Page Replacement Mechanism for Small Foot-Print Database in Android Devices

Dissertation Report

Submitted in partial fulfillment of the requirements for the degree of

Master of Technology

by

Pratik Patodi
Roll No: 10305917

under the guidance of

Prof. Deepak B. Phatak

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai
Abstract

Android devices are becoming popular due to their lower cost and increased integration with Google services. Open Source Android-SDK encourages the development of variety of applications for mobile devices. Mobile devices use flash drives as their memory resource. Considerable amount of data needs to be stored and organized for these applications. Flash drives have certain limitations for writing data. Database logging adds a major bottleneck against the fast response time of update transactions, especially for large update transaction, since a large amount of log should be flushed during commit. In this report, we propose a new page replacement technique and compares it with existing approach used by android’s native database, SQLite. An overview of adaptive logging approach is also provided in the report which can be further used as an enhancement.
Acknowledgements

I would like to express my deepest gratitude to my guide Prof. D. B. Phatak, for his patience and guidance throughout the project. I would also like to thank Mr. Nagesh Karmali for his continuous inputs in my work. I would also like to thank every one else who supported me in this work.
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Fields of Table tbl1</td>
<td>10</td>
</tr>
<tr>
<td>3.2</td>
<td>Opcode program for <strong>INSERT</strong> statement</td>
<td>11</td>
</tr>
<tr>
<td>3.3</td>
<td>Opcode program for <strong>UPDATE</strong> statement</td>
<td>17</td>
</tr>
<tr>
<td>6.1</td>
<td>Time Taken and DB size for Small delete (1 Page)</td>
<td>31</td>
</tr>
<tr>
<td>6.2</td>
<td>Time Taken and DB size for Small delete (3 Page)</td>
<td>31</td>
</tr>
<tr>
<td>6.3</td>
<td>Time Taken and DB size for Small update (1 Page)</td>
<td>32</td>
</tr>
<tr>
<td>6.4</td>
<td>Time Taken and DB size for Mass delete (Approx 77 Pages)</td>
<td>32</td>
</tr>
<tr>
<td>6.5</td>
<td>Time Taken and DB size for Mass update (Approx 233 Pages)</td>
<td>32</td>
</tr>
<tr>
<td>6.6</td>
<td>Time Taken and DB size for Mass update (Approx 233 Pages)</td>
<td>33</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Android Architecture\[1\] ................................................. 6

3.1 Block Diagram of SQLite Architecture\[2\] .......................... 9

3.2 SQLite file structure and header first 32 bytes .................. 12

3.3 SQLite b-tree structure .................................................. 13

3.4 SQLite page structure\[9\] ............................................... 15

3.5 Internal Page Header Contents ...................................... 16

4.1 Roolback Journal of SQLite\[6\] ........................................ 20

5.1 Working of the Flash Memory Page Manager ..................... 24

5.2 Flash Memory DBMS Model ............................................ 25

5.3 Functional Flow of our Approach ................................... 27

6.1 Schema of the Database used for Experiment ..................... 29

6.2 ER Diagram the Database used for Experiment ................. 30

6.3 Comparison\[10\] .......................................................... 34

7.1 Two Update patterns\[10\] ................................................. 36

7.2 Comparison between Berkley DB, Perst DB and SQLite ........ 39
Chapter 1

Introduction

Android has been developed by Google for different mobile devices like smartphones, tablets, etc. It is a Open Source Operating System. According to the research from IDC\[15\], after android’s launch in 2007, it now owns around 68% of smartphone market globally. Developers can now develop different applications on a propriety basis or free in Android by using Android SDK. Android provides a lot of services and device features that enable the developers to use public API’s without having any kind of knowledge related to Andriod internals. Also, we now have ‘Google Play’ where any developer can distribute his applications to Android users.

Most of the android applications require to store and update considerable amount of data. Android enabled devices uses flash drives as a storage media. Flash drives being expensive, puts an inevitable constraint on the internal memory of android enabled low cost devices such as mobile phones or tablet-PCs. This constraint directly limits the number of applications that can reside in the internal memory. To increase the number of applications, user needs to use external storage e.g. SD cards. Applications stored in the external storage introduce many performance overheads which may increase the response time of the application. Android uses SQLite as its native database engine.
1.1 Motivation

Response time is an important parameter that reflects the interactivity of the application. It can be decreased by the efficient use of the available memory. It is identified that, database engine plays a significant role in memory management. Database engine is responsible for allocation and deallocation of memory, for an application.

In SQLite, data is stored in the form of database files which are structured as B-tree. There are predominantly three types of operations i.e. insert, update, and delete, which may affect the performance of the device. SQLite uses journal mechanism to achieve these operations, which can be termed as log based mechanism. The maintenance of journal file degrades the system performance, as before making any changes in database, the original contents must be added in the journal file. Since this journal file is created during transactions execution, it delays transaction, which results in decrease in responsiveness of the android devices. In addition to this, there are certain limitations of flash drives:

- Flash memory offers random-access read and programming operations, but does not offer arbitrary random-access write or erase operations. It only allows to write or erase operations in fixed size block.

