

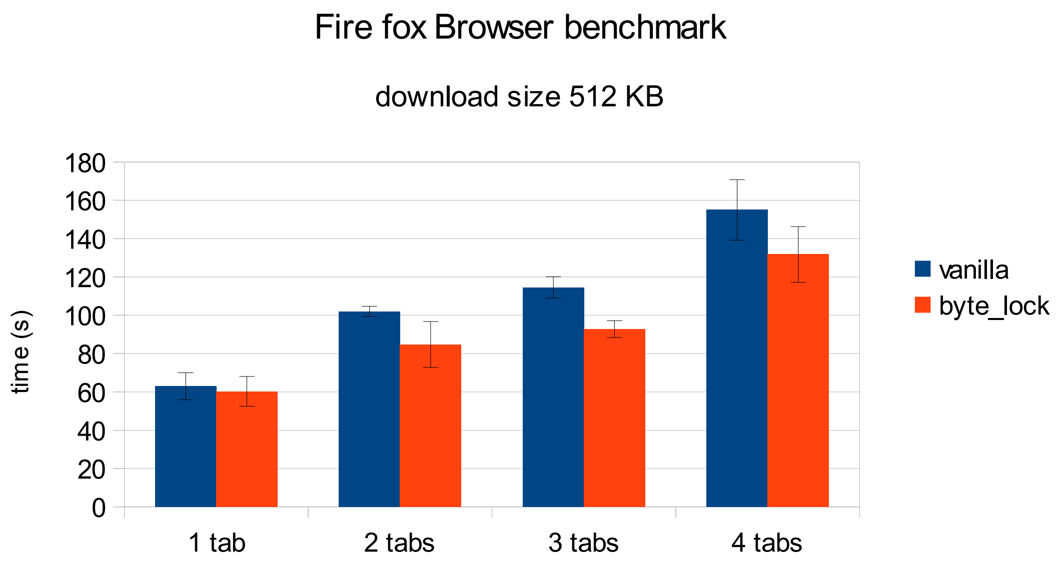


Figure 11: Firefox Benchmark : Download Times for Tabs 1-4 : Size 512KB

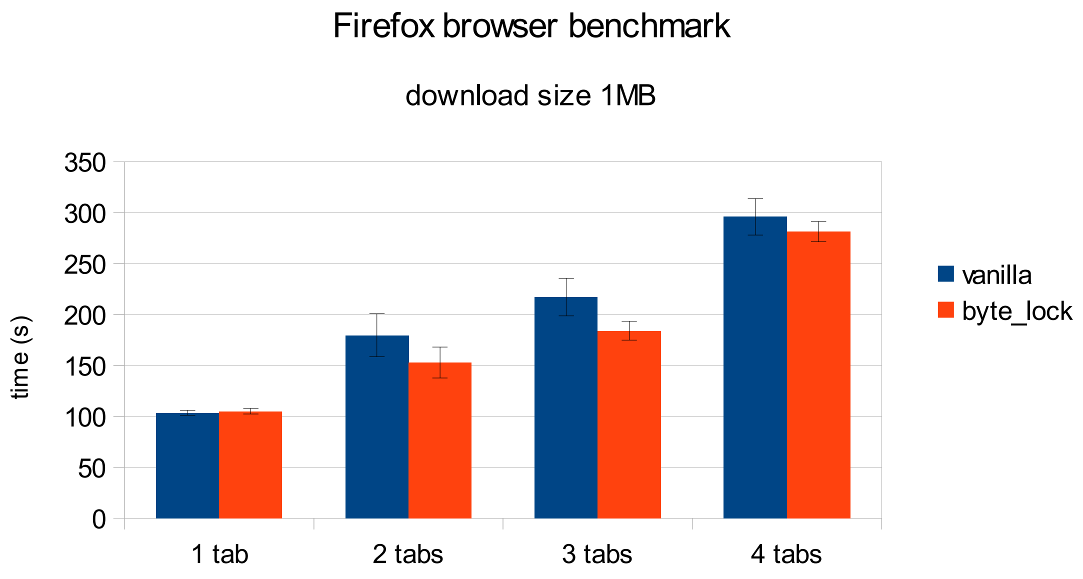


Figure 12: Firefox Benchmark : Download Times for Tabs 1-4 : Size 1MB

3.2.4 DB Bench

Next, we look at a customized benchmark extended from TestIndex benchmark [27]. The original benchmark is a single-threaded benchmark and the customized one is a multi-threaded benchmark. It performs 4 operations – inserting 1000 records, performing 20000 index searches, iterating through 20000 records and deleting 10000 records. The benchmark, DB Bench opens multiple connections to the database and accesses different tables from each thread. SQLite supports table-level locking and so the writes can go simultaneously to two different tables. At the database EXCLUSIVE lock level, we have byte-range locks and so, the pages would be locked independently.

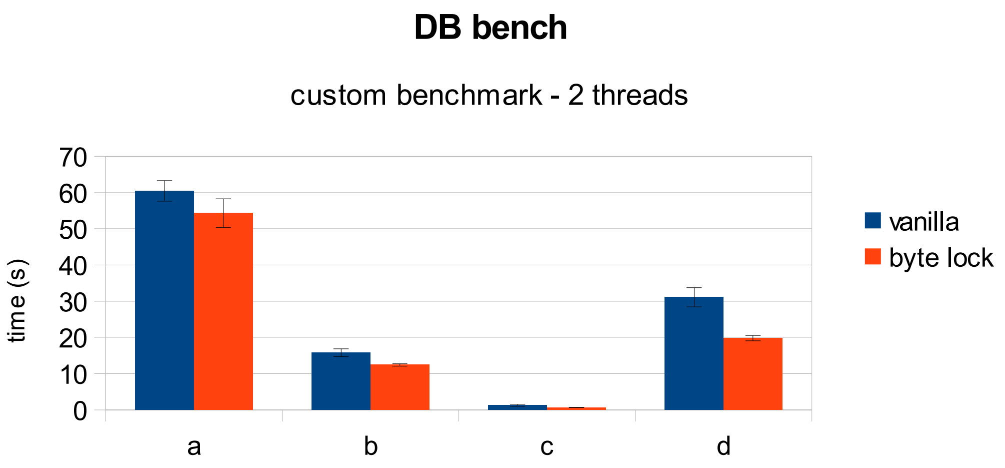


Figure 13: DB Bench : 2 Threads

Time taken for (a) inserting 1K records, (b) performing 20K index searches, (c) iterating through 20K records and (d) deleting 10K records

From Figure 13, we can notice a performance improvement of 10% in case of 2 threads inserting 1000 records each. There is an average improvement of 10% for index searches and 20% for deleting records. The performance gain also comes from the light-weight locking mechanisms that were used.

