Enhancement of Open Source Monitoring Tool for Small Footprint Databases

M.Tech. Project Stage-1 Report

Submitted in partial fulfillment of the requirements for the degree of

Master of Technology in
Computer Science and Engineering

Submitted by
Sukh Deo
Roll No. 123050061

Under the Guidance of
Prof. Deepak B. Phatak

Department of Computer Science and Engineering
INDIAN INSTITUTE OF TECHNOLOGY BOMBAY
Powai, Mumbai, India 400076
October 2013
Acknowledgement

I would like to thank my guide, Prof. Deepak B. Phatak, for the consistent encouragement and guidance throughout the project. Due to his consistent encouragement and right directions, we are able to do this project work. I would like to thank Mr. Nagesh Karmali for his continuous inputs in my work. I would also like to thank, Fируза Айбара, Prashant Kumar Singh, Arun Kumar, and B. Raju for proof reading this report and their valuable feedback.
Abstract

Nowadays, mobile devices are not only used for basic purpose like calls, but also for many different purposes and applications. We can say that mobile devices are siblings of personal computers. Their usage patterns are very different as compared to conventional PCs, as the devices are targeted for general users. They also do have some great features, e.g., light weight and efficiency, but has some drawbacks like small memory and battery issues. People want to use many applications, due to the growing number of purposes, but there is a challenge to manage the various applications in mobile device. Performance of mobile device depends on how to manage our applications and there data. There is a need to measure the performance of various commonly used applications.

Our work starts with simple questions like the: does storage affect the performance of mobile devices? What kinds of databases are required for mobile devices? What are the parameters to measure the performance of databases? How to measure the performance of mobile device and its possible tools. Finally, we will discuss the possible way to optimize performance of small foot print database and open source monitoring tool.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgement</td>
<td>2</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>3</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Small footprint databases</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Monitoring Tool</td>
<td>5</td>
</tr>
<tr>
<td>1.4 Overview</td>
<td>5</td>
</tr>
<tr>
<td><strong>2 Literature Survey on Small Footprint Databases</strong></td>
<td>6</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Berkeley DB</td>
<td>6</td>
</tr>
<tr>
<td>2.2.1 Access Methods</td>
<td>7</td>
</tr>
<tr>
<td>2.2.2 Use of Berkeley DB</td>
<td>7</td>
</tr>
<tr>
<td>2.2.3 Limitations</td>
<td>8</td>
</tr>
<tr>
<td>2.3 Perst DB</td>
<td>8</td>
</tr>
<tr>
<td>2.4 SQLite DB</td>
<td>9</td>
</tr>
<tr>
<td>2.4.1 Architecture of SQLite</td>
<td>9</td>
</tr>
<tr>
<td>2.5 Comparison among Berkeley DB, Perst DB, and SQLite DB</td>
<td>10</td>
</tr>
<tr>
<td>2.6 Journal Mode of SQLite</td>
<td>10</td>
</tr>
<tr>
<td>2.6.1 the Rollback Journal</td>
<td>11</td>
</tr>
<tr>
<td>2.6.2 Write-Ahead Log(WAL) Mode</td>
<td>12</td>
</tr>
<tr>
<td>2.6.3 OFF Mode</td>
<td>13</td>
</tr>
<tr>
<td>2.7 Shadow Paging</td>
<td>13</td>
</tr>
<tr>
<td><strong>3 Literature Survey on Monitoring Tool</strong></td>
<td>14</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>14</td>
</tr>
<tr>
<td>3.2 IO Characteristics</td>
<td>14</td>
</tr>
<tr>
<td>3.3 IOzone</td>
<td>15</td>
</tr>
<tr>
<td>3.3.1 Limitations</td>
<td>16</td>
</tr>
</tbody>
</table>
3.4 AndroBench