- The rewrite operation in flash drives follows erase-then-write process known as program/erase cycle. New data is written to the block, by first erasing old data from blocks. Therefore, the rewrite operation effectively involves to write operations.

The proposed mechanism uses a variation of shadow paging, which perform update-out-of-place technique. It performs update operations by writing the updated data in a new free page, leaving the original content ‘as it is’. By doing so, the mechanism removes the requirement of journal file and avoids erase-then-write process. Thereby, making the proposed mechanism suitable for android devices. In this way, proposed mechanism may ensures, decreased response time of android devices.
1.2 Overview

Chapter 2 gives an overview of the architecture of android system and talks about its native database SQLite. It also highlights difference between SQLite and Mysql and how it is used in android applications. Chapter 3 explains architecture, query execution plan, and the file structure of SQLite database. Chapter 4 talks about the page replacement algorithms of SQLite. Chapter 5 shows the model of the proposed mechanism, and describe how it is implemented. Chapter 6 depicts the performance of our mechanism under various conditions. Chapter 7 discusses about the further enhancement and SQLite’s alternate, and benefits of using object oriented embedded database instead of using SQLite. Finally, Chapter 8 concludes the report with conclusion and future work.
Chapter 2

Android

2.1 The Android Architecture

Android is a Google introduced Open Source software for mobile devices. It is capable of executing multiple application programs. Android is not only an Operating System, but it also includes applications important for mobile devices like messenger, call register, contacts, calendar, etc. Android is a Linux kernel 2.6 based complete Operating System which is bundled with the following:

- Large library set
- Rich multimedia User
- Powerful OS and
- Phone Application.

Android platform helps the developers to make new innovative mobile applications. It helps them to make maximum use of different functions related to mobile internet. The Software Development Kit (SDK) in android contains necessary API’s and tools required for developing applications in android platform and uses Java programming language for this purpose.

Different standard libraries are required to develop applications on Linux. Some of the necessary libraries have been rewritten by Andriod. For Example, a new library ‘bionic’
has replaced the existing ‘libc’ library [14].

Each application in Android system creates its own process. Android architecture consists of 4 layers as shown in the Figure 2.1 [11].

**Linux Kernel**: 2.6 version of linux provides some core system services to Android. These services includes:

- Process management
- Security
- Network stack
- Memory management and
- Driver model

The kernel behaves as a layer that separates hardware from software stack.

**Libraries**: Android system has various components that use set of C/C++ libraries. Android application framework helps the developers to explore the functionalities provided by these libraries.

**Android Runtime**: Core libraries of Andriod system contains various functionalities that are present in the core libraries of Java programming language. With the help of Davik we can run multiple Virtual Machines efficiently. In order to minimise the amount of memory used by the process, Dalvik Virtual Machine executes the files which are in ‘.dex’ format.

**Application Framework**: Android has an open development platform, which gives developers the competence to build innovative applications. Developers can take lots of advantages like:

- Access to local information.
- Running background services
- Adding notification to status bar
- Access device hardware and so on.
Applications- Andriod comes with lot of core applications that are written in Java. Some of these applications include maps, SMS program, contacts, calendar, Email client, browser, etc.

2.2 SQLite Database

Android provides several ways to store user and application data like XLM, file system, SQLite database, etc. SQLite is a very lightweight database engine which comes with Android OS. Android uses Java Native Interface (JNI) framework for using it, as it is written in C language. JNI provides a platform for an application, to communicate with SQLite or any other libraries.
SQLite can be considered as a lighter form of MySQL. Many of the functionalities of MySQL are not present in SQLite as mentioned below.\footnote{5}

- **User management:** SQLite does not have the user management capability. Hence for security it uses encryption.

- **Database Capabilities:** SQLite does not have merge join, and right outer join.

- **Alter command:** SQLite does not fully support ALTER TABLE statements. You can only rename table, or add columns. If you want to drop a column, your best option is to create a new table without the column, and to drop the old table in order to rename the new one.

- **Data type:** Boolean, binary, data time, enum, and set, datatypes are not supported by SQLite. For string it has only text database type, char and varchar are also not supported.

Absence of these functionalities makes it lighter than MySQL. These functionalities are mainly used for large database, but not here in case of Android, since we have memory, power, and processor limitation.

An application creates and manages its private database using two Android API, `android.database` and `android.database.SQLite`\footnote{7}. Content Providers, which resides on the application framework, are used when you want to share your data across applications. If you have a database attached with an application and you want another application to use some data, you can implement a content provider that exposes the data.
Chapter 3

SQLite Internal

The chapter describes the working of SQLite. It includes the architecture of SQLite library, its query execution plan, how the instruction are executed, etc. It also describes the structure of the database file which SQLite maintains.