The custom benchmark is also extended to perform overlapping updates, where data from the same table is accessed and non-overlapping updates, where data from two different tables are updated.

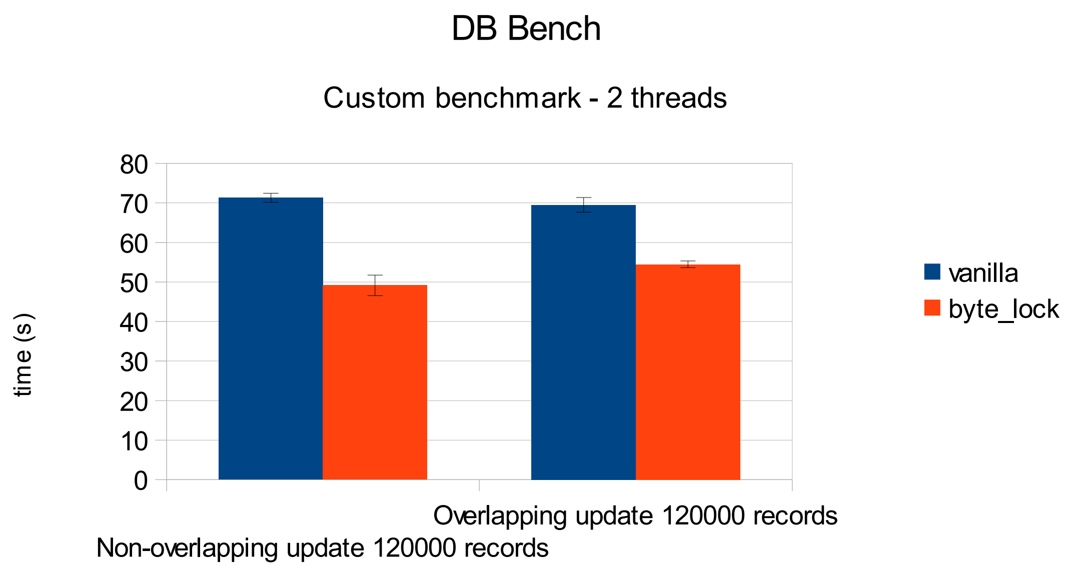


Figure 14: DB Bench : Time Taken for Overlapping vs Non-Overlapping Updates

From Figure 14, we see that there is a performance gain of about 30% for non-overlapping updates, mainly because we perform a fine-grained locking on the database pages. The performance gain for overlapping updates is about 22%. We notice that the performance does not really double for non-overlapping updates. This may be due to an inherent amount of work done before actually writing to the database file.

3.3 Conclusion of Enhancement using Byte-Range Locking

With byte-range locks obtained across pages of the database, we see a benefit in single-threaded and multi-threaded benchmarks. We see a consistent improvement for SQLite inserts. For SQLite updates, the benefit is not as great as inserts. The experiments were repeated as many times to obtain a consistent value. The graphs also show the standard deviation of the range of results.

4. ENHANCEMENT TO WRITE AHEAD LOGGING

In this section, an overview of write ahead log (WAL) is presented, followed by its implementation in SQLite and the details on the enhancements to write ahead log, including the recovery mechanisms in case of power failures.

4.1 Overview of Write Ahead Logging

Database systems are designed to support atomic transactions, have durability, consistency and isolation [28][29]. Write ahead logging is a mechanism which provides atomicity and durability for the database. One requirement for write ahead logs is that a version of a page present in the log must not be overwritten until that version of the page has been committed to a nonvolatile storage media [30]. Usually, the logs keep growing until a point where the pages in the log must be committed to the stable storage. A growing log-file ensures that all data required to restore the state of a transaction are available in case of a power failure. Failures can also occur in the system, storage media or the process itself. The log must ensure durability in all these cases.

With the current implementation of WAL as described in section 4.2, the log file is locked for each append. There is a potential for multiple threads to append to the log file if each of the threads know in advance the offset at which it has to write the data. Such an approach is discussed in section 4.3. Moreover, we notice that checkpoint frequency can further be reduced from the current implementation to enhance the performance. This approach is also discussed in section 4.3 along with a comparison of both the approaches.

4.2 WAL Implementation in SQLite [3]

SQLite uses two kinds of mechanisms to ensure durability of database – one is a rollback journal and the other is write ahead logging. The Write Ahead Log is a file in the same directory as the database file. The file contains a header of size 32 bytes stored in big endian format. It is organized as a sequence of frames. Each frame consists of a header and the database page it represents. The contents of the WAL header [31] is shown in Table 3.

WAL Header (32 bytes)
Byte offset 0: Magic number. 0x377f0682 or 0x377f0683
4: File format version. Currently 3007000
8: Database page size. Example: 512
12: Checkpoint sequence number
16: Salt-1, random integer incremented with each checkpoint
20: Salt-2, a different random integer changing with each checkpoint
24: Checksum-1 (first part of checksum for first 24 bytes of header).
28: Checksum-2 (second part of checksum for first 24 bytes of header)

Table 3: Header of the Write Ahead Log

The contents of the frame header [31] is shown in Table 4.

Frame header (24 bytes)
Byte offset 0 : Page number.
4: For commit records, the size of the database measured in pages after the commit. For all other records this field is zero.
8: Salt-1 (same as header)
12: Salt-2 (same as header)
16: Checksum-1
20: Checksum-2

Table 4: Frame Header of the Log

All changes to the database are recorded by writing frames into the WAL. Transactions commit when a frame is written that contains a commit marker. A single WAL can record multiple transactions. Periodically, the content of the WAL is transferred back into the database file in an operation called a "checkpoint". A single WAL file can be used multiple times. In other words, the WAL can fill up with frames and then be check-pointed and then new frames can overwrite the old ones. A WAL always grows from beginning toward the end. Check-sums and counters attached to each frame are used to determine which frames within the WAL are valid and which are leftovers from prior checkpoints.