3.4.1 Microbenchmarks

3.4.2 SQLite Benchmarks

3.4.3 History Tab

3.4.4 Limitation of AndroBench

3.5 MobiBench

3.5.1 History Tab

3.5.2 Setting Tab

3.5.3 Measure Tab

4 Problem Statement and Proposal

4.1 Possible Optimization on SQLite

4.2 Possible Enhancement in MobiBench

5 Conclusion and Future Work

5.1 Conclusion

5.2 Looking Ahead
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Architecture of Berkeley DB</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>Architecture of SQLite</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>Reo1back Journal of SQLite</td>
<td>11</td>
</tr>
<tr>
<td>3.1</td>
<td>Structure of AndroBench Measure Tab</td>
<td>17</td>
</tr>
<tr>
<td>3.2</td>
<td>AndroBench Setting Tab</td>
<td>18</td>
</tr>
<tr>
<td>3.3</td>
<td>Measure tab</td>
<td>19</td>
</tr>
<tr>
<td>3.4</td>
<td>Setting</td>
<td>21</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

Today’s mobile devices are multi-functional devices. It provides usage for many applications, both business and consumer purposes. Smart phones and tablets allow people to access Internet for email, messages, and web browsing. A business person can install many applications like streamline work-flow, save time, keep data accessible, automate daily tasks, and many others. Consumers can view new goods and their prices on Internet, and many more tasks can be done by mobile devices, i.e., smart phones and tablets. We can work on away from the office using these devices. We can synchronize with our personal computer and can get the updated information.

Due to the huge facility and light-weight property of mobile devices, there has been a huge market. People want to use many applications, but these device have some limitations, like small memory and battery backup. There are challenges to manage various applications in these devices. We have to analyze the possible optimization techniques on small footprint database. Research can be broadly done in the application and service, device architecture, and operating system area[7]. Traditionally, storage was not viewed as a critical component of these devices, in term of expected performance.

Performance of mobile devices are to be to continuously improved to fulfill the desire of users to install and execute more applications. Perfor-
mance of mobile devices are based on storage performance, e.g., quality of audio/video playback while number of applications are downloaded in background, storage of web browser history, and browsing speed all of which depends on storage performance. Most mobile devices use embedded Multi-media Card(eMMC), which is based on NAND flash memory. It is also required to use good file systems and I/O schedulers for optimizing the storage performance. In mobile devices, SQLite is used both, as system and application data. Example, SQLite uses to store contacts, message, and bookmarks of the web browser. It stores the entire database in one file. Thus, performance of SQLite depends on storage performance, i.e., how to handle the data and access method within that file.

Due to demand of mobile devices in market, there is need to measure its performance. This performance depends on storage of that device. Based on this motivation, we will discuss small footprint databases, its monitoring tools, and also possible ways for enhancing the performance of monitoring tool.

1.2 Small footprint databases

Small footprint databases are also called lightweight databases or embedded databases. It is specially designed for mobile devices. As these devices are different from personal computers in various ways, we cannot use heavy load databases. The limitation of mobile devices are:

- **Limited Memory:** Most devices use NAND based flash memory. It has limited capability to handle it. The reason for limiting memory is to reduce the battery consumptions.

- **Limited Processing Power:** They have small processing power. Due to the number of services provided over the Internet, such as gaming, music, video, etc., it is required to manage applications for fast responses in an efficient way.

- **Battery consumption:** Battery is an important part of mobile devices. Battery runs out very quickly in mobile device. Nowadays, there is exclusively one research area for reducing the battery consumptions.
Based on this limitations, these devices require such types of database that uses small memory and low power, called Small footprint Database. In the second chapter, we will discuss Berkeley DB, Perst DB, and SQLite DB. These are small footprint databases, each having specific feature. Based on those features, we will select good databases.

### 1.3 Monitoring Tool

The storage performance directly influences the overall performance of mobile device and depends on what kinds of storage medium and databases are used. Due to the low power consumption and high performance, most of the devices use NAND flash memory. Storage IO is one of key factors that govern the overall system performance. Each application generates the specific load. Before measuring the performance of particular application, we have to analyze the function of databases transaction and metadata of storage medium. Based on the throughput, we will be able to enhance the performance of that application. Thereafter, we can enhance the overall performance of these devices. Monitoring tool measures the load of each application that are running on mobile device. Some monitoring tools do not give correct result, because it does not analyze data types and file system of executing application. We have to choose a good monitoring tool, so that, we can enhance the performance of mobile device.