3.1 SQLite Architecture

The block diagram of SQLite architecture is shown Figure 6.2. It also gives a sense of how they are interrelated.

**Interface:** The SQLite library starts interacting with the user through the interface module. It generate the command line interface for the user. User enters the SQL command using this module. It then passes the input to SQL command processor.

**SQL Command Processor:** It acts as a compiler in the SQLite library. Tokenizer break down the string containing SQL statements into various tokens, and passes these tokens to the parser. Parser check fro the semantic and syntax errors. **Code generator** is then called to generate virtual machine code. Virtual machine executes the generated code.

**Data storage:** Data structure that is used to store SQLite database, is B-tree. Each table corresponds to a separate B-tree structure.

**Paging the database:** Information needed by B-tree has to be in fixed size pieces or chunks (by default each consist of 1024 bytes). These chunks of information is read,
write and cached by page cache. Page modification is done by the interaction of page cache and B-tree driver. The page to be modified, is requested by the B-tree driver from the page cache. It also notifies the page cache about the page modification, rollback or commit changes.

**Operating System Interface:** SQLite uses a layer called abstraction layer to communicate with operating system. This communication is achieved by an interface which is defined in a file named ‘os.h’ (‘SQLite3_android.h’ in android)
3.2 Query Execution Plan

Virtual machine executes the code that is generated by Intermediate code generator. An opcode and upto five operands are present in each instruction of the program:

- \( P_1 \): It is a 32 bit signed integer.
- \( P_2 \): It is a 32 bit signed integer.
- \( P_3 \): It is a 32 bit signed integer.
- \( P_4 \): It may be 32 bit signed integer, 64 bit floating point value or a string literal.
- \( P_5 \): It is an unsigned character.

Virtual machine keeps on execution instruction from instruction number 0 until a ‘Halt’ instruction is not encountered. Execution may also halts if the value of program counter reaches beyond the address of the last instruction or in case of any other execution error.

For example, consider a table \( tbl1 \) which has two columns key and value, as shone in the table 3.1.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>integer</td>
</tr>
<tr>
<td>value</td>
<td>text</td>
</tr>
</tbody>
</table>

Table 3.1: Fields of Table \( tbl1 \)

For **INSERT** statement, \( INSERT \ INTO \ tbl1 \ VALUES(01, 'pratik!'); \) the program generated is shown in table 3.2.

Similarly for **UPDATE** statement, \( UPDATE \ tbl1 \ SET \ value = 'bye pratik' \ WHERE key = 01; \) the program generated is shown in table 3.3.

The Virtual Machine starts executing from instruction address zero. The top two instruction and the last four instruction can be termed as a part of prologue and epilogue. The instructions 2 to 10 form a loop, which scans the table for the **WHERE** clause. In
this loop, all the rowid of the tuples, which satisfy the condition are added in the list. Instructions 15 to 19 constructs the updated tuple in the stack and then instruction 20, insert the this tuple in the rowid read by the instruction 13 from the list.

Similar program is generated for DELETE statement. The only difference is, instruction 15 to 19 will not be present, and the instead of insert instruction of line 20, there would have been delete instruction.

### 3.3 SQLite File Structure

The data-centric applications on android devices uses SQLite as a native database engine. SQLite is used by large number of local application software and embedded devices owing to its small size of database engine. There has been a substantial increase in the use and availability of portables devices like smartphones. This has immensely contributed to
the use of SQLite. SQLite is an Open Source database software that is based in ANSI-C. SQLite stores the results of all database usage in a single file of an application. The database file generated, is private for that application.

![SQLite file structure and header first 32 bytes](image)

Figure 3.2: SQLite file structure and header first 32 bytes

### 3.3.1 SQLite database file

According to the author[9], "the overall structure of a SQLite file is provided in Fig 3.2, and the file signature is 0x53 51 4c 69 74 65 20 66 72 6d 61 74 20 33 00 (16 bytes)". We use signature of target file to identify SQLite file, since it does not have a particular extension for a filename.

Page appearing at the beginning of the file is known as header page. Header page consists of two parts:

- Upper part
- Lower part

The upper part contains header and signature of database file. Lower part is in the form of schema table and contains information of tables in the database. The next pages
contains b-tree which are mostly separated in form of table b-tree and index b-tree. Table b-tree stores data contents. Cell is the basic unit of store operation in SQLite. Here data stored and cell structure may differ to leaf and internal pages. The tree has all the internal pages in the middle section. Each cell of the page holds pointers containing address of lower pages. The most bottom part of the tree contains leaf pages. These lower pages consists of cell that carry database records.

Figure 3.3: SQLite b-tree structure
3.3.2 Schema Table

The information about index, trigger and table, of SQLite is stored in schema table which is present in header page. All the names of the tables are present in schema name section. Schema name section also contains root page numbers and string which is nothing but the SQL statement.