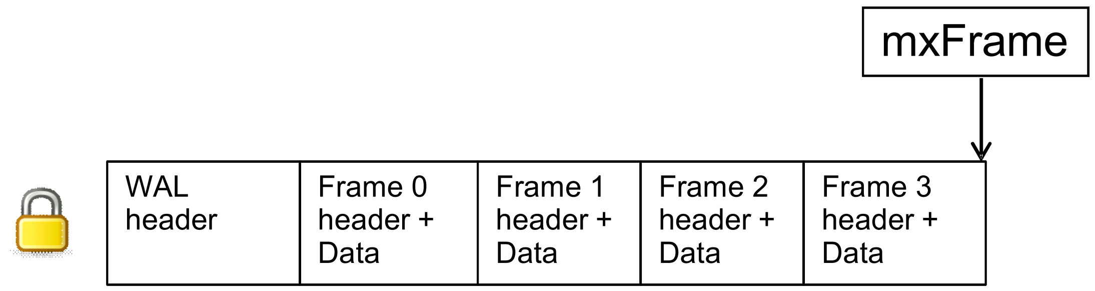


Figure 15: Write Ahead Log : Locking Mechanism and Appending Mechanism

In order to read a page from the database file, the reader algorithm first checks the log to see if it contains the page. If the log contains the requested page, the last valid page with the same page number before a commit frame will be returned. Otherwise, the requested page is read from the database file.

The Salt values and the check-sums are used to confirm the validity of a frame. The write ahead log is always written towards the end indicated by the 'mxFrame' value, as depicted in Figure 15. 'mxFrame' is the total number of frames in the WAL. The WAL is managed by using an in-memory data structure wal-index. It is reconstructed each time a system resumes after a failure, by reading each frame header from the log and gathering information on the valid frames that are present in the log. The wal-index contains all relevant information about the WAL, including the mxFrame. The wal-index can also be used to find the last occurrence of a page in the WAL.

The access to the write ahead log is limited to a single thread due to file level locking. It is however possible that multiple threads could append to the log provided they know the exact index to which data has to be appended. In section 4.3, the process of adding such a mechanism to the write-ahead log to support append by different threads is discussed.

In the trace collected for DB bench, it is noticed that the library checkpoints frequently, even when the write ahead log frames have not been completely filled. Limiting this checkpoint frequency can also have a significant impact on performance. Such a mechanism is also discussed in enhancements to write ahead log - section 4.3.

4.3 Enhancements to Write Ahead Log

As discussed in section 4.2, write-ahead logs only need to be appended at every write. In the enhanced implementation, a sequencer [32][33] is used by the write method of write-ahead log in order to allow multiple threads to write to the WAL. The function of the sequencer is to maintain the correct value of the end of the log. Any thread that wants to write to the WAL gets the last position in the log file from the sequencer. Any change to the sequencer happens under a lock so that multiple threads don't get the same end value. Once the position to write to the log has been obtained, the sequencer lock can be released and the thread can write to the log. The action of the sequencer is depicted in Figure 16. As long as every thread goes through the sequencer, multiple writes can proceed to the log simultaneously.

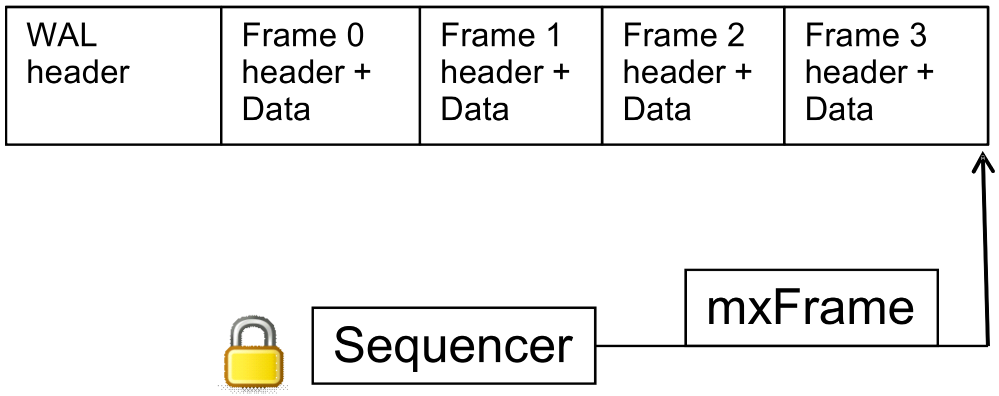


Figure 16: Write Ahead Log : Locking Mechanism with Sequencer

The method of writing to log using the index provided by the sequencer may introduce holes in the log in case of a failure. When power is restored, the library checks if a valid write ahead log is present. The log is read and the database is restored until the valid commit operations. The validity of a frame can be verified using the salt values and comparing them with the header of the WAL. If, while replaying the log, an invalid frame is encountered, that transaction is not committed to the database. The values that follow this frame are also ignored. This is performed to maintain the integrity of transactions.

Another improvement to the WAL is delaying checkpoints from applications. Check-pointing is the action of copying out the frames from the log file to the database file and truncating the log. By default, the WAL is check-pointed periodically, mostly when the log is full. The size of the log is 1000 frames by default, but it may be changed by using the configuration methods provided by SQLite. The application may also request to checkpoint the log at any time. By delaying such checkpoints, the latency can be reduced at the cost of relaxing the durability constraints. The durability is however not sacrificed by delaying checkpoints.

At the start of writing a set of frames to the log, the method checks if the log can be restarted, that is, started from index 0. Since we append to the log each time and delay checkpoint until the log is full, this check can be omitted.

Finally, at the end of each checkpoint, the sequencer is set to 0 so that the log can restart.

In the next section, the evaluation of changes to the write-ahead logging is discussed.

5. EVALUATION OF CHANGES TO WRITE AHEAD LOGGING

To find the performance impact of changes to write ahead log, the application must request the library to use WAL for the existing connection. The default journaling mode is rollback journal. There are two changes made to the library – a) delaying the checkpoint and b) introducing a sequencer to append to the log. We find the impact of enabling either a, b or both. The performance is measured using five benchmarks.

- DB Bench
- Comparison between SSD and HDD
- Maximum performance gain analysis - DB Bench
- Sandisk and Intel SSD comparison
- Comparison with Berkeley DB

5.1 DB Bench Benchmark

The DB Bench which was used for the evaluation of byte-range locks is used here too. However, it is slightly modified to perform only insert or update operations. For update operations, the records are first inserted to the database and then updated. It is seen that having both a and b is beneficial for two threads and does not cause decrease in performance in other cases. An Intel SSD is used for storage. Figure 17 shows the time taken for 10K inserts and Figure 18 shows the time taken for only 25K updates. While we can notice an equal improvement for (a) and (b), the reason behind this is not clear.