### 1.4 Overview

In this chapter, we have discussed introduction of mobile databases, i.e., light weight databases and monitoring tools. In the second chapter, we will discuss various features and limitation of light weight databases, i.e., Berkeley DB, Perst DB, and SQLite DB. In the third chapter, we will discuss monitoring tools, i.e., IOzone, AndroBench, and Mobibench.
Chapter 2

Literature Survey on Small Footprint Databases

2.1 Introduction

This chapter includes literature study in several small footprint databases. Each database has its own characteristics, based on which we can decide more preferable database. We will analyze the features of Berkeley DB, Perst DB, and SQLite DB.

2.2 Berkeley DB

Berkeley DB is an embedded database. It is open source and written in C language with callable APIs for C#, C++, PHP, and so on. Berkeley DB originated at The University of California Berkeley, but later developed at Oracle Corporation. It supports concurrent access by multiple users. It used to store the data, key/value technique. For example, we can use it to store the user name and password, i.e., user name as key and password as value. It is not a relational database, but we can store multiple data using one key. Architecture of Berkeley DB is shown in figure 2.1. It has three layers. The first layer is the application program, which is directly connected to access methods and recovery.
2.2.1 Access Methods

Berkeley DB provides three types of access methods:

- **B+tree**: B+tree stores records in leaf page. Key in the tree is stored in sorted order. B+tree reduces tree size, and gives better search performance.

- **Hash**: Berkeley DB uses extended linear hashing.

- **Fixed or variable length records**: In this method, each record has a record number, which serves as a key. *For example*, if we want to store a text file, then each line is treated as a record.

2.2.2 Use of Berkeley DB

Berkeley DB provides concurrent access to databases. It can be used as a standalone application, linked by other application. *For example*, in mobile applications, we can use Berkeley DB to store name and mobile number.
2.2.3 Limitations

- It does not support RDBMS. It has no schema.
- It stores only the record key/value technique, so that we cannot compare the data other than key value.

2.3 Perst DB

Perst Database is an open source embedded database, based on Java object oriented architecture. The supported features are as follows:

- Classic B-Tree implementation
- R-tree indexes for spatially-oriented applications such as GIS and navigation
• Time Series class to efficiently deal with small fixed-size objects

• Specialized versions of collections for thick indices (indices with many duplicates), and bit indices (keys with a restricted number of possible values).

It is application specific database for storing real object information. Example, we can use Perst DB for Restaurant searching application, because we can access and store information using making object.

2.4 SQLite DB

SQLite is an embedded database. Due to the limitation of small memory, it does not provide all the feature of RDBMS. Some other features that are not present in RDBMS are zero configuration, self-content, and server-less database[14]. SQLite is a native database. It has a schema, thus, we can design our own tables as we want.

2.4.1 Architecture of SQLite

The architecture of SQLite[2] explain the working of SQLite. This information is very useful, if we want to change inner code of SQLite. Block diagram is shown in figure 2.2.

• **Interface:** This module provides external interface to SQLite library. It is implemented in main.c.

• **SQL command process:** This module has three parts.
  
  − *Tokenizer:* This module, breaks the SQL statement into tokens and calls the parser module.
  
  − *Parser:* This module, parses the meaning of tokens based on the context.
  
  − *Code generator:* After assembling the meaning of token, it generates code that will be used in virtual machine.

• **Virtual Machine:** It executes the generator code.
• **B-tree:** SQLite maintains the data using B-tree. A separate B-tree is made for each table. All B-trees are stored in same file.

• **Pager:** B-tree’s modules request data in the form of chunks. The default size is 1024KB. Page cache takes care of rollback journal and information.

• **OS Interface:** It provides facility for OS service, i.e., disk access, and critical section problems.

In NAND based flash memory, write operation is ten times slower than read operation, because of write before erase and garbages overheads[15]. In SQLite, rollback Journal file system is used[16]. It is good for small transaction and sequential writes. It is costly for random write transactions. For large transaction, we can use shadow paging technique, as it is good for random write.