This SQL statement is passed to the SQLite while schema is being created. SQLite consists of four data types viz text, integer, numeric and blob. Data types in standard SQL query such as big int, varchar and date are modified to the most relevant data type in SQLite as determined by system.

3.3.3 Table B-Tree

Actual data is stored in table B-Tree. Figure 3.3 shows the structure table B-Tree. The leaf page contains actual data while internal page consists of the pointers that points to the child pages.

3.3.4 Page Structure

Both types of pages i.e. internal and leaf pages as depicted in Figure 3.4, resides and shares the structure. Location of the cells containing actual data is specified by the page header which is followed by a list of big endian integers at 2-byte offset. A space is created which is initially filled by zeros between a cell offset and a cell. This space is termed as free space. There is another type of space referred as free block in the leaf page, which is used to store a cell until further operations.

3.3.5 Internal Page Header

An internal page header shown in the Figure 3.5, contains 12-byte, the first byte of this 12 byte sequence is called as page flag and it has a value 0x05. The first free block offset is indicated by 2-byte big endian which are first two offsets. The number of cells is indicated by third and fourth offsets which are also written in 2-byte big endian integers. A page is called free page when all the cells cleared to zeros. The offset of the first appeared cell is
3.3.6 Leaf Page Header

The structure of leaf page header is comparable with internal page header. The size of leaf page header is 8 bytes. First byte represents a flag, this flag contains the value of $0 \times 0D$. Rest of the seven bytes are same as internal page. Since there is no child node present in the leaf page, there is no need of 4-byte information which is present in internal page.

3.3.7 Internal Cell

In SQLite, cells are used as an elementary unit of information storage. Database records are stored in the leaf pages. Pointers that points to the leaf pages are present in internal
A cell can be uniquely identified by a key which is the identification value of the cell, is represented by a variable length integer. Child page number is depicted by the first four bytes in a cell in big endian format.

### 3.3.8 Leaf Cell

Records are stored in three areas namely: cell header, record header, and record data area. Record header length is stored in record header which is represented in a variable length integer format. Byte length information is stored in record data area where each individual byte field of a record is stored.
<table>
<thead>
<tr>
<th>Addr</th>
<th>opcode</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_4$</th>
<th>$P_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Trace</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Goto</td>
<td>0</td>
<td>24</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Null</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>OpenWrite</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>Rewind</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>Column</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>Integer</td>
<td>1</td>
<td>6</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>Ne</td>
<td>6</td>
<td>10</td>
<td>5</td>
<td>collseq(BINARY)</td>
<td>6c</td>
</tr>
<tr>
<td>8</td>
<td>Rowid</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>RowSetAdd</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>Next</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>Close</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>OpenWrite</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>RowSetRead</td>
<td>1</td>
<td>22</td>
<td>2</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>NotExists</td>
<td>0</td>
<td>13</td>
<td>2</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>Null</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>Column</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>String8</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>bye pratik</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>NotExists</td>
<td>0</td>
<td>19</td>
<td>2</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>MakeRecord</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>da</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>Insert</td>
<td>0</td>
<td>5</td>
<td>2</td>
<td>tbl1</td>
<td>5</td>
</tr>
<tr>
<td>21</td>
<td>Goto</td>
<td>0</td>
<td>13</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>22</td>
<td>Close</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>23</td>
<td>Halt</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>24</td>
<td>Transaction</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>25</td>
<td>VerifyCookie</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>TableLock</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>tbl1</td>
<td>0</td>
</tr>
<tr>
<td>27</td>
<td>Goto</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.3: Opcode program for **UPDATE** statement
Chapter 4

Page Replacement of SQLite

SQLite uses logging mechanism to update the content of the database file. The original content of the database page is stored in a journal file.

4.1 Rollback Journal

4.1.1 Updating Disk Content

- Initially all the data reside in the disk, the OS cache and the main memory are empty.

- **Acquiring a Read Lock:** When a transaction wants to read a data from the disk, first it has to obtain a shared lock on the OS cache. A ‘shared’ lock allows two or more database connections to read from the database file at the same time, but a shared lock prevents another database connection from writing to the database file while we are reading it.

- **Reading Data from the Database:** Data is first transferred from the disk to the OS cache and the user space(main memory) reads from OS cache.

- **Obtaining A Reserved Lock:** Like shared lock, reserved lock also gives the liberty to processes to read from database file. Multiple shared locks can be applied along with single reserved lock but, reserved lock cannot be applied on file already
having a reserved lock. This ensures the isolation of write operations from different processes.

- **Creating a Journal File:** SQLite creates a file called journal file, the purpose of this file is to make a copy of the original content in database. The size of the original content is stored in this file, in the form of a small header.

- **Changing Database Pages in User Space:** When the content of the database is successfully saved in the file mentioned in the above point, the pages can be modified in the database.

- **Flushing the Journal File:** The content of the journal file is to be flushed to nonvolatile storage. The first flush writes out the base journal content. The header of the journal is then modified to show the number of pages in the journal. Thereafter, the header is flushed to the disk.