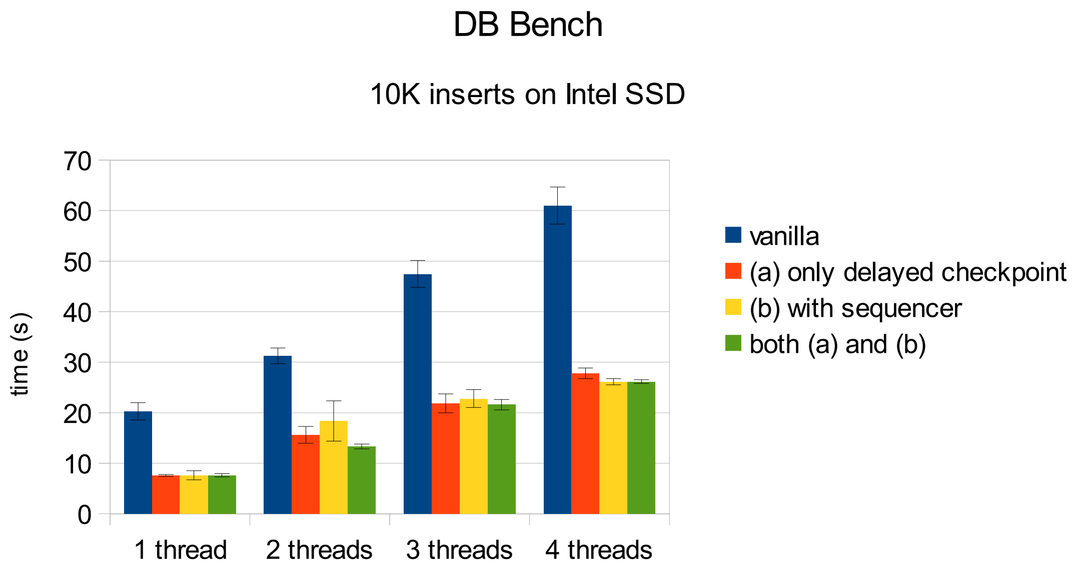


Figure 17: DB Bench - 10K Inserts on Intel SSD

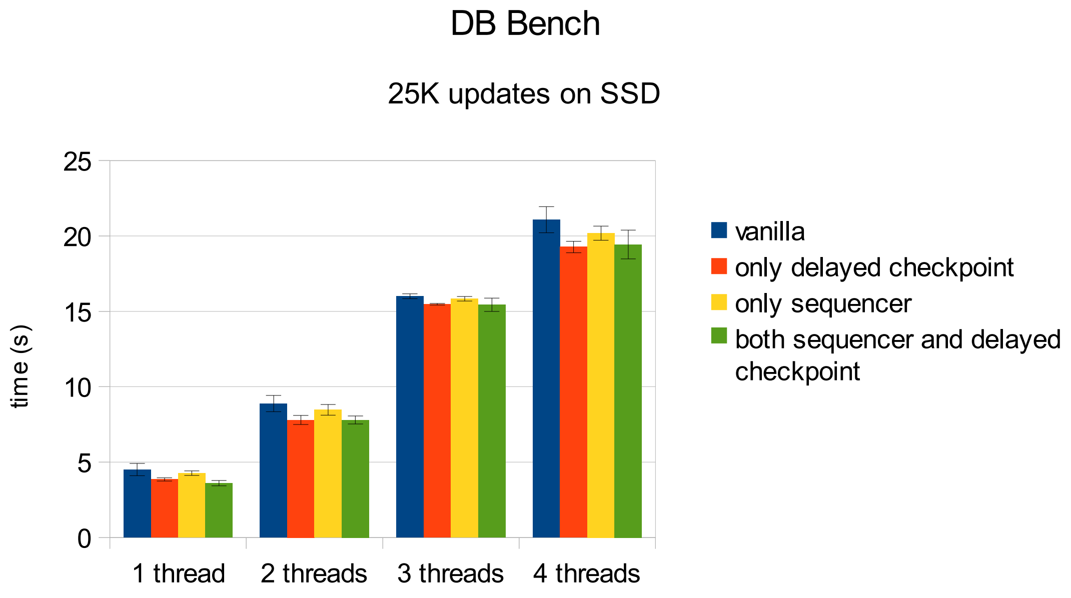


Figure 18: DB Bench - 25K Updates on Intel SSD

5.2 Comparison between SSD and HDD

Next, the enhancements are compared on Solid State Drive and Hard Disk Drive (Figure 19 to Figure 22). For the vanilla code, the response times almost go in parallel. There is not much deviation in the the slope of these two curves. But, for delayed checkpoint and using a sequencer, the hard disk time increases rapidly with increase in the number of threads. This is due to the conversion of sequential writes to random writes using a sequencer. The increase is not so much in case of SSD since SSDs don't incur head seeks for random writes. With both delayed checkpoint and the sequencer, we see a similar behavior.