### 2.5 Comparison among Berkeley DB, Perst DB, and SQLite DB

 SQLite is a native database. It has a schema, so that we can design our own tables as we want. Berkeley DB is an embedded database[17]. It stores the data by key/value schema. Perst DB is an object oriented, embedded database. It is application specific database for storing real object information.

We cannot really compare properly, because each has some special property. Since, SQLite has schema, we can use as the way we want. On an average, SQLite gives better performance amongst others.

### 2.6 Journal Mode of SQLite

There are three journal modes in SQLite, namely, rollback journal, write ahead logging(WAL), and OFF mode. Rollback journal is the default recovery mechanism[18].
2.6.1 the Rollback Journal

In SQLite, when the process wants to change the database file, it first creates a separate rollback journal file and writes the original content of the database pages that are to be altered into the rollback journal file. It contains a header that records the original size of the database file. So if a change causes the database file to grow, we will still know the original size of the database. The page number is stored together with each database page that is written into the rollback journal.

After the original page content has been saved in the rollback journal, the pages can be modified in user memory. Before making changes to the database file itself, we must obtain an exclusive lock on the database file. Once an exclusive lock is held, we know that no other processes are reading from the database file, and it is safe to write changes into the database file. Usually those changes affect operating systems disk cache, and not the mass storage. Another flush must occur to make sure that all the database changes are written into nonvolatile storage.
After the database changes are all safely completed on the mass storage device, the rollback journal file is deleted which is as shown in figure 2.3.

2.6.2 Write-Ahead Log(WAL) Mode

The traditional rollback journal works by writing a copy of the original unchanged database content into a separate rollback journal file and then writing changes directly into the database file[18]. In the event of a crash or rollback journal, the original content contained in the rollback journal is played back into the database file to revert the database file to its original state. The COMMIT occurs when the rollback journal is deleted. The WAL approach inverts this. The original content is preserved in the database file and the changes are appended into a separate WAL file. A COMMIT occurs when a special record indicating a commit is appended to the WAL. Thus a COMMIT can happen without ever writing to the original database, which allows readers to continue operating from the original unaltered database while changes are simultaneously being committed into the WAL. Multiple transactions can be appended to the end of a single WAL file.

Advantages:

- WAL is significantly faster in most scenarios.
- WAL provides more concurrency as readers do not block writers and a writer does not block readers. Reading and writing can proceed concurrently.
- Disk I/O operations tends to be more sequential using WAL.
- WAL uses fewer fsync() operations and is thus less vulnerable to problems on systems where the fsync() system call is broken.

Disadvantages:

- WAL normally requires that the VFS support shared-memory primitives. The built-in unix and windows VFSes support this but third-party extension VFSes for custom operating systems might not.
- All processes using a database must be on the same host computer. WAL does not work over a network file system.
• Transactions that involve changes against multiple attached databases are atomic for each individual database, but are not atomic across all databases as a set.

• It is not possible to change the database page size after entering WAL mode, either on an empty database or by using VACUUM or by restoring from a backup using the backup API. You must be in a rollback journal mode to change the page size.

• It is not possible to open read-only WAL databases. The opening process must have write privileges for ”-shm” wal-index shared memory file associated with the database, if that file exists, or else write access on the directory containing the database file if the ”-shm” file does not exist.

• WAL might be very slightly slower than the traditional rollback-journal approach in applications that do mostly reads and seldom write.

• There is an extra operation of check pointing.

2.6.3 OFF Mode

OFF mode journal is used when there is no need of page replacement technique. Example, while playing the game, there is no need of such page replacement technique, as at time only speed matters. So, we can use OFF mode journal.

2.7 Shadow Paging

In shadow paging approaches, a database maintains two images per page during the transaction, i.e., shadow page and a new page. The shadow page is remain same the duration of the transaction. The new page will be changed when a transaction performs a write operation. In undo process, it will frees new page. In commit process, it will modifie all pointers to old (shadow) page to now point to new page, and frees the shadow page. If the system fails, then shadow pages are used to recover and set the status of the system as it was before the failure[19].
Chapter 3

Literature Survey on Monitoring Tool

3.1 Introduction

IO characteristics of Android based mobile devices are different from desktop computers. We require monitoring tools to measure the storage performance of mobile devices. We have studied different monitoring tools, that are used to monitor the embedded databases. Based on existing monitoring tools, we will discuss what possible features can be enhanced.