- **Exclusive Lock:** An exclusive lock has to be requested before making any changes in database block or file. Exclusive lock can be obtained in two steps. In the first step SQLite gets a ‘pending’ lock. In step two, this ‘pending’ lock can be amplified to an exclusive lock. The key purpose of the pending lock is not to allow other processes for establishing shared locks on a given file. This also prevents starvation of writer which can be caused by large pool of readers.

- **Writing Changes in the Database File:** A successful accomplishment of an exclusive lock indicates that, its impossible for other processes to read from the file under consideration and it can be written safely. Now, the disk is updated with all the changes done in the database file.

- **Deleting the Journal File:** The Journal file is deleted, once all the changes are done in the disk, as shown in Figure 4.1. The transaction is now said to be committed.

- **Releasing acquired Locks:** Once the transaction is committed, locks is released in order to allow other processes to access the database file.
4.2 When Something Goes Wrong: Rollback

- During the updation of the disk if power cut happens then there may be the situation of partial writes on the disk when the power is re-established. Hence, there is need of rollback mechanism.

- **Hot Rollback Journals:** In the situation of failure, the transaction is not reached to its commit state which ensures the presence of journal file. A journal file is known as hot journal if and only if the following conditions are met:
  - The existence of journal file.
  - The journal file must not be empty.
  - The main database file must not be under any reserved lock.
  - The file header must not be cleared to zero.

- **Rolling back to the previous state:** Rollbacking the database to the previous state is a three step process. In the first step, exclusive lock is requested on the
database file, ensuring mutual exclusion among the processes. The next step is to rollback the incomplete changes, which is done by reading original pages from the journal file and writing it back to the database file. The final step is to delete the journal file, after the successful rollback of the data is completed.
Chapter 5

Shadow Paging

5.1 Problem Description

Flash memory place a key role in embedded system, due its property of non-volatility, low power consumption and resistivity to shock. Although flash memory is not as fast as RAM, it is hundred times faster than a hard disk in read operations. These attractive features make flash memory one of the best choices for portable information systems. However, flash memory has three critical drawbacks.

- If you want to update a particular data segment you have to update the whole page. That is, if you want to update a particular data entry, you have to update the whole page in which the data record exist.

- In contrast to conventional methods of rewriting a particular block, flash memory follow erase-then-write property. In erase-then-write method the memory block is rewritten in two steps, in the first operation all the bits of the block are cleared to binary zeros, its step is termed as erase operation. In the next step all the required bits are toggled. Flash memory technology only allows individual bits to be toggled in one way for writes. The key reason behind this method of updation is the property of the flash memory that allows only one way toggling in case of write operation.

- It is found out that, the maximum life span of a given memory block is atmost 10,00,000 write.
These limitations cause delay in responsiveness of the flash memory device in case of update-in-place mechanism. SQLite uses journal mechanism, which can be termed as log based mechanism. The log management degrades the system performance.

5.2 Proposed Solution

We are proposing a variation of shadow paging. In this mechanism, the original page is never modified, instead the modified data are written on a free page. This rule is applied on both intermediate as well as leaf page, but not on root page.

If we shift the root page from top of the file, then the next time the application is used, it won’t be able to recognize the file. As we have already mentioned in section 3.3.1, that SQLite database does not have a specific file type, it looks for the signature of SQLite at the start of the file. Hence, this rule is not applicable for root page.

When a write-item operation is performed:

- **Step 1**- The modified or new data is written on free or new leaf pages.

- **Step 2**- A new intermediate page is generated which points to the pages generated in step 1.

- **Step 3**- Recursively all the intermediate page are created till we reach the root page.

- **Step 4**- The entry of the highest intermediate page generated in the above step 2 or step 3, is updated on the root page following the conventional journaling mechanism.

Once all the new leaf pages and intermediate pages are written and, its entry is updated in the schema table, the cleaner, erase all the pages keep in clean list, at this state the transaction is said to be committed.

If the system fails while transaction has started and yet not reached its commit state, we can return back to its prior state, since we have not modified the original page contents. Therefore, there is no recovery process involved. We just have to free all the new pages written after transaction started.
5.3 Implementation Detail

This section of the report describes the working model of our approach, as well as it gives the functional flow of the woke done.

The objective is to extend SQLite to use a variation of shadow paging and increase performance by reducing the response time for update, delete, and insert operations. Since, the number of disk write is less in write operation, this approach will yield better result as compared to conventional SQLite journaling implementation: as data is written once to the database (the updated data), not twice (original content in journal and then updated one).

Note, that since in our proposed model we write the updated page in a new page, during this operation there is no need to lock the original page. Hence, it also adds up the ability for read and write transactions to run concurrently.
This design also provides the following:

- Read transactions are never blocked by any other transactions.
- Write transactions are never blocked by read transactions.