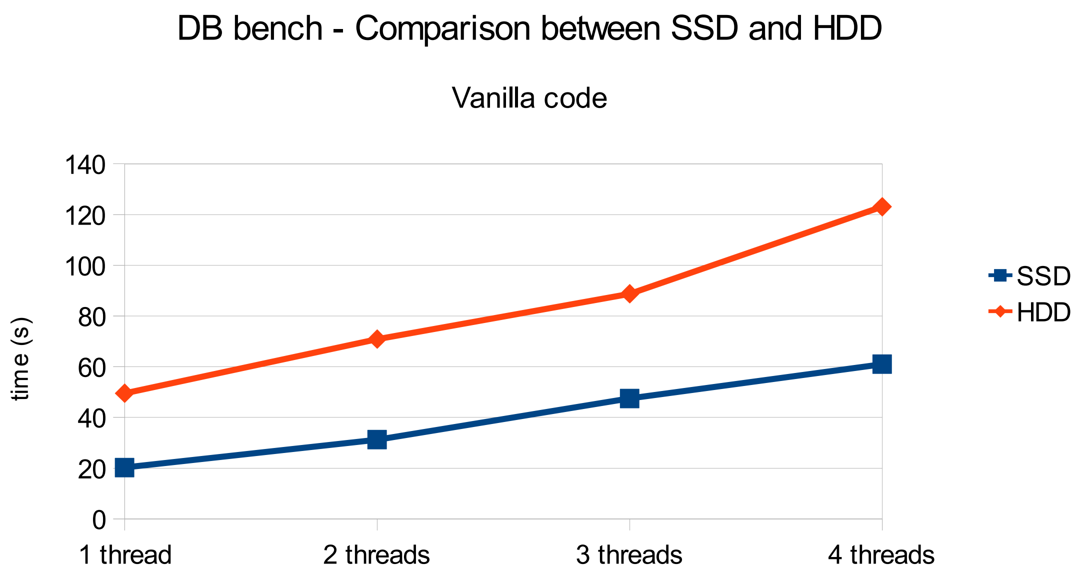


Figure 19: DB Bench : Comparison of SSD and HDD - Vanilla Code

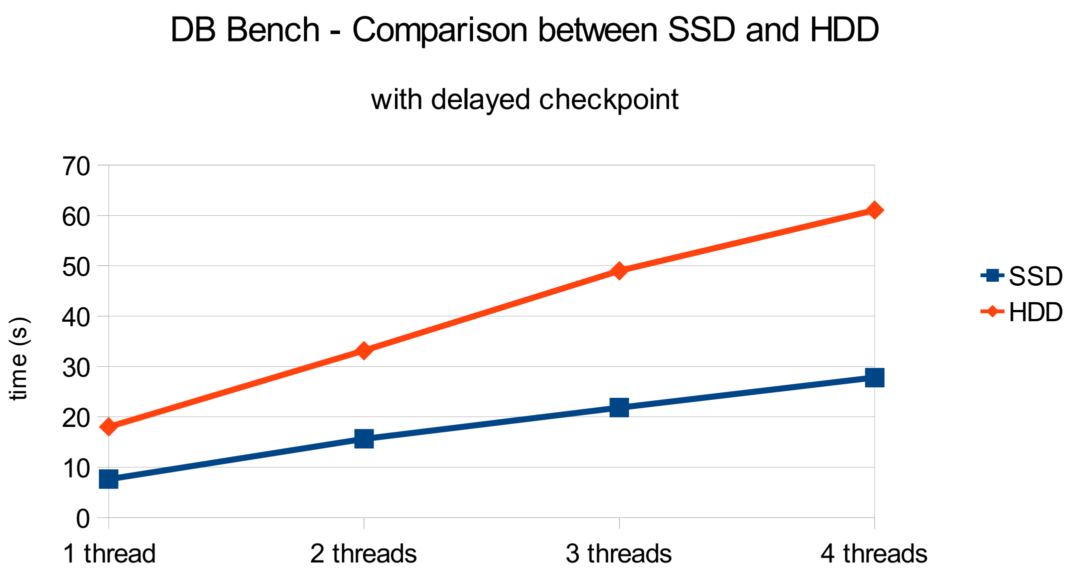


Figure 20: DB Bench : Comparison of SSD and HDD - Delayed Checkpoint

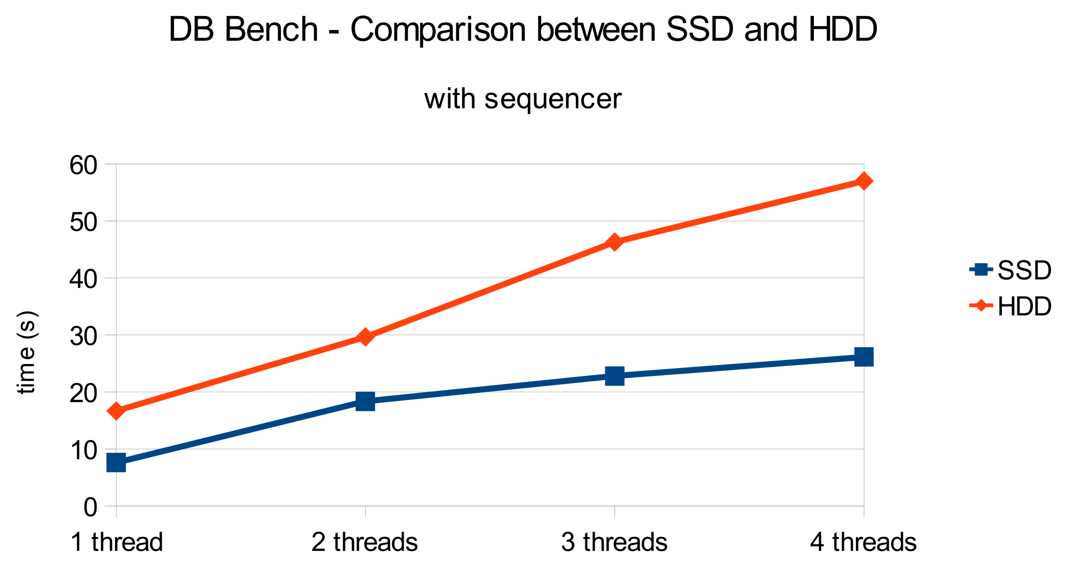


Figure 21: DB Bench : Comparison of SSD and HDD - Sequencer

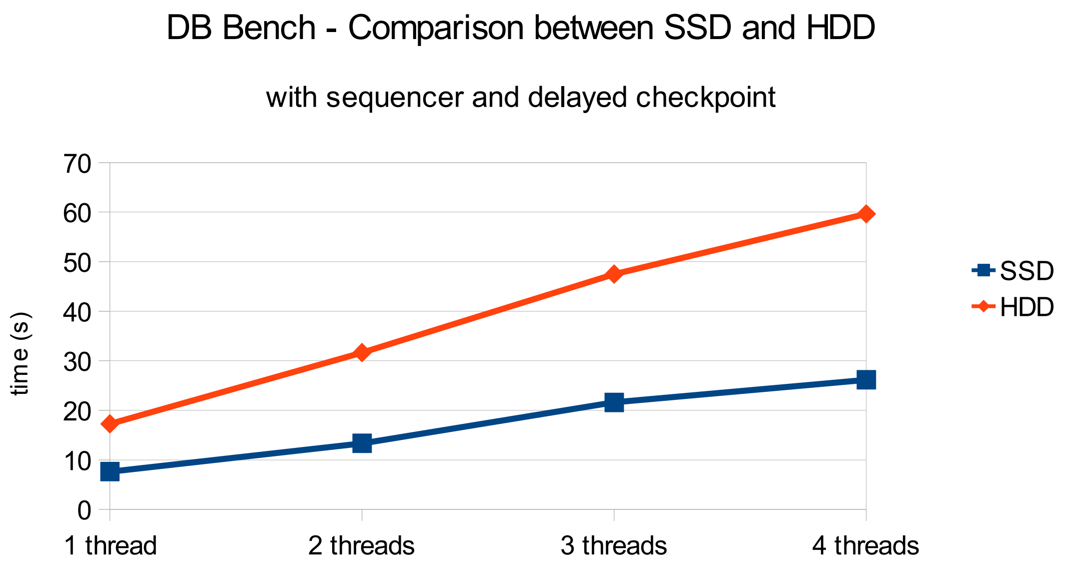


Figure 22: DB Bench : Comparison of SSD and HDD - Both Enhancements

5.3 Maximum Performance Gain Analysis - DB Bench

The byte-range locking from phase one is combined with enhancements to WAL and the performance is compared using DB Bench (Figure 23). With both enhancements, the performance can get a lot better as seen from Figure 23. However, it should be noted that this is only a measure of potential performance that can be reached. The locking primitives were turned off for write ahead log and the byte-range lock was also added. This greatly reduces the overhead of locking. This only goes to show that there is scope for more improvement in performance by minimizing the locking primitives and that they do significantly add a overhead by heavily synchronizing database operations.