3.2 IO Characteristics

There are four types of IO characteristics:

- IO type(Read vs. Write).
- IO size(KB).
- Spatial locality(Sequential vs. Random).
- Temporal locality(Hot vs. Cold).

*Synchronous IO:* Works on polling basis [20]. While polling, if data is available, then we get the data. Otherwise application will block and retry for the data.

*Buffered IO:* Default IO of programs is in buffer, i.e., they are used from
a temporary storage to the requesting program. It will then increase the throughput of the application. We should analyze all the characteristics of an Android device, based on the above characteristics, we can measure correct workload of application process. For example, we need to measure the number of synchronous write operation that takes place in metadata as well as Journal file. Some existing workload does not identify the file types, thus, accuracy level is not good. We will discuss workload tool based on the above four characteristics.

### 3.3 IOzone

IOzone is benchmark tool for file system[21]. It is good for measuring the performance of a file system in computer platform. It is generally used by computer vendors. It measures the IO performance of accessing read, write, fread, and fwrite operations of a single file.

- **Write**: This test measures the performance of writing a new file. It contains overhead of creation of new file and metadata. Metadata contains the overhead address like directory information and space allocation.

- **Re-write**: This test measures the performance of writing a file that is already written. It takes low cost, as there is no overhead of metadata.

- **Read**: This test measures the performance of reading the file.

- **Re-Read**: This test measures the performance of reading the file, that was recently read.

- **Random Read and Random Write**: This test measures performance of read/write, that requires random access file. It includes overhead of seek time and latency.

- IOzone can test file write and read operations, using library function fwrite and fread. It can also test recently fwrite and fread operations, using freeread and freewrite.

- **Mmap**: This maps a file into a users address space. It provides easy access to the data for applications and does not require msync() to waiting for the data.
3.3.1 Limitations

It is file system benchmark tool [6].

- It does not include fsync(). But, SQLite forces to commit data via fsync() to commit on Journal modes. Thus, we cannot measure the correct performance of android device.

- In Android devices [22], there is no way of getting the superuser permission in user level for remounting and unmounting file, but IOzone needs superuser permission. So, it cannot measure the performance of Android devices.

- IOzone must write sequential write operation, before running any other benchmarks, because SQLite uses Journal mode.

Based on these limitations, we cannot measure the performance of SQLite in an Android device.

3.4 AndroBench

AndroBench is a storage benchmarking tool for Android mobile device [22]. It measures the sequential and random IO performance and throughput of various SQLite transaction, i.e., insert, delete, and update. It has two methodology, i.e., microbenchmarks and SQLite Benchmarks [22].

3.4.1 Microbenchmarks

AndroBench uses four micro-benchmarks for measuring the performance of sequential and random IO. For measuring the sequential read performance the step are given below:

- AndroBench creates a file (32MB by default) in target partition.

- The file is sequentially read with fixed buffer size (256kb by default).

- As write operation consumes lot of time, the file is reduced to 2MB for sequential write, which is similar to read operations.
• For random read/write operation: AndroBench measures the number of IO operations per second. The size of each read/write operation is 4KB default. 32MB of file is used for read and 2MB for writes. For measuring the correct performance, it takes the average of each of the three transactions.

![Figure 3.1: Structure of AndroBench Measure Tab](image]

Measure

For measure the sequential read/write IO operation, below step are required:

- First, go to measure tab.
- Second, click the micro button.

It will measure the performance with default configurations. Java does not provide bypass buffer cache, so it is implemented by C.

18
3.4.2 SQLite Benchmarks

For testing the performance of SQLite transaction, AndroBench creates a table, that has 17 columns (twelve integer types and five text types) in target location. It performs three types of tests, i.e., insert, update, and delete. By default, the number of transactions are set to 300, but we can change from setting button. First, it release cache page by using SQLiteDatabase.releaseMemory().