So, the only blocking that takes place is, that write transactions block other write transactions. Write transactions always write to free disk pages, so the written data cannot interfere with read transactions. This freedom for read transactions to run concurrently with write transaction, results in increase in database size. The increased database size in the experiment conducted was found to be negotiable.

### 5.3.1 Model

We have used the model shown in Figure 5.2. It has three important components.

![Flash Memory DBMS Model](image)

**End Users:** End Users are the people who interact with the database through applications or utilities. The user interacts by firing queries to read, write, delete, and modify data.

**DBMS:** The DBMS is a software layer between the end user and the database. It consists of three parts. The query is first transferred to the query processor.
• **Query processor:** The query processor is that subcomponent of the DBMS which processes SQL requests. It parses the query and converts it into tokens and checks for the syntax and semantic errors. In addition to this, it also generates the intermediate code for the given query.

• **Transaction Manager:** The intermediate code consists of transactions. Transaction Manager controls the execution of these transactions. It ensures that the transaction does not violate ACID property of the database.

• **Data Manager:** The data manager is the central software component of the DBMS also known as database control system. It controls all the access of the data, stored on the disk, by the database management system. It also controls handling buffers in main memory.

**Flash Memory Page Manager:** The page manager manages the file structure of database. It manages the page table and controls the allocator and cleaner.

It first allocates the free page when required. If there is no free page then it asks the cleaner to clean the previously used pages. If there are no previously used pages, then cleaner cleans a new page from the flash drive and adds it to the free page list. Cleaning a page means changing all the bits to 0.

Although traditional shadow paging schemes is a well-known technique used in many stable applications and softwares, but they just write the leaf page in a new location and update this new page address in the page table. In flash memory updating page table will result in forming a journal for page table and then updating it. This will give no benefit.

Here we proposed the scheme which gives better performance for portable devices, that uses flash memory for data storage.

### 5.3.2 Functional flow

As described in section 3.2, the query execution plan of an update or an insert statement is a VDBE program, which calls insert instruction. Now, we have changed the behavior of this insert instruction to reach our goal. Figure 5.3 shows the functional flow of our approach.
The insert instruction of the VDBE code calls the following algorithm:

- Step 1- Call for insert function with argument current page pointer (pdbpage) and
new data (Ndata).

- Step 2- Check if the current page (pdbpage) is a root page. If no, go to step 3, else goto step 7.
- Step 3- Call for a new page whose address is returned in ifreepage.
- Step 4- Insert the new data in ifreepage.
- Step 5- Add pdbpage in clean list.
- Step 6- Goto step 1 with argument parent page of pdbpage and page map table with updated entry of the ifreepage.
- Step 7- Update the header page of journal file.
- Step 8- Insert the original content of pdbpage in journal file.
- Step 9- Erase the content of pdbpage and insert the new data.
- Step 10- Erase the journal file and the pages keep in clean list.
- Step 11- Set the commit flag and exit.

### 5.3.3 Compilation Process

The android version of SQLite code is different from the conventional SQLite code. Android developers have modified the original code to make it compatible for Android OS. To compile this code we need some of the android libraries. Hence, we have modified the SQLite code and compiled the source code of whole android system to make it work. It is present in the folder `/external/SQLite`. After compiling the android code, it generates SQLite executable code as well its shared object file. Replacing the original SQLite files present in Aakash tablet with those file generate after compiling android code make it work on Aakash tablet.

SQLite executable code is present at the folder `/out/target/generic/system/xbin` and shared object file at `/out/target/generic/system/lib` folder. Once these file are replaced, we can use the modified version of SQLite.
Chapter 6

Experiment and Results

6.1 Schema Used

For experimental purpose, we have used a simple university schema, diagram shown in Figure 6.1 and its ER diagram is shown in figure 6.2. It has six tables:

Figure 6.1: Schema of the Database used for Experiment
For simplification and prediction of number of entries in a particular page, the tables are populated with following entries.

- 5 departments - Electrical, Mechanical, Civil, Maths, Computer Science.
- 10,000 students with 2,000 in each department. tot_grades is zero for all students.
- 15 courses - 3 courses with name basic, intermediate and advance for in each department.
- 17 instructors.
- 15 entries in teaches table - 1 for each subject.
- 20,000 entries of student in takes table distributed evenly.