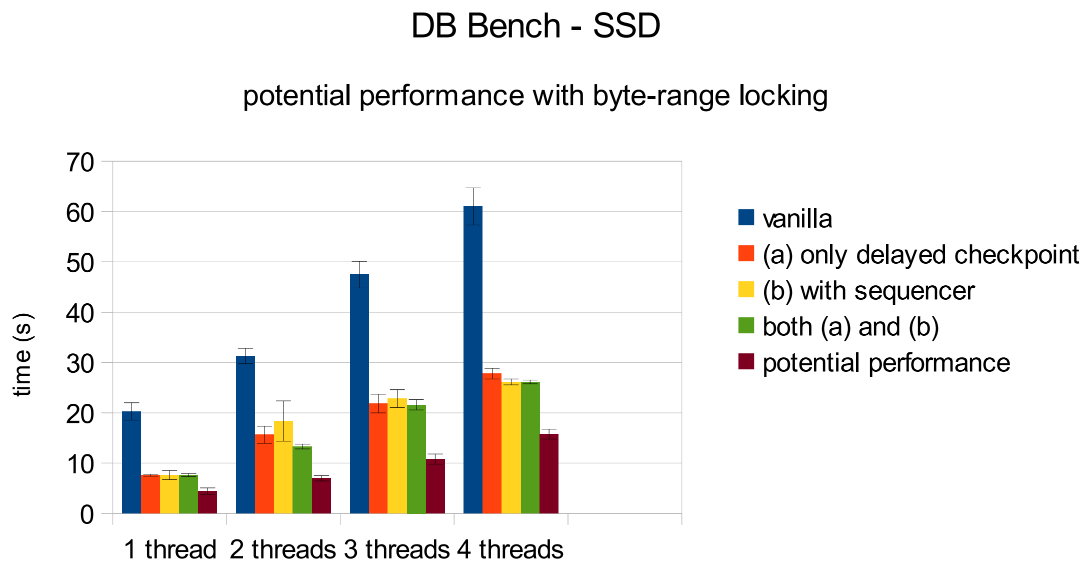


Figure 23: DB Bench : Performance Comparison for Various Enhancements

5.4 Sandisk and Intel SSD – A Comparison

Next, we compare the response times obtained from Intel SSDs to Sandisk Flash drives (Figure 24 to Figure 27). Both show similar performance in most of the cases. There is considerable performance improvement over vanilla code for both flash drives. Since both are flash drives, it can be inferred that the performance variation is not as visible as a HDD and a SSD.