Changing Parameters: Users can change, target partition, file size, and buffer size used in micro benchmarks, and SQLite benchmarks.

![Figure 3.2: AndroBench Setting Tab](image)

After successfully completion of all benchmarks, it transmits the measurement result to the central server. Result is associated by model name and parameter. It measure the performance by transaction per second. Figure 3.1 shows the structure of measuring tab. We can measure the overall performance, SD card, and SQLite transaction by clicking specified button.
The setting tab is shown in figure 3.2. This tab has provides features, we can change buffer size, file size, and partition location as desired.

### 3.4.3 History Tab

In this tab has two button:

- *View history:* Using this button, we can see history of measured performance.

- *Clear history:* Using this button, we can clear the history.

### 3.4.4 Limitation of AndroBench

Although, AndroBench is a powerful tool for measuring the performance of android device, it has some limitations, which are listed below:
• It does not allow changing synchronization option such as O_SYNC, O_DIRECT, and Mmap.

• It has no method to measure the performance of fsync() call.

• Androbench does not support multi-threading benchmark environment.

Based on the above mentioned limitations, AndroBench is not able to measure the correct performance.

3.5 MobiBench

Mobibench is open source mobile storage performance benchmark tool[6]. It combines both characteristics of IOzone and AndroBench. Mobibench is also called workload generator. It analyzes the different application behavior in Android IO and generates the file system work load of applications, i.e., file system workload, random and sequential, and synchronous and buffered IO. It is implemented in two versions. First one is shell application and second one is Android application. Both versions are implemented in C language. It uses Java Native Interface(JNI) to run the API, since Java cannot call the App in C directly. Mobibench provides similar environment to smart-phones, to create the workload.

Mobibench measures databases operation of insert, update, and delete in SQLite. Performance of SQLite depends on PRAGMA options, e.g., synchronous mode or journal mode. It generates three result, i.e., throughput, CPU utilization, and number of context switches. Units of throughput in SQLite operation is transactions per second. There are three button on measure tab, that are:

• **ALL**: Runs File IO and SQLite simultaneously.

• **File IO**: Runs sequential and random read/write operations simultaneously.

• **SQLite**: Runs transactions of insert, update, and delete SQLite operations

• **Custom**: Runs set of tests that an user chose from the Setting tab.

In figure 3.3 shows measure tab of MobiBench.
3.5.1 History Tab

History tab keeps the records of previous runs of benchmarks.

![Figure 3.4: Setting](image)

3.5.2 Setting Tab

In figure 3.4 shows setting tab. In this tab we can select various operations of file IO and SQLite transaction, i.e., insert, delete, and update. It provides options to change partition, number of threads in a benchmark, IO size, File size, IO modes, and journal modes.

3.5.3 Measure Tab

- **Partition**: Changes target partition of the device from “/data” to following partitions.
- `/data`: /data partition of internal storage.
- `/sdcard`: /sdcard partition of internal storage.

- **Number of Threads**: Default mode runs in single thread and it allows to increase the number up to 100 threads.

- **Custom check box**: Using check box, user can select specific tests in a File IO and SQLite benchmarks.

- **File size**: Default file size for write and read is 1MB and 32MB, respectively, but we can change according we want.

- **IO Size**: Changes the size of IO issued in File IO benchmark. Default size is 4KB.

- **Mode(File IO)**: Changes synchronization mode for File IO benchmark. Default mode run in `O_DIRECT` mode. As we can change modes available are as follows:
  - **Normal**: Buffered IO
  - **O_SYNC**: Synchronous IO
  - **fsync**: SQLite use to flush the data via `fsync()` system call.
  - **O_DIRECT**: Direct IO
  - **mmap**: `mmap()` creates file to virtual address space. Then, the program copies the specified size record, e.g., 4KB, to the mmaped memory region. Upon completing copy of all memory copy, this mode flushes file through `msync()` to storage device.

- **Transaction(SQLite)**: We can set number of SQL transaction, where default number of transactions is 100. Maximum number of transaction is 10,000.

- **Mode(SQLite)** We can change the mode of SQLite. Default mode in SQLite test is `FULL`.
  - **OFF**: This mode does not use synchronization mechanism to guarantee that all data is written to the storage device.