### 6.2 Results

**Case 1: Small Delete**- Deleting the teaches form all department who are not teaching any courses. **Query**- `DELETE FROM instructor where ins_id NOT IN (SELECT ...`
TINCT ins_id FROM teaches); DB size after the query execution and time taken by both the technique for this query is shown in table 6.2

<table>
<thead>
<tr>
<th>Technique used</th>
<th>Time in milliseconds</th>
<th>Database Size in KB</th>
<th>Expected no. of pages effected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Journaling</td>
<td>3796</td>
<td>788</td>
<td>1</td>
</tr>
<tr>
<td>Shadow Paging</td>
<td>5468</td>
<td>790</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6.1: Time Taken and DB size for Small delete (1 Page)

**Case 2: Small Delete different pages** - De-registering a student for the university.

**Query** - DELETE FROM student where name = 'Pratik Patodi'; Since, the student has taken two subjects, it is likely, that 3 pages will be updated. DB size after the query execution and time taken by both the technique for this query is shown in table 6.2

<table>
<thead>
<tr>
<th>Technique used</th>
<th>Time in milliseconds</th>
<th>Database Size in KB</th>
<th>Expected no. of pages effected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Journaling</td>
<td>6478</td>
<td>788</td>
<td>3</td>
</tr>
<tr>
<td>Shadow Paging</td>
<td>6592</td>
<td>790</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 6.2: Time Taken and DB size for Small delete (3 Page)

**Case 3: Small Update** - Updating the salary of a particular department. UPDATE instructor SET salary = salary + 5000 WHERE dept_id = (SELECT dept_id FROM department WHERE dept_name = 'Computer Science'); DB size after the query execution and time taken by both the technique for this query is shown in table 6.3.

**Case 4: Mass Delete** - Suppose AICTE affiliation of some department is canceled and the university have to shut down the department. In these case there will be large delete. **Query** - DELETE FROM department WHERE dept_name = 'Computer Science');
### Table 6.3: Time Taken and DB size for Small update (1 Page)

<table>
<thead>
<tr>
<th>Technique used</th>
<th>Time in milliseconds</th>
<th>Database Size in KB</th>
<th>Expected no. of pages effected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Journaling</td>
<td>3894</td>
<td>788</td>
<td>1</td>
</tr>
<tr>
<td>Shadow Paging</td>
<td>5643</td>
<td>790</td>
<td>1</td>
</tr>
</tbody>
</table>

DB size after the query execution and time taken by both the technique for this query is shown in table 6.4.

### Table 6.4: Time Taken and DB size for Mass delete (Approx 77 Pages)

<table>
<thead>
<tr>
<th>Technique used</th>
<th>Time in milliseconds</th>
<th>Database Size in KB</th>
<th>Expected no. of pages effected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Journaling</td>
<td>69149</td>
<td>788</td>
<td>77</td>
</tr>
<tr>
<td>Shadow Paging</td>
<td>39236</td>
<td>863</td>
<td>77</td>
</tr>
</tbody>
</table>

### Case 5: Mass Update-

At the end of the semester, if we want to calculate and update the total grades of each student. **Query-**

```
UPDATE student as s SET tot_grade = (SELECT sum(grade) FROM takes where s.roll_no = takes.roll_no GROUP BY roll_no);
```

At the end of the semester update the total grades of all the students. DB size after the query execution and time taken by both the technique for this query is shown in table 6.5.

### Table 6.5: Time Taken and DB size for Mass update (Approx 233 Pages)

<table>
<thead>
<tr>
<th>Technique used</th>
<th>Time in milliseconds</th>
<th>Database Size in KB</th>
<th>Expected no. of pages effected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Journaling</td>
<td>259396</td>
<td>788</td>
<td>233</td>
</tr>
<tr>
<td>Shadow Paging</td>
<td>149086</td>
<td>1021</td>
<td>233</td>
</tr>
</tbody>
</table>

32
Case 6: Select after update- Count the number of subject taken by each student.

Query- 
```
SELECT name, CNT FROM student AS A JOIN (SELECT roll_no, COUNT(*) AS CNT FROM takes GROUP BY roll_no) B ON A.roll_no = B.roll_no;
```

DB size after the query execution and time taken by both the technique for this query is shown in table 6.6.

<table>
<thead>
<tr>
<th>Technique Used</th>
<th>Time in Milliseconds</th>
<th>Database Size in KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Journaling</td>
<td>15223</td>
<td>788</td>
</tr>
<tr>
<td>Shadow Paging</td>
<td>15325</td>
<td>1021</td>
</tr>
</tbody>
</table>

Table 6.6: Time Taken and DB size for Mass update (Approx 233 Pages)

6.3 Analysis

Data from tables 6.4 and 6.5 clearly shows that, by applying our approach, the performance is improved significantly, as compared to conventional journaling mechanism in case of large update and large delete. Read time is almost same for both the mechanisms inspite of different DB size, as the data can be read directly from any location.

Graph in Figure 6.2 summarizes the results shown in the above table.

It is also noted that, the size of database file is increased, but this extra overhead can be negotiated at the cost of decreased response time. In the case of small updates, journaling mechanism performs better then our approach. As we are not only writing the data page (leaf page), but also the page map table (intermediate page) and database header page.
Figure 6.3: Comparison of shadow paging and journaling.
Chapter 7

Further Extension and Related Options

7.1 Adaptive Logging

The results show that no single page replacement mechanism is best for all types of transactions. The performance of a page replacement mechanism depends on the number of pages modified in the transaction. For less number of pages, journaling is better, and for large no of pages shadow paging is better. Hence, the decision of which page replacement mechanism to be used, should be taken at the runtime.