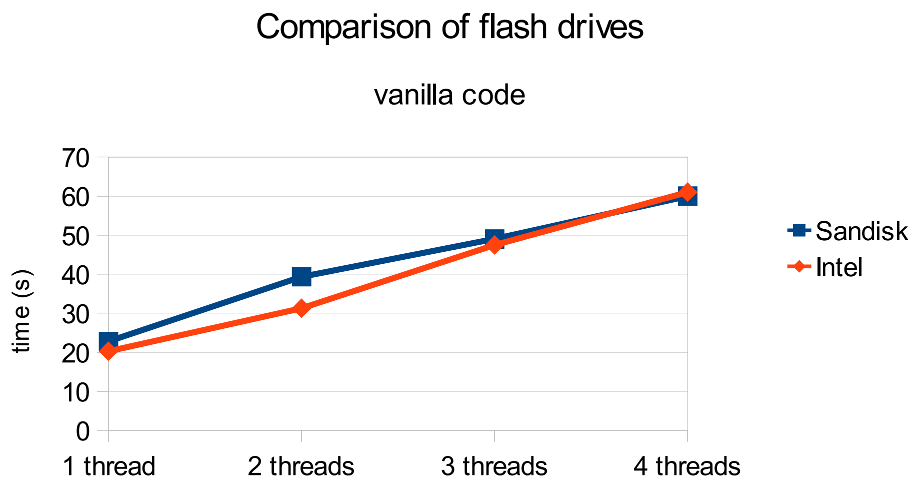


Figure 24: DB Bench : Comparison of Flash Drives - Vanilla Code

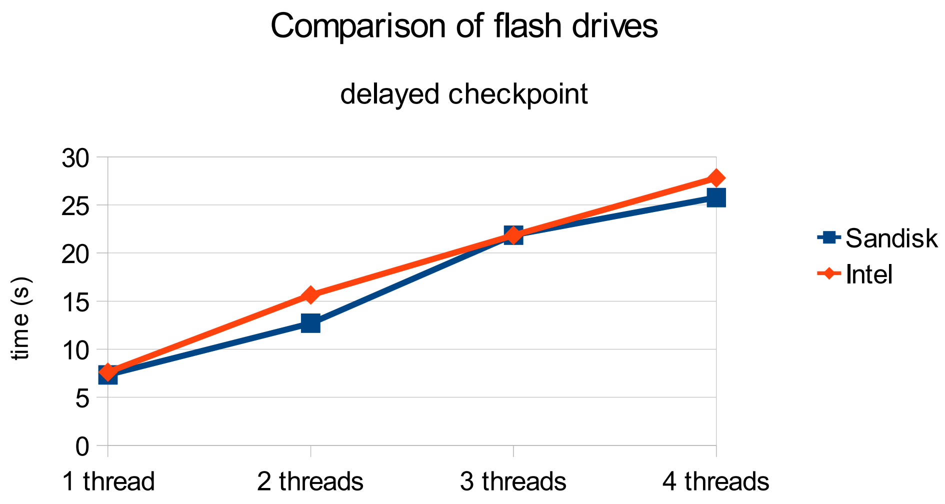


Figure 25: DB Bench : Comparison of Flash Drives - Delayed Checkpoint

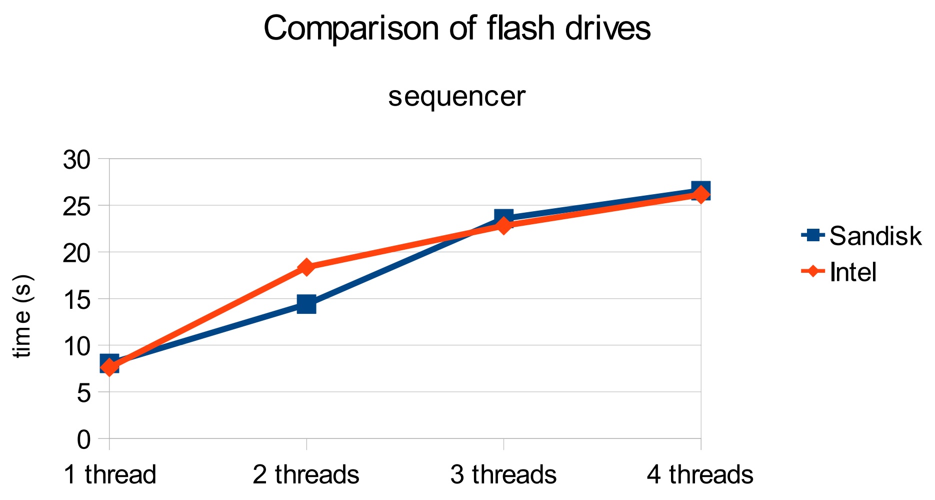


Figure 26: DB Bench : Comparison of Flash Drives - Sequencer

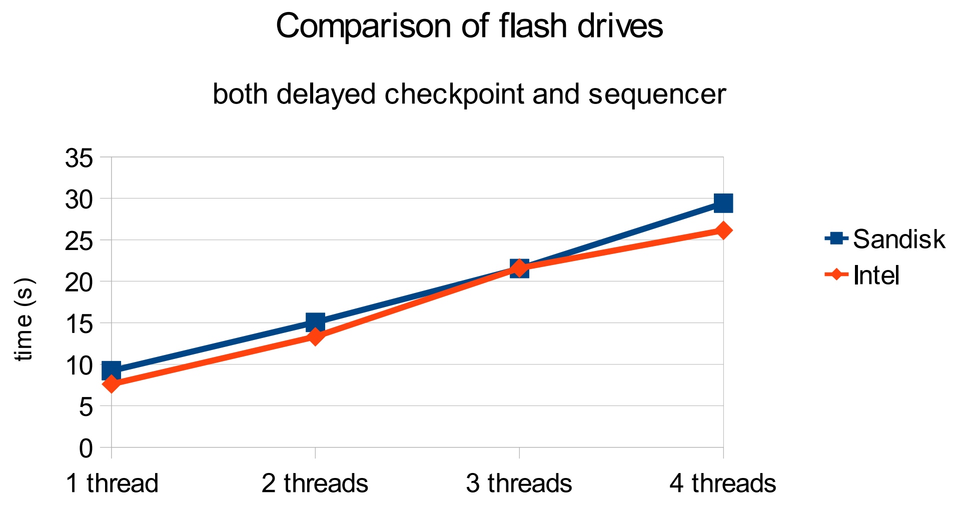


Figure 27: DB Bench : Comparison of Flash Drives - Both Enhancements

5.5 Comparison with Berkeley DB

Finally, we compare the SQLite database with Berkeley DB [34]. Berkeley DB supports many features like fine grained locking and improved concurrency [35]. But, it is seen that Berkeley DB performs well for inserts and SQLite performs well for all other cases (Figure 28). It could be that Berkeley DB is not optimized for a lot of operations that are performed. The final bar in Figure 28 gives the total time for the entire RL bench benchmark. It is significantly less for SQLite than it is for Berkeley DB.

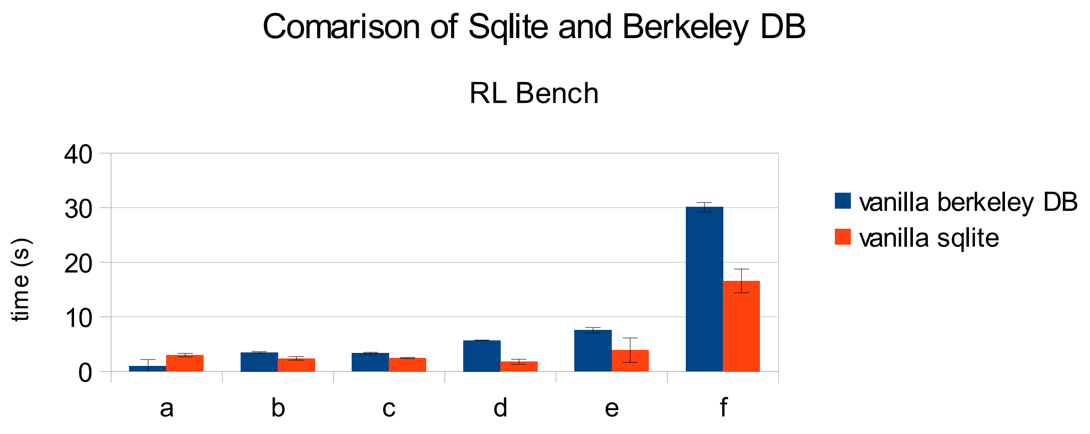


Figure 28: Comparison of SQLite and Berkeley DB

x-axis : (a) 1 K inserts (b) 25 K inserts in a transaction

(c) 25 K inserts into an indexed table in a transaction

(d) 1K updates without index (e) 25 K updates with an index (f) Total time

6. CONCLUSION AND FUTURE WORK

The thesis covered two enhancements to the SQLite library, used by mobile systems to store structured data. The SQLite library was modified to provide light-weight locking mechanisms. Such a locking mechanism can improve performance and reduce some locking overhead of writes to the database. This was verified from the various benchmarks that were run. A byte-range locking mechanism was introduced to enable better concurrent access to the database file. The effect of adding byte-range locks is measured from various multi-threaded benchmarks.

The second enhancement is done to the write ahead log. A sequencing mechanism is added to enable concurrent access to the log. The performance is measured using multi-threaded benchmarks. We note that performance can be improved by reducing the checkpoint frequency to the minimum value. The performance of using a sequencer and reducing checkpoint frequency was compared across various benchmarks as well as using different types of storage devices. It is inferred that flash drives can see more improvements from enhancements to write ahead log.

While setting up benchmarks for write ahead log, it was noticed that many applications do not enable the write ahead log. Since WAL is not enabled by default, such applications may see a lower performance.

The results obtained from changes to write ahead log indicate that there is scope for more improvement in this area in terms of minimizing the locking primitives. Mobile systems use flash memory for data storage. As a result, we can see significant gain in performance if the entire software stack is designed for flash memory. Most storage subsystems were designed for disk-based systems and are reused for flash based systems. By changing the storage stack, significant improvements can be noticed.