23
- **NORMAL:** SQLite uses synchronization mechanism to sync the data to the storage device at certain critical moments, but less often than in FULL mode.

- **FULL:** SQLite database is forced to use synchronized via fsync() system call to ensure that all content is safely written to the storage before processing following IO requests.

• **SQLite Journal:** We can change SQLite journal mode. Default mode of SQLite is rollback journal mode. As available, WAL: Write Ahead Logging mode, OFF mode, and rollback journal mode.
Chapter 4

Problem Statement and Proposal

4.1 Possible Optimization on SQLite

Recently, flash memory has become a critical component in building embedded systems or portable devices, because it is non-volatile, shock-resistant, and uses little power. Although flash memory is not as fast as RAM, it is hundred times faster than a hard disk where read operations are concerned. These attractive features make flash memory as one of the best choices for portable information systems. However, flash memory has two critical drawbacks\(^{19}\).

- A segment, i.e., blocks of flash memory, need to be erased before they can be rewritten. This is because flash memory technology only allows individual bits to be toggled in one way for write operation. The erase operation writes ones or zeros into all the bits in a segment. This erase operation takes much longer than a read or write operation.

- The life of each memory block is limited to 10,000,000 writes.

SQLite uses Rollback journaling or write ahead logging, which follow update-in-place approach. In update-in-place approach, both undo and redo logs gets saved in a log file before the transaction commits, in order to make failure recovery possible. As the name suggests, update-in-place rewrites the same memory block, which does not overcome the above mentioned problems of flash memory. Shadow paging technique is a better solution\(^{18}\). It will not
only increase increase the transaction time in SQLite, but also increase the Life of the flash memory. This is because shadow paging is like a concept of wear-leaving. It will optimize the random write transaction is a very efficient way. As only mapping of pointer in B-tree is required. It is good for mass as well as small transactions. Implementing shadow paging can benefit in the following ways:

- Shadow paging helps in overcoming the above mentioned constraints of flash memory.
- Log based mechanism will not be needed which eliminates the overhead of creating and storing logs, resulting in optimization of storage space.
- At the time of failure there is no need to read from a log file first and then update database. Instead just change the next and previous page pointers.
- Changing pointers rather than reading from file and then updating database, is much simpler and faster resulting in speedy recovery mechanism.

4.2 Possible Enhancement in MobiBench

After analysis of IOzone, AndroBench, and MobiBench, we have found, MobiBench gives better performance among IOzone, AndroBench, and Mobibench. Mobibench has some limitations, that needs to be analyzed.

- MobiBench has no procedure to analyze the IO characteristics of Hot and Cold chunk. If we want to measure the performance of P-SQLite[23], then it will not give the correct performance, because P-SQLite uses hot and cold chunk storage technique for enhancement of performance.
- It has no procedures to measure Shadow paging technique, if we use shadow paging technique then, it will not give the correct result.
- It measures only insert, delete, and update transaction of SQLite, but if we want to measure the select transaction, then it does not give the result.
If we can remove the above mentioned limitations, then it will measure the accurate performance of SQLite databases after shadow paging is implemented.
Chapter 5

Conclusion and Future Work

5.1 Conclusion

Based upon the literature survey, we found SQLite is better for mobile devices, due to feature of RDBMS. We have studied architecture and different journal mode of SQLite, i.e., rollback journal, WAL, and OFF mode. We found that existing journal mode of SQLite is not good for flash memory. We have showed that shadow paging is better for flash memory. Thus, there is a need to implement the shadow paging in SQLite.

The problem is now to determine how shadow paging will give better performance than WAL. Currently, we don’t have any existing monitoring tool, that can measure the performance of shadow paging. This is motivated us to study different monitoring tool, for small foot print databases, i.e., IOzone, AndroBench, and MobiBench. We found MobiBench is an open source benchmark monitoring tool.

5.2 Looking Ahead

In the next stage we plan to do following tasks:

• Implement shadow paging in SQLite.

• Change the source code of MobiBench, so that we can compare the performance between shadow paging and WAL.
References