Adaptive logging is a novel page replacement mechanism which can switch from log based method (which is further referred as journal) to shadow paging adaptively at a page level, according to the update state of each page at run time.

It focuses on reducing the update log size in a way, that different logging methods are applied dynamically on run time at a page level switching from Journal to shadow paging.

It is found that, there are two update patterns called A-pattern (Journal-favorable update pattern); and S-pattern (Shadow-paging-favorable update pattern). Figure 7.1 shows these two patterns. A-pattern updates a small area of a data page in a transaction, which generates a small amount of logs. Small update transaction is usually found in traditional OLTP applications.

S-pattern updates a large area of a data page or repeatedly updates the same area
of a data page in a transaction, which causes a large amount of logs. In this situation, shadow paging is better since it doesn’t make any log record, and less number of pages are written as compared to log pages by Journal. We call this type of transaction as large update transaction. Updating large objects is one typical example of the large update transaction.

7.2 Overview

Whenever a transaction changes the pages, log records corresponding to the modification is generated using Journal method, by making a journal file. If the size of this file exceeds our predefined threshold value, there would be a switch in replacement mechanism to
our method. But, if due to other transaction’s operation we are not able to acquire the page lock of the page, journaling will continue to be used. After the switch takes place, every update coming to the page gets reflected into a new page, without generation of log records, till the transaction commits. Log records still gets generated for updates on those table where threshold value has not exceeded. At the time of commit, all the new pages and the logs are written in the disk.

From the next transaction onwards, same journal logging method will be used for updates on the pages. Here application of over approach adaptively takes place only in case where total size of the journal file surpass the threshold value.

7.3 Problem statement

When SQLite is used in Android, we know that the database creates a single database file and limited number of pages within that file. Whenever there is an update request for large area of a data page, lots of logs are being generated, causing large number write operations.

7.4 Solution

We can further enhance replacement mechanism by implementing adaptive logging, which will dynamically switch from journaling to shadow paging, once the size of journal file exceeds the predefined threshold value. The benefits of using adaptive logging are:

- Shifting to shadow paging does not require further logging of records.
- Since the replacement mechanism is selected at the page level, at the time of failure, rollback to the original state can be achieved much quicker.

7.5 Use of Object Oriented Database

An object database (also object-oriented database management system) is a database management system in which information is represented in the form of objects as used
in object-oriented programming. Object databases are different from relational databases which are table-oriented.

We have experimentally observed, Object Oriented Database yield better performance as compared to relational database. Experiments are conducted on SQLite as a relational database and, perst and Berkley J.E. as Object Oriented Database.

- **SQLite:** The modified It is written in C language. Hence, to use it, android uses JNI. In the experiment we have used our modified SQLite which uses shadow paging.

- **Berkley DB JE:** Berkley DB Java edition(JE) is developed at Berkley university. It is written in Java. Hence, it does not uses any JNI to execute its instructions. It uses logging mechanism.

- **Perst DB:** It is an Open Source Database Engine, developed by the McObject company. This company has developed many embedded Database system for different framework like C#, .NET, Java, etc. Here, we have used its Java version.

Berkley DB JE, and Perst DB are in-memory and embedded databases. Embedded means that there is no need to add these database in android source code, instead these are embedded in the application itself. We just have to include the ‘jar’ file of these databases in the application. As these two embedded application are written in Java, they are object-oriented databases. The table here is a class, and the fields of the table are the data member of the class. Each tuple of the table is an object of their respective class. Embedded databases does’t suffers from JNI overhead.

Figure 7.2 shows the time taken (in milliseconds) for all the four basic operations: insert, update, iterate, and delete, for the three database. First part shows time taken for 10 records (1 page), second shows for 100 records (2 pages), third for 1000 records (approx 20 pages), and forth shows for 10,000 records (approx 200 pages).

The results clearly shows, that SQLite suffers from JNI overhead, hence it is slower as compared to Berkley DB even though it uses logging. Embedded in application, helps Berkley DB to achieve fast response time, but Berkley DB is slower then Perst DB which uses shadow paging.
Figure 7.2: Comparison between Berkley DB, Perst DB and SQLite
Chapter 8

Conclusion and Future work

In this work, we identify the effect of page replacement mechanism on response time of android based devices. We proposed a new page replacement mechanism, which is a variation of shadow paging. It is experimentally proved that our approach outperforms the existing journaling approach employed by SQLite in the case of large updates.

8.1 Future Work

- Performance can further be enhanced by implementing ‘Adaptive logging’ as discussed in the previous chapter.

- Efforts are made to explore the possible replacement of relational database by object oriented database. The future work includes the analysis of pros and cons of these database systems.

- The scope of this work is limited to sequential transactions, at the implementation level. We have not taken care of concurrent transaction and neither experiments are not conducted on concurrent transactions, but conceptually we can say that this approach will yield better performance then journaling.
Bibliography