It has already been discussed that employing a Phase Change Memory (PCM) can improve performance. Even if the speed of hardware is increased, we need to employ the right software designed for these systems to see large gains. An implementation of a file system for Storage Class Memory [36][37] opens the possibility of more performance improvements by using faster memory as well as a file system designed for non-volatile media like PCM. Moreover, if a database system needs to be used for such systems, a database for PCM [38] can be used to design a well balanced system.

REFERENCES

[1] D. Richard Hipp, Accessed in Feb 2013, “About SQLite section”, SQLite : www.sqlite.org/about.html

[2] Hyojun Kim, Nitin Agrawal and Cristian Ungureanu, “Revisiting Storage for Smartphones”, in *Proceedings of the 10th Conference on File and Storage Technologies (FAST '12)*, Feb 2012

[3] SQLite Development Community, Available from Jul 2010, “Write Ahead Logging section”, http://www.sqlite.org/wal.html

[4] Linus Torvalds, Accessed in Nov 2012, “Download and Wiki Page of Kernel version 3.0”, http://kernel.org/

[5] Wiki Contribution, Available from Jun 2011, “Android Architecture Diagram”, http://elinux.org/Android_Architecture

[6] Android Developer APIs Web Contributors, Accessed in Feb 2013, “SQLite Package”,http://developer.android.com/reference/android/database/sqlite/package -summary.html

[7] SQLite Web Community, Accessed in Apr 2013, “Overview Documents section”, http://www.sqlite.org/docs.html

[8] D. Richard Hipp, Accessed in Apr 2013, “SQLite Architecture Diagram”, http://www.sqlite.org/arch.html

[9] SQLite Web Community, Accessed in Nov 2012, “SQLite Locking”, http://www.sqlite.org/lockingv3.html

[10] SQLite Web Community, Accessed in Nov 2012, “Shared Cache Mode section”, http://www.sqlite.org/sharedcache.html

[11] Sibsankar Haldar, “Inside SQLite”*, O'Reilly Media, Inc.,* 2007, pp 26-36

[12] SQLite Web Community, Available from Feb 2003, “Virtual DataBase Engine for SQLite as used in version 2.8.0”, http://www.sqlite.org/vdbe.html

[13] The Open Group Base Specification, Accessed in Jan 2013, “fcntl() description”, http://pubs.opengroup.org/onlinepubs/009695399/functions/fcntl.html

[14] W. Richard Stevens, “Advanced Programming in UNIX Environment”, Addison-Wesley Professional Publishing, 2nd edition, section 14.3, 2005

[15] Chih-Wei Huang, Available from Jul 2012, “Android x86 4.0 Download section”, http://www.android-x86.org/getsourcecode

[16] AOSP Developer Community, Accessed in Sep 2012, Building Customized Kernel for Android-x86, http://www.android-x86.org/documents/customizekernel

[17] Oracle Virtual Box Developer Community, Accessed in Sep 2012, “i386 version for Ubuntu 10.04”, https://www.virtualbox.org/wiki/Linux_Downloads

[18] Denys Vlasenko, Accessed in Apr 2013, “Busybox Utility FAQ and About sections”, http://www.busybox.net/about.html

[19] Sergei Viznyuk, Accessed in Jan 2013, “DHCP Command Reference and Description”, http://www.phystech.com/download/dhcpcd_man.html

[20] Wiki Contribution, Available from May 2011, “Android Networking: Setting the DNS Server”, http://elinux.org/Android_Networking

[21] Wiki Contribution, Accessed in Feb 2012, “Secure Copy Program Usage”, http://en.wikipedia.org/wiki/Secure_copy

[22] Open Source File Format Wiki Contribution, Accessed in Apr 2013, “APK File Format”, http://en.wikipedia.org/wiki/APK_(file_format)

[23] Android Developer APIs Web Contributors, Accessed in Feb 2013, “Android Debug Bridge (ADB)”, http://developer.android.com/tools/help/adb.html

[24] Computer Systems Lab at Sungkyunkwan University, Available from May 2011, “Androbench Benchmark”, http://www.androbench.org/wiki/AndroBench

[25] Open Source App on Google Play, Accessed in Oct 2012, “RL Benchmark”, https://play.google.com/store/apps/details?id=com.redlicense.benchmark.sqlite

[26] Gerhard Smith, Accessed in Mar 2013, “Firefox for Android-x86”, http://www.android-x86.org/download

[27] McObject Developers, Accessed in Jan 2013, “TestIndex Benchmark for Android”, http://www.mcobject.com/index.cfm?fuseaction=download&pageid=581&sectionid=133

[28] Bruce G. Lindsay, “The Transaction Processing Revolution ”*,* in *SIGMOD*, Vol 37 No. 2, June 2008, pp 38-39

[29] Brahim Medjahed, Mourad Ouzzani, Ahmed K. Elmagarmid, *“Generalization of ACID Properties ”,* in *Encyclopedia of Database Systems,* 2009, pp 1221-1222

[30] C. Mohan, Don Haderle, Bruce Lindsay et. al., “ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks

Using Write-Ahead Logging”, in *ACM Transactions on Database Systems*, Vol 17, No. 1, March 1992, pp 94-97

[31] D. Richard Hipp, Accessed in November 2012, “WAL Header Format ; sqlite.c Source Code Comments section”

[32] Dahlia Malkhi et. al., “From Paxos to CORFU: A Flash-Speed Shared Log”, in *ACM SIGOPS Operating Systems Review,* Volume 46 Issue 1, 2012, pp 47-51

[33] Mahesh Balakrishnan et. al., “CORFU: a shared log design for flash clusters”, in *NSDI'12: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation,* 2012

[34] Michael A. Olson, Keith Bostic, and Margo Seltzer, “Berkeley DB”, in *USENIX Annual Technical Conference*, 1999

[35] Oracle Berkeley DB, Accessed in Mar 2013, “Berkeley DB”, http://www.oracle.com/technetwork/database/berkeleydb/db-faq-095848.html

[36] X. Wu and A.L.N. Reddy, "SCMFS: A File System for Storage Class Memory", in *Supercomputing Conference 2011*: *Proc. of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis,* 2011

[37] Qian Cao, “SCMFS Performance Enhancement and Implementation on Mobile Platform”, *Texas A&M Thesis Repositories,* August 2012

[38] John Coburn, Trevor Bunker, Rajesh K. Gupta, Steven Swanson, “From ARIES to MARS: Reengineering Transaction Management for Next-Generation, Solid-State Drives”, in *UCSD CSE Technical Report CS2012-0981*, 2012